USER GUIDE

LINUX DEVICE DRIVER
FOR
SC16IS750
(SPI-UART BRIDGE)

TABLE OF CONTENTS

1. Overview

The SC16IS750 is a bridge that provides I2C/SPI interface to a single-channel high performance UART. It acts in slave mode only and can offer data rates up to 5Mbps. A driver is written for the Linux platform that can communicate with the device and is specifically coded for LPC3250 board. But it can be modified to support other development boards and is explained in section 6.

This driver supports only the SPI interface to the UART. There is a separate driver for the I2C interface. It uses the inbuilt SPI driver framework for linux 2.6 kernel running on LPC3250. It is highly recommended to go through this manual before using the driver.

Before accessing the driver, Linux OS has to be prepared for LPC3250 using Linux Target Image Builder (LTIB). LTIB is a tool that integrates the build and configuration of the multiple software packages and components required for a typical embedded Linux distribution. LTIB greatly reduces the complexity of gathering, configuring, and building all of these required components. The source code is maintained at www.bitshrine.org and can be downloaded for free.

The driver package consists of following folders inside the ‘Driver’.

i) sc16is750_spi-1.1.tar.gz where driver source code is located
ii) sc16is750_spi which contains the spec file required to install the driver.

The files/folders inside the ‘Application’ are:

i) spitest-1.1.tar.gz contains a sample user-space application program to test the driver.
ii) spitest which contains the spec file required to install the application program.
iii) tx.txt/rx.txt are the text files to send/receive data using the driver.

2. Installation

In order to use this driver there is a need of some modification in the Linux kernel. But before that we need to get the Linux operating system running on LPC3250. Following are the quick steps that will download the Linux source code and bring about the required modification.

2.1 Source code download

The source code can be downloaded from http://www.bitshrine.org/ltib/resources-download. It's better to use CVS to check out the latest and build the platform by running the following line:

$ cvs -z3 -d:pserver:anonymous@cvs.savannah.nongnu.org:/sources/ltib co -P ltib

This will create a folder called 'ltib' on the present working directory. Then move to the ltib directory and run the following command:

$ cd ltib

ltib$ ./ltib

During the linux OS building, choose the platform and then LTIB configuration box will appear. Here, select the following option:

[*] leave the kernel source after building

The detailed OS installation instructions can be found in 'phyCORE-LPC3250 Quick start for Linux' that can downloaded from http://www.phytec.com/products/sbc/ARM-XScale/phyCORE-ARM9-LPC3250.html

2.2 Kernel source code modification

After LTIB is installed for LPC3250, the board initialization and configuration file has to be changed. We need to create and add a new SPI device in the system that is used by the device driver. For this, following is the code that has to be modified in *phy3250.c* located at rpm/BUILD/linux/arch/arm/mach-lpc3250 inside the ltib folder.

```
static int __init phy3250_spi_board_register(void)
{
#if defined(CONFIG_SPI_SPIDEV) || defined(CONFIG_SPI_SPIDEV_MODULE)
        static struct spi_board_info info[] = {
                {
                        .modalias = "spidev",
                        .max_speed_hz = 5000000,
                        .bus_num = 0,
                        .chip_select = 0,
                        .controller_data = &spi0_chip_info,
                },
        };

#elif defined(CONFIG_EEPROM_AT25) || defined(CONFIG_EEPROM_AT25_MODULE)
        static struct spi_eeprom eeprom = {
                .name = "at25256a",
                .byte_len = 0x8000,
                .page_size = 64,
                .flags = EE_ADDR2,
        };

        static struct spi_board_info info[] = {
                {
                        .modalias = "at25",
                        .max_speed_hz = 5000000,
                        .bus_num = 0,
                        .chip_select = 0,
                        .platform_data = &eeprom,
                        .controller_data = &spi0_chip_info,
                },
        };
#else
     static struct spi_board_info info[] = {
          {
                .modalias = "my_spi_driver1",
                .max_speed_hz = 1000000,
                .bus_num = 0,
                .chip_select = 0,
                .controller_data = &spi0_chip_info,
          },
     };
#endif
        return spi_register_board_info(info, ARRAY_SIZE(info));
}
```

arch_initcall(phy3250_spi_board_register);

Note: Locate the phy3250_spi_board_register code and add my_spi_driver1 as shown above.

