AN5633

Configuring PCI1xxxx Through Serial Communication

Author: *Christopher Brisco*
Microchip Technology, Inc.

1.0 INTRODUCTION

The PCI1xxxx family of PCIe fanout switches allows for several different configuration options. EEPROM and OTP provide for a static configuration, but the option of configuration over I^2C or SPI allows for dynamic configuration on startup. This gives greater system flexibility. To properly configure the devices via this serial interface, there are a few straps that need to be set and registers read/written, which are detailed in this document.

1.1 Sections

This application note covers the following topics:

- Section 2.0, Hardware Settings
- Section 3.0, Registers
- Section 4.0, I^2C
- Section 5.0, SPI

1.2 References

Consult the following documents when using this application note:

- *PCI12000 Data Sheet*
- *PCI11010 Data Sheet*
- *PCI11101 Data Sheet*
- *PCI11400 Data Sheet*
- *PCI11414 Data Sheet*
- *AN5213 Configuration and Programming Options for the PCI1xxxx*

2.0 HARDWARE SETTINGS

PCI1xxxx loads its configuration in the following order: *Hardware defaults>OTP>EEPROM>SERIAL (I2C or SPI)*. See Figure 1.

FIGURE 1: CONFIGURATION FLOWCHART

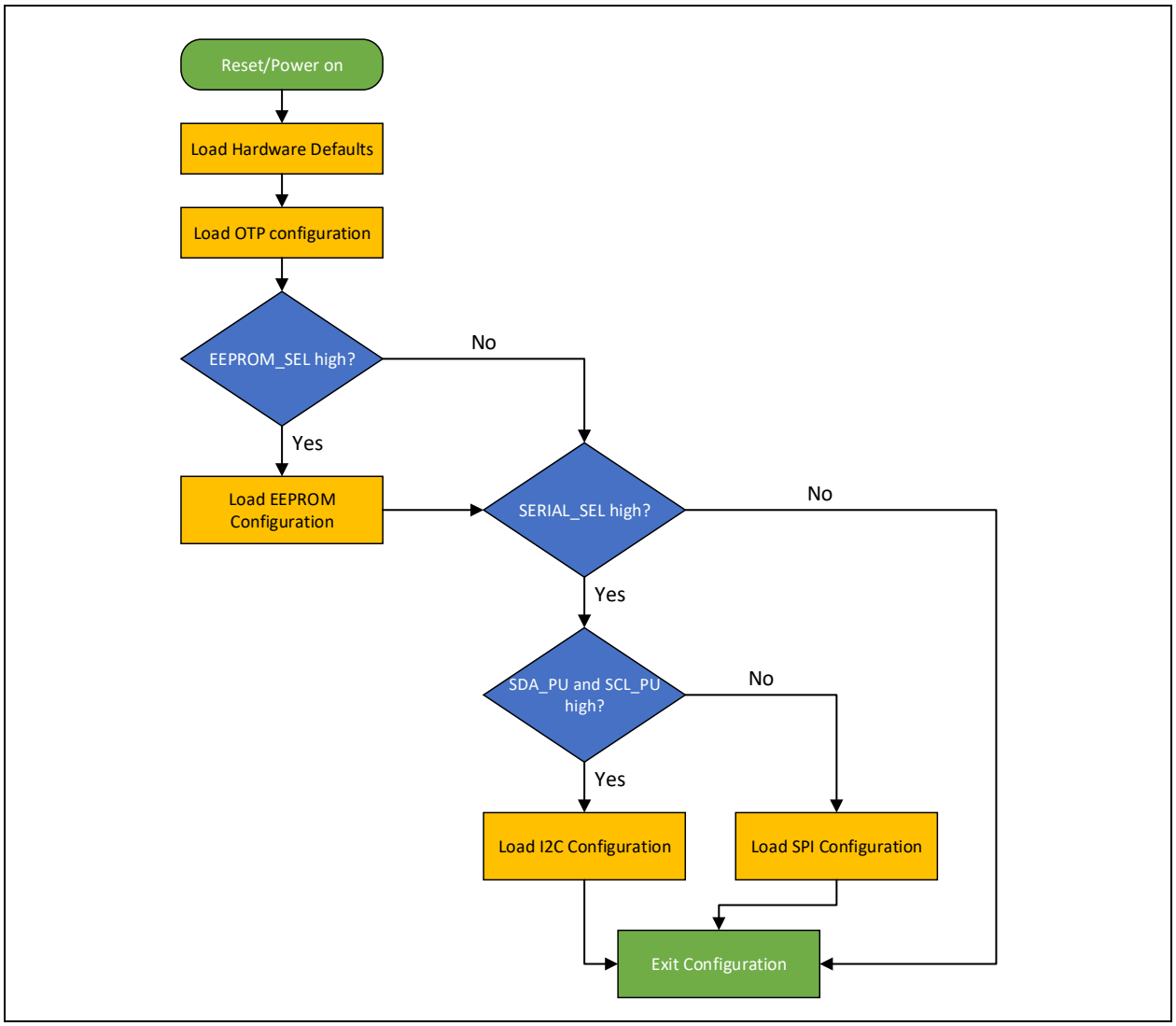

Table 1 gives an overview of strap settings for serial configuration.

TABLE 1: STRAP SETTINGS

	EEPROM_SEL	SERIAL_SEL	SDA_PU	SCL_PU
I^2C Config	X	High	High	High
SPI Config	X	High	Low	Low

Note that the value of EEPROM_SEL strap does not matter. PCI1xxxx can be configured by EEPROM and Serial or it can be configured just by Serial.

PCI1xxxx uses PERST# as a Warm reset. Whenever PERST# is asserted and then deasserted (brought low and then high), PCI1xxxx will go through the configuration process again. This consideration is important when designing a system that utilizes serial configuration. PCI1xxxx will not enumerate until it has completed its configuration. Therefore, if PERST# is asserted and then deasserted, but a host does not reconfigure PCI1xxxx, the device will appear unresponsive on PCIe until the serial configuration is completed.

3.0 REGISTERS

During serial configuration, the host configuring PCI1xxxx should start by polling the BYTE_TEST_REG (offset 0x00240120). Once the correct pattern (0x87654321) is recognized, the host can then go on to configure the chip.

