To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

H8/300L

Usage of Port to Implement I2C (I2Cport)

Introduction

This application note describes the necessary information and usage specification when implementing I^2C with I/O ports of SLP series.

The I^2C-bus supports any IC fabrication process (NMOS, CMOS, bipolar). Two wires, serial data (SDA) and serial clock (SCL), carry information between the devices connected to the bus. All I^2C-bus compatible devices incorporate an on-chip interface, which allows them to communicate directly with each other via the I^2C-bus.

While SLP series do not possess dedicated on-chip I^2C support hardware, this serial bus can be simulated by software control of it's two I/O pins.

*Please refer to Application Note on Interfacing with EEPROM with Emulating I^2C for an example of implementation of SLP series I/O ports as the I^2C bus ports.

Target Device

SLP – H8/38024

Contents

1. I²C General Characteristics

*Please refer to Application Note on SPI and I^2C for I^2C General view.

2. Hardware Configuration

It is important to make sure that any devices on an I^2C bus have the capability to pull the bus signals low. When the slave device wants to acknowledge the master polling, it must pull the SDA signal low. When a slow device cannot keep pace with a faster device, it will hold the SCL signal low to slow down the speed of communication.

When more than one device attempts to become master of the bus, the one that holds SDA low longer wins the arbitration. For this reason, it is critical that each I^2C device releases the bus after it finishes its activity and allows other devices to pull the bus signal low as needed. To implement this feature, I/O ports of the SLP series are configured as open-collector drivers.

To drive the signal line low, the direction register of the I/O port is set to be output and the data register is set to zero. To drive the signal line high, the data register is set to one, which prevents other devices from pulling the signal line low. The direction register of that I/O port is set to zero so that the I/O port is configured as an input. Then, this port will show logic one because of the external pull-up resistor. This approach is implemented in routines: SciOut() and SdlOut(), shown in below:

```
/* Both SDA_DATA_REG and SCL_DATA_REG are the data register for each respective I/O
port. It may change to map to other ports. The setting example as following:
        #define SDA_DATA_REG            P_IO.PDR7.BYTE
        #define SCL_DATA_REG            P_IO.PDR8.BYTE
SCL_IO_REG and SDA_IO_REG are the control register for each respective I/O port. The
setting example as following and is changeable by user.
        #define SDA_IO_REG              P_IO.PCR7.BYTE
        #define SCL_IO_REG              P_IO.PCR8.BYTE
*/
/* Drive SCL bus */
void SclOut (unsigned char status)
{   if (status == LOW)
    {    SCL_DATA_REG = 0;                //Drive Port LOW
         SCL_IO_REG |= SCL_IO_SET_BIT;    //Port is output
    }
    else
    { SCL_DATA_REG = 1;                   //Port is Input & using external
                                          //pull-up resistor to go high
      SCL_IO_REG |= SCL_IO_SET_BIT;       //Port is output
    }
}
/* Drive SDA bus */
void SdaOut (unsigned char status)
{   if (status == LOW)
    {    SDA_DATA_REG = 0;                //Drive Port LOW
         SDA_IO_REG |= SDA_IO_SET_BIT;    //Port is output
    }
    else
    {    SDA_DATA_REG = 1;             //Port is Input & using external pull-up
                                       //resistor to go high
         SDA_IO_REG |= SDA_IO_SET_BIT;       //Port is output
    }
}
```

3. Timing

Another consideration is the time. The timing for the I^2C clock does not have to be very accurate. When a slow device is connected to the bus, it will slow down a higher-speed device. There are many ways to set up a timer for the clock. NOP commands or a delay `for` loop can be used to create a delay in software. The built-in timer in the MCU can also be employed. The timer interrupt can be used to flip the clock output.

However, when setting the clock, you should pay attention to some of the important parameters, which will affect the I^2C protocol. Parameters such as rise time, fall time, hold time and setup time for all the conditions such as START, STOP etc. must be appropriately set according to the specification of the device being used (min and max time). Thus, it is important for user to check the AC characteristics for the device to ensure the result of I^2C implementation.

