1. I²C Software Implementation

1.1 Introduction

The implementation is to use SGPIO as master and PCA9673PW together with push buttons, joystick and LEDs as slave.PCA9673PW is a remote 16-bit I/O expander for I2C-bus. The desired result should be that LED0 is on when the user presses button SW3, and LED0 is off when the user releases button SW3. Similarly button SW5, SW6, JOY1 control the states (on or off) of LED1, LED2, LED3 respectively. In a result, SGPIO should continually read the state of buttons then send the desired LEDs state to slave. In this example, the LPC1850EVA-A4 evaluation board with the debugger is the only device used to implement the code. The main instrument to do the testing is a 4-chennel 500 MHz digital storage oscilloscope (DSO6054A) from Agilent Technologies. The block diagram of hardware construction in this demo is in Figure 1-1.

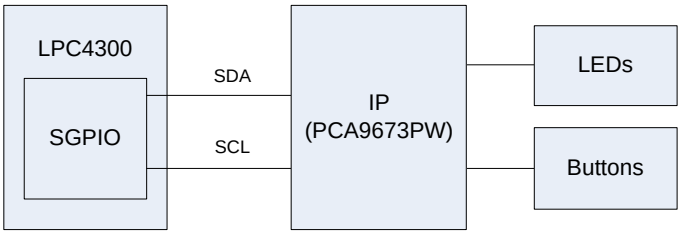

Figure 1-1: I2C hardware connection

At the initial state of project, the LPC1850EVA-A2 board with engineering version core is the device to use. Before software implementation, the work has been done to check whether the LPC1850EVA-A2 board could be working, which means if there is SGPIO feature in the board. It proves that this LPC1850EVA-A2 board meets the requirement.

1.2 I²C Protocol

Overview

I²C (Inter-Integrated Circuit) is a multi-master serial single-ended computer bus invented by Philips that is used to attach low-speed peripherals to a motherboard, embedded system, cell phone, or other electronic device. Here are some features of the I²C-bus:

- Two wires, serial data (SDA) and serial clock (SCL), carry information between the devices connected to the bus.
- Each device connected to the bus is software addressable by a unique address and simple master/slave relationships exist at all times. A master is the device which initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered a slave. Each device can work as either a transmitter or receiver.
- Serial, 8-bit oriented, bidirectional data transfers can be made at up to 100 kbit/s in the Standard-mode, up to 400 kbit/s in the Fast-mode, up to 1 Mbit/s in Fast-mode Plus, or up to 3.4 Mbit/s in the High-speed mode.

Data validity

The data on the SDA line must be stable during the HIGH period of the clock. The HIGH or LOW state of the data line can only change when the clock signal on the SCL line is LOW (see Figure 1-2). One clock pulse is generated for each data bit transferred.

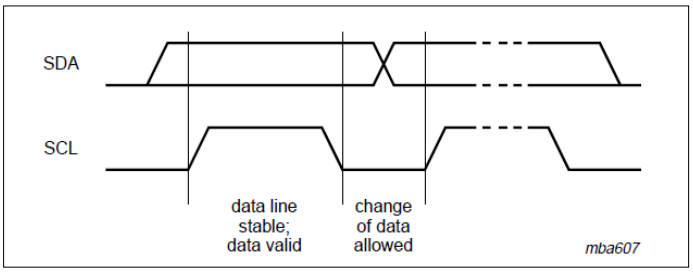

Figure 1-2: Data validity on the I2C bus

START and STOP conditions

All transactions begin with a START (S) and can be terminated by a STOP (P) (see Figure 1-3). A HIGH to LOW transition on the SDA line while SCL is HIGH defines a START condition. A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition.

START and STOP conditions are always generated by the master. The bus stays busy if a repeated START (Sr) is generated instead of a STOP condition. In this respect, the START (S) and repeated START (Sr) conditions are functionally identical.

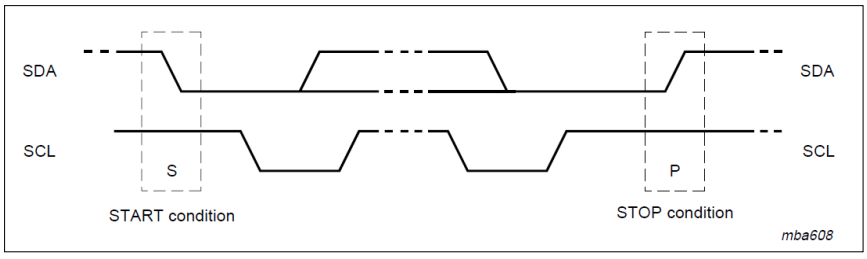

Figure 1-3: I2C Start and Stop condition

Byte format

Every byte put on the SDA line must be 8 bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte has to be followed by an Acknowledge bit. Data is transferred with the Most Significant Bit (MSB) first (see Figure 1-4). If a slave cannot receive or transmit another complete byte of data until it has performed some other function, for example servicing an internal interrupt, it can hold the clock line SCL LOW to force the master into a wait state. Data transfer then continues when the slave is ready for another byte of data and releases clock line SCL. Sr represents repeated Start.

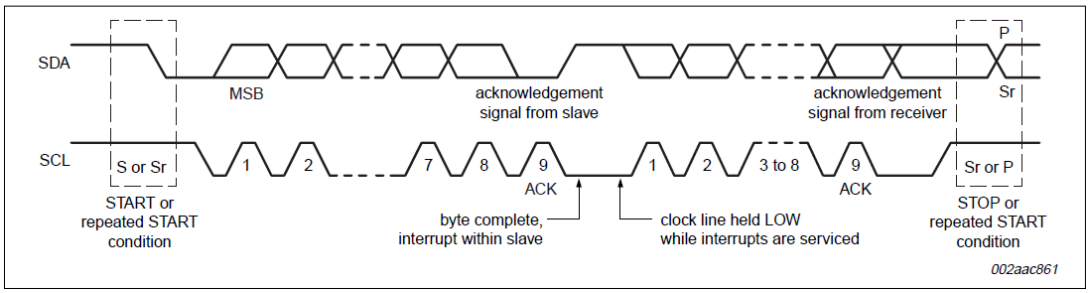

Figure 1-4: Data transfer on I2C bus

ACK and NACK

The acknowledge takes place after every byte. The acknowledge bit allows the receiver to signal the transmitter that the byte was successfully received and another byte may be sent. All clock pulses including the acknowledge 9th clock pulse are generated by the master.

