AN926: Reading and Writing Registers with SPI and I²C

The applicable timing devices use either SPI or I²C as the communication protocol for changing internal register settings in the chip. This application note clarifies how to access the registers with examples using these two protocols.

While a microcontroller is often used to read/write registers on the device, an SPI or I²C bus analyzer can also be used. Companies, such as Corelis (www.corelis.com) or Total Phase/Aardvark™ (http://www.totalphase.com), sell both I²C and SPI bus analyzers that connect through USB to the PC and have GUI interfaces. The user simply connects the wires (GND, CS, MISO, MOSI, and SCLK pins for SPI or SDA, SCLK, and GND pins for I²C) of the hardware (slave). The bus analyzer handles low-level control of the bus. The commands and data are sent through a GUI on the PC. This is often a quick way to evaluate a chip requiring one of these serial interfaces when the driver code is not easily available. It is always important to make sure the hardware is connected and correctly configured for proper operation.

The examples provided in this application note can be followed regardless of the hardware chosen to connect to the applicable timing device.

KEY POINTS

- SPI and I²C communication available
- SPI command protocol provided
- Step-by-step code examples provided for SPI and I²C

APPLIED TIMING DEVICES

- Si534x
- Si538x
- Si539x

1. Introduction

In the applicable timing devices for this application note, the registers are only 8-bit addressable. The register map addresses are 2 bytes (16-bits) wide, but the upper byte is ALWAYS the PAGE. To change the upper byte of the address, the PAGE register must be changed. The PAGE is always located at the lower byte address, 0x01. It is repeated regardless of the upper byte value. Therefore, address 0001 = Page; 0101 = Page; 0201 = Page, etc.

It is recommended to write a function to handle the paging. The MSB is always the page value. If the page has changed, it will have to be updated. When parsing the register file exported from CBPro, the MSB of each address will need to strip out the MSB from the register byte and determine each time if the page has changed before writing the LSB register value. If the page has changed, then the updated page value must be written before the LSB register value. An example of writing some simple functions to strip out the page from the register value are shown below.

Example: Verify the page (MSB)

```
PAGE = 0x01;
byte get_page(reg_address)
{
    byte page_byte = reg_address> 8; //MSB is page
    byte reg_byte = reg_address; //LSB is register
return page_byte
}
```

Example: Update the page: SPI Communication

```
PAGE = 0x01;
SPI_SET_ADDRESS_CMD = 0x00;
SPI_WRITE_CMD = 0x40;
update_page_spi (page_byte)
{
    SPI_Run(SPI_SET_ADDRESS_CMD, PAGE);
    SPI_Run(SPI_WRITE_CMD, page_byte);
}
```

Example: Update the page: I2C Communication

```
PAGE = 0x01;
update_page_iic (page_byte)
{
    IIC_Write(PAGE, page_byte); //S/SlaveAddress/0/Ack/Reg_Address=PAGE/Ack/Data=page_byte/Ack/P
}
```

Note that a named setting, such as “MXAXB_NUM”, is a contiguous range of bits within one or more registers. A named setting will never span a page boundary. This avoids having to write the PAGE in the middle of updating a setting. For example:

Named Setting	Location	Description
DEVICE_GRADE	0x0004[7:0]	A full byte at page 0x00, serial interface address 0x04
MXAXB_NUM	0x0235[0:43] 0x0235[7:0] 0x0236[7:0] 0x0237[7:0] 0x0238[7:0] 0x0239[7:0] 0x023A[3:0]	5 full bytes, 1 partial byte located at page 0x02 interface address 0x35 through 0x3A

The register map section of the device Reference Manual has more information on registers and addressing.

2. SPI Protocol

The applicable timing devices for this application note can operate in 4-wire or 3-wire SPI mode. There is a configuration bit called "SPI_3WIRE", which must be set when in 3-wire mode and cleared when in 4-wire mode. The hardware connections for both SPI configurations are shown in the following figure.