I^2C does not specify any minimum bus frequency but provides two modes of operation, the STANDARD MODE up to 100 KHz and the FAST-MODE up to 400 KHz. Thus, port implementation is possible.

I^2C also does not specify any timeout limit. Thus, user may need to specify the timeout limit for transmitting or receiving data to improve the efficiency of the protocol.

4. Bit Transfer

Due to the variety of different technology devices (CMOS, NMOS, bipolar) which can be connected to the I^2C-bus, the levels of the logical ‘0’ (LOW) and ‘1’ (HIGH) are not fixed and depend on the associated level of V_{cc}. One clock pulse is generated for each data bit transferred.

One of the commonly used logic when sending the bits is ‘0’(LOW) and ‘1’(HIGH) as the following example:

```
void SendBit (unsigned char data_byte)
{
        if (data_byte != 0)
   {
        SdaOut(HIGH);
    }
    else
    {
        SdaOut(LOW);
    }
}
```

The number of data bytes transferred between the START and STOP condition from transmitter to receiver is not limited. Each byte, which must be eight bits long, is transferred serially with the most significant bit first, and is followed by an acknowledge bit.

5. START and STOP conditions

Within the procedure of the I^2C-bus, unique situations arise which are defined as START (S) and STOP (P) conditions. One of the major concerns when generating these conditions is the timing. As mentioned earlier, attention should be given, particularly on the hold time and set up time.

As a transmitter, the device must provide an internal minimum delay time of 300ns to bridge the undefined region of falling edge of SCL to avoid unintended generation of START and STOP conditions.

Generally, the START and STOP conditions with delay (device and clock dependent) as below:

```
void SendStartBit(void)
{
      SclOut(HIGH);    //SCL && SDA must be HIGH to indicate bus-free.
      SdaOut(HIGH);
      SdaOut(LOW);

      Delay();
      SclOut(LOW);
      Delay();
}

void SendStopBit(void)
{
    SdaOut(LOW);
    Delay();
    SclOut(HIGH);
    Delay2x();
    SdaOut(HIGH);
}
```

6. Errors

Any transfer can be aborted by either the slave or the master. The master can issue a STOP condition and the slave can withhold acknowledgement after any byte to terminate the transfer. If the device detects an error, it will withhold acknowledgement. Withholding acknowledgement is required for the last byte in a read operation under the I^2C specification

A device may decide to generate an error indication for one or more of the following reasons:

- Device is not ready to process the request for data (either read or write)
- Device does not recognize the command code or function requested
- Device does not permit the command code or function requested
- Overflow or underflow condition
- Incorrect size of data in a block read/write transfer
- Unrecognized or unsupported data transfer protocol used in transaction
- Any other known or unknown error condition

7. Software Routines

There are only two routines needed to implement an I2C depending on the on the functionality. There are i2cRead(), or i2cWrite() or both i2cRead() and i2cWrite(). Obviously, an LCD driver is only a receiver, while a memory or I/O chip can be both transmitter and receiver.

i2cWrite() is called with:-

i. the slave address to be written,

ii. the start address of the data buffer where the data to be written is located, and

iii. the number of data bytes to write.

i2cRead() sends out :-

i) the address of the slave, and

ii) the number of data bytes the master is requesting to read.

If this operation is successful, it will send an acknowledgment upon receiving or writing and then proceed to the following operation.

*Please refer to Application Note on Interfacing with EEPROM with emulating I^2C for example of various types of read and write operation available.

Reference

1. *The I^2C-Bus Specification (Version 2.1)*, January 2000, Philips Semiconductor
2. *H8/38024 Series,H8/38024F-ZTZT Hardware Manual(version 2.0)*, 20 Feb 2002, Hitachi Ltd.
3. Leonard Haile, *Hitachi H8/3437 Series Microcontroller I2C Peripheral- A Practical SMBus/I2C Firmware Design Guide (Revision 1.2)*, 12 June 1998, Hitachi Semiconductor (America) Inc.
4. http://www.esacademy.com/faq/i2c/

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Sep.03	—	First edition issued