**Project 1: RHMS & Device Management**

RHMS ( Remote Health Monitoring System ) is a technique, used to monitor the health status of a device remotely.

It examine the device module wise performance at that instant and reports to the server.

**\*** Need to RHMS

To monitor the device on regular basis depends on User requirement.

Device Management is a mechanism to upgrade a u-boot, kernel, application and other files related to the root file system.

**\*** Need for Upgrade

Enhancements

Bug fixes

Improve Product Performance

Maximum reliability

Coming to Working functionality of the DM and RHMS.

It is a server and client communication, I.e, request and response.

\* Device registration,

Each device is identified by unique serial number.

RHMS Server application supports multiple projects, multiple firmware , multiple application.

We are registering appropriate project, firmware, applications as per the production department selling.

\* Communication Part

RHMS is supports https communication.

The communication is divided into 5 types.

1. Hardware Status call:-

2. Boot Status Call

3. Periodic Status Call

4. Firmware Call

5. Application Call

First three calls are managed by RHMS application and second Two calls are managed by device Management application.

RHMS application is call all required hardware information related routines. And placed the hardware related information into hardware.xml file . And pushes the same report to RHMS server.

Hardware Status call will happened only when hardware related changes.

Boot time Status is call will send only once after the device boot on. This routine initializes and examine module wise details and software versions ( Firmware / Applications / Other PDS related versions ). And send information to server.

Periodic Health calls having information related GPS, Communication mode, Battery, RAM and internal memory used,free and total information. The same will send to the server.

Each xml Have Serial ID information.

Above each call requested the server with xml information. And this will get configuration information for next periodic report sends.

Device Management will request firmware and application calls to server , On Successful server will send registered related Firmware name , version and URL . No of Applications related Name , version , Patch URL.

Internally application will maintain registration related all Firmware and Application information.

Version .

On Every hit device version will compared with server versions and downloads if there any update found. And install to the device on next boot.

**Project 2: IO Control Driver Implementation**

This driver basically implemented for power control of all the modules.

This driver was developed for POS machine in which we are accessing the GPIO pins to do the power control of all peripherals like Bluetooth, GSM, GPS, Audio, Buzzer, Flash, Camera, Finger Print Scanner, USB e.t.c,

Driver Implementation

1. Registration of static character driver , define ioctl operations for the inode.

2. hardware all peripherals GPIO Pin initialization , gpio request, gpio set direction , and values

3. initialization of kthread

4. Timer initialization and start a timer with 10+jiffies // Usually HZ 100 = 10ms HZ 1000 = 1ms jiffies

So Every 10 jiffies counts, timer will wake up the wait\_event

**void add\_timer(struct timer\_list \****timer***); // Start a timer**

static struct timer\_list gl11\_timer ;

gl11\_timer.function = gl11\_timer\_work ;

gl11\_timer.expires = jiffies+10;

In Thread,

{

Work thread

{

we demeonize this thread and we are entering into continuous while loop

in that loop we are waiting for timer interrupt through

wait\_event\_interruptible(wq,(atomic\_read (&data\_present)

in Which loop we are handling, on key click or touch click , we giving buzzer sound and LCD bkl on

If any want to off the background sound also we are giving support in this driver only.

Basically How we did is Exporting the symbols of key and touch variables

}

static int ioc\_ioctl(struct file \*flip, unsigned int cmd, unsigned long arg);

Another way to do IO cntrl operations :-

mount -t debugfs debugfs /sys/kernel/debug/gpio // See GPIO pins Using list.

/sys/class/gpio if want to access the gpios , must and should enable the kernel.

use export and unexport

echo 24 >/sys/class/gpio/export

echo out >/sys/class/gpio/gpio24/direction

echo 24 >/sys/class/gpio/unexport

GPIO Driver:

A GPIO chip handles one or more GPIO lines. To be considered a GPIO chip, the lines must conform to the definition: General Purpose Input/Output. If the line is not general purpose, it is not GPIO and should not be handled by a GPIO chip.

methods to establish GPIO line direction

methods used to access GPIO line values

method to set electrical configuration for a given GPIO line

method to return the IRQ number associated to a given GPIO line

**Project 3: Matrix Keyboard Driver**

This driver is used to reports key events to kernel input subsystem. Matrix Keypad is input device to CPU.

We are using matrix keyboard I.e, 8\*8. So it can produce 64 combinations of unique key positions.

The Matrix keypad supports multiple rows and columns, a key can be placed at intersection of a unique row and unique column.

When a key is pressed, a row and a column make a contact. otherwise, there is no connection between rows and columns. So this is the logic we are going to use.

Scanning is detecting a row number and a column number is pressed

The keyboard produces scan codes, The scan codes are assembled into key codes.

( One unique code for each key ). and Key codes are converted to input characters using the kernel maps.

We are registering the keypad as a platform device.

Devices which are directly memory mapped are referred to as platform devices. ( Devices found on Soc

platform like Device controller, Bus controller, and port devices / bus interfaces ) fall into this category.

\* Kernel provides a layer called platform core to manage drivers of platform devices.

\* Kernel BSP code enumerates data structures which represent platform devices found on soc, this list

is managed by platform core

**Changes are Done in kernel arm board file**

\* Registering or adding a device to platform core.

