Invotronics
Embedding Ideas into Reality

I2C Driver for Linux Based Embedded System

Posted on September 30, 2014 by invo-tronics

Today Linux is the operating system choice for a wide range of special-purpose electronic devices known as embedded systems. An embedded system is specifically designed to perform a set of designated activities, and it generally uses custom, heterogeneous processors. This makes Linux a flexible operating system capable of running on a variety of architectures, such as ARM and many others.

Linux has highly modular architecture and it facilitates the porting and a lot of efforts are required to build new kernel components to fully support the target platform. A big part of these efforts are in developing the low-level interfaces commonly referred to as device drivers. A device driver is a piece of software designed to direct control a specific hardware resource using an hardware-independent well defined interface.

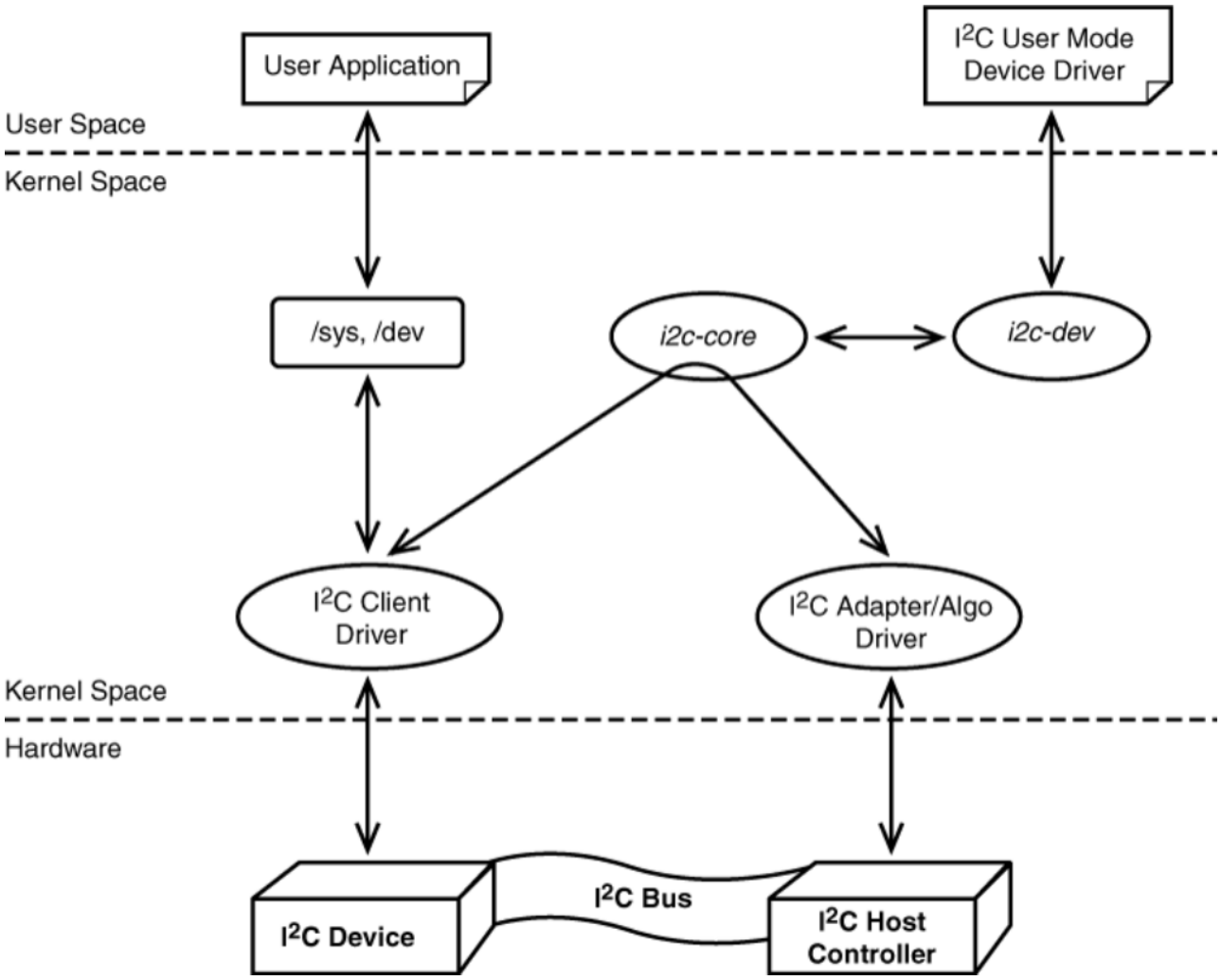
The kernel I2C subsystem is divided into Buses and Devices, and then further buses into Algorithms and Adapters, and devices into Drivers and Clients.