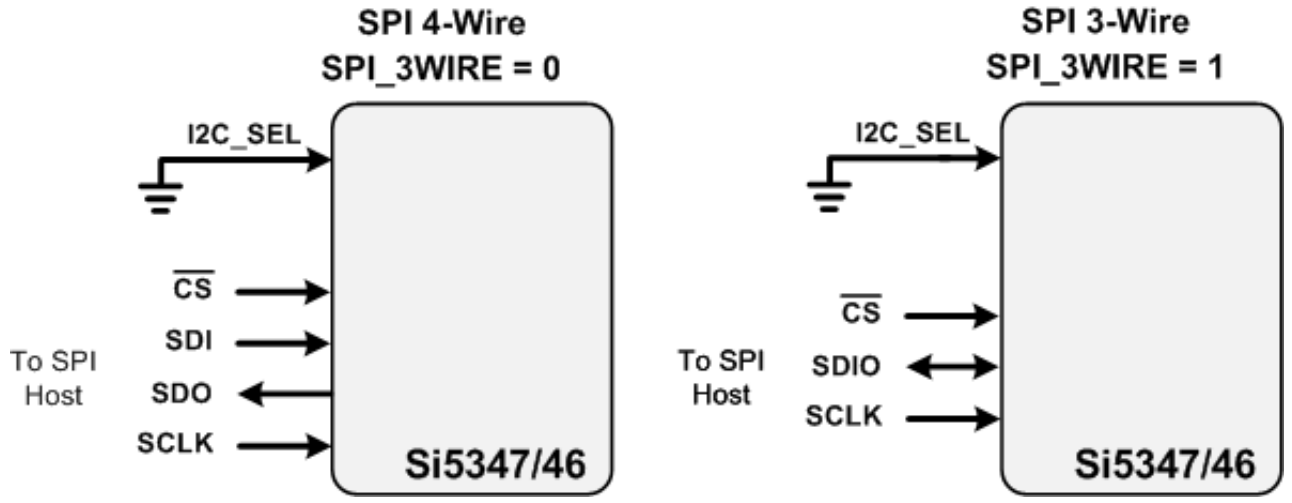

Figure 2.1. SPI Connections Diagram

The I2C_SEL pin must be low for SPI. The SPI protocol has a command format that must be followed for reading and writing transactions. The SPI commands that the applicable timing devices support are defined in the following table:

Table 2.1. SPI Commands and Expected Command Format

Instruction	1st Byte[1]	2nd Byte	3rd Byte	Nth Byte[2, 3]
Set Address	000x xxxx	8-bit Address	—	—
Write Data	010x xxxx	8-bit Data	—	—
Read Data	100x xxxx	8-bit Data	—	—
Write Data + Address Increment	011x xxxx	8-bit Data	—	—
Read Data + Address Increment	101x xxxx	8-bit Data	—	—
Burst Write Data	1110 0000	8-bit Address	8-bit Data	8-bit Data

Note:

1. X = don't care (1 or 0).
2. The Burst Write Command is terminated by deasserting /CS (/CS = high).
3. There is no limit to the number of data bytes that follow the Burst Write Command, but the address will wrap around to zero in the byte after address 255 is written.

The following transactions can be performed from SPI with the applicable timing devices:

1. Read Single Register
2. Write Single Register
3. Auto-Increment Read Register
4. Auto-Increment Write Register
5. Burst Write Data

Note that, in all the examples that follow, the registers chosen are completely arbitrary. Please take care and understand the purpose of each register in the product before writing to it.

The following is an example of the SPI_RUN function written on an 8051 Silicon Labs MCU and used to send and receive data over the SPI bus to communicate with the timing chip using the SPI command protocol.

