**PROJECT**

1. **Building Linux on to Raspberry Pi and Bring up the board and understand and explain in detail of all the steps involved in kernel compilation?**
2. **Write a character device driver with file operations and communicate with the character driver and user application and explain in detail withs sample code that works on Raspberry pi ?**
3. **Explain in detail the Linux device driver framework considering any one of the Hardware protocols ?**
   1. **I2C**
   2. **SPI**
   3. **UART**

I2C is an acronym for the “Inter-IC” bus, a simple bus protocol which is widely used where low data rate communications suffice. I2C is also the acronym for the microprocessor I2C peripheral interface. Around the microprocessor device, the user can add many I2C external devices to create a custom board. Each external device can be accessed through the I2C from the user space or the kernel space.

Here are some details from the Linux kernel website,

## Driver Structure

Driver structure contains general access routines, and should be zero-initialized except for fields with data you provide. A client structure holds device-specific information like the driver model device node, and its I2C address.

static struct i2c\_device\_id foo\_idtable[] = {

{ "foo", my\_id\_for\_foo },

{ "bar", my\_id\_for\_bar },

{ }

};

MODULE\_DEVICE\_TABLE(i2c, foo\_idtable);

static struct i2c\_driver foo\_driver = {

.driver = {

.name = "foo",

.pm = &foo\_pm\_ops, /\* optional \*/

},

.id\_table = foo\_idtable,

.probe = foo\_probe,

.remove = foo\_remove,

/\* if device autodetection is needed: \*/

.class = I2C\_CLASS\_SOMETHING,

.detect = foo\_detect,

.address\_list = normal\_i2c,

.shutdown = foo\_shutdown, /\* optional \*/

.command = foo\_command, /\* optional, deprecated \*/

}

## Extra client data

/\* store the value \*/

void i2c\_set\_clientdata(struct i2c\_client \*client, void \*data);

/\* retrieve the value \*/

void \*i2c\_get\_clientdata(const struct i2c\_client \*client);

## Accessing the client

int foo\_read\_value(struct i2c\_client \*client, u8 reg)

{

if (reg < 0x10) /\* byte-sized register \*/

return i2c\_smbus\_read\_byte\_data(client, reg);

else /\* word-sized register \*/

return i2c\_smbus\_read\_word\_data(client, reg);

}

int foo\_write\_value(struct i2c\_client \*client, u8 reg, u16 value)

{

if (reg == 0x10) /\* Impossible to write - driver error! \*/

return -EINVAL;

else if (reg < 0x10) /\* byte-sized register \*/

return i2c\_smbus\_write\_byte\_data(client, reg, value);

else /\* word-sized register \*/

return i2c\_smbus\_write\_word\_data(client, reg, value);

}

## Initializing the driver

static int \_\_init foo\_init(void)

{

return i2c\_add\_driver(&foo\_driver);

}

module\_init(foo\_init);

static void \_\_exit foo\_cleanup(void)

{

i2c\_del\_driver(&foo\_driver);

}

module\_exit(foo\_cleanup);

The module\_i2c\_driver() macro can be used to reduce above code.

module\_i2c\_driver(foo\_driver);

## I2C communication

int i2c\_master\_send(struct i2c\_client \*client, const char \*buf,

int count);

int i2c\_master\_recv(struct i2c\_client \*client, char \*buf, int count);

int i2c\_transfer(struct i2c\_adapter \*adap, struct i2c\_msg \*msg,

int num);

Submission requirement:

**Individual recording of each module should be uploaded along with the proper documentation.**

**Due Date : 10/12/2021**