Technical notes on using Analog Devices DSPs, processors and development tools
Visit our Web resources http://www.analog.com/ee-notes and http://www.analog.com/processors or e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

Blackfin® Processor SCCB Software Interface for Configuring I2C® Slave Devices

Contributed by Thorsten Lorenzen

Rev 2 – March 13, 2007

Introduction

This EE-Note describes the implementation of the Serial Camera Control Bus (SCCB) interface using software and general-purpose pins on Blackfin® processors. Because of its architecture and video processing capabilities, Blackfin processors are well-suited to interface with video devices. Many of these video devices in the signal chain must be configured by an I2C®-compatible hardware interface. For those Blackfin derivatives not equipped with this two-wire interface (TWI), the software described in this document can be used to emulate the function of I2C with the help of two general-purpose pins. The protocol is compliant with the I2C protocol and supports slave devices only. The Blackfin processor is always acting as the master. No multi-master bus network can be accessed.

Basics

The SCCB interface can be realized with the use of two general-purpose pins. In this example, `PF0` is used to generate the clock line, and `PF1` is used to transmit and receive data. This functionality is most common in configuring video devices. The associated software for this EE-Note includes the protocol stack (`I2C_BF5xx_revXX.asm`, `I2C_BF5xx_revXX.h`) and the C/ASM API (`I2C_BF5xx_ASM_C_API.c`, `I2C_BF5xx_ASM_C_API.h`) to call the protocol within a C program. One programmable timer (timer0) is used to run the two-wire state machine. The timer generates an interrupt after its counter has expired. All activities of the SCCB interface are performed during the timer interrupt. The rate of the SCCB clock is determined by the interrupt interval of the timer. For example, the time between these interrupts can be used to process video data as it is received. The associated software contains two projects, `ADSP-BF533_I2C_ASM.dpj` and `ADSP-BF561_I2C_ASM.dpj`, to implement the software SCCB protocol for the ASDP-BF533 and the ADSP-BF561 processors, respectively.

I2C Software Protocol

Common video devices are fitted with an I2C-compatible interface that is dedicated to set up all the registers in the devices. Because an addressing scheme is used, multiple devices can be connected on the same lines. Figure 1 shows an example connection.

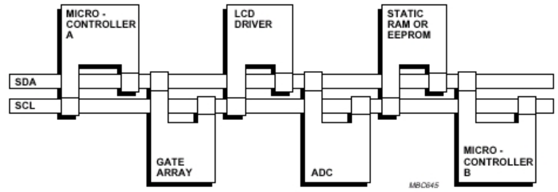

Figure 1. Device connection

In order to generate a write to a device's register, the data line must first send the "device address", followed by the register's "word address". At this point, the actual data can be transferred. Each bit must be aligned to a clock cycle generated by the clock line. As long as the clock is continuous, data words will be sent to the slave. The address will be incremented internally by the slave.

After every 8th bit transferred, an acknowledge bit (9th bit) is tested by the master (Blackfin processor). The SCCB interface will set the data line high and reconfigure `PF1` as an input. If the slave (video device) does not pull down the data line, the code will end in an error routine indicating a failed access.

Figure 2 shows a write access. As can be seen, the 8th bit of the device address indicates that the following data will be written in the device.

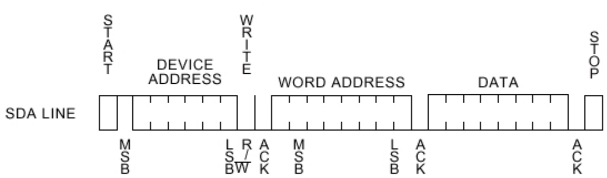

Figure 2. I2C write access

Read accesses start in the same fashion, writing the "device address" and the register's "word address" to the device. After this is done, an extra start condition must be inserted, as shown in Figure 4. The device address will be written to the device again, but the 8th -bit is set to high, indicating that a read access follows. If a second device address is received, as shown in Figure 3, the device starts sending the contents of the location that the internal address pointer points to. Multiple data words can be sent from the slave device to the processor (master).

Similar to the write sequence, sending the device address, word address and device address again ends with an "acknowledge" test, as described above. This time, if the master does not pull down the data line, the slave will go into power down mode and keep the data line high. Finally, the read sequence will end in a "no acknowledge" bit (9th bit is driven high by the master). This is required to gracefully complete the access. If the "no acknowledge" bit is missed, some devices may remain in read mode, regardless whether a stop condition follows.

