Popular

- Home
- Gumstix and Yocto
- Gumstix kernel dev
- Gumstix u-boot dev
- Gumstix I2C
- Gumstix GPIO
- Gumstix PWM
- Gumstix network boot
- Gumstix USB networking
- Writing a Linux SPI driver
- Gumstix developer tips
- Gumstix ADC
- Gumstix Buttons and LEDs

Writing a Linux SPI driver - Part 1

Last Updated on Thursday, 14 April 2011 07:38

A Linux SPI system overview

Introduction

What follows is a guide for developing a SPI driver for the Gumstix Overos. The steps should translate to any OMAP3 system running Linux.

Contents

The project code can be found at http://github.com/scottellis/spike

You can use the Github 'Switch Branches' menu in the upper left if you just want to browse the the code for a particular section.

Overview

The kernel documentation for the SPI framework is a good place to start. You can find it in your kernel source documentation directory. Here is a link spi-summary.

Linux SPI drivers have two parts. A controller driver that handles direct communication with the hardware and a protocol driver to handle the details for a particular device.

This article will be describing a SPI protocol driver.

The mainline kernel has a generic SPI protocol driver with a character device interface called spidev. Footnote 2 has some instruction on getting it to work with Gumstix Overo boards. This article is not about spidev, but about writing a customized SPI device driver.

There are a couple of reasons to write a customized driver.

1. You will likely to get better performance with a customized driver.
2. You won't have to patch the kernel board file. The SPI framework fully supports doing all device driver setup in a loadable kernel module. This allows you to keep all your work outside the kernel source tree.

Kernel Configuration

For the OMAP3's, the SPI controller driver is omap2_mcspi.c.

The kernel config option to include the omap2_mcspi driver is CONFIG_SPI_OMAP24XX. You can configure it as built-in or as a loadable module. If you are using the GUI menuconfig tool, you can find the module here

```
Device Drivers
    SPI support
        McSPI driver for OMAP
```

The default Gumstix Overo kernels already have the omap2_mcspi driver enabled.

Gumstix brings out SPI1 to the expansion board with two cable select lines, CS0 and CS1.

CS0 is assigned to the ADS7846 touchscreen driver and CS1 to the LGPHILIPS LB035Q02 display in the Overo Linux board file, board-overo.c. If you want to use SPI bus 1 off the Overo expansion board, then you need to completely disable one or both of these kernel modules in your kernel config and rebuild your kernel.

If you are using gumstix kernels >= 2.6.36, then SPIDEV is conditionally assigned to both CS0 or CS1 if either the ADS7846 or LGPHILIPS drivers are not enabled. This is convenient if you want to use SPIDEV, but if you want to load your own driver you will also have to disable SPIDEV in your kernel config.

Instructions for modifying your kernel configuration and rebuilding can be found here - Gumstix kernel development

If you are working in the menuconfig application, you can find the ADS7846 config option here

```
Device Drivers
    Input Device Support
        Touchscreens
```

```
            ADS7846/...
```

the LGPHILIPS LB035Q02 config option here

```
Device Drivers
    Graphics Support
        OMAP2/3 Display Subsystem support (EXPERIMENTAL)
            OMAP2/3 Display Device Drivers
                LG. Philips LB035Q02 LCD Panel
```

and the SPIDEV config option here

```
Device Drivers
    SPI support
        User mode SPI device driver support
```

Here are some shortcuts, to directly modify the defconfig file for the ADS7846 touchscreen and the LGPHILIPS display

```
sed -i 's:CONFIG_TOUCHSCREEN_ADS7846=m:# CONFIG_TOUCHSCREEN_ADS7846 is not set:g' defconfig
sed -i 's:CONFIG_PANEL_LGPHILIPS_LB035Q02=y:# CONFIG_PANEL_LGPHILIPS_LB035Q02 is not set:g' defconfig
```

And this will disable SPIDEV

```
sed -i 's:CONFIG_SPI_SPIDEV=y:# CONFIG_SPI_SPIDEV is not set:g' defconfig
```

Pin Multiplexing

The OMAP3s have 4 SPI buses. Different boards may bring out different SPI bus pins.