SPI Run Function:

```
u8 SPI_Run(u8 command, u8 val)
          {
                P0_3 = 0;                              // Set Chip Select line low
                SPI0CN |= 0x01                         // Enable the SPI Port
                SPI0DAT = command                      // send the command byte to the DUT
                while (SPIF == 0 ) {}                  // Wait for transfer to complete
                SPIF = 0;                              // Clear the SPI interrupt flag
                spi _read_data = SPI0DAT;              // Clear the SPI receive buffer

                SPI0DAT = val;                         // Send the value or data to the DUT
                while (SPIF == 0 ) {}                  // Wait for transfer to complete
                SPIF = 0;                              // Clear the SPI interrupt flag
                P0_3 = 1;                              // Set Chip Select line high
                spi_read_data=SPI0DAT;
                return spi_read_data;                  // return the data from the receive buffer
           }
```

2.1 SPI Read Function

The following is an example of a read function using SPI communication:

```
SPI Read Function: u8 reg_read_spi(u16 register)
const PAGE = 0x01;
SPI_SET_ADDRESS_CMD = 0x00;
SPI_WRITE_CMD = 0x40;
SPI_READ_CMD = 0x80;
u8 reg_read_spi(u16 register)
{
    u8 page_byte;
    u8 reg_byte;
    page_byte =  register >> 8;
    reg_byte = register;
    SPI_Run(SPI_SET_ADDRESS_CMD, PAGE);
    SPI_Run(SPI_WRITE_CMD, page_byte);

    SPI_Run(SPI_SET_ADDRESS_CMD, reg_byte);

return SPI_Run (SPI_READ_CMD, 0xFF);
}
```

Example: Read register 0x052A. The first step is to write the PAGE register to 0x05. Then, write Register 0x2A and read the data out. Note that a read sequence always consists of first writing the register and then reading the data. It is advised to handle the paging with a separate function and compare when the page has changed. The following example includes the paging.

1. Run the SPI bus to send out 0x00 for the “Set Address” command followed by the PAGE register location (0x01). This sets the address to point to 0x01, which is the register for changing the page value:

   ```
   SPI_Run (SPI_SET_ADDRESS_CMD, PAGE) // Set Address to Register 0x01 (PAGE register)
   ```

2. Run the SPI bus to write the page value of 0x05, since the address is pointing at the PAGE register. First send out the “write” command 0x40, followed by the value 0x05:

   ```
   SPI_Run (SPI_WRITE_CMD, 0x05) // Write Value 0x05
   ```

3. Run the SPI bus to point to the register 0x2A. This is done by using the “Set Address” command followed by the register 0x2A:

   ```
   SPI_Run (SPI_SET_ADDRESS_CMD, 0x2A) //Set Address to Register 0x2A
   ```

4. Send the Read Command (0x80) followed by a dummy write. The dummy write is needed because the SPI bus sends data out to receive data in:

   ```
   data = SPI_Run (SPI_READ_CMD, 0xFF) //Send the read command and receive data from register 0x052A
   ```

2.2 SPI Write Function

This is an example of a write function using SPI communication.

```
SPI Write Function: void reg_write_spi(u16 register, u8 value)
const PAGE = 0x01;
SPI_SET_ADDRESS_CMD = 0x00;
SPI_WRITE_CMD = 0x40;
void reg_write_spi(u16 register, u8 value)
{
    u8 page_byte;
    u8 reg_byte;
    page_byte =  register >> 8;
    reg_byte = register;
    SPI_Run(SPI_SET_ADDRESS_CMD, PAGE);
    SPI_Run(SPI_WRITE_CMD, page_byte);

    SPI_Run(SPI_SET_ADDRESS_CMD, reg_byte);
    SPI_Run (SPI_WRITE_CMD, value);
 }
```

Example: Write 0x23 to register 0x03B9. The first step is to write the PAGE register to 0x03. Then, write Register 0xB9, and write in the data. It is advised to handle the paging with a separate function and compare when the page has changed. The following example includes the paging:

1. Run the SPI bus to send the “Set Address” command, followed by the PAGE register:

```
SPI_Run (SPI_SET_ADDRESS_CMD, PAGE) // Set Address to Register 0x01 PAGE
```

2. Run the SPI bus to write in the page value:

```
SPI_Run (SPI_WRITE_CMD, 0x03) // Write Value 0x03 to set the page to 3
```

3. Run the SPI bus to point to register address 0xB9:

```
SPI_Run (SPI_SET_ADDRESS_CMD, 0xB9) // Set the register address to 0xB9
```