Once a new SPI device is added to the code, the kernel has to be reconfigured and recompiled.

ltib$ ./ltib --configure

The LTIB configuration dialog will appear. Select the option:
[*] always rebuild the kernel
[*] configure the kernel

Then save & exit.

Now the 'Linux Kernel Configuration' dialogue will appear. Go to **Device Driver->Misc devices->EEPROM Support** and make sure that 'SPI EEPROM from most vendors' is not set. See the figure below.

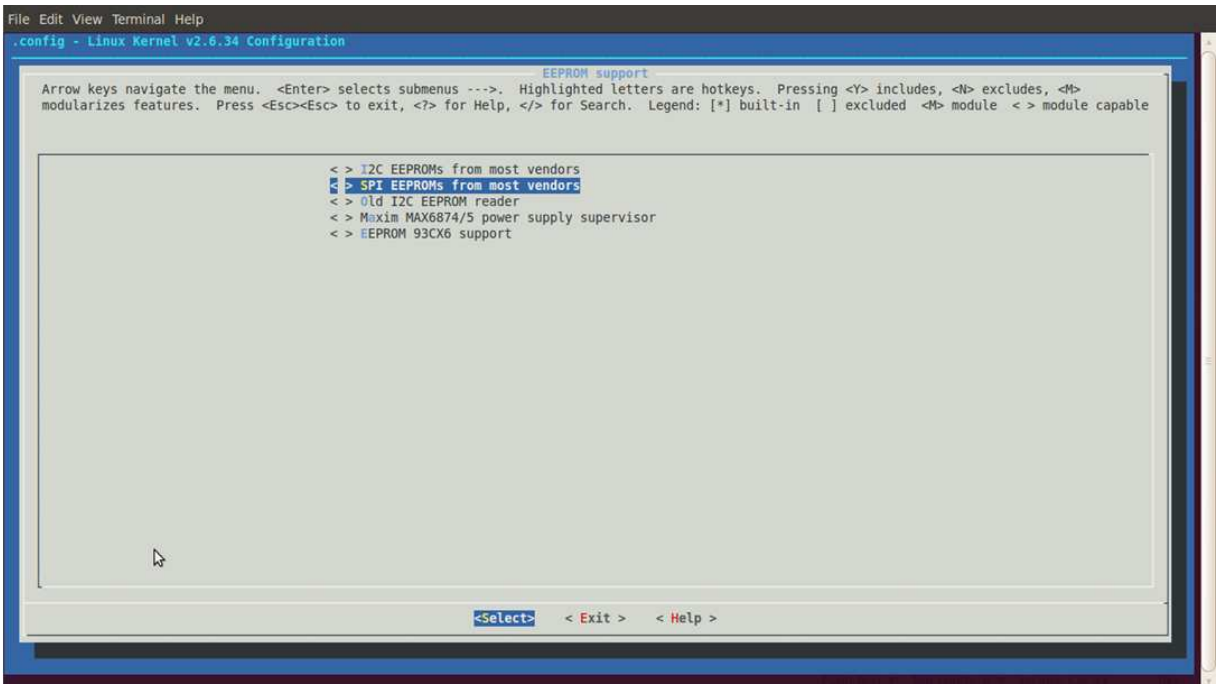

Figure 1. Linux kernel configuration

Next we need to make sure that *CONFIG_SPI_SPIDEV* is also not set. If both of them are unselected then, my_spi_driver1 will be added as an SPI device by the board initialization code. Now go to **Device Driver->SPI support** and make sure that 'User mode SPI device driver support' is not set. See the figure below.