The ACK (Acknowledge signal) is defined as follows: the transmitter releases the SDA line during the acknowledge clock pulse so the receiver can pull the SDA line LOW and it remains stable LOW during the HIGH period of this clock pulse. When SDA remains HIGH during this 9th clock pulse, this is defined as the NACK (Not Acknowledge signal).

Slave address and R/$\overline{W}$ bit
Data transfers follow the format shown in Figure 1-5. After the START condition (S), a slave address is sent. This address is 7 bits long followed by an eighth bit which is a data direction bit (R/$\overline{W}$)—a 'zero' indicates a transmission (WRITE), a 'one' indicates a request for data (READ) (refer to Figure 1-6). A data transfer is always terminated by a STOP condition (P) generated by the master. However, if a master still wishes to communicate on the bus, it can generate a repeated START condition (Sr) and address another slave without first generating a STOP condition. (11)

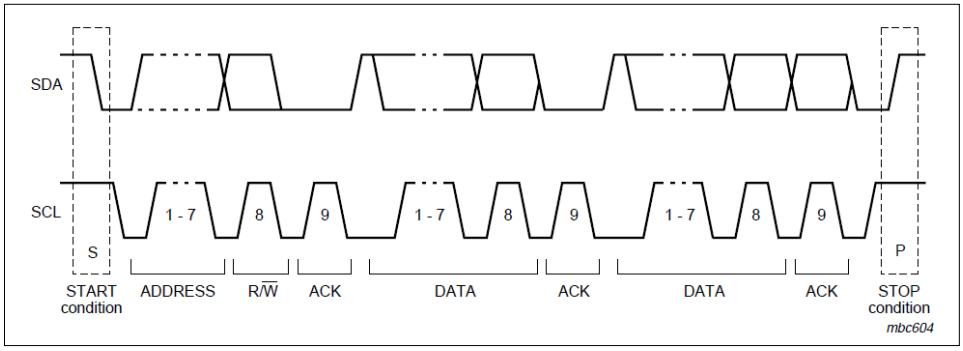

Figure 1-5: A complete I2C data transfer

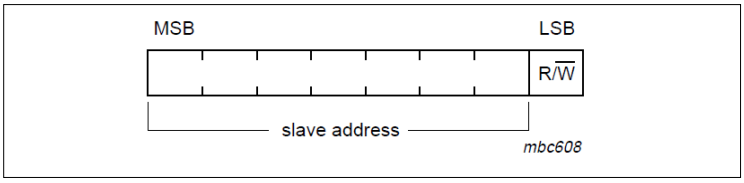

Figure 1-6: The first byte after the START procedure

1.3 I²C SGPIO Configuration

SGPIO is a hardware feature of LPC4300 series. There are 16 SGPIO pins called from SGPIO 0 to SGPIO 15. SGPIO is one of the functions of LPC 4300 pins which can be chosen. SGPIO could work as standard GPIO pins, or do stream processing. So it provides a number of possibilities for interface implementation.

Each pin of SGPIO has a configuration as shown in Figure 1-7 . The pin function has been chosen by SCU (System Control Unit) as SGPIO function. What left should be configured with SGPIO pin is set by bits of OUT_MUX_CFG Register. OUT_MUX_CFG consists of two parts, P_out_cfg and P_oe_cfg. P_out_cfg decides whether a slice or GPIO_REG generates the output signal (dout). P_oe_cfg selects whether slice or GPIO_OEREG to be output enable (doe). All the relevant SGPIO registers information is in Appendix B.

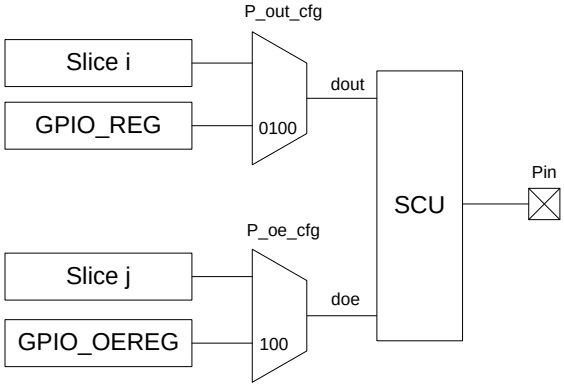

Figure 1-7: Pin configuration for SGPIO

Slice is an enhanced feature of SGPIO to accelerate serial streaming. The construction of one slice is in Figure 1-8. REG register is a shift register to generate output or get input for one slice. REG_SS is the slice data shadow register. POS register is a down counter. The first 8 bits (POS_COUNTER) of POS register indicates the current number of shifts left before the exchange between REG and REG_SS. The last 8 bits (POS_PRESET) of POS register sets the reload value of POS_COUNTER. REG and REG_SS will exchange the content when the down counter POS_COUNTER reaches 0x0. This feature can be used to change new output value of on slice. PRESET register sets the reload value of the counter, in other words, it controls the rate of shift clock. COUNT register reflects the slice clock counter value. When COUNT equals 0x0, POS_COUNTER counts down, and REG shifts once. All the details about setting of one slice are in section 5.4.