4. Run the SPI bus write the value (0x23 in this example) into that register location:

```
SPI_Run (SPI_WRITE_CMD, 0x23) // Write Value 0x23
```

2.3 SPI Multi-Read Function

The following is an example of a multi-read function using SPI communication:

```
SPI Multi Read Function: u8 reg_read_multi_spi(u16 register, u8 num_bytes)
const PAGE = 0x01;
SPI_SET_ADDRESS_CMD = 0x00;
SPI_WRITE_CMD = 0x40;
SPI_READ_INC_CMD = 0xAA;
u8 data [] = {0,0,0};
u8 reg_read_multi_spi(u16 register, u8 num_bytes)
{
    u8 page_byte, reg_byte, i;
    page_byte = register >> 8;
    reg_byte = register;
    Run_SPI(SPI_SET_ADDRESS_CMD, PAGE);
    Run_SPI(SPI_WRITE_CMD, page_byte);

    Run_SPI(SPI_SET_ADDRESS_CMD, reg_byte);
    for (i=0; i < num_bytes; i++)
    {
        data[i] = Run_SPI (SPI_READ_INC_CMD, 0xFF);
    }
return data;
}
```

Example: Read registers 0x0130, 0x0131,0132. In this example, the purpose is to read from registers 0x0130, 0x0131, and 0x0132 consecutively. The first step is to manage the paging. This can be done by a separate function. Point to the starting register value 0x30. Then, using the read increment command, read the values out until complete.

1. Set the Address to point to the PAGE register:

```
SPI_Run (SPI_SET_ADDRESS_CMD, PAGE);
```

2. Write the page value to 0x01:

```
SPI_Run (SPI_WRITE_CMD, 0x01);
```

3. Set the address to 0x30:

```
SPI_Run (SPI_SET_ADDRESS_CMD, 0x30);
```

4. Send the read increment command. This will read out the value from the current register address and automatically point to the next consecutive address. A dummy byte is sent after the read increment command. Data is clocked out and in during the read transaction.

```
        data[0]= SPI_Run (SPI_READ_INC_CMD, 0xFF);
        data[1]= SPI_Run (SPI_READ_INC_CMD, 0xFF);
        data[2]= SPI_Run (SPI_READ_INC_CMD, 0xFF);
```

2.4 SPI Multi-Write Function

The following is an example of a multi-write function using SPI communication. In this example, the purpose is to wrtie to registers 0x0711, 0x0712, and 0x0713 consecutively.

```
SPI Multi Write Function: void reg_write_multi_spi (u16 register, u8 num_bytes)
const PAGE = 0x01;
SPI_SET_ADDRESS_CMD = 0x00;
SPI_WRITE_CMD = 0x40;
SPI_WRITE_INC_CMD = 0x60;
u8 data [] = {0xA3, 0xB5, 0x2C};

void reg_write_multi_spi (u16 register, u8 num_bytes, u8* data)
{
    u8 page_byte, reg_byte, i;
    page_byte = register >> 8;
    reg_byte = register;
    Run_SPI(SPI_SET_ADDRESS_CMD, PAGE);
    Run_SPI(SPI_WRITE_CMD, page_byte);

    Run_SPI(SPI_SET_ADDRESS_CMD, reg_byte);
    for (i=0; i < num_bytes; i++)
    {
        Run_SPI (SPI_WRITE_INC_CMD, data[i]);
    }
}
```

Example: Multi byte write to 0x0711, 0x0712, 0x0713.