Algorithms
An Algorithm performs the reading and writing of I2C messages to the hardware, this may involve bit banging GPIO lines or writing to an I2C controller chip. An Algorithm is represented by the structure i2c_algorithm and allows you to define function pointers to functions that can write I2C messages (master_xfer) or SMBus messages (smbus_xfer).

Adapters
An Adapter represents a bus which is used to tie up a particular I2C/SMBus with an algorithm and bus number. It is represented by the structure i2c_adapter. If you imagine a system where there are many I2C buses – perhaps two controlled by a controller chip and one bit-banged – then you would expect to see 3 instances of an i2c_adapter and 2 instances of an i2c_algorithm.

Clients
A Client represents a chip (slave) on the I2C/SMBus such as a Touchscreen, RTC or ADC-DAC peripheral and is represented by a structure i2c_client. This includes various members such as chip address, name and pointers to the adapter and driver.

Drivers
A driver, represented by a structure i2c_driver represents the device driver for a particular class of I2C/SMBus slave devices. The structure contains a bunch of function pointers like probe and remove pointers.



Linux I2C Subsystem

The I2C core layer and its attendant benefits is an example of how Linux helps portability. For instance, enabling I2C on a new platform requires only to design the hardware-dependent components, namely the bus driver and the client drivers, whereas the core layer needs not to be changed.

Initializing and probing of I2C Client Driver

During initialization the driver registers itself with the I2C core. This is achieved by populating a structure i2c_driver and passing it as argument to the function i2c_add_driver(), as shown in (Listing-1). The structure i2c_driver holds pointers to the probe and remove functions that are executed respectively on device probing and when the device is removed. The id_table member of the structure i2c_driver informs the I2C framework about which slave devices are supported by the driver. In this case the only chip supported is named lis3lv02d. The names of the supported devices are important for binding, as explained next.

```
1 /* Device and driver names */
2 # define DEVICE_NAME “lis3lv02d”
```

```
3
4/* I2C client structure */
5 static struct i2c_device_id lis3lv02d_idtable [] = {
6 { DEVICE_NAME , 0 },
7 {}
8 };
9 MODULE_DEVICE_TABLE (i2c , lis3lv02d_idtable );
10
11 static struct i2c_driver lis3lv02d_driver = {
12 . driver = {
13 . name = DRIVER_NAME
14 },
15 . probe = lis3lv02d_probe ,
16 . remove = __devexit_p ( lis3lv02d_remove ),
17 . id_table = lis3lv02d_idtable ,
18 };
19
20 /* Module init */
21 static int __init lis3lv02d_init ( void )
22 {
23 return i2c_add_driver (& lis3lv02d_driver );
24 }
```

Listing-1: Registration of the LIS3LV02DL driver.

The binding process consists of associating a device with a driver that can control it. In embedded systems where the number of the I2C bus and the devices connected to it are known for a fact, it is possible to declare in advance the I2C slaves which live on the bus. This is typically done in the board setup file (Listing-2).

