DEVELOPMENT OF A DRIVER IN LINUX/ANDROID

by: Abisai Negrete
Application Engineer

Table Of Contents

MQX Board Support Package Porting Guide..**Error! Bookmark not defined.**
1 Introduction 1
2 CW7.2 BSP Creation**Error! Bookmark not defined.**
3 IAR BSP Creation**Error! Bookmark not defined.**
3.1 .... Copy BSP and PSP Projects **Error! Bookmark not defined.**
3.2 ....Copy BSP Folder **Error! Bookmark not defined.**
3.3 .... Copy the configuration folder **Error! Bookmark not defined.**
3.4 .... Edit Batch Files **Error! Bookmark not defined.**
3.5 .... Edit IAR Project Settings **Error! Bookmark not defined.**
3.6 .... Add BSP and PSP Files **Error! Bookmark not defined.**
3.7 .... Modify the other libraries **Error! Bookmark not defined.**
3.8 .... Use the new library in a project **Error! Bookmark not defined.**

1 Introduction

2 Required Knowledge

Before explain how to program and use a driver into the i.MX families running Linux or Android as OS, this document will explain a few concepts that the developer should know to understand and program drivers or applications.

What is a driver?
A driver is a computer program (software) that allows higher-level computer programs and operating system how to interact with a hardware device.
The Role of a device driver
When you have designed and programmed a driver you will provide a mechanism, not policy, to the users and

© Freescale Semiconductor, Inc., 2007. All rights reserved.

freescale semiconductor

operating system. The distinction between mechanism and policy is one of the best ideas behind the UNIX design

Most programming problems can indeed be split into two parts: “what capabilities are to be provided” (the mechanism) and “how those capabilities can be used” (the policy). If the two issues are addressed by different parts of the program, or even by different programs altogether, the software package is much easier to develop and to adapt to particular needs.

When writing drivers, a programmer should pay attention to this fundamental concept: write kernel code to access the hardware, but don’t force particular policies on the user, since different users have different needs. The driver should deal with making the hardware available, leaving all the issues about how to use the hardware to the applications. As a conclusion a driver is flexible if it offers access to the hardware capabilities without adding constraints. Sometimes, however, some policy decisions must be made. For example, a digital I/O driver may only offer byte-wide access to the hardware in order to avoid the extra code needed to handle individual bits.

Task in the kernel

The kernel is the big chunk of executable code in charge of handling all such requests. And the kernel’s role can be split (as shown in Figure 1-1) into the following parts:

- *Process management :* The kernel is in charge of creating and destroying processes and handling their connection to the outside world (input and output). Communication among different processes (through signals, pipes, or interprocess communication primitives) is basic to the overall system functionality and is also handled by the kernel. In addition, the scheduler, which controls how processes share the CPU, is part of process management. More generally, the kernel’s process management activity implements the abstraction of several processes on top of a single CPU or a few of them.
- *Memory management :* The computer’s memory is a major resource, and the policy used to deal with it is a critical one for system performance. The kernel builds up a virtual addressing space for any and all processes on top of the limited available resources. The different parts of the kernel interact with the memory-management subsystem through a set of function calls, ranging from the simple malloc/free pair to much more complex functionalities.
- *Filesystems:* Unix is heavily based on the filesystem concept; almost everything in Unix can be treated as a file. The kernel builds a structured filesystem on top of unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system. In addition, Linux supports multiple filesystem types, that is, different ways of organizing data on the physical medium. For example, disks may be formatted with the Linux-standard ext3 filesystem, the commonly used FAT filesystem or several others.
- *Device control:* Almost every system operation eventually maps to a physical device. With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called a device driver. The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape drive. This aspect of the kernel’s functions is our primary interest in this book.

- *Networking:* Networking must be managed by the operating system, because most network operations are not specific to a process: incoming packets are asynchronous events. The packets must be collected, identified, and dispatched before a process takes care of them. The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity. Additionally, all the routing and address resolution issues are implemented within the kernel.

