Altera DE2 I2C Driver

Daniel Fiske
Michael Lam
Daniel Tiam

Preface

This application note is based on the previous application note[1] released by Braedan Jongerius, Michael Jun and Scott Hewson in the winter semester of 2013. The I2C driver was refactored and extended to include additional I2C behaviour.

Changes include:

- Explicitly start and stop bus transactions
- Read/write directly to devices in addition to internal device registers
- Continue previous read/write transactions
- Clock stretching support

The Original i2c.c and i2c.h files were written by Altera. The files can be found in C:\altera\DE2_v.2.0.2_CDROM\DE2_demonstrations\DE2_SD_Card_Audio\software\DE2_SD_Card_Audio\terasic_lib

Table of Contents

Background

Inter-Integrated Circuit (I2C) is a serial bus protocol developed by NXP Semiconductors (formerly Philips) that supports multiple masters and slaves using only 2 lines. These 2 lines are the Serial Data Line (SDA) and the Serial Clock Line (SCL). The SDA line is bi-directional and is used for transferring data between master and slave devices. The SCL line is used for synchronizing data transfers. While the SCL line is always driven by a master device, slave devices can hold the line low in a technique called clock stretching. Slave devices stretch the clock to force the master to wait until they are ready to proceed. The I2C protocol does not define a time limit for clock stretching. Furthermore, I2C devices are assigned 7-bit addresses used by a master device to select which slave should get the bus. An additional 8th bit differentiates between read and write operations. This is shown in Table 1. Consult the datasheet for your device to determine its I2C address.

Slave Address	Effective Write Address	Effective Read Address
0bXXXXXXX	0bXXXXXXX0	0bXXXXXXX1

Table 1: Device Addressing

The SDA and SCL lines are open-drain and thus require external pull-up resistors. A single resistor is attached between each line and the input high voltage required for your devices. I2C devices typically operate on +3.3V or +5V and therefore common pull-up resistor values are 1.8kΩ, 4.7kΩ or 10kΩ. An example configuration is shown in Figure 1.

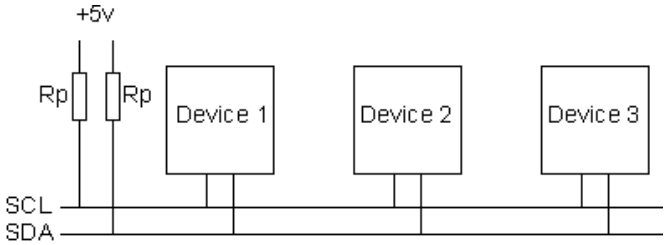

Figure 1: External Pull-up Resistors[2]

I2C transactions are initiated by a master device. A transaction begins with an I2C start sequence where the SDA line goes low while the SCL line is high. Transactions end when the master device sends the I2C stop sequence where the SDA line goes high while the SCL line is high. Figure 2 denotes these sequences as S and P respectively. During data transfer, the SDA line must remain stable when the SCL line is high. Furthermore, data is transferred in 8 bit sequences starting with the MSB. After each 8 bit transfer, the receiver will send a single acknowledgement bit.

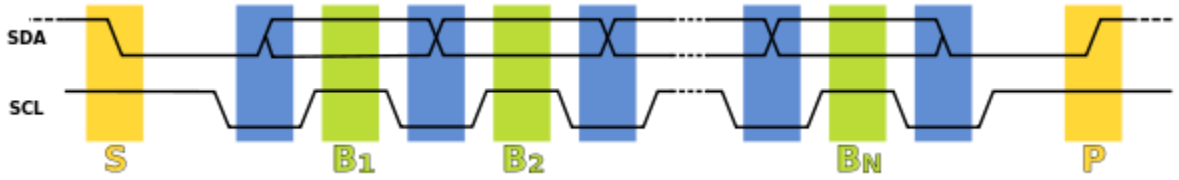

Figure 2: Device Transaction Sequence[3]

Project Setup

Qsys

1. Add a PIO from the component library under Peripherals > Microcontroller Peripherals > PIO:
    I. Set Width to 1 bit. Set Direction to output. Click finish.
    II. Rename the component to something more descriptive such as 'I2C_SCL'.
    III. Hookup the component, remembering to export the conduit.
2. Add a second PIO:
    I. Set Width to 1 bit. Set Direction to bidir. Click finish.
    II. Rename the component to something more descriptive such as 'I2C_SDA'.
    III. Hookup the component, remembering to export the conduit.