After the host is done configuring a subsystem, it should write the appropriate bit in EXT_SYS_CONFIG_DONE_REG (offset 0x0020084). Once all the configuration is done, the value 0X01073F3F will have been written to the register, but trying to read any register will return 0x00000000. This indicates that configuration is done, and PCI1xxxx has now started PCIe enumeration.

TABLE 2: EXT_SYS_CONFIG_DONE_REG

Bits	Name	Description
31:25	Reserved	Always reads ‘0’
24	EXT_SYSTEM_CONFIG_DONE	Set by external system to indicate configuration of the system registers is done
23:19	Reserved	Always reads ‘0’
18	EXT_PHYC_CONFIG_DONE	Set by external system to indicate configuration of PHY C is done
17	EXT_PHYB_CONFIG_DONE	Set by external system to indicate configuration of PHY B is done
16	EXT_PHYA_CONFIG_DONE	Set by external system to indicate configuration of PHY A registers is done
15:14	Reserved	Always reads ‘0’
13	EXT_SW_P4_CONFIG_DONE	Set by external system to indicate configuration of SW Port 4 is done
12	EXT_SW_P3_CONFIG_DONE	Set by external system to indicate configuration of SW Port 3 is done
11	EXT_SW_P2_CONFIG_DONE	Set by external system to indicate configuration of SW Port 2 is done
10	EXT_SW_P1_CONFIG_DONE	Set by external system to indicate configuration of SW Port 1 is done
9	EXT_SW_P0_CONFIG_DONE	Set by external system to indicate configuration of SW Port 0 is done
8	EXT_SW_MAIN_CONFIG_DONE	Set by external system to indicate configuration of the SW Main configuration is done
7:6	Reserved	Always reads ‘0’
5	EXT_USB_CONFIG_DONE	Set by external system to indicate configuration of the USB subsystem is done
4	EXT_ENET_CONFIG_DONE	Set by external system to indicate configuration of the Ethernet subsystem is done
3	EXT_PERI_GEN_CONFIG_DONE	Set by external system to indicate configuration of the General Peripheral subsystem is done
2	EXT_PERI_SPI_CONFIG_DONE	Set by external system to indicate configuration of the SPI subsystem is done
1	EXT_PERI_SMBUS_CONFIG_DONE	Set by external system to indicate configuration of the SMBUS subsystem is done
0	EXT_PERI_UART_CONFIG_DONE	Set by external system to indicate configuration of the UART subsystem is done

4.0 I²C

The SMBUS_SCL_PU and SMBUS_SDA_PU lines must be pulled high when PCI1xxxx checks the straps to enable I²C configuration. These lines also double as the SCL and SDA lines for I²C configuration. I²C Target mode supports 100 KHz and 400 KHz operation.