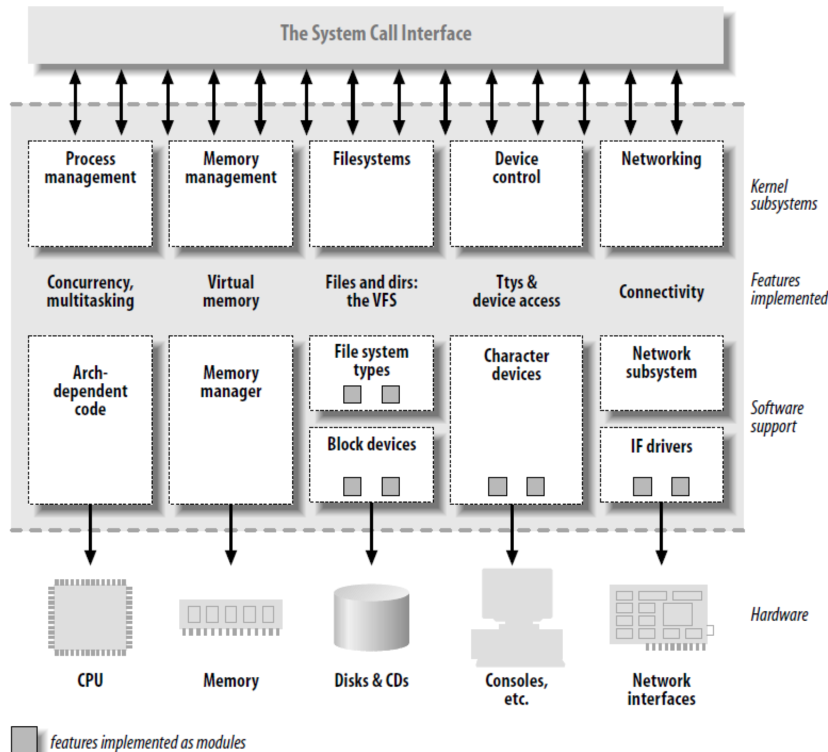

Figure 1 Kernel Split

The Linux way of looking at devices distinguishes between three fundamental device types. Each module usually implements one of these types, and thus is classifiable as a char module, a block module, or a network module. This division of modules into different types, or classes, is not a rigid one; the programmer can choose to build huge modules implementing different drivers in a single chunk of code.
The three classes are:

- Character devices

It is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls. The text console (/dev/console) and the serial ports (/dev/ttyS0 and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as /dev/tty1 and /dev/lp0.

The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless, char devices that look like data areas, and you can move back and forth in them.

- Block devices

Like char devices, block devices are accessed by filesystem nodes in the /dev directory. A block device is a device (e.g., a disk) that can host a filesystem. Linux allows the application to read and write a block device like a char device—it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node, and the difference between them is transparent to the user. Block drivers have a completely different interface to the kernel than char drivers.

- Network interfaces

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are, usually, designed around the transmission and receipt of packets. A network driver knows nothing about individual connections; it only handles packets. Not being a stream-oriented device, a network interface isn't easily mapped to a node in the filesystem, as /dev/tty1 is. The way to provide access to interfaces is still by assigning a unique name to them (such as eth0), but that name doesn't have a corresponding entry in the filesystem. Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of read and write, the kernel calls functions related to packet transmission.

Loadable Modules

One of the good features of Linux is the ability to extend or reduce at runtime the set of features offered by the kernel. This means that you can dynamically add/remove functions running in the kernel.

Each piece of code that can be added to the kernel at runtime is called a module. The Linux kernel offers support for quite a few different types (or classes) of modules, including, but not limited to, device drivers. Each module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel by the "insmod" program and can be unlinked by the "rmmod" program. The modules are identify with the extension .ko (Kernel Objects) and when the system starts this modules are no loaded.

No loadable Objects

There are others functions that are present in the kernel all the time and these have the extension .o (Objects Files) this files are complete executable. Both files .o and .ko are cross-compiled to the specific architecture and there is not a difference in the program way and functions that provided.

Kernel Space and User Space

System memory in Linux can be divided into two distinct regions: kernel space and user space. Kernel space is where the kernel (i.e., the core of the operating system) executes (i.e., runs) and provides its services. When you write device drivers, it's important to make the distinction between "user space" and "kernel space".

