**Matrix Keyboard Driver**

Keyboard is input device to CPU. The keyboard is interfaced with GPIO controller.

The Matrix keypad supports multiple rows and columns, a key can be placed at intersection of a unique row and unique column. We are using matrix keyboard I.e, 8\*8. So it can produce 64 combinations of unique key code.

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);