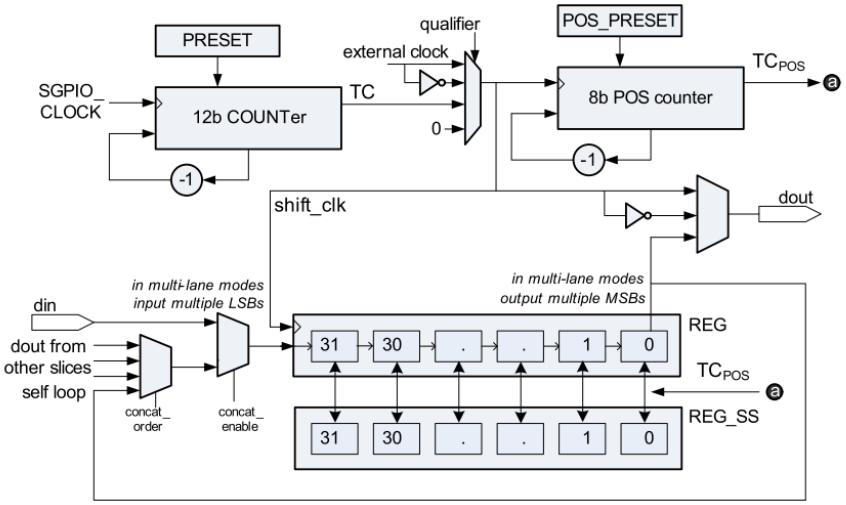

Figure 1-8: Slice basic operation and construction

To configure LPC4350 pins to be I²C signal pins, two SGPIO pins should be chosen first. In this demo, SGPIO 1 and SGPIO 13 are defined as SDA and SCL respectively. With looking up the table in section 6.2 (Pin description) of LPC4350 datasheet [6], the available pins to be SGPIO 1 and SGPIO 13 could be found. Possible pins for SGPIO 1 are P0_1, P9_1, and PF_2. Possible pins for SGPIO 13 are P1_20, P2_4, P4_8, PC_14, and PD_9.

Not all the pins in the list above are available to use on the real board. In order to choose the proper pins from them, the schematics of the evaluation board has been consulted. The requirements to define free pins on the board are as follows,

- It should have a jumper between the pin of LPC4350 and a certain chip or circuit.

- The signal of SDA or SCL will not affect the previous function of the pin.

According to these requirements, the choices left for SGPIO 1 are P0_1, P9_1, and PF_2. For SGPIO 13, the pins meet the requirements are P1_20, P2_4, P4_8. Finally, P9_1 and P1_20 are picked. The overall information of selected LPC4350 pins is in Table 1-1.

Table 1-1: I2C function and SGPIO number of selected LPC4350 pins

SGPIO i	LPC4350 Pin number	I2C signal
1	P9_1	SDA
13	P1_20	SCL

The System Control Unit (SCU)/ IO configuration should be set for each pin. The pin configuration registers bit description is in page 227 of LPC4350 user manual, which is also in Appendix C Table C-1. What should be set in pin configuration register are pin function, pull-up enable, pull-down enable, slew rate, input buffer enable and input glitch filter. As for pin function, both of P9_1 and P1_20 work in SGPIO mode numbered as function 6. For I^2C Standard-mode, the signal rates are below 30 MHz, so the pins should be set as slow rate and input glitch filter enable. The pins are set as pull-up enable due to the construction of I^2C masters and slaves (see next paragraph). Input buffer is enabled to make SGPIO streaming possible.

As to create a suitable SGPIO configuration for I^2C protocol, first should look into the construction both master side and slave side (see Figure 1-10). The SDA line is originally high with the pull up, so SGPIO could not always drive it high. An example can explain this in Figure 1-9. If master 2 is always driving SDA line, assume the voltage on master 2 is 5 V and master 1 now wants to transfer data and drive SDA line low. In this situation, the final voltage on SDA line is 2.5V. In conclusion, what should be done is to drive SDA low when there's a need. The condition is exactly the same for SCL line.

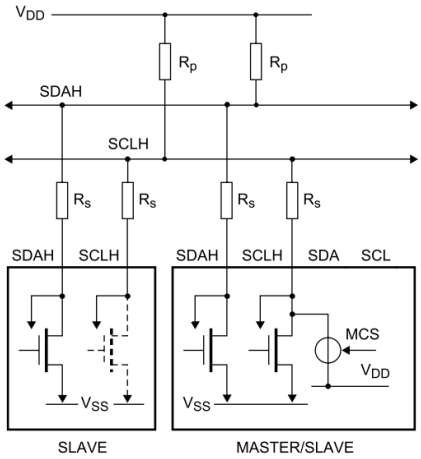

Figure 1-10: Internal circuit in Slave and Master

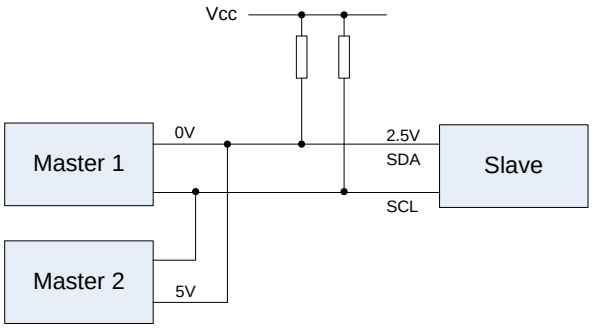

Figure 1-9: always driving condition

The setting of SGPIO pin configuration is in Figure 1-11. For SGPIO 1, the SDA line, GPIO mode is to generate output, and output value is set by GPIO_OUTREG register. Slice M is output enable as a driver. Slice M is in a self-loop. Slice I gets the input from SGPIO 1 during I^2C data receiving. The way to choose Slice I as input is according to Table 1-5 in section 5.4.

In terms of SGPIO 13, GPIO mode is to generate output. Slice K is self-looped and set to be output enable. Slice D is the clock source for Slice I, M, and K. To find the way to select clock source for each slice, please refer to Table **1-5** in section 5.4.The selection of slices that are related to output mode is according to the Table 1-2 and Table 1-3. The blue blocks represent slices mapping of SGPIO 1. The green blocks indicate slices mapping of SGPIO 13.

The output mode and output enable are set by the register called OUT_MUX_CFG as shown in Table **1-4**.