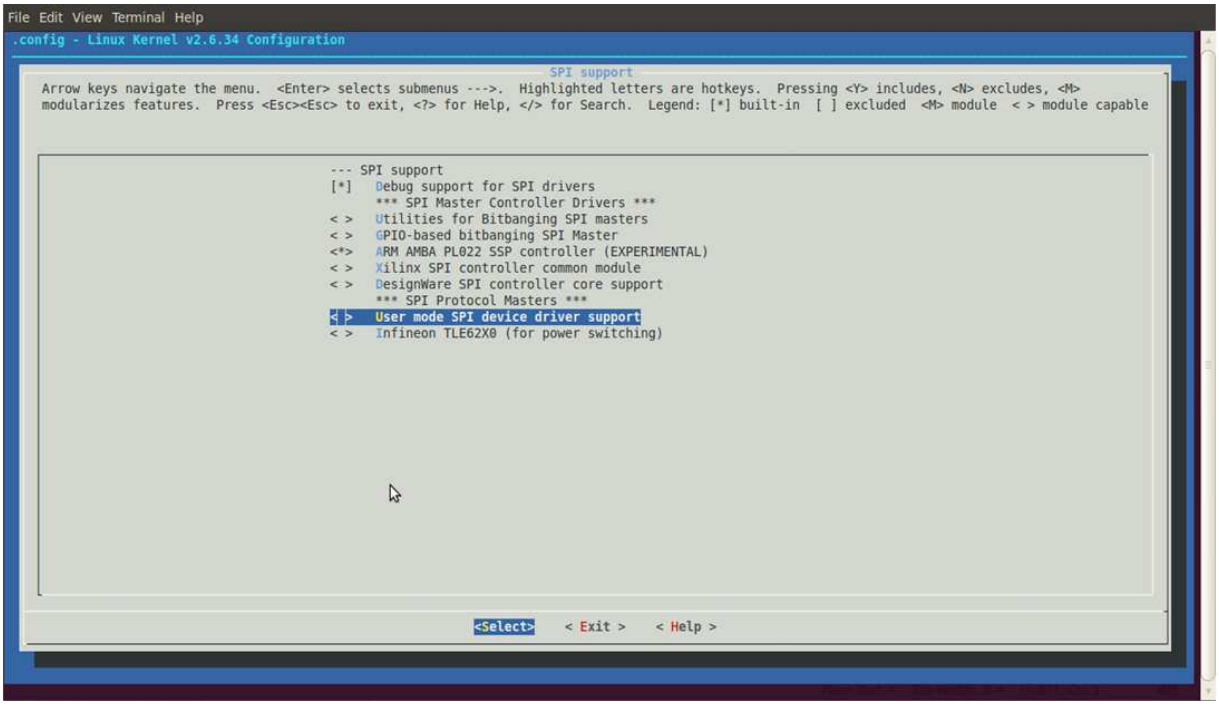

Figure 2. Linux kernel configuration

2.3 Adding a device node

A static device node is created for the device driver by adding the major/minor number to the file ltib/bin/device_table.txt. The driver takes 156 as the major number and 0 as the minor number. But it can be changed to any different value provided similar changes are also made in sc16is750_spi.h file located in sc16is750_spi-1.1.tar.gz folder.

Following line has to be added under 'Normal system devices heading' in device_table.txt:

/dev/sc750spi c 755 0 0 156 0 - - -

The new entry will create the node once the devices are force rebuilt by executing the following command on the terminal.

ltib$ ./ltib -p dev -f

2.4 Adding the driver to the local package pool

The driver source code has to be added to the local package pool of LTIB before it can be installed. When LTIB is installed on the system it creates a local package directory at opt/ltib/pkgs. The sc16is750_spi-1.1.tar.gz has to be moved to this directory. To do this, execute the following command:

$ sudo mv /<path to the location of the tar file>/sc16is720_spi-1.1.tar.gz /opt/ltib/pkgs

2.5 Installing the dver

The driver is installed using the spec file that contains the package name, version, steps to build, install and clean the package. This spec file is located in sc16is750_spi folder which must be copied to ltib/dist/lfs-5.1 folder by executing the following command:

$ mv /<path to the location of spec folder>/sc16is750_spi /..../ltib/dist/lfs-5.1

Then, move to the ltib directory to build and deploy the driver into the Linux root file system.

ltib$ /.ltib -p sc16is750_spi.spec -m prep
ltib$ /.ltib -p sc16is750_spi.spec

2.6 Loading the module

Note: Please make the hardware connections before loading the module. See section 3.