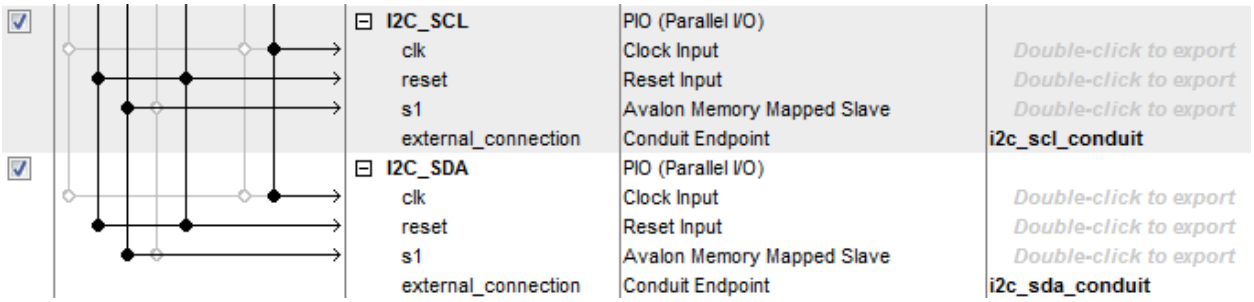

Figure 3: Qsys Interconnect

3. If you have conflicting addresses, simply click System > Assign Base Addresses.
4. Generate the SOPC.

Quartus

In your top level .vhd file:

1. Add 'GPIO_0 : inout std_logic_vector(35 downto 0) := (others => 'X');' to the project's top level entity ports.
2. Add 'I2C_SCL : out std_logic;' and 'I2C_SDA : inout std_logic := 'X';' to the NiosII component declaration ports.
3. Add 'I2C_SCL => GPIO_0(X1)' and 'I2C_SDA => GPIO_(X2)' to the NiosII component instantiation port map where X1 and X2 are integers corresponding to available GPIO pins.
4. Compile the design.

NiosII SBT for Eclipse

Download I2C.c and I2C.h and add them to your project. Include I2C.h where necessary.

Software API

void I2C_Start

Parameter	Description
alt_u32 clk_base	Base address of the I2C_SCL PIO
alt_u32 data_base	Base address of the I2C_SDA PIO

Initiates a new transaction.

void I2C_Stop

Parameter	Description
alt_u32 clk_base	Base address of the I2C_SCL PIO
alt_u32 data_base	Base address of the I2C_SDA PIO

Ends the current transaction.

bool I2C_WriteToDevice

Parameter	Description
alt_u32 clk_base	Base address of the I2C_SCL PIO
alt_u32 data_base	Base address of the I2C_SDA PIO
alt_8 deviceAddr	Effective write address of slave device
alt_u8* pData	Data buffer to write from
alt_u16 len	Number of bytes to write

Writes the specified number of bytes from the data buffer to the slave device addressed.

Usage: Call I2C_Start first. Call I2C_Stop when finished to release the bus.

bool I2C_WriteToDeviceRegister

Parameter	Description
alt_u32 clk_base	Base address of the I2C_SCL PIO
alt_u32 data_base	Base address of the I2C_SDA PIO
alt_8 deviceAddr	Effective write address of slave device
alt_8 controlAddr	Address of internal register on slave device
alt_u8* pData	Data buffer to write from
alt_u16 len	Number of bytes to write

Writes the specified number of bytes from the data buffer to an internal register on the slave device addressed.

Usage: Call I2C_Start first. Call I2C_Stop when finished to release the bus.

bool I2C_WriteMore

Parameter	Description
alt_u32 clk_base	Base address of the I2C_SCL PIO
alt_u32 data_base	Base address of the I2C_SDA PIO
alt_u8* pData	Data buffer to write from
alt_u16 len	Number of bytes to write

Writes the specified number of bytes from the data buffer to the last addressed slave device.

Usage: Call I2C_WriteToDevice or I2C_WriteToDeviceRegister first. Call I2C_Stop when finished to release the bus.

bool I2C_ReadFromDevice

Parameter	Description
alt_u32 clk_base	Base address of the I2C_SCL PIO
alt_u32 data_base	Base address of the I2C_SDA PIO
alt_8 deviceAddr	Effective write address of slave device
alt_u8* pBuf	Data buffer to read into
alt_u16 len	Number of bytes to read

Reads the specified number of bytes into the data buffer from the slave device addressed.

Usage: Call I2C_Start first. Call I2C_Stop when finished to release the bus.

bool I2C_ReadFromDeviceRegister

Parameter	Description
alt_u32 clk_base	Base address of the I2C_SCL PIO
alt_u32 data_base	Base address of the I2C_SDA PIO
alt_8 deviceAddr	Effective write address of slave device
alt_8 controlAddr	Address of internal register on slave device
alt_u8* pBuf	Data buffer to read into
alt_u16 len	Number of bytes to read

Reads the specified number of bytes into the data buffer from an internal register on the slave device addressed.