Table 1-2: OUT_MUX_CFG register (P_out_cfg)

pin nr.	Output mode - OUT_MUX_CFG P_out_cfg											
	1011	1010	1001	0111	0110	0101	0011	0010	0001	0000	1000	0100
	8b	8b	8b	4b	4b	4b	2b	2b	2b	1b	clk	gpio
0	L0	J0	A0	J0	I0	A0	J0	I0	A0	A0	Bck	0
1	L1	J1	A1	J1	I1	A1	J1	I1	A1	I0	Dck	1
2	L2	J2	A2	J2	I2	A2	I0	J0	E0	E0	Eck	2
3	L3	J3	A3	J3	I3	A3	I1	J1	E1	J0	Hck	3
4	L4	J4	A4	L0	K0	C0	L0	K0	C0	C0	Cck	4
5	L5	J5	A5	L1	K1	C1	L1	K1	C1	K0	Fck	5
6	L6	J6	A6	L2	K2	C2	K0	L0	F0	F0	Ock	6
7	L7	J7	A7	L3	K3	C3	K1	L1	F1	L0	Pck	7
8	N0	M0	B0	N0	M0	B0	N0	M0	B0	B0	Ack	8
9	N1	M1	B1	N1	M1	B1	N1	M1	B1	M0	Mck	9
10	N2	M2	B2	N2	M2	B2	M0	N0	G0	G0	Gck	10
11	N3	M3	B3	N3	M3	B3	M1	N1	G1	N0	Nck	11
12	N4	M4	B4	P0	O0	D0	P0	O0	D0	D0	Ick	12
13	N5	M5	B5	P1	O1	D1	P1	O1	D1	O0	Jck	13
14	N6	M6	B6	P2	O2	D2	O0	P0	H0	H0	Kck	14
15	N7	M7	B7	P3	O3	D3	O1	P1	H1	P0	Lck	15

Table 1-3: OUT_MUX_CFG register (P_oe_cfg)

pin nr.	OE control - OUT_MUX_CFG P_oe_cfg				
	111	110	101	100	000
	8b	4b	2b	1b	gpio
0	H0	H0	H0	B0	0
1	H1	H1	H1	M0	1
2	H1	H1	D0	G0	2
3	H1	H1	D1	N0	3
4	H1	O0	G0	D0	4
5	H1	O1	G1	O0	5
6	H1	O1	O0	H0	6
7	H1	O1	O1	P0	7
8	P0	P0	P0	A0	8
9	P1	P1	P1	I0	9
10	P1	P1	B0	E0	10
11	P1	P1	B1	J0	11
12	P1	N0	N0	C0	12
13	P1	N1	N1	K0	13
14	P1	N1	M0	F0	14
15	P1	N1	M1	L0	15

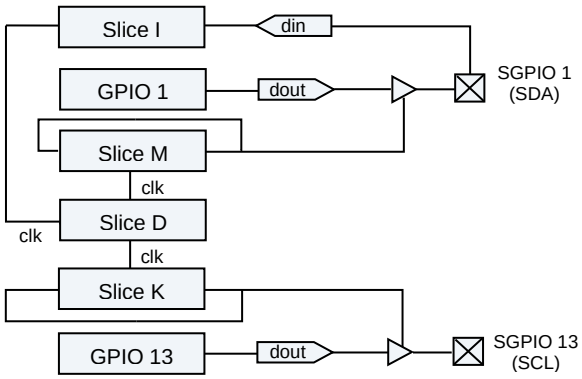

Figure 1-11: SGPIO output pin mapped to slices for I2C

Table 1-4: SGPIO output pin configuration registers setting for I2C

OUT_MUX_CFGi	**SGPIO 1 (i = 1)**	**SGPIO 13 (i = 13)**
P_out _cfg	0x4: gpio_out	0x4: gpio_out
P_oe_cfg	0x4: dout_oem1	0x4: dout_oem1

In this demo, the value of bit 1 in GPIO_OUTREG is set to be always 0 in I²C implementation. So, during I²C data sending, the output of SGPIO 1 is 1 when slice M generates 0. The output of SGPIO 1 is 0 when slice M generates 1. The method above is applied to SGPIO 13 as well. In this way, SGPIO could work well with I²C slave/master hardware construction (refer to Figure 1-10).

1.4 I²C Slice Configuration

SGPIO 1, the SDA signal pin, needs to get the state of buttons by receiving data. So, one slice should be chosen to get the external data input from SGPIO 1. According to Table 1-5 (see blue blocks), it is Slice I that is mapped as input to SGPIO 1. As to get external data pin input for Slice I, set 0 in concat_enable bit in SGPIO_MUX_CFG register (refer to Table 1-6).

Table 1-5: Slices selection for Input or clock source (SGPIO_MUX_CFG register)

Slice	Slice Din				
	concat_enable				
	0	1			
		concat_order			
		00	01	10	11
A	Pin0	A	I	J	L
I	Pin1	I	A	A	A
E	Pin2	E	J	I	I
J	Pin3	J	E	E	E
C	Pin4	C	K	L	J
K	Pin5	K	C	C	C
F	Pin6	F	L	K	K
L	Pin7	L	F	F	F
B	Pin8	B	M	N	P
M	Pin9	M	B	B	B
G	Pin10	G	N	M	M
N	Pin11	N	G	G	G
D	Pin12	D	O	P	N
O	Pin13	O	D	D	D
H	Pin14	H	P	O	O
P	Pin15	P	H	H	H

slice	Clock Slice Din			
	clk_source_ slice_mode			
	00	01	10	11
A	D	H	O	P
I	D	H	O	P
E	D	H	O	P
J	D	H	O	P
C	D	H	O	P
K	D	H	O	P
F	D	H	O	P
L	D	H	O	P
B	D	H	O	P
M	D	H	O	P
G	D	H	O	P
N	D	H	O	P
D	-	-	-	-
O	-	-	-	-
H	-	-	-	-
P	-	-	-	-

Another important part to configure slices is the shift rate setting. In order to keep all the slices work at the same shift rate, 4 registers (SGPIO_MUX_CFG, PRESET, COUNT, and POS) should be set:

- SGPIO_MUX_CFG: slice I, M, K are set to use slice D as their clock source. (refer to orange blocks in Table **1-5**)
- PRESET: controls the shift clock frequency by formula below. So set same value to slice I, M, D, K.

  $$frequency_{shift\_clock} = frequency_{SGPIO\_CLOCK} / (PRESET+1)$$
- COUNT: controls the phase of shift clock. Set 0 to slice I, M, D, K in this situation.
- POS: contains POS_COUNTER and POS_PRESET. In this demo, slice exchange content between REG and REG_SS every 32 bits. So values in POS_COUNTER and POS_PRESET should be 32 - 1 = 0x1F.