1. Step 1: Set the Address to point to the PAGE register:

```
SPI_Run (SPI_SET_ADDRESS_CMD, PAGE);
```

2. Set the page value to 0x07:

```
SPI_Run (SPI_WRITE_CMD, 0x07);
```

3. Set the address of the register to 0x11:

```
SPI_Run (SPI_SET_ADDRESS_CMD, 0x11);
```

4. Send the write increment command. This will write the value to the current register address and automatically point to the next consecutive address.

```
        SPI_Run (SPI_WRITE_INC_CMD, data[0]);
        SPI_Run (SPI_WRITE_INC_CMD, data[1]);
        SPI_Run (SPI_WRITE_INC_CMD, data[2]);
```

2.5 SPI Burst Write Function

There is an SPI burst command that will write data consecutively into the part auto-incrementing between each data byte. The following is an example showing how this is used. Note that the paging should be handled before running the burst.

```
SPI Burst Write Function: void reg_write_burst_spi(u8* src_data, u8 num_bytes)

SPI_WRITE_BURST_CMD = 0xE0;

void reg_write_burst_spi(u8* src_data, u8 num_bytes)
{
    chip_select = 0;               //Set Chip Select line low
    SPI0DAT = SPI_WRITE_BURST_CMD; // Send the burst command byte to the DUT
    while (SPIF == 0 ) {}          // Wait for transfer to complete
    SPIF = 0;                      // Clear the SPI interrupt flag

    while ( (i < num_bytes) || (i == num_bytes))
    {
        SPI0DAT = src_data[i];    //send the register and data to the DUT
        while (SPIF == 0) {}      // wait for the transfer to complete
        SPIF = 0;                 // clear the interrupt flag
        i++;
    }
    chip_select = 1;             //Set Chip Select line high
}
```

Example: Burst write starting at register 0x002B.

1. Set the Address to point to the PAGE register:

```
SPI_Run (SPI_SET_ADDRESS_CMD, PAGE); //Set the Address to point to the PAGE register
```

2. Set the page value to 0x00:

```
SPI_Run (SPI_WRITE_CMD, 0x00); //Write 0x00 into the PAGE register
```

3. Send the Burst Command 0xE0, followed by the Register (0x2B), followed by the Data {0x01 through 0x0F}:

```
u8 src_data [] = {0x2B, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E,
0x0F};
```

   This function points to the data array above. The length “num_bytes” would be 16 for 15 bytes of data plus one byte for the register in this example. This code is specific to the Silicon Labs 8051 MCU architecture, but the concept is similar for other MCUs.

Note: There is no Burst Read command for the applicable timing devices.

3. I2C Hardware Connections

For I2C communication, the I2C_SEL pin must be set high. Please review the following hardware connection diagrams. They can also be found in in the reference manual.