The mysoc_platform_init function is executed on board startup and, among other tasks, registers the I2C slave devices by invoking the i2c_register_board_info function with arguments that specify the number of the bus (zero in this case) and the devices connected with it. This is done through an array of structure i2c_board_info(), each item of which specifies the device name and the device address, with the former that must match with the name registered by the driver in order for binding to succeed. In this case structure i2c_board_info holds only one item which corresponds to the LIS3LV02DL inertial sensor.

Since the sensor's chip has an interrupt line tied to the cpu, the irq member is also specified with the respective IRQ number. By means of another member called platform_data it is possible to define custom data for the driver.

```
1 /* I2C devices */
2 static struct i2c_board_info mysoc_i2c_devices [] = {
3 {
4 I2C_BOARD_INFO (" lis3lv02d ", 0 x1D ),
5 . irq = MYSOC_GPIO_TO_IRQ (82) ,
```

```
6 /* No platform data : use driver defaults */
7 },
8 };
9
10 static void __init mysoc_platform_init ( void )
11 {
12 …
13 /* Register I2C devices on bus #0 */
14 i2c_register_board_info (0, mysoc_i2c_devices ,
15 ARRAY_SIZE ( mysoc_i2c_devices ));
16 …
17 }
```

Listing 2: Registration of the I2C devices.

During boot the kernel looks for any I2C driver that has registered a matching device name, that is “lis3lv02d”. Upon finding such a driver, the kernel invokes its probe() function passing a pointer to the LIS3LV02DL device as a parameter. This process is called probing. The probe function is responsible for the per-device initialization, that is initializing hardware, allocating resources, and registering the device with any appropriate subsystem.

More in detail, the LIS3LV02DL probe function takes the following actions:

1. Allocate memory for lis3lv02d_priv private data structure.
2. Load the device settings.
3. Identify the LIS3LV02DL chip.
4. Configure the device hardware.
5. Create the per-device sysfs nodes.
6. If the free-fall feature is enabled, request the interrupt and register the IRQ for the free-fall detection.
7. If the device polling feature is enabled, register the device with the input subsystem.

On successful completion of all the above steps, meaning a successful probing, the device is bound to the driver.

Initializing and probing of I2C Bus Driver

Initializing and probing the dummy MYSOC I2C bus driver is performed in a similar way as for LIS3LV02DL client driver, with the major difference being that the former uses a platform bus. The platform bus requires that any I2C adapter (or equivalently controller), which is controlled by the bus driver, be registered using a platform_device structure. This structure represents the bus adapter and provides information such as the device name, the device resources and the adapter number, to the bus driver.

Usually the registration of the I2C adapters with the platform bus is performed by the board initialization file, as the information needed are highly board specific. Listing-3 shows the part relevant to this matter.

```
1 /* first bus : i2c0 */
2 static struct platform_device mysoc_i2c_dev0 = {
```

```
3 .name = “mysoc_i2c “,
4 .id = 0,
5 . resource = &mysoc_i2c_resources [0] ,
6 . num_resources = 2,
7 . dev = {
8 . platform_data = &mysoc_i2c_dev0_data ,
9 },
10 };
11
12 /* second bus: i2c1 */
13 static struct platform_device mysoc_i2c_dev1 = {
14 . name = “mysoc_i2c”,
15 .id = 1,
16 . resource = &mysoc_i2c_resources [2] ,
17 . num_resources = 2,
18 /* No platform data : use driver defaults */
19 };
20
21 static int __init mysoc_i2c_init ( void )
22 {
23 …
24 platform_device_register (&mysoc_i2c_dev0 );
25 …
26 platform_device_register (&mysoc_i2c_dev1 );
27 …
28 }
29 arch_initcall ( mysoc_i2c_init );
```

Listing 3: Registration of the I2C platform device with the platform bus.