Table 1-6: Slices configuration registers setting

SGPIO_MUX_CFGi	Slice I (i = 8)	Slice M (i = 12)	Slice D (i = 3)	Slice K(i = 10)
ext_clk_enable	0: internal clock signal	0: internal clock signal	x	0: internal clock signal
clk_source_slice_mode	00: Slice D	00: Slice D	x	00: Slice D
qualifier_mode	00: enable	00: enable	x	00: enable
concat_enable	0: external data pin	x[1]	x	1: concatenated data
concat_order	x[1]	x[1]	x	00: self loop

[1] set to be 0 in demo

Table 1-7: Frequency-related registers setting of slices

	Slice I (i = 8)	Slice M (i = 12)	Slice D (i = 3)	Slice K(i = 10)
PRESETi	0xFF	0xFF	0xFF	0xFF
COUNTi	0	0	0	0
POSi	0x1F1F	0x1F1F	x	0x1F1F

1.5 Board Connection

To turn on the board, the USB interface is used to provide power. The USB cable connects from the evaluation board to PC. A JTAG connector is used to connect the board and PC via J-link debugger.

As for connection from SGPIO to slave (PCA9673PW), it needs two wires, SDA and SCL. The wires are connected between SGPIO pins and I^2C signal jumpers on the board. The board position of selected SGPIO 1 pin is SV6 Pin7. Since SGPIO 1 is the output pin for SDA signal, it should be connected into SDA line which has been already on the board. There is a jumper between SDA line, and the position is SV10 Pin 1. Therefore, one wire is needed between SV6 Pin7 and SV10 Pin 1. Similarly, there should be one wire between SV3 Pin9 and SV10 Pin 3 for SCL. Details of I^2C signal connections are in Table **1**-**8**. All the connections on board are shown in Figure 1-12.

Table 1-8: I2C signal connection on the evaluation board

SGPIO i	I^2C signal	LPC4350 Pin number	Board pin position	I^2C board pin position
1	SDA	P9_1	SV6 Pin7	SV10 Pin 1
13	SCL	P1_20	SV3 Pin9	SV10 Pin 3

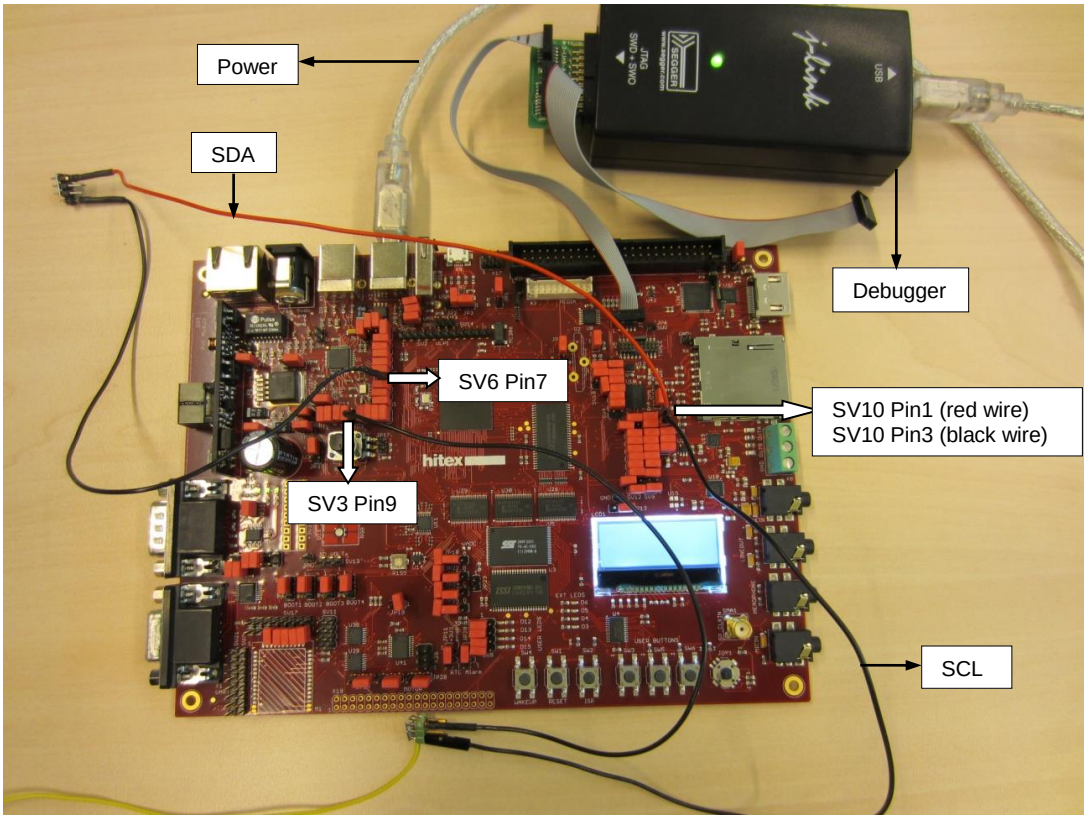

Figure 1-12: Board connection for I2C example

1.6 I²C Programming

1.6.1 Demo Level

The desired result of this demo is that LED is on when the user presses button. The demo is designed to be in a while loop, continually check the button state and compare it with last read value, transfer it into LED state if the button state changes. It's the responsibility of SGPIO to read the states of touch buttons and transfer corresponding LED states. Below is the flow chart on demo level.

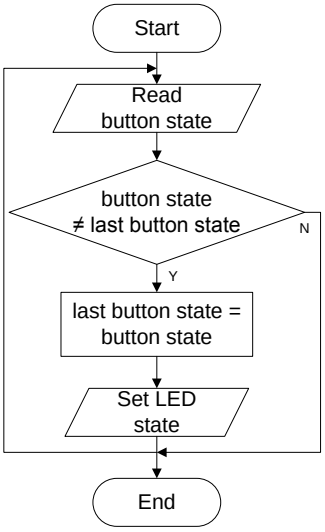

Figure 1-13: Flow chart of I2C demo-level programming

1.6.2 SGPIO level

A basic I2C write transfer consists of a START condition, followed by 7 address bits, a one to indicate that it's a write, an ACK (send by the slave), 8 data bits and another ACK, finalized with a stop condition. The transfer is not an upfront fixed sequence as it depends on two conditions. One is that what the acknowledge responses from the slave is. The other is how many bytes of data would be sent. Due to this, it makes sense to split the pattern to make it possible to respond to the feedback from the slave. It's also easier to perform a read transfer by separating protocol format into parts. The I2C protocol is such that it is possible to stall the transfer.