The I²C target address is stored in `SMBUS_TGT_CONFIG_REG` (offset `0x00240110`). By default this is `0x04`, but it can be changed in EEPROM or OTP configuration before serial configuration begins.

TABLE 3: PER SKU I²C PINS

SKU	SMBUS_SCL_PU	SMBUS_SDA_PU
PCI11414	PROG64 (Pin 89)	PROG65 (Pin 90)
PCI11400	PROG64 (Pin 66)	PROG65 (Pin 67)
PCI11101	PROG64 (Pin 74)	PROG65 (Pin 75)
PCI11010	PROG64 (Pin 60)	PROG65 (Pin 61)
PCI12000	PROG64 (Pin 38)	PROG65 (Pin39)

TABLE 4: I²C COMMAND FORMAT

	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
Read	Control	Reg[31:24]	Reg[23:16]	Reg[15:8]	Reg[7:0]	—	—	—	—
Read Response	—	—	—	—	—	Data[31:24]	Data[23:16]	Data[15:8]	Data[7:0]
Write	Control	Reg[31:24]	Reg[23:16]	Reg[15:8]	Reg[7:0]	Data[31:24]	Data[23:16]	Data[15:8]	Data[7:0]
Write Response	—	—	—	—	—	—	—	—	—

Appendix A contains an example python script of configuring PCI1xxxx over I²C using an Aardvark.

5.0 SPI

The SMBUS_SCL_PU and SMBUS_SDA_PU lines must be pulled low when PCI1xxxx checks the straps to enable SPI configuration. On the PCI1xxxx EVBs, there are 0Ω resistors that connect the I^2C SDA and SCL configuration lines to the SMBus sideband signals on the upstream connector. The hosts on the other end of the upstream connector typically have pull-up resistors on SDA and SCL, so the 0Ω resistors should be removed if SPI configuration is desired. Refer to the user manual for your EVB for more information.

TABLE 5: PER SKU SPI PINS

SKU	SDI	SDO	SCK	CE#
PCI11414	PROG66 (Pin 91)	PROG65 (Pin 90)	PROG64 (Pin 89)	PROG67 (Pin 92)
PCI11400	PROG66 (Pin 68)	PROG65 (Pin 67)	PROG64 (Pin 66)	PROG67 (Pin 69)
PCI11101	PROG66 (Pin 76)	PROG65 (Pin 75)	PROG64 (Pin 74)	PROG67 (Pin 77)
PCI11010	PROG66 (Pin 62)	PROG65 (Pin 61)	PROG64 (Pin 60)	PROG67 (Pin 63)
PCI12000	PROG66 (Pin 40)	PROG65 (Pin39)	PROG64 (Pin 38)	PROG67 (Pin 41)

To complete the configuration in SPI mode, the `SPI_ALERT_SC` bit in the `SPI_PERI_CONFIG_REG` register must be set to 1. This is done automatically in I^2C configuration, but not in SPI configuration. SPI peripheral mode supports SPI modes 0 and 3. CS is an active-low signal.

PCI1xxxx SPI Target mode only supports SPI read and SPI write.

TABLE 6: SPI PACKET FORMAT

	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10
Read	0x03	Addr[31:24]	Addr[23:16]	Addr[15:8]	Addr[7:0]	0x00	0x00	0x00	0x00	0x00	0x00
Read Response	0x00	0x00	0x00	0x00	0x00	0x00	0x00	Data[7:0]	Data[15:8]	Data[23:16]	Data[31:24]
Write	0x02	Addr[31:24]	Addr[23:16]	Addr[15:8]	Addr[7:0]	Data[7:0]	Data[15:8]	Data[23:16]	Data[31:24]	—	—
Read Response	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	—	—

Appendix B contains an example python script of configuring PCI1xxxx over SPI using an Aardvark.