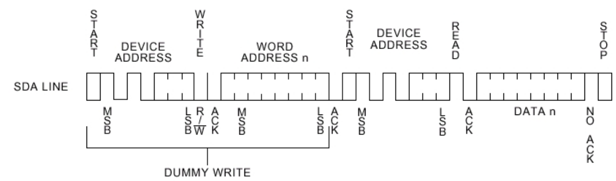

Figure 3. I2C read access

Figure 4 shows the start condition, stop condition, and the data and clock alignment. After the start condition, the data bit must be available before the rising edge of the clock. In addition, it must not be released before the falling edge.

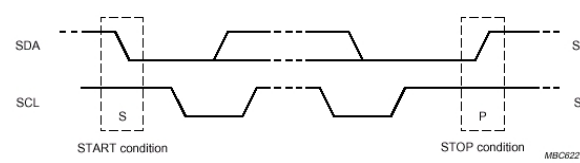

Figure 4. I2C start, stop condition

The SCCB software interface meets all these requirements. More details about the I2C timing can be found in dedicated literature or on the World-Wide Web.

Functional Support

As explained in the "I2C Software Protocol" section above, many devices function on an 8-bit basis. However, more and more devices can be found today with either 16-bit address capability or even 16-bit register width (e.g., Micron, Atmel). This has been adapted by the protocol.

The list below summarizes the four different address and data widths supported within the ASM/C API.

- 8-bit address, 8-bit register width support
- 8-bit address, 16-bit register width support
- 16-bit address, 8-bit register width support
- 16-bit address, 16-bit register width support

The protocol stack itself is capable of expanding address width or data width to a higher degree, as explained below.

Assembly Program Use

The SCCB interface is defined and used in the `I2C_BF5xx_revXX.asm` / `I2C_BF5xx_revXX.h` files. Before calling the transfer startup routine (`SCCB_Interface()`), some configuration is required. To understand the usage of the interface, the following steps are provided.

1. Open the `I2C_BF5xx_revXX.h` file to select the register addresses and bit settings for a dedicated Blackfin derivative. For example, configurations for the ADSP-BF561 and ADSP-BF533 processors can be found, along with explanatory comments.

   Both hardware description blocks (for ADSP-BF533 and ADSP-BF561 processors) are selected automatically via a preprocessor definition (`__ADSPBF533__` or `__ADSPBF561__`). This must be taken into account when making modifications to it.

2. Review some variables defined in `I2C_BF5xx_rev_XX.asm` to configure the interface before calling it.

   Transmission or reception is chosen using `_SCCB_Control`. Setting this variable to one (1) will trigger a read sequence. Setting this variable to two (2) triggers a write sequence.

   `_SCCB_Word_Count` holds the number of general transfers to be performed. Device addresses and word addresses are included. This generic form allows expanding address and data widths, if required.

   The transmission data (including device and word address) is held in the `_SCCB_DataIn[x]` array. The receive data from a read sequence is held in the `_SCCB_DataOut[x]` array.

   `_SCCB_In_Progress` can be polled for notification of when a transmission has completed. Therefore, before calling `_SCCB_Interface()`, this variable must be set to a non-zero value by the user application. The final step of the protocol will be to clear this variable (set it to zero).

   Missing acknowledges during transmission will trigger an error message. To be notified of errors, `_SCCB_Error` can be used. The user application must clear this variable before calling the SCCB interface. In the event of an error, `_SCCB_Error` is set to one and the transfer is aborted by execution of the I2C stop condition.

3. After these settings are established, the `_SCCB_Interface()` subroutine (defined in the `I2C_BF5xx_rev_XX.asm` file) must be called to start the transfer.

   `_SCCB_Interface()` stores and restores all registers used by the I2C protocol. It starts the timer and sets up the interrupt and PF pins on a user-defined basis.

   User-defined interrupt priority cannot be selected within the protocol because the SIC_IARx registers are not modified. The default values are:

   - ADSP-BF533: Timer0 = IVG11

   - ADSP-BF561. Timer0 = IVG10

   If a change is required, it must be made by the user application.

4. When the initialization process is completed, the core returns to the user application code. A timer interrupt is raised after a certain

time, following its generation in the initialization process. Each timer interrupt will be taken to drive a signal change or insert an extra delay.

5. During the SCCB transfer, all used PF pins and the selected timer must not be used for any other purposes. The registers and pointers may be used because they will be stored and restored before and after the end of each interrupt.