The SGPIO is using 32 bit registers. So the implementation is simpler if the transfer is split into parts which fit in 32 bit words. This approach results in 8 parts: Start, Send Data, Receive ACK/NACK, Repeated Start, Receive Data, Send ACK, Send NACK, and Stop. These eight parts could group into several different combinations. Three typical combinations are as follows:

1. Write to slave: Start, Send slave address ($\overline{\mathbf{W}}$), Receive ACK, Send data, Send NACK, Stop
2. Read from slave: Start, Send slave address (R), Receive ACK, Receive data, Send NACK, Stop
3. Write and read: Start, Send slave address ($\overline{\mathbf{W}}$), Receive ACK, Send data, Receive ACK, Repeated Start, Send slave address (R), Receive ACK, Receive Data, Send NACK, Stop (see Figure 1-14).

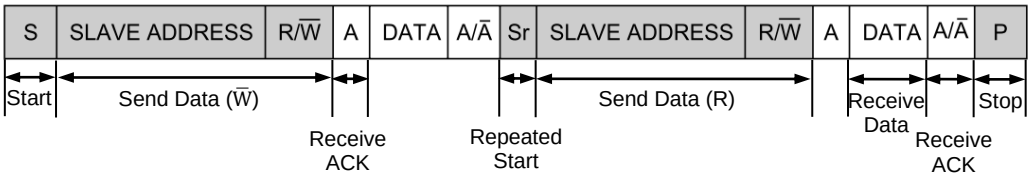

Figure 1-14: One I2C data transmission consists of read and write transfer

The data transmission is programmed to be a function named I2C_TransferData. Eight parts mentioned above are programmed as eight functions and could be called in function I2C_TransferData.

The aim of function I2C_TransferData is to combine all possibilities of transfer formats. It can choose whether to read or write and when. Moreover, the function can react to acknowledge signal sent by slave. It's also possible to judge the validity of data sent by slave and response to it.

The input of I2C_TransferData function is a structure which contains slave address, data length (means how many bytes of data) to send or receive, value of data to send or receive, counter of transferred bytes. The data received is transferred back by this structure as well. The output of the function is state of transmission, error or success.

Below are the flow charts of the I2C_TransferData function. The gray block (Receive) in the left chart points to the gray block (Start Receive) in the right chart. The Stop Receive block in the chart on the right points back to end of chart on the left. The function has Start and Stop at the beginning and the end respectively, and in between is data streaming. Followed by START condition, it's the decision whether to send data or not. If sending data is needed, it starts to send 7-bit slave address with 8th bit low ($\overline{\mathbf{W}}$), and then send data. Followed by every byte sent, there are acknowledge receiving and judging if all the data has been sent. The transmission could stop and send error information if the received acknowledge bit is high (NACK).

