MARK BIRDSALL

CONNECTING PERIPHERAL DEVICES TO A PANDA BOARD USING PSI
This is an application note that will help somebody use the Serial Programming Interface that is available on the OMAP-based PandaBoard’s expansion connector and also explains how use the SPI to connect with a real-time clock (RTC) chip. Ever since the PandaBoard came out, there has been a community of eager programmers constructing creative projects and asking questions about where else and what more they could do to extend the PandaBoard’s abilities. This application note will document a way to connect devices to a OMAP-based devise like a PandaBoard

**What is the Serial Programming Interface**

“Serial Programming Interface” (SPI) is a simple standard that was developed my Motorola. SPI can also be called “4-wire” interface (as opposed to 1, 2 or 3-wire serial buses) and it is sometimes referred to like that because the interface has four wires defined. The first is Master-Out-Slave-In (MOSI) and the second is the Master-In-Slave-Out (MISO). There is also a Serial Clock from the Master (SCLK) and a Chipselect Signal (CS#) which can allow for more than one slave devise to be able to connect with one master.

**Why do we want to use the SPI?**

There are various ways to connect a peripheral device to the PandaBoard like USB, SPI, etc... and SPI has some advantages. Using the Serial Programming Interface costs less in terms of power usage and it is easy to connect different devises to the PandaBoard and also to debug any problems that occur all while maintaining an acceptable performance rate.

![Figure 1](http://www.omappedia.com/wiki/PandaBoard_SPI)
What can you connect to the SPI?

A lot of different types of devices can be connected to the PandaBoard via the SPI bus and there are drivers for many SPI devices in the SPI subsystem in Linux. The list of drivers include Ethernet adapters, SD controllers, SPI flash memory (MTDs), etc… and also includes a generic driver called “spidev” that introduces an Application Programming Interface to the userspace and allows control from application level.

Connecting the PandaBoard to an SPI device.

There is an expansion connector on the PandaBoard.

![Figure 3](http://semiaccurate.com/static/uploads/2010/09_september/pandaboard.jpg)

Within the highlighted expansion connector in Figure 3 there are the pins associated to the multi-channel serial port interface (MCsPI).

![Figure 4](http://semiaccurate.com/static/uploads/2010/09_september/pandaboard.jpg)

These pins are (in order from left to right):
- Pin 4 MCSPI_CS3
- Pin 10 MCSPI_CS1
- Pin 12 MCSPI_SIMO
- Pin 14 MCSPI_CS2
- Pin 16 MCSPI_CS0
- Pin 18 MCSPI_SOMI
- Pin 20 MCSPI_CLK
In Figure 4 we can see pins 16, 10, 14, 4 highlighted. These pins are the CS0/CS1/CS2/CS3 lines respectively and they are the pins that allow up to four devices to be connected to the master. These are not all needed, of course, so therefore pins MCSPI_SIMO, MCSPI_SOMI, MCSPI_CLK and MCSPI_CS0 (pins 12, 18, 20, 16) are the pins that are most commonly used.

Now if we want to use a DS3234 chip seen in Figure 5 (left), some extra measures need to be taken. The chip wants 5V to power it but the BandaBoard only has 1.8V to give. In this case we would need two PCA9306 chips that acts as a bidirectional voltage-level translator. All you need to do is connect these chips as seen in Figure 6: for both PCA9306 chips, drive the ENABLE pins to high, 1.8V to VREF1 and 5V to VREF2. Doing this successfully translates those voltage levels between pins SDA1/SCL1 and SDA2/SCL2.

After this, all you have to do is connect the MCSPI_SOMI, MCSPI_SIMO, MCSPI_CS0 and MCSPI_CLK pins from the PandaBoard to one side of the voltage translators and MISO, MOSI, SS and CLK from the DS3234 chip to the other side. Refer to Figure 6.

Figure 5: http://www.alpha-crucis.com/1150-1602-large/deadon rtc-ds3234-breakout.jpg & http://nicegear.co.nz/obj/images/10403-01.jpg
Drivers stack in Linux for SPI

Inside the Linux kernel, you can find already existing code relating the SPI subsystem mentioned above. It provides SPI master drivers, SPI board information and SPI functional device drivers (protocol masters).

- SPI board information:
  This is code that performs the registration of the peripheral devices with the SPI subsystem.
- SPI master drivers:
  These are drivers that already know how to information to the peripheral devices and they deal with the SPI controller.
- SPI functional device drivers:
  These drivers are charged with managing the peripheral devices connected to the SPI bus as they keep track of where information is supposed to go (sending or receiving). Also, these drivers are in charge of providing an interface for users.

What should we change inside the kernel?

SPI devices cannot provide information about themselves like other connections like USB can so the master driver cannot even know if anything is connected to the Chipselect (CS#) so we have to use some code that will provide this sort of information. We will call the function `spi_register_board_info` to do the registration of the devices. Here is an example:

```
spi_register_board_info(panda_spi, ARRAY_SIZE(panda_spi));
//Two parameters: list of peripheral devices attached to PCI bus and the how big the list is
//Two peripheral devises are registered next:
static const struct spi_board_info panda_spi[] __initconst =
{  
    {  
        .modalias = "spidev",  
        //first devise controlled by “spidev” driver
        .chip_select = 1,
```
Using the Peripheral Devices

When you are recompiling all of the code after your changes, you should check to see if all of these options are switched on:

CONFIG_SPI = y
CONFIG_SPI_MASTER = y
CONFIG_SPI_OMAP_24xx = y
CONFIG_RTC_DRV_DS3234 = y
CONFIG_SPI_SPIDEV = y

After compilation has started this message will appear:

ds3234 spi1.0: rtc core: registered ds3234 as rtc0

This display means that your devise has been successfully connected to the ds3234 driver and the RTC interface. Now you can access any of the tools utilizing the real-time clock.
Try entering this command to use a clock function:

# hwclock --show

You can also utilize the spidev interface which allows for userspace applications. Try this sample code from http://www.omappedia.com/wiki/PandaBoard_SPI:

import os, sys

def spidev_test(devnode):
    //Simple code; no exception handling.
    //Slight error handling
    spi = os.open(devnode, os.O_RDWR, 0777)  //Open the devise

    write_bytes = "\x00"  //Write zero byte
w_res = os.write(spi, write_bytes, len(write_bytes))
if written != len(write_bytes):
    raise Exception("You wrote less bytes than requested")

rd = 8
rd_bytes = os.read(spi, rd)  # Read 8 bytes back
if len(rd_bytes) != rd:
    raise Exception("Read less than expected")

print ["%x" % ord(rd) for rd in rd_bytes]  # Print

os.close(spi)  # Close

if __name__ == "__main__":
    spidev_test ("/dev/spidev1.0")
sysexit(0)
else:
    print "How do you want to use spidev-test?"

This code tries to send the zero byte to the peripheral devise and it then prints a response of 8 bytes.

**Debugging**

Sometimes things go wrong! If something does check these things for some easy fixes.

- The line “#define VERBOSE” will print all the date that is sent or received from the peripheral devises if you insert it in the master driver source.
- Enable CONFIG_DEBUG_KERNEL and CONFIG_SPI_DEBUG and you will get more debugging information from the kernel.
- Double check your wiring
- Make sure the devise is powered properly.

**References**

http://www.omappedia.com/wiki/PandaBoard_SPI
http://nicegear.co.nz/obj/images/10403-01.jpg
http://www.techonlineindia.com/Libraries/tol/SPI3.sflb.ashx