Many of the pins of an OMAP3 system can be multiplexed for different functions. So even if a particular SPI bus is brought out, you need to verify that the pins are multiplexed for SPI functionality and not for another function such as GPIO.

The Gumstix Overo SPI1 pins are correctly mux'd for SPI use by default.

The SPI1 pins on the expansion board are

Pin 3 - SPI1_CLK
Pin 5 - SPI1_MOSI
Pin 7 - SPI1_MISO
Pin 6 - SPI1_CS0
Pin 8 - SPI1_CS1

Pin multiplexing is still done primarily in the bootloader, u-boot. There is a continuing effort by the Linux OMAP developers to move the pin mux'ing into the kernel. You can do your muxing in either place, but since the kernel follows the bootloader, values set in the kernel will override.

Here are some instructions for modifying the pin muxing in u-boot - Gumstix u-boot development

The values currently set in the Overo u-boot overo.h file to get SPI bus 1 functionality on the expansion boards are as follows

```
MUX_VAL(CP(MCSPI1_CLK),       (IEN  | PTD | DIS | M0))
MUX_VAL(CP(MCSPI1_SIMO),      (IEN  | PTD | DIS | M0))
MUX_VAL(CP(MCSPI1_SOMI),      (IEN  | PTD | DIS | M0))
MUX_VAL(CP(MCSPI1_CS0),       (IEN  | PTD | EN  | M0))
MUX_VAL(CP(MCSPI1_CS1),       (IDIS | PTD | EN  | M0))
```

The macros MUX_VAL() and CP() and the constants used in overo.h are defined in the u-boot source file arch/arm/include/asm/arch-omap3/mux.h.

You'll see slightly different configurations in Beagleboard examples, but the important thing is to make sure that both the CLK and SOMI pins are mux'd as input enabled (IEN). The remaining SPI pins can be mux'd as outputs, pulled up or down or neither (DIS) doesn't matter. The Overo default that muxes the SIMO line as input enabled (IEN) also doesn't make a difference. IEN still allows output.

If you wanted to do the pin mux'ing in the kernel, you would do it in board-overo.c like this

```
omap_mux_init_signal("mcspi1_clk", OMAP_PIN_INPUT);
omap_mux_init_signal("mcspi1_somi", OMAP_PIN_INPUT);
omap_mux_init_signal("mcspi1_simo", OMAP_PIN_OUTPUT);
omap_mux_init_signal("mcspi1_cs0", OMAP_PIN_OUTPUT);
omap_mux_init_signal("mcspi1_cs1", OMAP_PIN_OUTPUT);
```

The beagle board file is board-omap3beagle.c

See the footnote on spidev at the bottom of this page for a fuller example.

Driver Setup

The are two steps you need to get your driver recognized by the Linux SPI system.

1. You need to register a SPI slave device on a particular SPI bus and cable-select position. You can do this during kernel init or you can do it dynamically in your driver code. The device name (modalias) is specified as a part of this process.

2. You need to register a SPI protocol driver. This driver should use the same name (modalias) as the devices it will be handling so the kernel can link the two.

The two steps can be done in any order, but when they are both completed, the SPI framework will respond with a call to the probe() callback in your driver. You will be given a handle to a struct spi_device in the callback that you'll use to interact with the SPI system. Once your probe function has been called you can start using the SPI bus.

Static registering of SPI devices is covered in the spidev notes section at the bottom of this page.

To dynamically register a SPI device, you can follow these steps

1. Get a handle to the spi_master controller driver managing the bus
2. Allocate a spi_device structure for that bus
3. Verify that no other device has been registered for that particular SPI bus.cs position
4. Populate the spi_device fields with device specific values (speed, data size, etc...)
5. Add the new spi_device to the bus