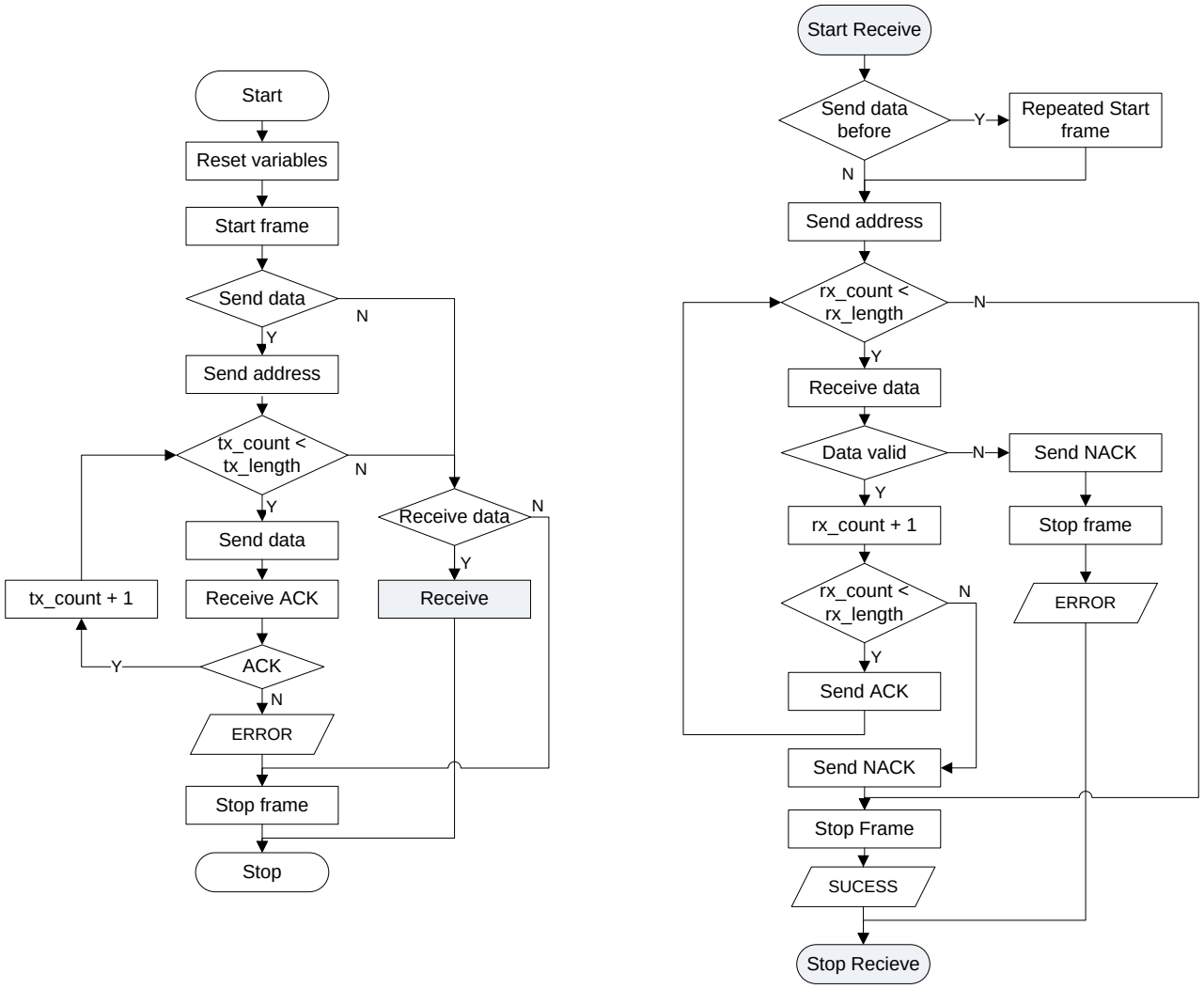

Figure 1-15: Flow chart for SGPIO-level I2C programming

If there's no need to send any data, or all data bytes have been sent successfully, the code will go to next decision. This decision is to decide whether to read or not. If reading is required, the routine will go to the flow chart on the right, and decide whether to send a Repeated START condition or not. If the transmission has sent data before receiving, a Repeated START condition is needed. Otherwise, the data transfer directly goes to address sending part. Next, it sends 7-bit slave address with 8th bit high (R), receives 8-bit data, and checks the data validity to decide to send ACK or NACK. If data is not stable on SDA when SCL is high, the transmission would generate a NACK and STOP condition. Error information will be returned as well. After receiving every data byte and sending ACK, the code will check if it reaches the amount of byte desired to receive.

If the decision is no data to receive, or all data bytes have been received successfully, the transmission will end with a STOP condition. An important point to notice is that the last acknowledge bit sent before STOP condition should be NACK. Otherwise, the slave would not recognize it's the STOP condition to come.

Due to the 32-bit shift register and 8-bit data format, extending 8-bit data into 32 bits is used in Send Data part, and shortening 32 bits into 8 bits is used in Receive Data.

In this project, there are two boards used. At the beginning of the project, an old board was used, which contains the touch buttons, LED, and PCA9502 as the slave. The new board that was delivered later has push buttons, LED, joystick and PCA9673PW. PCA9502 is an 8-bit I/O expander with I2C-bus or SPI interface. It

requires writing a register address of PCA9502 before read or write data. Therefore, some sequences with using the old board contain read and write transfer, and appear like what is in Figure 1-14. Since the working way of the new chip in the new board is different, data transmissions is either a read transfer or a write transfer. So, not all the eight parts are actually working in the latest demo.

The structure of six functions above is very similar. The basic idea is to change the values in REGi (i represents the slice number) and REG_SSi of output enable slices to generate different patterns. Changing the values at the right time is the main difficulty. For example, if a new value is written into REGi without any condition, the former pattern is changed to new one before it totally shifts out. In a result, the pattern generated in this way is mixed with the previous value and the new value. So a feature of SGPIO is used in this case. The shift register REGi can shift out 32 bits and stop after exchanging with REG_SSi. In other words, POSi could countdown once in one cycle of 32 bits if it is desired. Detail steps are included in the example with Slice A (0) below.

CTRL_ENABLED = 0	(clear enable of all slices)
CTRL_DISABLED = 0	(clear disable of all slices)
REG[0] = 0xFFFF0000	(give a initial value of output)
CTRL_ENABLED = 1	(enable slice A which is now shifting sixteen 0s and sixteen 1s)
CTRL_DISABLED = 1	(slice A will shift one more old pattern and stop)
Wait	(until POS_COUNTER reloads its value)
REG[0] = 0xFFFFFFFF	(change a new value to output)
CTRL_DISABLED = 0;	(slice A generates all high continually as new pattern)

In terms of right time to change value, the delay part above is made of two while loops in the demo, here is one example for slice K:

```
while (LPC_SGPIO->POS[10]==0x1F1F);
while (LPC_SGPIO->POS[10]!=0x1F1F);
```

The second while-loop is to wait until POS reloads its value. The reason to have the first while-loop is that, in the program, the delay sections (while loop) are often used. When POS equals 0x1F1F and it costs some time counting down to 0x1F1E, 0X1F1D and so on. The code would run very fast and skip some while loops if there's only the second while. So the first while is to avoid skipping steps in transmission. Another important thing is that value in while-loop condition should be 0x1F1F rather than 0x0. It's because the shifting disabled function is designed to pause the sequence at the moment POS has reloaded the new value.

The flow chart of Start function is in Figure 1-16. For Send Data, Receive ACK/NACK, Repeated Start, Receive Data, Send ACK, Send NACK, and Stop, only one thing should be changed. That is the desired value to be written in REG and REG_SS of slice K and slice M.

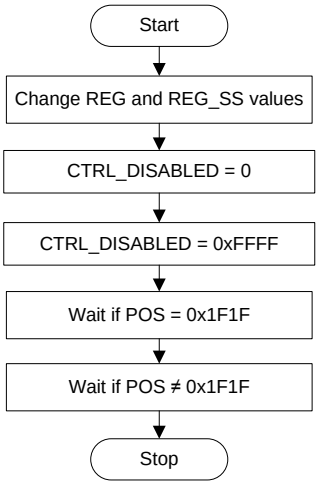

Figure 1-16: Start function flow chart

1.7 Testing

Test Setup

At the beginning of implementation, the first part finished is generating a clock signal. At that time, a LED on the board was used in testing to see whether it could be on, off, or flashing. With the code developing into more complex state, a scope is to test the sequence. The setup for testing is as shown in Figure 1-17. The whole setup consists of PC, a LPC1850EVA-A4 evaluation board, J-link debugger digital and a storage oscilloscope. Use the edge trigged mode of scope so that the sequence could be immediately captured on the screen by the rising or falling edge.