Kernel space

Linux (which is a kernel) manages the machine's hardware in a simple and efficient manner, offering the user a simple and uniform programming interface. In the same way, the kernel, and in particular its device drivers form a bridge or interface between the end-user/programmer and the hardware. Any subroutines or functions forming part of the kernel (modules and device drivers, for example) are considered to be part of kernel space. Kernel space can be accessed by user processes only through the use of system calls.

System calls are requests in a Unix-like operating system by an active process for a service performed by the kernel, such as input/output (I/O) or process creation. An active process is a process that is

currently progressing in the CPU, as contrasted with a process that is waiting for its next turn in the CPU. I/O is any program, operation or device that transfers data to or from a CPU and to or from a peripheral device (such as disk drives, keyboards, mice and printers).

A module runs in kernel space, whereas applications run in user space. This concept is at the base of operating systems theory. The role of the operating system, in practice, is to provide programs with a consistent view of the computer's hardware. In addition, the operating system must account for independent operation of programs and protection against unauthorized access to resources. This nontrivial task is possible only if the CPU enforces protection of system software from the applications.

User space

User space is that set of memory locations in which user processes (i.e., everything other than the kernel) run. A process is an executing instance of a program. One of the roles of the kernel is to manage individual user processes within this space and to prevent them from interfering with each other. End-user programs, like the UNIX shell or other GUI based applications are part of the user space. Obviously, these applications need to interact with the system's hardware. However, they don't do so directly, but through the kernel supported functions.

Kernel space and user space is the separation of the privileged operating system functions and the restricted user applications. The separation is necessary to prevent user applications from ransacking your computer. It would be a bad thing if any old user program could start writing random data to your hard drive or read memory from another user program's memory space.

User space programs cannot access system resources directly so access is handled on the program's behalf by the operating system kernel. The user space programs typically make such requests of the operating system through system calls.

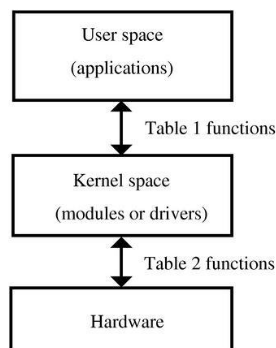

Figure 2: User space where applications reside, and kernel space where modules or device drivers reside

Linux device number (Major and minor number)

Char devices are accessed through names in the filesystem. Those names are called special files or device files or simply nodes of the filesystem tree; they are conventionally located in the /dev directory. Special files for char drivers are identified by a “c” in the first column of the output of ls –l. Block devices appear in /dev as well, but they are identified by a “b.”

If you issue the ls –l command, you'll see two numbers (separated by a comma) in the device file entries before the date of the last modification, where the file length normally appears. These numbers are the major and minor device number for the particular device. The following listing shows a few devices as they appear on a typical system.
Their major numbers are 1, 4, 7, and 10, while the minors are 1, 3, 5, 64, 65, and 129.

```
crw-rw-rw-    1 root root     1,   3  Apr 11 2002 null
crw-------    1 root root    10,   1  Apr 11 2002 psaux
crw-------    1 root root     4,   1  Oct 28 03:04 tty1
crw-rw-rw-    1 root tty      4,  64  Apr 11 2002 ttys0
crw-rw----    1 root uucp     4,  65  Apr 11 2002 ttyS1
crw--w----    1 vcsa tty      7,   1  Apr 11 2002 vcs1
crw--w----    1 vcsa tty      7, 129 Apr 11 2002 vcsa1
crw-rw-rw-    1 root root     1,   5  Apr 11 2002 zero
```

Traditionally, the major number identifies the driver associated with the device. For example, /dev/null and /dev/zero are both managed by driver 1, whereas virtual consoles and serial terminals are managed by driver 4.

The minor number is used by the kernel to determine exactly which device is being referred to. Depending on how your driver is written (as we will see below), you can either get a direct pointer to your device from the kernel, or you can use the minor number yourself as an index into a local array of devices. Either way, the kernel itself knows almost nothing about minor numbers beyond the fact that they refer to devices implemented by your driver.

Within the kernel, the dev_t type (defined in <linux/types.h>) is used to hold device numbers—both the major and minor parts. As of Version 2.6.0 of the kernel, dev_t is a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor number. Your code should, of course, never make any assumptions about the internal organization of device numbers; it should, instead, make use of a set of macros found in <linux/kdev_t.h>.