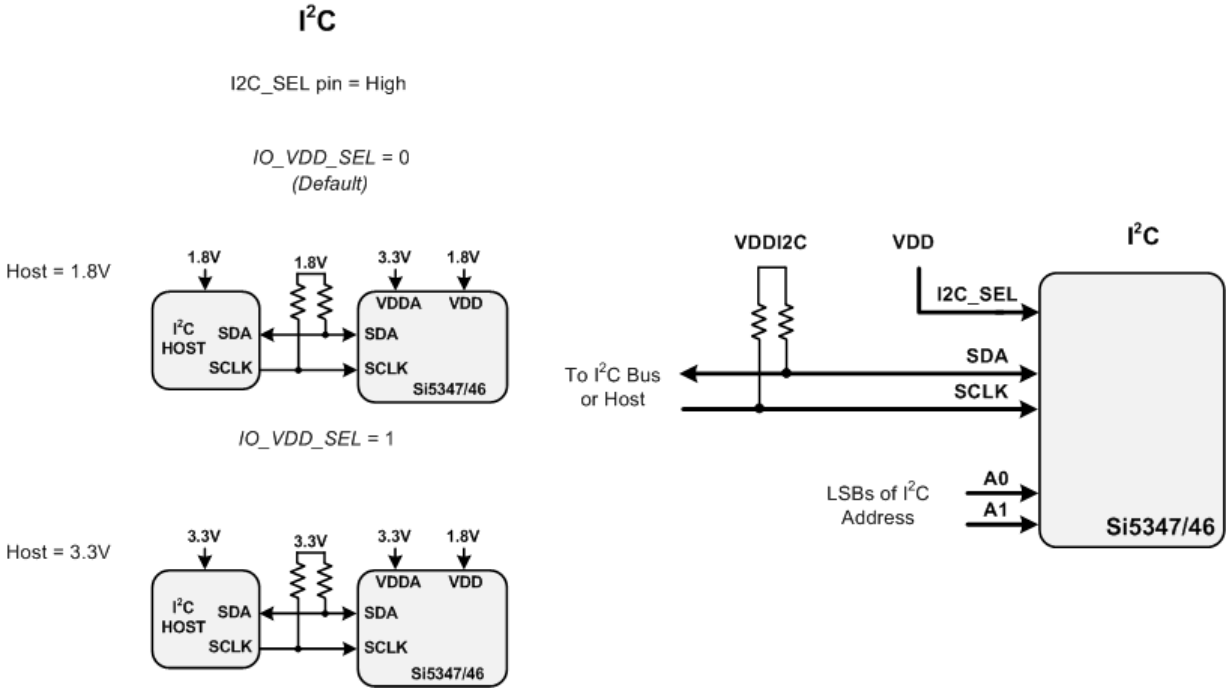

Figure 3.1. Hardware Connections I2C Communication

The I2C protocols used with the Si534x/8x devices are shown in the figures below. When reading and writing registers, it is important to remember that the PAGE register address must be set correctly each time. Also, pay attention to the Slave I2C Address. The first five bits starting from MSB of the slave I2C address are fixed, while A1 and A0 can be high or low. The eighth bit of the I2C address implies reading (1) or writing (0).

The 7-bit slave device address of the Si5347/46 consists of a 5-bit fixed address plus two pins that are selectable for the last two bits, as shown below.

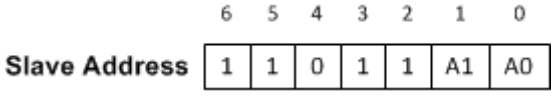

Figure 3.2. 7-Bit I2C Slave Address Bit-Configuration

Data is transferred MSB-first in 8-bit words as specified by the I2C specification. A write command consists of a 7-bit device (slave) address + a write bit, an 8-bit register address, and 8 bits of data as shown in the figure below. A write burst operation in which subsequent data words are written using an auto-incremented address is also shown:

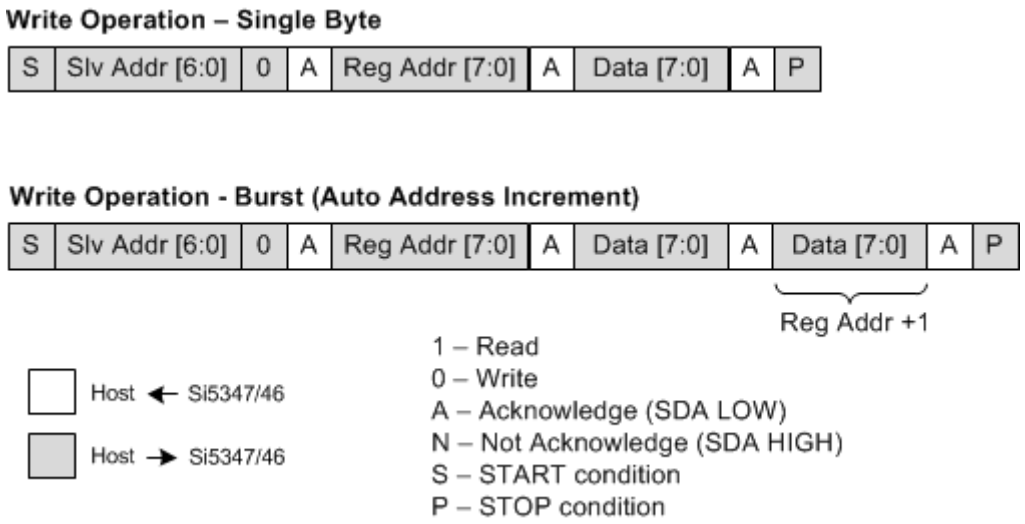

Figure 3.3. I2C Write Operation

A read operation is performed in two stages. A data write is used to set the register address; then, a data read is performed to retrieve the data from the set address. A read burst operation is also supported. This is shown in the following figure.