After the successful completion of above steps, the driver module spi_bridge.ko can be found at /ltib/rootfs/lib/modules/<kernel_version>/misc directory. Now the board has to be booted with the kernel uImage and the rootfs directory has to be deployed either using NFS or an SD card. Information on how to deploy boot-loader, Linux kernel and the root file system can be found at www.lpclinux.com and www.phytec.com. Once Linux is running on the board, you will see something in the terminal output:

PHY3250>

Change the directory to the location where the driver is present and use insmod to install the loadable kernel module.

PHY3250> cd lib/modules/<kernel-version>/misc

Before executing insmod, it is always safe to apply hardware reset to sc16is750 device.

PHY~misc> insmod spi_bridge.ko

If the module is successfully loaded, the following lines are shown on the terminal.

Dev num allocation...done
Device registration...done
Interrupt line acquired successfully
PHY3250>

3. Hardware Connection

The LPC3250 has a SPI bus which is connected to the SPI bus of sc16is750. Apart from this, the driver sets GPI-4 of LPC3250 as the edge-falling trigger interrupt input pin. This pin is used to receive interrupt from sc16is750 and so the SPI-IRQ line from sc16is750 should be connected to GPI-4.

Following are the pin connections between LPC3250 extension board and sc16is750.

DESCRIPTION	**SC16IS750 (JP6)**	**LPC3250 (EXTENSION BOARD)**
IRQ	Pin 7	17D (GPI-4)
CS	Pin 2	17B (SSEL0)
MISO	Pin 3	17A (MISO0)
MOSI	Pin 4	17E (MOSI0)
SCLK	Pin 5	16F (SCK0)
GND	Pin 6	3C (GND) OR ANY OTHER GND

The UART of sc16is750 can be connected to any other development board or to serial communication software like HyperTerminal (for Windows) and minicom (for Linux). Then it can be used to send and receive the data. For driver testing, UART of sc16is750 was serially connected to the HyperTerminal.

4. Understanding the Driver

The sc16is750 is a board from NXP Semiconductors that provides both I2C and SPI interface to a high performance UART. The device driver is written specifically for LPC3250 board to interact with sc16is750. It is a Linux driver and it supports only SPI bus interface to UART. There is a separate driver for the I2C to UART interface. The driver utilizes inbuilt SPI framework to send and receive the data. It customizes the configuration registers of sc16is750 to obtain the best performance. The SPI bus of LPC3250 is connected to the SPI bus of the sc16is750 board for data transfer as explain in this section later. The driver can be accessed from user-space by writing an application program a sample of which is provided in the package. Before delving into how the driver is accessed, the driver code should be read to understand its working. Here is a brief explanation.

The driver uses two buffers of 2KB (2048 characters) each for transmission and reception of data. So the data to be sent to sc16is750 via SPI bus should not exceed more than 2KB characters at a time. Else the size is automatically truncated to 2KB by the driver. Similarly, all the received data is stored in 2KB buffer and if the size exceeds, the data is overwritten on the buffer thus loosing the previous stored data. It is important that the received data that is kept stored in a buffer by the driver is first read out by the user application before it exceeds the capacity.

After executing insmod, the user must write an application program and perform the following settings in sequence before trying to send/ receive the data. Please do not attempt to change any configuration register values. Use the ioctl() command to set the baud rate, word frame (parity, word length, no. of stop bits), hardware flow control and to send/receive data.

1. Open the device file
2. Set the baud rate
3. Set Hardware Flow control
4. Set the word frame
5. Start sending/receiving

4.1 Opening the driver

The driver can be opened as follows:

```
int file;
file = open (“dev/sc750spi”, O_RDWR);
```

4.2 Setting baud rate

By default the baud rate for the UART is set to 115200 bps. But it can be configured to other value. To set a different baud rate use the following command:

```
unsigned long div_value;
div_value = op_freq / (baudrate * 16);
ioctl(file, SET_BAUD, div_value);
```

Note:

SET_BAUD value is 0x0508 (see sc16is750_spi.h inside sc16is750_spi-1.1 folder)
op_freq is the operating frequency of the sc16is750 which is 14745600.
baudrate is the baud rate that has be set for the UART (eg. 115200, 9600 etc)

4.3 Setting hardware flow control

By default, there is no hardware flow control enabled. In order to enable it use the following command:

```
ioctl(file, HW_CONTROL, 0);
```