Here is what the code might look like (https://gist.github.com/716613)

```
static int __init add_spike_device_to_bus(void)
{
        struct spi_master *spi_master;
        struct spi_device *spi_device;
        struct device *pdev;
        char buff[64];
        int status = 0;

        spi_master = spi_busnum_to_master(SPI_BUS);
        if (!spi_master) {
                printk(KERN_ALERT "spi_busnum_to_master(%d) returned NULL\n",
                        SPI_BUS);
                printk(KERN_ALERT "Missing modprobe omap2_mcspi?\n");
                return -1;
        }

        spi_device = spi_alloc_device(spi_master);
        if (!spi_device) {
                put_device(&spi_master->dev);
                printk(KERN_ALERT "spi_alloc_device() failed\n");
                return -1;
        }

        /* specify a chip select line */
        spi_device->chip_select = SPI_BUS_CS1;

        /* Check whether this SPI bus.cs is already claimed */
        snprintf(buff, sizeof(buff), "%s.%u",
                        dev_name(&spi_device->master->dev),
                        spi_device->chip_select);

        pdev = bus_find_device_by_name(spi_device->dev.bus, NULL, buff);
        if (pdev) {
                /* We are not going to use this spi_device, so free it */
                spi_dev_put(spi_device);

                /*
                 * There is already a device configured for this bus.cs combination.
                 * It's okay if it's us. This happens if we previously loaded then
                 * unloaded our driver.
                 * If it is not us, we complain and fail.
                 */
                if (pdev->driver && pdev->driver->name &&
                                strcmp(this_driver_name, pdev->driver->name)) {
                        printk(KERN_ALERT
                                "Driver [%s] already registered for %s\n",
                                pdev->driver->name, buff);
                        status = -1;
                }
        } else {
                spi_device->max_speed_hz = SPI_BUS_SPEED;
                spi_device->mode = SPI_MODE_0;
                spi_device->bits_per_word = 8;
                spi_device->irq = -1;
                spi_device->controller_state = NULL;
                spi_device->controller_data = NULL;
                strlcpy(spi_device->modalias, this_driver_name, SPI_NAME_SIZE);
                status = spi_add_device(spi_device);

                if (status < 0) {
                        spi_dev_put(spi_device);
                        printk(KERN_ALERT "spi_add_device() failed: %d\n",
                                status);
                }
        }

        put_device(&spi_master->dev);

        return status;
}
```

gistfile1.txt hosted with ❤ by **GitHub** **view raw**

It's a little messy, but it's all boilerplate stuff. You can copy and paste the code and only modify the spi_device fields specific to

devices your driver is handling. These are the same fields you would have to set in a board data structure if you did it that way.

To register a spi driver you need to initialize a spi_driver structure with your driver name and some callback functions. At a minimum, you need to provide a probe() callback.

Something like this

```
static struct spi_driver spike_driver = {
        .driver = {
                .name = this_driver_name,
                .owner = THIS_MODULE,
        },
        .probe = spike_probe,
        .remove = spike_remove,
};
```

And then somewhere in your driver code call spi_register_driver().

```
spi_register_driver(&spike_driver);
```

Code for Part 1

The code for the project is here http://github.com/scottellis/spike

You need a cross-compiler and the kernel source code available to build the module. There are some notes for using the OpenEmbedded tools here - Gumstix Kernel Development

Once you have your tools and the linux source, you can fetch and compile the driver like this

```
$ git clone git://github.com/scottellis/spike.git
$ cd spike
$ source overo-source-me.txt
$ make
```

This will build a driver spike.ko

Copy this driver to your system and load it with insmod.

```
root@overo:~# insmod spike.ko
```

You might get this error message if omap2_mcspi is not loaded.

```
root@overo:~# insmod spike.ko
[   74.586791] spi_busnum_to_master(1) returned NULL
[   74.591613] Missing modprobe omap2_mcspi?
[   74.595642] add_spike_to_bus() failed
insmod: cannot insert 'spike.ko': Operation not permitted
```

If so, load omap2_mcspi first

```
root@overo:~# modprobe omap2_mcspi
root@overo:~# insmod spike.ko
root@overo:~# lsmod
Module                  Size  Used by    Not tainted
spike                   2397  0
omap2_mcspi             5888  0
```

The driver declares a spi device and registers the driver spike for this device.