On the driver’s side, the registration with the platform bus is achieved by populating a structure platform_driver and passing it to the macro module_platform_driver() as argument (Listing-4). The platform bus simply compares the driver.name member against the name of each device, as defined in the platform_device data structure (Listing 3); if they are the same the device matches the driver.

```
1 # define DRIVER_NAME “mysoc_i2c”
2
3static struct platform_driver mysoc_i2c_driver = {
4 . probe = mysoc_i2c_probe ,
5 . remove = __devexit_p ( mysoc_i2c_remove ),
6 . driver = {
7 . name = DRIVER_NAME ,
8 . owner = THIS_MODULE ,
```

```
9 .pm = &mysoc_i2c_pm_ops ,
10 },
11 };
12 module_platform_driver ( mysoc_i2c_driver );
```

Listing 4: Registration of the I2C platform driver with the platform bus.

As usual, binding a device to a driver involves calling the driver's probe() function passing a pointer to the device as a parameter.

The sequence of operations performed on probing are the following:

1. Get the device resource definitions.
2. Allocate the appropriate memory and remap it to a virtual address for being accessed by the kernel.
3. Load the device settings.
4. Configure the device hardware.
5. Register with the power management system.
6. Create the per-device sysfs nodes.
7. Request the interrupt and register the IRQ.
8. Set up the struct i2c_adapter and register the adapter with the I2C core.

On successful completion of above steps the driver is bounded to the devices representing the two mysoc I2C controllers.

In the Linux I2C subsystem an I2C bus driver consists of an adapter driver and an algorithm driver. This division is to improve the software reuse and to allow portability. An algorithm driver is intended to contain general code that can be used for a whole class of I2C adapters, while each specific adapter driver either depends on one algorithm driver, or includes its own implementation.

However, while having a generic algorithm that works for multiple adapters is suitable for many cases, in embedded systems, where each I2C bus adapter has its own way of interfacing with the processor and the bus, it is usual to develop the adapter driver together with its corresponding algorithm driver. The bus driver registers with the I2C subsystem by using a structure i2c_adapter that is instantiated and initialized by the I2C platform driver's probe() function, as shown in Listing 5.

The i2c_adapter structure's algo member is set up to point to a structure i2c_algorithm which in turn holds two pointers:

- master_xfer points to the function that implements the actual I2C transmit and receive algorithm.
- functionality points to a function that returns the features supported by the I2C adapter.

To communicate with a client the I2C subsystem offers two class of functions: one for I2C plain communication which includes i2c_master_send(), i2c_master_recv() and i2c_transfer(), and a second one that uses SMBus commands. However, whichever method is used, the data transfer relies on the bus driver's function pointed to by master_xfer, as the I2C core ultimately calls this function for the actual transfer to take place.

```
1 /* I2C algorithm structure */
2 static struct i2c_algorithm mysoc_i2c_algo = {
3 . master_xfer = mysoc_i2c_xfer ,
4 . functionality = mysoc_i2c_func ,
5 };
6
7/* Probe function */
8 static int __devinit mysoc_i2c_probe
9 ( struct platform_device * pdev )
10 {
11 …
12 adap = kzalloc ( sizeof ( struct i2c_adapter ),
13 GFP_KERNEL );
14 …
15 adap -> algo = &mysoc_i2c_algo ;
16 …
17 err = i2c_add_numbered_adapter ( adap );
18 …
19 }
```

Listing 5: Registration of the I2C adapter.

As shown in listing 5, mysoc_i2c_xfer() is the transfer function installed by the I2C bus driver. This function receives an array of messages as argument and processes them in sequence by calling mysoc_i2c_xfer_rd() or mysoc_i2c_xfer_wr() depending on whether the message being processed is marked for read or write. Once all messages have been sent mysoc_i2c_xfer() successfully returns, otherwise, upon detecting a communication error, aborts the transmission and returns an appropriate error code.

This entry was posted in Technical and tagged Driver, Embedded, I2C, Linux. Bookmark the permalink.

One Response to *I2C Driver for Linux Based Embedded System*

Jagannath Pattar *says:*
February 21, 2016 at 12:34 PM

I2C driver/adapter/client explained nicely!

Reply

Invotronics
Proudly powered by WordPress.