Note:

The third argument value does not matter. It can be set to '0'.
HW_CONTROL value is 0x0551 (see sc16is750_spi.h inside sc16is750_spi-1.1 folder)

The hardware flow control can be disabled as follows:

```
ioctl(file, NO_HW_CONTROL, 0);
```

Note:

The third argument value does not matter. It can be set to '0'.
HW_CONTROL value is 0x0550 (see sc16is750_spi.h inside sc16is750_spi-1.1 folder)

4.4 Setting word frame

The sc16is750 also provides the option to set the word frame. The word length, stop bits length and the parity bit can be set by LCR register of sc16is750. Use the following ioctl() commands to configure them.

Parity set commands:

```
ioctl(file, NO_PARITY, 0);              //To set no parity
ioctl(file, ODD_PARITY, 0);             //To set odd parity
ioctl(file, EVEN_PARITY, 0);            //To set even parity
ioctl(file, FORCED_ONE_PARITY, 0);      //To set forced one parity
ioctl (file, FORCED_ZERO_PARITY, 0);    //To set forced zero parity
```

Note:

The third argument value does not matter. It can be set to '0'.
NO_PARITY value is 0x0750 (see sc16is750_spi.h inside sc16is750_spi-1.1 folder)
ODD_PARITY value is 0x0751.
EVEN_PARITY value is 0x0752.
FORCED_ONE_PARITY value is 0x0753.
FORCED_ZERO_PARITY value is 0x0754.

Word Length and Stop bit set commands:

```
ioctl (file, WL5_1SB, 0);               //To set Word Length 5 and no. of Stop Bits 1
ioctl (file, WL5_1_5SB, 0);             //To set Word Length 5 and no. of Stop Bits 1.5
ioctl (file, WL6_1SB, 0);               //To set Word Length 6 and no. of Stop Bits 1
ioctl (file, WL6_2SB, 0);               //To set Word Length 6 and no. of Stop Bits 2
ioctl (file, WL7_1SB, 0);               //To set Word Length 7 and no. of Stop Bits 1
ioctl (file, WL7_2SB, 0);               //To set Word Length 7 and no. of Stop Bits 2
ioctl (file, WL8_1SB, 0);               //To set Word Length 8 and no. of Stop Bits 1
ioctl (file, WL8_2SB, 0);               //To set Word Length 8 and no. of Stop Bits 2
```

Note:

The third argument value does not matter. It can be set to '0'.
WL5_1SB value is 0x0850 (see sc16is750_spi.h inside sc16is750_spi-1.1 folder)
WL5_1_5SB value is 0x0851.
WL6_1SB value is 0x0852.
WL6_2SB value is 0x0853.
WL7_1SB value is 0x0854.
WL7_2SB value is 0x0855.
WL8_1SB value is 0x0856.
WL8_2SB value is 0x0857.

4.5 Sending/Receiving data

Below is the pseudo code to explain how the data communication should be done using the driver.

Transmission

1. Give the ioctl() command to tell the driver that the data has to be send via THR (Transmit Holding Register).
   ioctl(file, WRITE_THR, 0);

Note:

The third argument value does not matter. It can be set to '0'.
WRITE_THR value is 0x0507 (see sc16is750_spi.h inside sc16is750_spi-1.1 folder)

2. Break the data into portion of 2048 characters.

3. Send each portion of data one after another using write() command as follows:
   write(file, buffer, sizeof(buffer));

   Here buffer is the pointer to the location of the first character to be sent and size of the buffer should not exceed 2048 characters. Also the first byte of the buffer should be set to the register value. So if we need to send 5 characters to the THR, we append THR address (0x00) at the start of the buffer and so a total of 6 characters is send.
   E.g. *ioctl(file, buffer, 6)* ;where buffer[0] = RHR address (0x00) and the rest 5 data bytes follow.