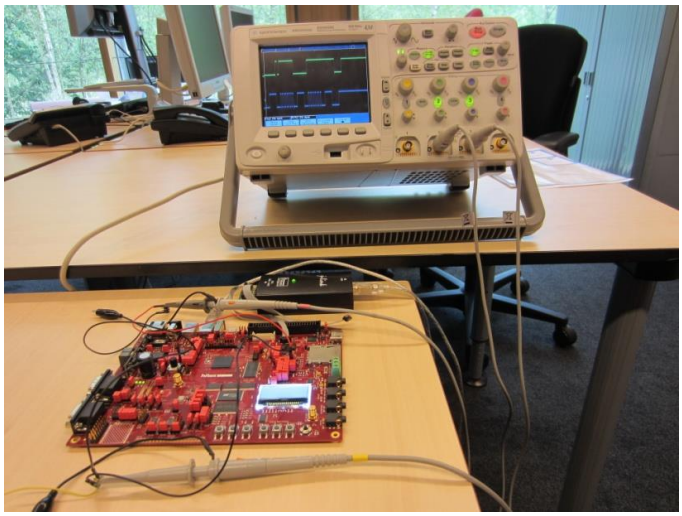

Figure 1-17: Testing Setup

Typical Mistakes and Tips

1. Testing equipment
   - Before testing, it's better to set the oscilloscope back to default setting. It is to prevent the setting of the scale or delay between channels to give an illusion of testing result. One example is that an unexpected phase appears because there is delay between two channels on scope.
2. Documentation
   - To do SCU setting for pins, Table 3 in section 6.2 (Pin description) of LPC4350 datasheet [6] is referred. Reserved functions should be counted when deciding SGPIO function number.
   - When looking up document about SGPIO, pin number is corresponding to SGPIO number rather than Slice number.
3. SGPIO programming
   - POS register should be set to be 0x1F1F instead of 0x1F.
   - SCL and SDA lines should not be always driving. (explained in section 5.3)
   - To change the value in REGi should be at the right point. (explained in section 5.6.2)
4. I2C programming
   - When it's master's responsibility to send A/$\bar{A}$and before stop, the acknowledge bit should be NACK instead of ACK.

Testing result

SGPIO clock frequency is 12 MHz. The fastest rate of the final demo is 11.76 kbit/s. As shown in Figure 1-18, the shortest time to transfer one bit is 85 µs which is measure by scope. So the highest transfer rate of I²C is

$$f_{max} = \frac{1}{85 \times 10^{-6}} = 11764.7\ bit/s = \ 11.76\ kbit/s$$

This transfer speed comes from the implementation, it's not a limitation of i2C itself. One typical sequence is in Figure 1-18 which is saved from scope. It comprises one complete transfer, the line above (green line) is SCL signal, and below (purple line) is SDA signal.

As shown in the figure, it's a complete read transfer. The master first generates START condition, sends slave address with 8th bit high, and then keep SDA high on 9th clock to wait for the slave to send the acknowledge bit. At this time the slave keeps driving SDA line low during 9th clock, it means the slave has received the address. The master receives data in the next 8 clocks and then sends an ACK signal. During next 8 clocks, the master receives another byte of data. Then it's the master's responsibility to send NACK to stop. The master keeps SDA high to inform the slave to stop. The whole transmission is terminated by a STOP condition.

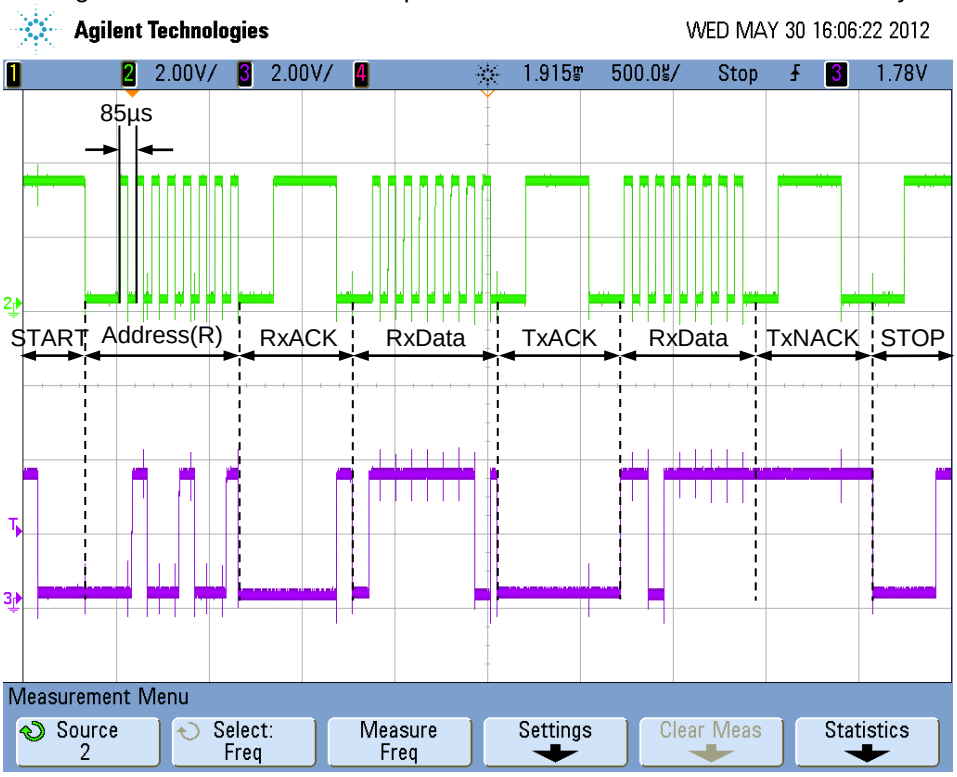

Figure 1-18: One complete read transfer of I²C on scope

1.8 Conclusion

I²C protocol could be implemented into SGPIO interface by software programming. The demo proves it's possible for SGPIO to work in rule of communication protocols. Error detecting is also realizable. There are still some problems left to be improved with I²C protocol:

1. The output of function I2C_TransferData is ERROR or SUCCESS. It can be detected if there is any error during executing I2C_TransferData. However, two conditions could lead to error state. One is receiving NACK from slave when it sends address or data. The other is the data received from slave couldn't meet data validity. When the routine get an error state of I2C_TransferData, it couldn't define which part of the transfer generates error. So it's better to have overall information about where and what the error is.
2. The wait part in Figure 1-16 is made of a while loop. To wait until the exact moment, It's better to have a smarter and safer solution instead of while loop.
3. Now the code is working on ARM M4 core. The desired core to deal with peripherals is ARM M0.
4. Between M4/M0 core and SGPIO, there is a bridge. The data stream should get over the bridge to reach SGPIO. So it might cause latency when the sequence is complex and long. A test of long data transmission is recommended to see the performance.
5. I²C protocol contains Standard-mode, Fast-mode, Fast-mode Plus, and the High-speed mode. The protocol achieved now is under Standard-mode. More work is left to realize other modes.