To obtain the major or minor parts of a dev_t, use:
MAJOR(dev_t dev);
MINOR(dev_t dev);

If, instead, you have the major and minor numbers and need to turn them into a dev_t, use:

MKDEV(int major, int minor);

Some Important Data Structures

Most of the fundamental driver operations involve three important kernel data structures, called file_operations, file, and inode. A basic familiarity with these structures is required to be able to do much of anything interesting, so we will now take a quick look at each of them before getting into the details of how to implement the fundamental driver operations.

File Operations

So far, we have reserved some device numbers for our use, but we have not yet connected
any of our driver's operations to those numbers. The file_operations structure is how a char driver sets up this connection. The structure, defined in *<linux/fs.h>*, is a collection of function pointers. Each open file is associated with its own set of functions. The operations are mostly in charge of implementing the system calls and are therefore, named *open*, *read*, and so on. We can consider the file to be an "object" and the functions operating on it to be its "methods," using object-oriented programming terminology to denote actions declared by an object to act on it.
Conventionally, a file_operations structure or a pointer to one is called fops. Each field in the structure must point to the function in the driver that implements a specific operation, or be left NULL for unsupported operations.

The following list introduces all the operations that an application can invoke on a device. We've tried to keep the list brief so it can be used as a reference, merely summarizing each operation and the default kernel behavior when a NULL pointer is used.

*ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)*
Use to retrieve data from a device. A null pointer in this position causes the *read* system call to fail with -EINVAL ("Invalid argument"). A nonnegative return value represents the number of bytes successfully read (the return value is a "signed size" type, usually the native integer type for the target platform).

*ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *)*
Sends data to the device. If NULL, -EINVAL is returned to the program calling the *write* system call. The return value, if nonnegative, represents the number of bytes successfully written.

*int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long)*
The *ioctl* system call offers a way to issue device-specific commands (such as formatting a track of a floppy disk, which is neither reading nor writing). Additionally, a few *ioctl* commands are recognized by the kernel without referring to the fops table. If the device doesn't provide an *ioctl* method, the system call returns an error for any request that isn't predefined (-ENOTTY, "No such ioctl fordevice").

*int (*open) (struct inode *, struct file *)*
Though this is always the first operation performed on the device file, the driver is not required to declare a corresponding method. If this entry is NULL, opening the device always succeeds, but your driver isn't notified.

*int (*release) (struct inode *, struct file *)*
This operation is invoked when the file structure is being released. Like *open*, *release* can be NULL.*

```
struct file_operations example_fops = {
.owner = THIS_MODULE,
.read = example_read,
.write = example_write,
.ioctl = example_ioctl,
.open = example_open,
.release = example _release,
};
```

The file Structure

struct file, defined in *<linux/fs.h>*, is the second most important data structure used in device drivers. Note that a file has nothing to do with the FILE pointers of user-space programs. A FILE is defined in the C library and never appears in kernel code. A struct file, on the other hand, is a kernel structure that never appears in user programs. The file structure represents an *open file*. (It is not specific to device drivers; every open file in the system has an associated struct file in kernel space.) It is created by the kernel on *open* and is passed to any function that operates on the file, until the last *close*. After all instances of the file are closed, the kernel releases the data structure.

In the kernel sources, a pointer to struct file is usually called either file or filp ("file pointer"). To avoid confusions file refers to the structure and filp to a pointer to the structure. The most important fields of struct file are shown here. As in the previous section, the list can be skipped on a first reading

unsigned int f_flags

These are the file flags, such as O_RDONLY, O_NONBLOCK, and O_SYNC. A driver should check the O_NONBLOCK flag to see if nonblocking operation has been requested. In particular, read/write permission should be checked using f_mode rather than f_flags. All the flags are defined in the header *<linux/fcntl.h>*.

*struct file_operations *f_op*
The operations associated with the file. The kernel assigns the pointer as part of its implementation of *open* and then reads it when it needs to dispatch any operations.

The value in filp->f_op is never saved by the kernel for later reference; this means that you can change the file operations associated with your file, and the new methods will be effective after you return to the caller. For example, the code for *open* associated with major number 1 (*/dev/null*, */dev/zero*, and so on) substitutes the operations in filp->f_op depending on the minor number being opened. This practice allows the implementation of several behaviors under the same major number without introducing overhead at each system call. The ability to replace the file operations is the kernel equivalent of “method overriding” in object-oriented programming.