4. Give a small delay between two write() commands so that the driver has enough time to complete first transmission.

Reception

The driver constantly saves the data in a buffer of size 2048 KB as it is received. In case there is an over flow then the driver looses the previous data and restarts saving the new incoming data in that buffer. So care must be taken when data is sent to sc16is750 from the UART end. The user can ask for the size of the received data by issuing read() after this ioctl() command:

```
unsigned char ch[2];
ioctl(file, READ_RX_SIZE, 0);
read(file, ch, 0)
```

Note:

The third argument value does not matter. It can be set to '0'.
READ_RX_SIZE value is 0x0506 (see sc16is750_spi.h inside sc16is750_spi-1.1 folder)

The driver will send the received buffer size at 'ch' location. So ch[0] will have the LSB and ch[1] will have the USB. In order to retrieved the size in decimal format:

```
int size;
size = *(ch+1); //MSB
size = (size*100) + (*ch);                    //size = MSB*100 + LSB
```

There are two ways to read the data received by the driver. Either the whole saved data can be read out at once or just a part of it.

```
char *ch;
.
.
.
ioctl(file, READ_RHR, 0);
read(file, ch, count);
```

If count = 0, the whole data is read out of the driver.
If count <> 0, data of size count is read out of the driver.

To make it more clear, suppose the current size of the received data is 1200 bytes (characters) stored in a buffer of size 2KByte. If the read() command is issued with the third argument set to ‘0’ then it will read out all 1200 bytes and store it at 'ch' pointer location. Now READ_RX_SIZE issued with ioctl() will return 0. If the read() command is issued with the third argument set to ‘200’ then it will read out only 200 characters and store it at 'ch' pointer location. Now READ_RX_SIZE issued with ioctl() will return size 1000. (1200-200)

The following sample code will do the reading in a chunk of 200 bytes:

```
char *ch = malloc(sizeof(char) * 200);
unsigned int size, count;

ioctl(file, READ_RX_SIZE, 0);
read(file, ch, 0);
size = *(ch+1); //MSB
size = (size*100) + (*ch); //size = MSB*100 + LSB

while(size > 0)
{
        ioctl(file, READ_RHR, 0);

        if(size> 200)
        {
                read(file,ch , 200);
                size = size – 200;
        }
        else
        {
                read(file, ch, 0);
                size = 0;

        }
        printf(“%s”, ch);
        //The delay will ensure that the previous read is completed before issue the next one.
        delay(100);
}
```

4.6 Writing/Reading configuration registers

Any of the configuration registers of sc16is750 should not be modified. The driver takes care of the configuration in order to optimize the output. If any of the register value is changed, it may cause kernel panic.

5. Sample Application Program

The testing of the driver is done using the application program that comes with the package. To run the program, go through the part of installation and hardware connections explained in this section. This section also explains how a user can create its own application program and what all care must be taken.

5.1 Installation

The application program has to be added to the local package pool first and then it should be installed using its spec file.

Adding the application program to LPP:

$ sudo mv /<path to the location of the tar file>/spitest-1.1.tar.gz /opt/ltib/pkgs

Adding the spec file to the spec directory:

$ mv /<path to the location of spec folder>/spitest /..../ltib/dist/lfs-5.1

Then, move to the ltib directory to build and deploy the application program into the linux root file system.

ltib$ /.ltib -p spitest.spec -m prep
ltib$ /.ltib -p spitest.spec

There is rx.txt and tx.txt file that has to be copied to the ltib/rootfs/lib/modules/<kernel version>/misc/ folder.

sudo mv /path to the location of rx.txt file/rx.txt /.../ltib/rootfs/lib/modules/<kernel version>/misc
sudo mv /path to the location of tx.txt file/rx.txt /.../ltib/rootfs/lib/modules/<kernel version>/misc

5.2 Hardware Connection

Hardware connection is the same as explained in section 3. It is important to serially connect the UART of sc16is750 to the other system that has HyperTerminal or minicom running on it.

5.3 Description

The application program is meant to guide a user on how to access a driver from user-space. It uses the same methods to access the driver as explained in section 4. The program provides a menu that helps the user to do the following operations:

1. Set Baud Rate
2. Set Hardware Flow Control (Auto CTS/RTS)
3. Set Word frame
4. Read Register
5. Write Register
6. Write Data
7. Read Data
8. I/O Pins Control

Baud rate, hardware flow control and the word frame are set as explained in section 4.