If you read from /dev/spike device you'll get the following

```
root@overo:~# cat /dev/spike
spike ready on SPI1.1
```

The next section will start reading and writing some data Part 2 - SPI message basics

Notes

1. For kernel versions < 2.6.34 you will need a patch for a bug in the omap2_mcspi cleanup code that impacts dynamically introduced slave devices. Without it you'll oops under certain module insertion conditions. You can find the patch for the oops in the spike project patches directory.

2. spidev is a generic SPI protocol driver that exposes a userland char-dev interface. You can find the official documentation for spidev here.

I haven't used spidev, but I can provide some instruction for getting enabled. Spidev is not setup to handle dynamic declaration of spi devices, so you have to modify the kernel to enable it, typically in your kernel board file.

In the kernel header **include/linux/spi/spi.h** is a **struct spi_board_info** structure that should be populated for any statically registered spi device. You then add the device to the system using the **spi_register_board_info()** function.

Here is what it looks like (https://gist.github.com/736463)

```
1  struct spi_board_info {
2          /* the device name and module name are coupled, like platform_bus;
3           * "modalias" is normally the driver name.
4           *
5           * platform_data goes to spi_device.dev.platform_data,
6           * controller_data goes to spi_device.controller_data,
7           * irq is copied too
8           */
```

```
 9         char            modalias[SPI_NAME_SIZE];
10         const void      *platform_data;
11         void            *controller_data;
12         int             irq;
13
14         /* slower signaling on noisy or low voltage boards */
15         u32             max_speed_hz;
16
17
18         /* bus_num is board specific and matches the bus_num of some
19          * spi_master that will probably be registered later.
20          *
21          * chip_select reflects how this chip is wired to that master;
22          * it's less than num_chipselect.
23          */
24         u16             bus_num;
25         u16             chip_select;
26
27         /* mode becomes spi_device.mode, and is essential for chips
28          * where the default of SPI_CS_HIGH = 0 is wrong.
29          */
30         u8              mode;
31 };
```

gistfile1.txt hosted with ❤ by **GitHub** **view raw**

You could register a spidev device on SPI bus 1 CS0 this way

```
static struct spi_board_info spidev_board_info {
    .modalias = "spidev",
    .max_speed_hz = 48000000,
    .bus_num = 1,
    .chips_select = 0,
    .mode = SPI_MODE_0,
};
```

And then add it like this

```
spi_register_board_info(&spidev_board_info, 1);
```

Or if you wanted to register both CS 0 and CS 1 for spidev use

```
static struct spi_board_info spidev_board_info[] {
    {
        .modalias = "spidev",
        .max_speed_hz = 2000000,
        .bus_num = 1,
        .chips_select = 0,
        .mode = SPI_MODE_0,
    },
    {
        .modalias = "spidev",
        .max_speed_hz = 2000000,
        .bus_num = 1,
        .chips_select = 1,
        .mode = SPI_MODE_0,
    },
};

spi_register_board_info(spidev_board_info, ARRAY_SIZE(spidev_board_info));
```

There is an OE ready patch to enable spidev in the patches directory of the spike project on github here - spidev-enable.patch.

Refer to the Gumstix kernel development page, 'Patching the board file' section, for details on how to use this patch with OE.

To use this patch, you still have to disable either the ADS7846 or the LGPHILIPS display driver to free up one of the expansion board SPI1 CS lines.

Update - This patch is now incorporated in the Gumstix OE repository for kernel 2.6.36 so all you need to do is disable one or both of the ADS7846 touchscreen or LGPHILIPS display to use spidev. Older kernels still need the spidev-enable patch.

Of course, you could do the same static device declaration with any custom SPI driver. It would look like this for the spike driver in this article

```
static struct spi_board_info spike_board_info {
    .modalias = "spike",
    .max_speed_hz = 1000000,
    .bus_num = 1,
    .chips_select = 1,
    .mode = SPI_MODE_0,
};

spi_register_board_info(&spike_board_info, 1);
```

Then you would have to remove the add_spike_device_to_bus() code in spike.c or at least the call to that function. The end result would be a spike driver that works the same way.