Usage: Call I2C_Start first. Call I2C_Stop when finished to release the bus.

void I2C_ReadMore

Parameter	Description
alt_u32 clk_base	Base address of the I2C_SCL PIO
alt_u32 data_base	Base address of the I2C_SDA PIO
alt_u8* pBuf	Data buffer to read into
alt_u16 len	Number of bytes to read

Reads the specified number of bytes into the data buffer from the last addressed slave device.

Usage: Call I2C_ReadFromDevice or I2C_ReadFromDeviceRegister first. Call I2C_Stop when finished to release the bus.

Sample API Code

Sample Project

Included in the app notes is the 2014w G5 partial project (G5_2014w_I2C_NFC.qar).
This project contains working I2C for the real time clock and NFC communication.
Areas of interest:

- main.c
- hw/rtc.c - uses I2C code
- hw/pn532.c - uses more advanced I2C code
- hw/i2c/i2c.h - contains the modified Altera I2C code
- hw/i2c/i2c.c - contains the modified Altera I2C code

App note readers who do not have similar hardware can use the oscilloscope to read I2C transactions from the running project.

Rtc.c

The following sample code is a snippet from Group 5 2014's rtc.c code. It contains two functions, one that writes and another that reads to the RTC Module through I2C. The time conversion functions are not shown since they are application specific, and do not affect the I2C communication. Note that additional calls to I2C_Start are considered restart sequences without releasing the bus.

```
#include <time.h>
#include "i2c/I2C.h"

#define RTC_SCL_BASE I2C_RTC_SCL_BASE
#define RTC_SDA_BASE I2C_RTC_SDA_BASE

#define RTC_I2C_ADDR 0xD0
#define RTC_REG_BASE 0x00
#define RTC_REG_LEN 7
#define RTC_DST 0x08

bool setRtcFromStruct(struct tm rtcTime)
{
	alt_u8 timeReg[RTC_REG_LEN];

	timeReg[RTC_SECONDS] = setSeconds(rtcTime.tm_sec);
	timeReg[RTC_MINUTES] = setMinutes(rtcTime.tm_min);
	timeReg[RTC_HOURS] = setHours(rtcTime.tm_hour);
	timeReg[RTC_DAY] = setDay(rtcTime.tm_wday);
	timeReg[RTC_DATE] = setDate(rtcTime.tm_mday);
	timeReg[RTC_MONTH] = setMonth(rtcTime.tm_mon);
	timeReg[RTC_YEAR] = setYear(rtcTime.tm_year);

	I2C_Start(RTC_SCL_BASE, RTC_SDA_BASE);
	if (!I2C_WriteToDeviceRegister(RTC_SCL_BASE, RTC_SDA_BASE, RTC_I2C_ADDR,
RTC_REG_BASE, timeReg, RTC_REG_LEN))
		return false;

	I2C_Start(RTC_SCL_BASE, RTC_SDA_BASE);
	if (!I2C_WriteToDeviceRegister(RTC_SCL_BASE, RTC_SDA_BASE, RTC_I2C_ADDR,
RTC_DST, (alt_u8*)&rtcTime.tm_isdst, 1))
		return false;
```

```
	I2C_Stop(RTC_SCL_BASE, RTC_SDA_BASE);

	return true;
}

bool getRtcToStruct(struct tm* rtcTime)
{
	alt_u8 timeReg[RTC_REG_LEN];

	I2C_Start(RTC_SCL_BASE, RTC_SDA_BASE);
	if (!I2C_ReadFromDeviceRegister(RTC_SCL_BASE, RTC_SDA_BASE, RTC_I2C_ADDR,
RTC_REG_BASE, timeReg, RTC_REG_LEN, true))
		return false;

	I2C_Start(RTC_SCL_BASE, RTC_SDA_BASE);
	if (!I2C_ReadFromDeviceRegister(RTC_SCL_BASE, RTC_SDA_BASE, RTC_I2C_ADDR,
RTC_DST, (alt_u8*)&rtcTime->tm_isdst, 1, true))
		return false;
	I2C_Stop(RTC_SCL_BASE, RTC_SDA_BASE);

	rtcTime->tm_sec = getSeconds(timeReg[RTC_SECONDS]);
	rtcTime->tm_min = getMinutes(timeReg[RTC_MINUTES]);
	rtcTime->tm_hour = getHours(timeReg[RTC_HOURS]);
	rtcTime->tm_wday = getDay(timeReg[RTC_DAY]);
	rtcTime->tm_mday = getDate(timeReg[RTC_DATE]);
	rtcTime->tm_mon = getMonth(timeReg[RTC_MONTH]);
	rtcTime->tm_year = getYear(timeReg[RTC_YEAR]);

	return true;
}
```