Reading Registers

Not all the configuration registers of sc16is750 are accessible for reading directly. In many cases, when a particular bit position in one register is set, it allows reading out other registers. The application program takes care of all such details as the driver is ignorant of such conditions. Suppose the user wants to read out '*Enhanced Feature Register*' whose address is 0x10 (obtained by 0x02<<3). This register is accessible only when LCR = 0xBF. But do not directly set the LCR to 0xBF, instead the value of LCR is first read out and stored in a temporary variable. Later LCR is set to 0xBF and then EFR is read out. Once reading is done, LCR is again set to its original value using the same temporary register. Following sample code is given for better explanation for the above condition.

```
unsigned char read_buffer[2], write_buffer[2], temp;

write_buffer[0] = LCR;

//writing LCR address first lets this register to be read by the following read() instruction.
write(file, write_buffer, 1);
read(file, temp, 1); //this will read LCR and store it in temp variable
```

```
write_buffer[1] = 0xBF; //Setting LCR = 0xBF
write(file, write_buffer, 2);

write_buffer[0] = EFR;
write(file, write_buffer, 1);
read(file, read_buffer, 1);

printf("\nThe register value is %x", read_buffer[0]);

write_buffer[0] = LCR;
write_buffer[1] = temp; //Setting back the old reg. LCR value
write(file, write_buffer, 2);
```

Writing Register

Writing to a register should be completely avoided. In cases like above sample program where LCR has to be set to 0XBF to read out EFR, extreme precaution should be taken to reset the LCR to its previous values. Also such changes should only be made when there is no data transfer being done at that time. In other words, there should not be any use of THR or RHR when such temporary changes to configuration registers are being made. It is because the driver works with the pre-defined set of values for the registers and modifying them may cause the driver to behave awkwardly. The application does allow writing to Scratchpad Register (SPR) though, because it is not used by the driver anywhere.

Write/Read Data

When the user selects option 6 from the menu, the program will send the data from tx.txt file located inside lib/modules/<kernel version>/misc folder in a chunk of 200 bytes. The data can be seen on the HyperTerminal running on the other system serially connected to LPC3250 board.

Whatever data is send from the HyperTerminal can be received by the user by selecting option 7 from the menu. Make sure the data transfer from HyperTerminal to the sc16is750 is complete before choosing option 7. The best way is to wait for a while to ensure that the driver is not used while it is busy receiving the data from sc16is750. Once option 7 is selected, the size of the data received by the driver is shown. Now the user can either read out the whole data at once or it can choose a fix size to read. Whatever data is read, it is stored in rx.txt file which can be viewed anytime for cross verification.

I/O pins control

The program also configures the IODir, IOState, IOIntEna registers to set the 8 GPIO pins as input or output. The modifications in these registers will not create any harm to the driver as these registers are not utilized by the driver during the transmission of the data.

Note: Please read out the application program completely before trying to create your own. It provides a good understanding of how the driver can be accessed by the user. Once again, do not try to modify any registers without prior knowledge of the driver's working.

6. Support

The device driver uses some of the hardware specifics of LPC3250. The interfacing of LPC3250 with sc16is750 via SPI bus is interrupt driven. So the driver requests an interrupt line on GPI-4 (General Purpose Input Register – 04) and sets it as edge-falling trigger interrupt. Following is the instruction to request an interrupt line (according to the driver code):

request_irq(IRQ_LPC32XX_GPI_04, executethread, IRQF_DISABLED, "intr_handler", NULL);

Here IRQ_LPC32XX_GPI_04 is the interrupt line allocated for GPI-4 by the interrupt controller of LPC3250. To set the interrupt as falling-edge:

set_irq_type(IRQ_LPC32XX_GPI_04, IRQ_TYPE_EDGE_FALLING);

Both these code lines are present inside sc750spi_probe() function of the driver. If this driver has to be used for other development boards, then the above two lines of code has to be modified according to their hardware specifics. Rest part of the driver code need not to be changed.

Also the driver installation technique and hardware connections as explained in section 2 and 3 respectively will get changed.

***Author**:*
Malay Jajodia
Dated Dec' 2010
***Distributed By**:*
NXP Semiconductors
1151, McKay Drive
San Jose, CA95131-1706
Ph: (408) 433-3960