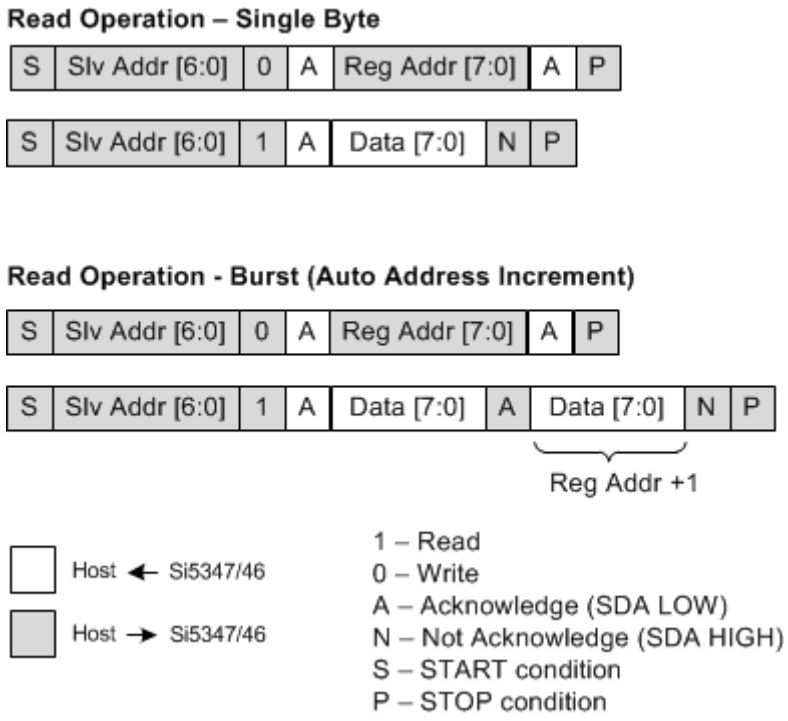

Figure 3.4. I2C Read Operation

The following sections describe the basic operations that can be performed using I2C:

1. Read a single register
2. Write a single register
3. Read increment
4. Write increment

3.1 I2C Read Function

This is a read function using I2C communication.

```
I2C Read Function: u8 reg_read_iic(u16 register)
const PAGE = 0x01;
u8 reg_read_iic(u16 register)
{
    u8 page_byte;
    u8 reg_byte;
    page_byte = register >> 8;
    reg_byte = register;
    IIC_Write(PAGE, page_byte); // Point to the correct page

    data = IIC_Read(reg_byte, 1); // Read 1 byte from the register address

return data;
}
```

Example: Read register 0x052A. The first step is to write the PAGE register to 0x05. Then, write Register 0x2A and then read the data out. Note that a read sequence always consists of first writing the register and then reading the data. It is advised to handle the paging with a separate function and compare when the page has changed. The following example includes the paging.

1. Write the Page Address (0x01) to a value of 0x05:

```
S/SlaveAddress/0/Ack/Reg_Address=0x01/Ack/Data=0x05/Ack/P
```

2. Read Address 0x2A.

   The read sequence requires first writing the register, then reading out the data.

```
S/Slave Address/0/Ack/Reg_Address=0x2A/Ack/P  //First Write the Register Address
```

```
S/Slave Address/1/Ack/DATA_RETURNED/N/P  //Then Read out the Data
```

When reading a single byte from the device, the data length of 1 is passed to the I2C firmware to tell it to read one byte of data and then stop. When reading multiple bytes, the data length will correspond to the number of bytes to read sequentially.

3.2 I2C Write Function

This is a write function using I2C communication.

```
I2C Write Function: void reg_write_iic(u16 register, u8 value)
const PAGE = 0x01;
void reg_write_iic(u16 register, u8 value)
{
    u8 page_byte;
    u8 reg_byte;

    page_byte = register >> 8;
    reg_byte = register;
    IIC_Write(PAGE, page_byte); // Point to the correct page

    IIC_Write (reg_byte, value, 1); // Write the 1 value into the register

return data;
}
```

Example: Write 0x23 to register 0x03B9. The first step is to write the PAGE register to 0x03. Then, write Register 0xB9, and then write in the data. It is advised to handle the paging with a separate function and compare when the page has changed. The following example includes the paging.