APPENDIX A: I²C CONFIGURATION

```
from aardvark_py import *
import time

DUT_ADDR = 0x04
#Register Offsets
BYTE_TEST_REGISTER = 0X00240120
EXT_SYS_CONFIG_DONE_REG = 0x240084
SYS_ENABLE_REG = 0x2400E0

#Register Values
DEVICE_READY = 0X87654321
WRITE_FINISHED = 0X01073F3F

#SS EN bits
SYS_EN_BASE = 0x700
PERI_ENABLE = 0b001
ENET_ENABLE = 0b010
USB_ENABLE = 0b100

def main():
        #Initialize the Aardvark. Note there is minimal error checking here and this code assumes
the Aardvark connected to the PCI1xxxx device is the first one connected to the host system
        handle = aa_open(0)
        #This is only here for debug. It helps verify if the Aardvark was correctly accessed
        #Any negative number means it was not
        print(aa_features(handle))

        #Check if device is ready for configuration
        read_write_reg(handle, BYTE_TEST_REGISTER, None, [DEVICE_READY], 0.1)
        read_write_reg(handle, SYS_ENABLE_REG, [SYS_EN_BASE | PERI_ENABLE | USB_ENABLE |
ENET_ENABLE], None, 0)
        #Do any PCI1xxxx configuration here. Below example enables the various subsystems
        #Finish configuration
        read_write_reg(handle, EXT_SYS_CONFIG_DONE_REG, [WRITE_FINISHED], [0], 0)

        aa_close(handle)

#handle: Aardvark handle, generated earlier
#reg: The register to access (read or write)
#data: The data to write to the register. Should be None if only reading the register
#verify: Any extra values that the register could read. Should be None if not different from data
#delay: How long to wait before reading the register again. Should only be used for checking
BYTE_TEST_REGISTER
def read_write_reg(handle, reg, data, verify, delay):
        DONE = False
        #Filter out any None type data
        if data is not None and verify is not None:
                check = data + verify
        elif data is not None:
                check = data
        elif verify is not None:
                check = verify
        #Continue to read/write the register until the expected value is there
        #Note there is no limit to retries, this can result in an eternal loop if your function
arguments are wrong
        while not DONE:
                #Compile the register address and any data into a list, write the list to I2c
```

```
                payload = format_reg(reg)
                if data is not None:
                        for entry in data:
                                payload = payload + format_reg(entry)
                aa_i2c_write(handle, DUT_ADDR, AA_I2C_NO_STOP, array('B', payload))
                #Read the register back
                out = aa_i2c_read(handle, DUT_ADDR, AA_I2C_NO_FLAGS, 4)
                num = 0
                #out is a list of 4 bytes, combine into single variable, check if it is expected
                for i in out[1]:
                        num = (num << 8) + i
                if num in check:
                        DONE = True
                else:
                        #Through experimentation, a delay here offers much greater reliability when
checking BYTE_TEST_REGISTER, should not otherwise be used
                        #Without this delay, the SDA_PU and SCL_PU straps could be pulled down when
PCI1xxxx checks the straps, resulting in PCI1xxxx expecting SPI operation
                        time.sleep(delay)
                #Print the register and the value read
                print(hex(num), end='')
                print(" on register: ",end='')
                print(hex(reg))

def format_reg(reg):
        #Break 4 byte register into a list of 4 bytes
        return [(reg & 0xff000000)>>24,(reg & 0x00ff0000)>>16,(reg & 0x0000ff00)>>8,(reg &
0x000000ff)]

if __name__ == '__main__':
main()
```