The inode Structure

The *inode* structure is used by the kernel internally to represent files. Therefore, it is different from the file structure that represents an open file descriptor. There can be numerous file structures representing multiple open descriptors on a single file, but they all point to a single inode structure. The inode structure contains a great deal of information about the file. As a general rule, only two fields of this structure are of interest for writing driver code:

dev_t i_rdev
For inodes that represent device files, this field contains the actual device number.

*struct cdev *i_cdev*
struct cdev is the kernel’s internal structure that represents char devices; this field contains a pointer to that structure when the inode refers to a char device file.

The type of i_rdev changed over the course of the 2.5 development series, breaking a lot of drivers. As a way of encouraging more portable programming, the kernel developers have added two macros that can be used to obtain the major and minor number from an inode:

*unsigned int iminor(struct inode *inode);*
*unsigned int imajor(struct inode *inode);*

In the interest of not being caught by the next change, these macros should be used instead of manipulating i_rdev directly.

Device Manager

The kernel device manager used is udev. It runs as a daemon on a Linux system and listens to uevents the kernel sends out if a new device is initialized or a device is removed from the system. The system provides a set of rules that match against exported values of the event and properties of the discovered device. A matching rule will possibly name and create a device node and run configured programs to set-up and configure the device.

udev rules can match on properties like the kernel subsystem, the kernel device name, the physical location of the device, or properties like the device's serial number. Rules can also request information from external programs to name a device or specify a custom name that will always be the same, regardless of the order devices are discovered by the system. Primarily, it manages device nodes in /dev.

3 Debugging a driver

As a debugging tool in kernel drivers you only available to use “printk()” functions to know what is happens into the driver. The kernel print function, printk(), behaves almost identically to the C library printf() function, function is simply the name of the kernel's formatted print function.It is callable from just about anywhere in the kernel at any time. It can be called from interrupt or process context. It can be called while a lock is held.

Loglevels
The major difference between printk() and printf() is the capability of the former to specify a loglevel. The kernel uses the loglevel to decide whether to print the message to the console. The kernel displays all messages with a loglevel below a specified value on the console.

You specify a loglevel like this:

```
printk(KERN_WARNING "This is a warning!\n");
printk(KERN_DEBUG "This is a debug notice!\n");
printk("I did not specify a loglevel!\n");
```

The complete list of loglevels available is

Loglevel	Description
KERN_EMERG	An emergency condition; the system is probably dead
KERN_ALERT	A problem that requires immediate attention
KERN_CRIT	A critical condition
KERN_ERR	An error
KERN_WARNING	A warning
KERN_NOTICE	A normal, but perhaps noteworthy, condition
KERN_INFO	An informational message
KERN_DEBUG	A debug messagetypically superfluous

The library to be included in the driver to use printk is <linux/kernel.h>

4 Compile the driver

You need to have your driver under drivers/XXX where XXX is a existed folder or a new one.
To compile the driver a Makefile is need, you can use the same Makefile in the folder where is your driver and add at the end
“obj-m := i2c_subdriver.o” for module and for compiling you need to do a “make modules”
“obj-y := i2c_subdriver.o” as object file to be loaded since OS starts, use “make ”

You need to do the make command in your host where the BSP was installed or using the LTIB and chose the option Configure Kernel.
If, instead, you have a module called module.ko that is generated from two source files (called, say, file1.c and file2.c), the correct incantation would be:
obj-m := module.o
module-objs := file1.o file2.o

5 Loading a module

If a driver was compiled as module (.ko) as not as object file (.o) it is necessary to mount the module in the system and create the node to establish the communication and allow the use of them by other drivers in kernel space or applications in user space.
The steps to load a module are the next

1. You need to have the name of the module “name”, without .ko, in your rootfs and enter in the console the next command

insmod name

2.-You need to have the major and minor number for the node that you will create. Then
sudo mknod /dev/${device} c ${major} ${minor}
where
${device} Name of the node to be create
${major} Major number used to identify the driver controller
${minor} Minor number of the device controlled by the driver.