Avoid starting the SCCB interface a second time before the pending transfer has been completed. For multiple device setups, use conditional loops or place code between each call that guarantees the delay required to finish the process.

6. The final timer interrupt will turn off the timer and disable all I2C resources.

C Program Use (ASM Interface)

In order to make the protocol functional in C, a C/ASM API was created. This section explains how to use the protocol stack in this case.

As discussed in the previous section, open the `I2C_BF5xx_revXX.h` file and select the register addresses and bit settings for a specific Blackfin derivative.

Both hardware description blocks (for ADSP-BF533 and ADSP-BF561 processors) are selected automatically via a preprocessor definition (`__ADSPBF533__` or `__ADSPBF561__`). This must be taken into account when making modifications to it.

In addition to the `I2C_BF5xx_revXX.asm` and `2C_BF5xx_revXX.h` source files, the `I2C_BF5xx_ASM_C_API.c` and `I2C_BF5xx_ASM_C_API.h` files must be added to the project.

The API sources include two types of accesses: blocking and non-blocking. A blocking access means that, for the duration of an I2C transfer, the processor will stall (i.e., it polls `SCCB_In_Progress` internally in the subroutine). In contrast, the non-blocking access just triggers the transmission and immediately returns to the application code. The advantage is that application code can be executed in parallel to the I2C transfer. However, `SCCB_In_Progress` must be polled externally (i.e., in the user application) to detect completion of a pending transmission. It is left to the user to determine which type of access to use.

The `I2C_NonBlocked_Write()` and `I2C_Blocked_Write()` functions can be found in the `I2C_BF5xx_ASM_C_API.c` file. They are the entry points to the write functions. The Boolean `addr_size_16` and `data_size_16` elements configure the interface to either send/receive 16-bit or 8-bit addresses or data values. `TWIBase_Addr` will hold the device address to identify the target external device. The `start_address` parameter can either include 16 or 8 bits and identifies the register/memory address within the identified device. The `values` pointer requires the address to an array where the data is located to be sent. `Num_Transactions` indicates the number of transfers to be executed. It includes the number of data transfers only! For blocking accesses, an integer return value returns the number of successful transfers. For non-blocking accesses, `_SCCB_Error` must be polled in order to recognize whether the transfer completed gracefully.

For non-blocking accesses, avoid starting the SCCB interface a second time before the pending transaction has completed. For multiple device setups, use conditional loops (e.g., `while(SCCB_In_Progress)`) or place code between each call that guarantees the delay required to finish the process.

Similarly, the `I2C_NonBlocked_Read()` and `I2C_Blocked_Read()` read functions use the same variables. For blocking accesses, the received data can be obtained directly from the `values` variable. For non-blocking accesses, the data cannot be obtained before the transfer has been completed. Therefore, it is left to the user application to get the values.

The Listings below show how to use the C interface in the user application.

```
unsigned int I2C_Blocked_Write(bool addr_size_16,          // Addr width select 8/16
                             bool data_size_16,            // Data width select 8/16
                             unsigned char TWIBase_Addr,   // I2C device addr
                             unsigned short start_address,// I2C register addr
                             unsigned short* values,       // Values to send
                             int Num_Transactions);        // Number of transfers
```

Listing 1. I2C blocked write prototype

```
void I2C_NonBlocked_Write   (bool addr_size_16,            // Addr width select 8/16
                             bool data_size_16,            // Data width select 8/16
                             unsigned char TWIBase_Addr,   // I2C device addr
                             unsigned short start_address,// I2C register addr
                             unsigned short* values,       // Values to send
                             int Num_Transactions);        // Number of transfers
```

Listing 2. I2C non-blocked write prototype

```
unsigned int I2C_Blocked_Read(bool addr_size_16,           // Addr width select 8/16
                             bool data_size_16,            // Data width select 8/16
                             unsigned char TWIBase_Addr,   // I2C device addr
                             unsigned short start_address,// I2C register addr
                             unsigned short* values,       // Values to send
                             int Num_Transactions);        // Number of transfers
```

Listing 3. I2C blocked read prototype

```
void I2C_NonBlocked_Read    (bool addr_size_16,            // Addr width select 8/16
                             bool data_size_16,            // Data width select 8/16
                             unsigned char TWIBase_Addr,   // I2C device addr
                             unsigned short start_address,// I2C register addr
                             int Num_Transactions);        // Number of transfers
```