\* Registering the keypad related information using struct platform\_device ,

Customized keypad or key codes data , gpio columns , gpio rows data and their sizes are placing in one structure, I.e, matrix\_keypad\_platform\_data and that structure object is passing to platform\_device of .platform\_data, I.e, member of struct platform\_device

static struct platform\_device mx6sl\_matrix\_gpio\_device = {

.name = "matrix-keypad",

.id = -1,

.dev = {

.platform\_data = &mx6sl\_matrix\_gpio\_pdata,

},

};

static struct matrix\_keypad\_platform\_data mx6sl\_matrix\_gpio\_pdata = {

.keymap\_data = &mx6sl\_matrix\_gpio\_keymap\_data,

.row\_gpios = mx6sl\_matrix\_gpio\_row\_gpios,

.col\_gpios = mx6sl\_matrix\_gpio\_col\_gpios,

.num\_row\_gpios = ARRAY\_SIZE(mx6sl\_matrix\_gpio\_row\_gpios),

.num\_col\_gpios = ARRAY\_SIZE(mx6sl\_matrix\_gpio\_col\_gpios),

.wakeup = 1,

};

mx6sl\_gpio\_matrix\_keymap[]{ All keys }

standard prototype

struct platform\_device {

const char \*name;

u32 id;

struct device dev;

u32 num\_resources;

struct resource \*resource;

};

Platform drivers

~~~~~~~~~~~~~~~~

Platform drivers follow the standard driver model convention, where

discovery/enumeration is handled outside the drivers, and drivers

provide probe() and remove() methods. They support power management

and shutdown notifications using the standard conventions.

struct platform\_driver {

int (\*probe)(struct platform\_device \*);

int (\*remove)(struct platform\_device \*);

void (\*shutdown)(struct platform\_device \*);

int (\*suspend)(struct platform\_device \*, pm\_message\_t state);

int (\*suspend\_late)(struct platform\_device \*, pm\_message\_t state);

int (\*resume\_early)(struct platform\_device \*);

int (\*resume)(struct platform\_device \*);

struct device\_driver driver;

};

**Changes are done in keypad driver**

Registering driver with platform core through

platform\_driver\_register(&matrix\_keypad\_driver); or

module\_platform\_driver(matrix\_keypad\_driver);

static struct platform\_driver matrix\_keypad\_driver = {

.probe = matrix\_keypad\_probe,

.remove = \_\_devexit\_p(matrix\_keypad\_remove),

.suspend = matrix\_keypad\_suspend,

.resume = matrix\_keypad\_resume,

.driver = {

.name = "matrix-keypad",

.owner = THIS\_MODULE,

},

};

In probe

we are getting platform data using various kernel defined function calls

platform\_get\_drvdata(pdev)

Allocating input device instance , and copying platform data into input data refrence

static int \_\_devinit init\_matrix\_gpio(struct platform\_device \*pdev,

struct matrix\_keypad \*keypad) //

This api will used to gpio request and direction set

input\_register\_device(keypad→input\_dev);

setup\_timer(&keypad->timer, kpad\_timer, (unsigned long) pdev);

mod\_timer(&keypad->timer, jiffies + 5) ;

device\_init\_wakeup(&pdev->dev, pdata->wakeup);

platform\_set\_drvdata(pdev, keypad);

Setup\_timer is set up a timer with data

That data is passing to handler,

kpad\_timer( data )

{

if ( encoder\_gs() == 0 ) // if any key is pressed then , gpio interrupt pin will be 0 // That is registered as input gpio

{

After 5 jiffes delay we get columns scan data //

By combing 3 columns data we will get decoder column number ,

code = MATRIX\_SCAN\_CODE((new\_state[rowdata]), rowdata, keypad→row\_shift);

#define MATRIX\_SCAN\_CODE(row, col, row\_shift) (((row) << (row\_shift)) + (col)) //

Getting Scan code value using above macro.

Inputs are encountered row and column,

**input\_report\_key(input\_dev,keypad→keycodes[code],0);** // Report key event to kernel subsystem

**In Function Key implementation , If we are adding column number + 7, Otherwise sending as usual**

**In Shift Key Implementation Sending input\_report\_key(input\_dev,keypad→keycodes[code],1);**

**Here Last argument if we pass that, 0 means No shift Pressed**

**1 Means Shift is Pressed , So Kernel sub system internaly map the appropriate special key information.**

**}**

else

{

\* MX6\_BRD\_LCD\_DAT8\_ROW0 // gpio\_set\_value // Setting ROWS gpios pins as per encoder logic

I.e, 000 001 002

\*After that checking if encoder\_ds ( I.e, key press event detection pin ), if this encounter we are scan value of the row , ie, row 0 1 2 3 4 5 6 7 8\*8 //

Getting Row Scan value from above loop

}

mod\_timer(&keypad->timer,(jiffies + 5)) ;

}

in remove

We de registering the same allocated instance

Keyboard drivers should use EV\_KEY to report key presses,

void input\_event(struct input\_dev \* dev, unsigned int type, unsigned int code, int value) ;

report new input event.

input\_event(input\_dev, EV\_MSC, MSC\_SCAN, code);

input\_report\_key(input\_dev,keypad→keycodes[code],1);