The first step is to set the PAGE register to a value of 3, and then write a value of 23 into register 0xB9.

1. Write the Page Address (0x01) to a value of 0x03:

```
S/SlaveAddress/0/Ack/Reg_Address=0x01/Ack/Data=0x03/Ack/P
```

2. Write a value of 0x23 into register 0xB9:

```
S/SlaveAddress/0/Ack/Reg_Address=0xB9/Ack/Data=0x23/Ack/P
```

When writing one byte of data, the firmware receives a data length instruction telling it to write in a single byte from the data passed in.

3.3 I2C Multi-Read Function

This is an example of a multi-read function using I2C communication.

```
I2C Multi Read Function: u8 reg_read_multi_iic(u16 register, u8 num_bytes)
const PAGE = 0x01;
u8 data [] = {0,0,0,0,0,0,0};
u8 reg_read_multi_iic(u16 register, u8 num_bytes)
{
    u8 page_byte, reg_byte, i;
    page_byte = register >> 8;
    reg_byte = register;
    IIC_Write(PAGE, page_byte);

    data = IIC_Read(reg_byte, num_bytes);  //read out multiple bytes and store in byte array
    }
return data;
}
```

Example: Read 7 bytes starting at register 0x0130. The first step is to set the PAGE register. Write the register address, and then read out the data, incrementing based on the length passed in.

1. Write the Page Address (0x01) to a value of 0x01:

```
S/SlaveAddress/0/Ack/Reg_Address=0x01/Ack/Data=0x01/Ack/P
```

2. Read Address 0x30:

```
S/Slave Address/0/Ack/Reg_Address=0x30/Ack/P
```

```
S/Slave Address/1/Ack/DATA_RETURNED[0]/Ack/ DATA_RETURNED[1]/Ack/…DATA_RETURNED[6]/N/P
```

When reading multiple bytes from the device, the data length is passed to the I2C firmware to tell it to read the number of bytes of data and then stop. The register address auto-increments as the data is incrementally read out.

3.4 I2C Multi-Write Function

This is an example of a multi-write function using I2C communication. In this example, the purpose is to wrtie eight consecutive values starting at register 0x021C.

```
I2C Multi Write Function: void reg_write_multi_iic (u16 register, u8 num_bytes, u8* data)
const PAGE = 0x01;
u8 data [] = {0xA3, 0xB5, 0x2C, 0x07, 0x08, 0x09, 0x0A, 0x04};

void reg_write_multi_spi (u16 register, u8 num_bytes, u8* data)
{
    u8 page_byte, reg_byte, i;
    page_byte = register >> 8;
    reg_byte = register;
    IIC_Write(PAGE, page_byte); // Point to the correct page

    IIC_Write(reg_byte, data, num_bytes); // Point to the register and send in all the data.
}
```

Example: Multi byte write to 0x021C.

After the page address is written, a multi-byte write is done by writing the register followed by all the data. The I2C driver will need to know the length expected to continue incrementing and writing in all the values passed in.

1. Write the Page Address (0x01) to a value of 0x02:

```
S/SlaveAddress/0/Ack/Reg_Address=0x01/Ack/Data=0x02/Ack/P
```

2. Write a data value data[0],data[1],…data[7] starting at register 0x1C:

```
S/SlaveAddress/0/Ack/Reg_Address=0x1C/Ack/data[0]/Ack/data[1]/Ack/…data[7]/Ack/P
```

When writing multiple bytes from the device, the data length is passed to the I2C firmware to tell it to write the number of bytes of data and then stop. The register address auto-increments as the data is incrementally written in.

ClockBuilder Pro

One-click access to Timing tools, documentation, software, source code libraries & more. Available for Windows and iOS (CBGo only).

www.silabs.com/CBPro

Timing Portfolio
www.silabs.com/timing

SW/HW
www.silabs.com/CBPro

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com