Listing 4. I2C non-blocked read prototype

```
ErrorIdent = I2C_Blocked_Write(false, false, DeviceAddr, RegAddr, &TxArray[i], CN);
// Application code can be executed as transfer has been completed
```

Listing 5. I2C blocked write function call

```
I2C_NonBlocked_Write(false, false, DeviceAddr, RegAddr, &TxArray[i], CN);
// Application code can be executed while I2C transfer is in progress
while(SCCB_In_Progress);  // Before kick off a subsequent I2C access check progress
if(SCCB_Error == 1) while(1);            // Check for errors after transfer complete
```

Listing 6. I2C non-blocked write function call

```
ErrorIdent = I2C_Blocked_Read(false, false, DeviceAddr, RegAddr, &RxArray[i], CN);
// Application code can be executed as transfer has been completed
```

Listing 7. I2C blocked read function call

```
I2C_NonBlocked_Read(false, false, DeviceAddr, RegAddr, CN);
// Application code can be executed while I2C transfer is in progress
while(SCCB_In_Progress);  // Before kick off a subsequent I2C access check progress
if(SCCB_Error == 1) while(1);           // Check for errors after transfer complete
for (i=0; i<1; i++) RxArray[j++] = SCCB_DataOut[i];     // Get values received
```

Listing 8. I2C non-blocked read function call

Performance

The following example shows the measured performance. The core clock (CCLK) is running at 432 MHz. The peripheral clock (SCLK) is running at 108 MHz. The timer0 is set to run the SCCB at 70 KHz (see Figure 5).

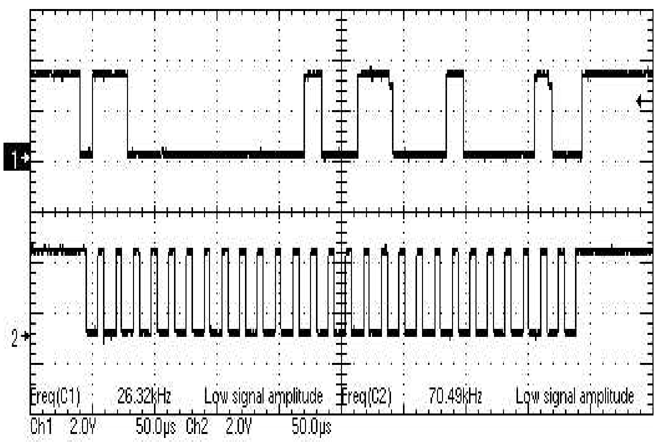

Figure 5. Write access timing

Figure 5 shows a register write. Three bytes are sent (channel 1 is SDA, channel 2 is SCL). The first 8 bits include the device address. In this case, it is "0xC0". The 9th bit is held low by the slave (acknowledge). The following 8 bits hold the word address, "0x13", within the identified device, followed by the second acknowledge. Finally, the last 8 bits carry the register's content, "0x21", followed by the third acknowledge.

Looking at Figure 5 again, the time between the signal changes of the data line and the clock line can be used by the core to execute other instructions. Figure 6 zooms in on the write transfer displayed in Figure 5 and shows some additional pins. Channel 1 shows the timer0 pin (TMR0), channel 2 shows PF4, which is programmed to toggle outside the timer interrupt. Channel 3 represents the SDA line, and channel 4 is the SCL line.

As can be seen, each positive edge of the timer0 pin (channel 1) will trigger an interrupt. The interrupt will cause the PF4 pin to stop and to start toggling.