4.-You need to give the group/permissions to the driver to
chmod 666 /dev/${device}

3.-To remove the module you must enter
rmmod name

The “modprobe” command is worth a quick mention. modprobe, like insmod, loads a module into the kernel. It differs in that it will look at the module to be loaded to see whether it references any symbols that are not currently defined in the kernel. If any such references are found, modprobe looks for other modules in the current module search path that define the relevant symbols. When modprobe finds those modules (which are needed by the module being loaded), it loads them into the kernel as well. If you use insmod in
this situation instead, the command fails with an “unresolved symbols” message left in the system logfile.

The “lsmod” program produces a list of the modules currently loaded in the kernel. Some other information, such as any other modules making use of a specific module, is also provided. lsmod works by reading the /proc/modules virtual file. Information on currently loaded modules can also be found in the sysfs virtual filesystem under /sys/module.

When the driver is compiled as object file, the last steps are not needed because the kernel mounts the driver automatically

6 How to use the driver in an application (user space)

Once the driver is loaded into the kernel and the node is available under /dev an application from user space can use it and access to the functionality using the next functions.

Open

```
      File_descriptor = open("/dev/NAME_NODE",PERMISSIONS)
            This function opens the node, and returns a file descriptor (int) to
identify the opened node.
      The permissions could be read, write or only one of them.
```

Close

```
      close(File_descriptor)
            This function opens the node, and returns a file descriptor (int) to
identify the opened node.
```

Write

```
      write(File_descriptor, output, length)
File descriptor with the node
      output (pointer) to the buffer when the data will be taken.
      Length data to write.
```

Read

```
      read(File_descriptor, buffer, length)
      File descriptor with the node
      Buffer (pointer) to the buffer when the data will be saved.
      Length of the read data.
```

Ioctl

When you need to do a different operation to read or write you can use the ioctl with the fuctions provide by the driver.

```
      ioctl(File_descriptor, function, ptr)
      Function is a value given by the API of the user to implement different
functions into ioctl.
      Ptr is the pointer passed to the ioctl.
```

From user space you cannot use different functions that the mentioned before.

7 Driver program

```
/*
*A simple char driver that use I2C subsystem in Linux.
*
 */

#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/slab.h> /* kmalloc() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
```