Pn532.c

The following sample code is a snippet from Group 5 2014's pn532.c code. It illustrates the use of the I2C_ReadMore() function. Note that I2C_Start() is **not** called between I2C_ReadMore() calls and I2C_Stop() is called on failures to end the I2C communication.

```
#include "i2c/I2C.h"

#define PN532_I2C_ADDR                          0x48
#define PN532_SCL_BASE                          I2C_NFC_SCL_BASE
#define PN532_SDA_BASE                          I2C_NFC_SDA_BASE

#define PN532_MAX_DATA_SIZE                     255
#define PN532_MAX_FRAME_SIZE                    262
#define PN532_FRAME_HEADER_SIZE                 5
#define PN532_FRAME_FOOTER_SIZE                 2
#define PN532_PREAMBLE                          0x00
#define PN532_STARTCODE1                        0x00
#define PN532_STARTCODE2                        0xFF
#define PN532_POSTAMBLE                         0x00
#define PN532_PN532TOHOST                       0xD5

bool readResponse()
{
	alt_u8 statusByte;
	alt_u8 dataLen;
	alt_u8 checksum;
	alt_u8 checksumRcvd;
```

```
	memset(frameBuffer, 0, PN532_MAX_FRAME_SIZE);
	memset(dataBuffer, 0, PN532_MAX_DATA_SIZE);

	I2C_Start(PN532_SCL_BASE, PN532_SDA_BASE);
	I2C_ReadFromDevice(PN532_SCL_BASE, PN532_SDA_BASE, PN532_I2C_ADDR, &statusByte,
1, false);

	if (statusByte != PN532_STATUS_READY)
		I2C_Stop(PN532_SCL_BASE, PN532_SDA_BASE);
		return false;
	}

	I2C_ReadMore(PN532_SCL_BASE, PN532_SDA_BASE, frameBuffer,
PN532_FRAME_HEADER_SIZE, false);

	if (frameBuffer[0] != PN532_PREAMBLE || frameBuffer[1] != PN532_STARTCODE1 ||
frameBuffer[2] != PN532_STARTCODE2)
	{
		printf("Invalid preamble\n");
		I2C_Stop(PN532_SCL_BASE, PN532_SDA_BASE);
		return false;
	}

	dataLen = frameBuffer[3];
	if (frameBuffer[4] != (alt_u8)~dataLen + 1)
	{
		printf("Invalid length check\n");
		I2C_Stop(PN532_SCL_BASE, PN532_SDA_BASE);
		return false;
	}

	I2C_ReadMore(PN532_SCL_BASE, PN532_SDA_BASE, frameBuffer +
PN532_FRAME_HEADER_SIZE, dataLen + PN532_FRAME_FOOTER_SIZE, true);
	I2C_Stop(PN532_SCL_BASE, PN532_SDA_BASE);

	if (frameBuffer[5] != PN532_PN532TOHOST || frameBuffer[6] != (lastCommand + 1))
	{
		printf("Invalid identifier\n");
		return false;
	}

	checksumRcvd = frameBuffer[PN532_FRAME_HEADER_SIZE + dataLen +
PN532_FRAME_FOOTER_SIZE - 2];
	checksum = PN532_PREAMBLE;
	checksum += PN532_STARTCODE1;
	checksum += PN532_STARTCODE2;
	checksum += PN532_PN532TOHOST;

	int i;
	for (i = 0; i < dataLen; i++)
	{
		checksum += frameBuffer[PN532_FRAME_HEADER_SIZE + i];
	}

	if (checksumRcvd != (alt_u8)~checksum)
	{
		printf("Invalid checksum\n");
		return false;
	}

	if (frameBuffer[PN532_FRAME_HEADER_SIZE + dataLen + PN532_FRAME_FOOTER_SIZE -
1] != PN532_POSTAMBLE)
	{
```

```
            printf("Invalid postamble\n");
            return false;
        }

        memcpy(dataBuffer, frameBuffer + PN532_FRAME_HEADER_SIZE, dataLen);
        return true;
}
```

References

[1] https://www.ualberta.ca/~delliott/local/ece492/appnotes/2013w/G6_I2C_Device_Integration/

[2] http://www.robot-electronics.co.uk/acatalog/I2C_Tutorial.html

[3] http://en.wikipedia.org/wiki/I%C2%B2C