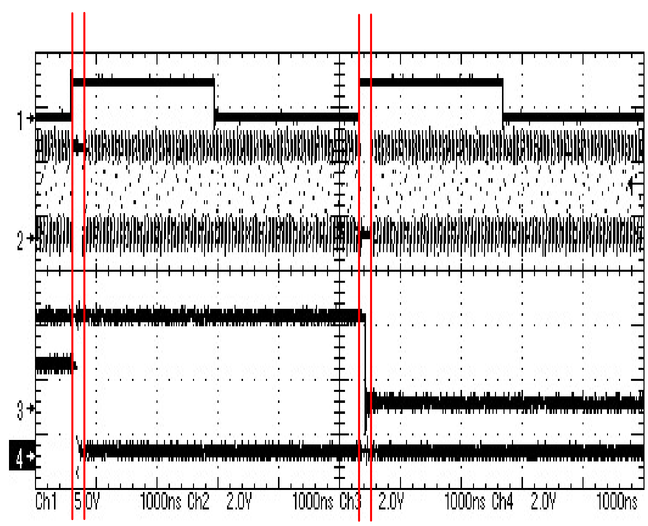

Figure 6. Performance test timing

For this test, the core is running in a loop outside of interrupt events. The loop executes the following instructions:

```
P0 = 0x02FF(Z);
LSETUP(Loop Start,performance test)
LC0=P0;
Loop_Start:
        p0.h = hi(FIO_FLAG_C);
        p0.l = lo(FIO FLAG C);
        r0.l = 0x10;
        w[p0] = r0;
        p0.l = lo(FIO FLAG S);
        r0.l = 0x10;
performance_test:w[p0] = r0;
```

Listing 9. Performance test instructions

These instructions just toggle the `PF4` pin continuously, as shown in Figure 6. `PF4` does not toggle when the core is executing SCCB instructions during the timer interrupt. Each positive edge of the `TMR0` pin causes the timer interrupt.

Figure 7 illustrates the processor load. At the beginning, `PF4` is toggling. The positive edge of `TMR0` generates the timer interrupt, which causes `PF4` to stop toggling. The first interrupt (Figure 6) forces the clock line (SCL) to clear its pin. After the interrupt is completed, `PF4` toggles again until the next positive edge of the `TMR0` pin appears. The next interrupt caused by `TMR0` again forces the data line to clear.

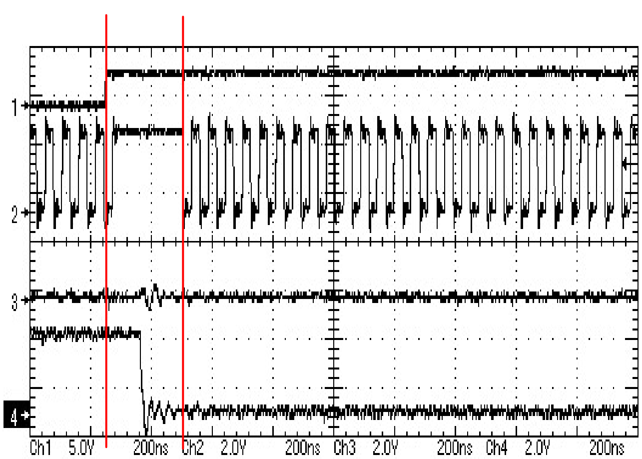

Figure 7. Performance test timing

Figure 7 shows the clearance of SCL zooming into Figure 6.

The frequency of `PF4` (channel 2) is not related to only the core frequency. Each instruction in the loop (Listing 9) will be executed in one or two cycles, but toggling the actual pin implies use of the system bus, which runs at 108 MHz.

The frequency of `PF4` is a combination of the core speed (`CCLK`) and the system speed (`SCLK`).

Conclusion

If the `CCLK` is running at 432 MHz, the `SCLK` is running at 108 MHz, and the SCCB interface is running at 70 KHz during the SCCB action, the entire transfer (shown in Figure 5) will take 404 µs. After the transfer completes, the core processes data at 100% again. As can be seen during the transfer, 5% of the timer period is used by the SCCB interrupt, leaving 95% of the timer period available to the core to process data. Additionally, the core performance can be increased by slowing down the timer. This results in a higher percentage of data processing performance, but it extends the SCCB transfer time.

This example was developed to emulate an I2C-compatible hardware interface for ADSP-BF53x and ADSP-BF561 Blackfin processors. Further performance optimizations may be realized via restructuring of the provided code.

References

[1] *ADSP-BF561 Blackfin Processor Hardware Reference.* Revision 1.1, February 2007. Analog Devices, Inc.

[2] *ADSP-BF53 Blackfin Processor Hardware Reference.* Revision 3.2, July 2006. Analog Devices, Inc.

[3] *ADSP-BF53x/BF56x Blackfin Processor Programming Reference.* Revision 1.1, February 2006. Analog Devices, Inc.

[4] *ADSP-BF561 Blackfin Embedded Symmetric Multi-Processor Data Sheet*. Revision A, May 2006. Analog Devices, Inc.

[5] *ADSP-BF533 Blackfin Embedded Processor Data Sheet*. Revision D, September 2006. Analog Devices, Inc.

Document History

Revision	**Description**
Rev 2 – March 13, 2007 *by Thorsten Lorenzen*	Initial public release.
Rev 1 – July 30, 2003 *by Thorsten Lorenzen*	Maintained internally.