APPENDIX B: SPI CONFIGURATION

```
from aardvark_py import *
import time

DUT_ADDR = 0x04
#Register Offsets
BYTE_TEST_REGISTER = 0X00240120
SPI_PERI_CONFIG_REG = 0X00240130
EXT_SYS_CONFIG_DONE_REG = 0x240084
SYS_ENABLE_REG = 0x2400E0

#Register Values
DEVICE_READY = 0X87654321
SPI_ALERT = 0x00000001
WRITE_FINISHED = 0X01073F3F

#SS EN bits
SYS_EN_BASE = 0x700
PERI_ENABLE = 0b001
ENET_ENABLE = 0b010
USB_ENABLE = 0b100

#SPI Codes
SPI_READ = 0X03
SPI_WRITE = 0X02

def main():
        #Initialize the Aardvark. Note there is minimal error checking here and this code assumes
the Aardvark connected to the PCI1xxxx device is the first one connected to the host system
        handle = aa_open(0)
        #This is only here for debug. It helps verify if the Aardvark was correctly accessed
        #Any negative number means it was not
        print(aa_features(handle))
        #Set the bit rate, polarity, phase, bit order, CE polarity, and disable the Aardvark as SPI
target (slave)
        configure_spi(handle)

        #Continuously read the BYTE_TEST_REGISTER until the expected value is shown
        dev_ready = data_to_array(DEVICE_READY)
        READY = False
        while not READY:
                READY = read_write_reg(handle, SPI_READ, BYTE_TEST_REGISTER, 0)[1][-4:].tolist() ==
dev_ready
                #Note that this timer isn't strictly necessary, but it does help avoid spamming SPI
read/writes before PCI1xxxx is ready for config
                time.sleep(0.1
        #Do configuration here. Below code can be used for enabling/disabling different subsystems
        read_write_reg(handle, SPI_WRITE, SYS_ENABLE_REG, (SYS_EN_BASE | USB_ENABLE | ENET_ENABLE
| PERI_ENABLE))

        #Automatically clear SPI_ALERT pin. This bit is set automatically in I2C config but must be
set manually in SPI config
        read_write_reg(handle, SPI_WRITE, SPI_PERI_CONFIG_REG, SPI_ALERT)

        #Finish configuration. Write our value to the register and read it back until configuration
is finished
        done = data_to_array(WRITE_FINISHED)
        ret = 0
```

```
        while (ret != done) and (ret != [0, 0, 0, 0]):
                read_write_reg(handle, SPI_WRITE, EXT_SYS_CONFIG_DONE_REG, WRITE_FINISHED)
                ret = read_write_reg(handle, SPI_READ, EXT_SYS_CONFIG_DONE_REG, 0)[1][-4:].tolist()

        #Release the reference to the Aardvark
        aa_close(handle)

def configure_spi(handle):
        #Set bitrate to 1000KHz,
        #polarity: rising/falling, phase: Sample/Setup, bitorder: MSB
        #Set CE to active low
        #Do not operate Aardvark as SPI target (slave)
        aa_spi_bitrate(handle, 1000)
        aa_spi_configure(handle, AA_SPI_POL_RISING_FALLING, AA_SPI_PHASE_SAMPLE_SETUP,
AA_SPI_BITORDER_MSB)
        aa_spi_master_ss_polarity(handle, AA_SPI_SS_ACTIVE_LOW)
        aa_spi_slave_disable(handle)

def read_write_reg(handle, rw, addr, data):
        #For SPI config, PCI1xxxx expects the address in MSB and the data in LSB
        #If this is a read, the data should be 2 dummy bytes + 1 bytes for every byte read (reading
4 bytes means 6 bytes total)
        payload = [rw] + address_to_array(addr)
        if rw == SPI_READ:
                payload = payload + [0x00, 0x00, 0x00, 0x00, 0x00, 0x00]
        else:
                payload = payload + data_to_array(data)
        ret = aa_spi_write(handle, array('B', payload), len(payload))
        return ret


def data_to_array(data):
        #Break 4 byte data into a list of 4 bytes. The byte ordering is reversed for LSB
        return [(data & 0x000000ff),(data & 0x0000ff00)>>8,(data & 0x00ff0000)>>16, (data &
0xff000000)>>24]

def address_to_array(addr):
        #Break 4 byte address into a list of 4 bytes
        return [(addr & 0xff000000)>>24,(addr & 0x00ff0000)>>16,(addr & 0x0000ff00)>>8,(addr &
0x000000ff)]


if __name__ == '__main__':
main()
```

APPENDIX C: REVISION HISTORY

TABLE C-1: REVISION HISTORY

Revision Level & Date	Section/Figure/Entry	Correction
DS00005633A (11-26-24)	Initial release	

Microchip Information

Trademarks

The “Microchip” name and logo, the “M” logo, and other names, logos, and brands are registered and unregistered trademarks of Microchip Technology Incorporated or its affiliates and/or subsidiaries in the United States and/or other countries (“Microchip Trademarks”). Information regarding Microchip Trademarks can be found at https://www.microchip.com/en-us/about/legalinformation/microchip-trademarks.

ISBN: 979-8-3371-0154-5

Legal Notice

This publication and the information herein may be used only with Microchip products, including to design, test, and integrate Microchip products with your application. Use of this information in any other manner violates these terms. Information regarding device applications is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. Contact your local Microchip sales office for additional support or, obtain additional support at www.microchip.com/en-us/support/design-help/client-support-services.

THIS INFORMATION IS PROVIDED BY MICROCHIP "AS IS". MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE, OR WARRANTIES RELATED TO ITS CONDITION, QUALITY, OR PERFORMANCE.

IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL, OR CONSEQUENTIAL LOSS, DAMAGE, COST, OR EXPENSE OF ANY KIND WHATSOEVER RELATED TO THE INFORMATION OR ITS USE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO THE INFORMATION OR ITS USE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THE INFORMATION.

Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip products:

- Microchip products meet the specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner, within operating specifications, and under normal conditions.
- Microchip values and aggressively protects its intellectual property rights. Attempts to breach the code protection features of Microchip product is strictly prohibited and may violate the Digital Millennium Copyright Act.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is “unbreakable”. Code protection is constantly evolving. Microchip is committed to continuously improving the code protection features of our products.