Driver program

```
#include <asm/uaccess.h> /* copy_from/to_user */
#include <linux/input.h>/*input_allocate_device*/
#include <linux/delay.h>
#include <linux/i2c.h>
#include <linux/interrupt.h>
#include <linux/workqueue.h> /*sleep workqueues*/
#include <linux/ctype.h>
#include <linux/ioctl.h>

///DEV Name
#define MAJOR_NUMBER 60  //Chosen Major number

//For IOCTL
#define IOCTL_SET _IOR(60,0,int)  //Define the prototipe of Function 0
#define IOCTL_GET _IOWR(60,1,char *) //Define the prototipe of Function 1
#define SET 0
#define GET 1


/*Private Struct of the driver*/
struct example_data {
		int d1;
		int d2;
		struct workqueue_struct *workqueue;
		struct work_struct work;
		struct input_dev *input_dev;
		struct i2c_client *client;
};

struct example_data *example;
struct i2c_client *clientPublic;

/*FOPS--for the subsystem*/

		int example_open(struct inode *inode, struct file *filp)
		{

				printk(KERN_DEBUG "\nOPEN DEVICE********************|\n\n");

		return 0;
}

		ssize_t example_read(struct file *filep, char *buf, size_t count, loff_t *fpos)
		{

				u8 dataRead;

				printk(KERN_DEBUG "\nREAD DEVICE********************\n");
				dataRead = i2c_smbus_read_byte_data(clientPublic,*buf);
				printk(KERN_DEBUG "Data Readed: %d", dataRead);
				if(copy_to_user(buf,&dataRead,1))
						printk(KERN_DEBUG "Error reading\n");
				return 1;
}

ssize_t example_write(struct file *filep, const char *buf, size_t size, loff_t *fpos)
{

		const char *copyBuf = NULL;
		printk(KERN_DEBUG "\nWRITE DEVICE********************\n\n");

		//Copy buffer
		copyBuf = buf;

	if(copyBuf == NULL)
		return -ENOMEM;

		 printk(KERN_DEBUG "Data: %x\n",*(copyBuf+1));
```

```
                        if((*(copyBuf+1) >= 0x00)&&(*(copyBuf+1) <= 0xFF))
                        {
                                    i2c_smbus_write_byte_data(clientPublic, *copyBuf, *(copyBuf+1));
                                    printk(KERN_DEBUG "Data Written: %d\n",i2c_smbus_read_byte_data(clientPublic,*copyBuf));
                        }
                        else
                                    printk(KERN_DEBUG "Invalid Range of data\n");
return 1; //Data written
}

            int example_ioctl(struct inode *inode, struct file *filp, unsigned int ioctl_num, unsigned long  ioctl_param)
            {

                        char *mem;
                        mem= kmalloc(sizeof(char *)*5, GFP_KERNEL); //Reserve  space to save data between data in user space.
                        mem="bye";

                        printk(KERN_DEBUG "\nIOCTL********************\n\n");

                        switch(ioctl_num)
                        {

                                    case  SET :
                                                            printk(KERN_ALERT "IOCTL_SET\n");
                                                            printk(KERN_ALERT "Num: %i\n", ioctl_param);
                                                            copy_from_user(mem, (void *)ioctl_param, 5);
                                                            break;


                                    case GET:
                                                            printk(KERN_ALERT "IOCTL_GET\n");
                                                            printk(KERN_ALERT "Num: %i\n", ioctl_num);
                                                            printk(KERN_ALERT "kERN_DATA: %s\n", mem);
                                                            copy_to_user((void *)ioctl_param,mem,5);  //Save data in the pointer received
                                                            printk(KERN_ALERT "Parameter: %s\n", (char *)ioctl_param);
                                                            break;

                                    default:                          printk(KERN_DEBUG "INVALID FUCTION \n");
                                                                      return EINVAL;

                        }

                        kfree(mem);
                        return 0;
            }

int example_release(struct inode *inode, struct file *filp)
{

                        printk(KERN_DEBUG "RELEASE\n");
                        printk(KERN_DEBUG "\nRELEASE DEVICE*****************|\n\n");

                        return 0;
}
//this are said the fuctions to be done by the driver from user space
struct file_operations file_ops_example = {
            open:                   example_open,
            release:    example_release,
            ioctl:                  example_ioctl,
            write:                  example_write,
            read:                   example_read,
};

//This is the fuction to be done in each thread.
            static void do_task(struct work_struct *work)
            {

                        unsigned int x;
```

```
        //Data Read
        u8 data = 0;
        data= i2c_smbus_read_byte_data(clientPublic,STATUS_REGISTER_1);
        printk(KERN_DEBUG "\nStatus_1: 0x%x", statusRegister_1);
        msleep(10);

}

static int examples_resume(struct i2c_client *client)
{
        printk(KERN_DEBUG "***RESUME***\n");
        return 0;
}

static int examples_suspend(struct i2c_client *client, pm_message_t mesg)
{
        printk(KERN_DEBUG "***SUSPEND***\n");
        return 0;
}

static int __devinit examples_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
        int result;
        struct input_dev *input_dev;

        clientPublic = client;

        //Allocate Private Structure
        example = kzalloc(sizeof(struct examples_data), GFP_KERNEL);
        if (!example)
                return -ENOMEM;

        //Allocate Input Device
        input_dev = input_allocate_device();
        if (!input_dev) {
                result = -ENOMEM;
                goto err_free_mem;
        }

        //save input in the Private structure
        example->input_dev = input_dev;

        //Create thread
        example->workqueue = create_singlethread_workqueue("examples");
        INIT_WORK(&example->work, do_tak);

        if (example->workqueue == NULL) {
                printk(KERN_DEBUG "couldn't create workqueue\n");
                result = -ENOMEM;
                goto err_wqueue;
        }

        //Send a command to probe the device was founded and initialized
        i2c_smbus_write_byte_data(clientPublic, HORIZONTAL_RESOLUTION_MBS , 0x11);


        //Verify that the I2C device received the sent data
        dataConfiguration = i2c_smbus_read_byte_data(clientPublic,CONFIGURATION);

        example->input_dev->name = "EXAMPLE Input Device";

        //Register Input Device
        result = input_register_device(example->input_dev);
        if (result)
                goto err_free_wq;

        //Register with DEV
```

```
                result = register_chrdev(MAJOR_NUMBER,DEV_NAME,&file_ops_examples);
                if(result)
                        goto err_unr_dev;

        //Launch thread
        queue_work(example->workqueue,&example->work);

        return 0;

err_unr_chrdev:
        unregister_chrdev(MAJOR_NUMBER,DEV_NAME);
err_unr_dev:
        input_unregister_device(example->input_dev);
err_free_wq:
        destroy_workqueue(example->workqueue);
err_wqueue:
        input_free_device(example->input_dev);
err_free_mem:
        kfree(example);
        return result;
}

        static int __devexit examples_remove(struct i2c_client *client)
        {

          cancel_work_sync(&example->work);
          destroy_workqueue(example->workqueue);
          input_unregister_device(example->input_dev);
          input_free_device(example->input_dev);
          unregister_chrdev(MAJOR_NUMBER,DEV_NAME);
          kfree(example);
          return 0;
        }

static const struct i2c_device_id examples_idtable[] = {
        {"examples_Id", 0},
        {}
};
///In this structure there are definied the functions done by kernel system
static struct i2c_driver examples_fops = {
  .driver = {
                .owner = THIS_MODULE,
                .name = "examples_Id",
          },
  .id_table = examples_idtable,
  .probe = examples_probe,
  .resume = examples_resume,
  .suspend = examples_suspend,
  .remove = __devexit_p(examples_remove),

};

MODULE_DEVICE_TABLE(i2c, examples_idtable);

static int __init examples_init(void)
{
        return i2c_add_driver(&examples_fops);  //Register the device with the i2c subsytem, and the corresponding file operations in the
subdriver
}

static void __exit examples_exit(void)
{
        i2c_del_driver(&examples_fops);                 //Remove the device from the i2c subsystem
}

module_init(examples_init);
module_exit(examples_exit);
```

```
MODULE_LICENSE("GPL");


MODULE_AUTHOR("Freescale Semiconductor, Inc.");
MODULE_DESCRIPTION("Example of a dummy driver");
```

8 Application to use driver

```
#include <stdio.h>
#include <sys/ioctl.h>
#include <string.h>
#define IOCTL_SET _IOR(60,0,int)
#define IOCTL_GET _IOR(60,1,char *)

int main()
{

int fd;
int io = 0;

char *data;
data=malloc(5*sizeof(char));
data="hello";

fd = fd = open("/dev/examples_ID",0

if(fd< 0 )
        printf("\nError opening file\n");
else
        printf("\nFile opened\n");


io = ioctl(fd,0,4321);
        printf("IO: %d\n",io);

//io = ioctl(fd,1,&hola2[0]);
io = ioctl(fd,1,hola);
if(io < 0)
        printf("Error on get message");
else
        printf("IO: %s\n",hola);

hola=NULL;
free(hola);
close(fd);

return 0;
}
```

9 Changes for i.MX51

In this case the driver was used in i.MX51EVK and there is some changes that you need to do before using this driver.

Mx51_babbage.c

In the next code add some lines, if this case the driver was added using the bus 1, if you want to use bus 0 modigy mxc_i2c0_board_info[]

```
static struct i2c_board_info mxc_i2c1_board_info[] __initdata = {
        {
         .type = "sgtl5000-i2c",
         .addr = 0x0a,
         },
        {
         .type = "isl29003",
         .addr = 0x44,
         .platform_data = &ls_data,
         },
        {
         .type = "mxc_ddc",
         .addr = 0x50,
         .irq = gpio_to_irq(BABBAGE_DVI_DET),
         .platform_data = &mxc_ddc_dvi_data,
         },
        {
          .type = "Name_driver"
          .addr = "0x49"
           // Pay attention this address corresponds to address device 0x96, there is a shift.
        },
};
```

In this case the configuration by default enables and registers all the devices and their address with the next lines

i2c_register_board_info(1, mxc_i2c1_board_info, ARRAY_SIZE(mxc_i2c1_board_info));

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSeminconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2007. All rights reserved.

ANxxxx
Rev.
11/20