Embedding it better...

µTasker Document

µTasker – ATMEL SAM7X SSC Driver

Copyright © 2010 M.J.Butcher Consulting

Table of Contents

1. Introduction

This document describes the use of the SSC driver interface together with the ATMEL SAM7X. The application is specific to a certain form of communication to and from a second processor (DSP) and so not general in nature.

2. Activating SSC support in the SAM7X Project

To work with the SSC define `SSC_INTERFACE` must be activated in `config.h`. This enables the driver code.

Specific details concerning the interface are defined in `app_hw_sam7x.h`:

```
// SSC Interface
//
#ifdef SSC_INTERFACE
    #define OUR_SSC_CHANNEL 0           // SAM7X has only one SSC interface
    #define NUMBER_SSC      1           // one SSC available
    #define SSC_SUPPORT_DMA             // enable SSC DMA support
    #define SSC_DMA_DOUBLE_BUF_RX       // use double buffering DMA on reception
    #define LOG_SSC0                    // log transmission to SSC0.txt
#endif
```

Note that the SSC is usually used as high speed communication interface with framing, which makes DMA operation particularly relevant. Double buffering at the receiver is necessary to keep up with high speed, back-to-back receptions (streaming).

3. Transmission

The application required a configuration allowing the application to define a fixed length block of data to be prepared and transmitted as a single frame. This frame is signalled by a synchronisation signal so that the remote receiver can identify its start. The synch signal and transmission clock are thus generated by the transmitter.

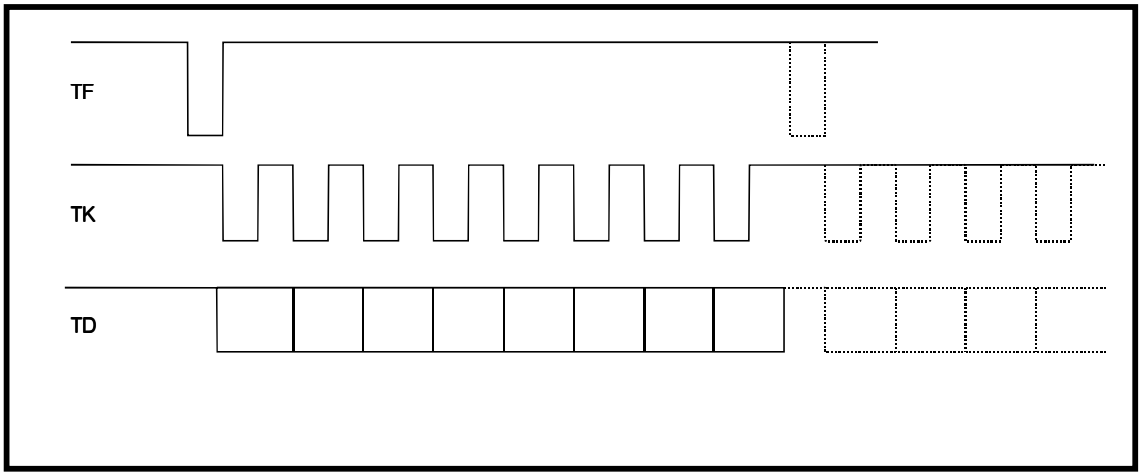


Figure 3-1 General transmission framing example

Figure 3-1 illustrates the general operation of the three transmission lines.

TF is the framing signal, in this case it is a negative pulse as the frame starts (exact timing of its position is configurable). The TK output from the processor is the clock signal – in this case it is shown generating 8 clocks only when the example 8 bit frame is valid. The TD signal sends the data bits (either in LSB or MSB format).

This example shows a single 8 bit byte being sent for simplicity. The works size can however be adjusted and 16 bits words are used in the application. The frame also contains only one word, whereby the application sends always 16 words in a frame; the application frame will thus be 16 x 16 (256) clocks in length and contain 16 discrete 16 bit words.

After a complete frame has been sent, the signals generally remain in their idle states until a further frame is to be sent. The dotted lines show a second frame being sent immediately after the first has finished, which is also the case when multiple frames are prepared and transmitted.

3.1. Transmission Problem and Workaround

It was found not to be possible to operate the TK signal only when the frame data was present since the clock, although generating the correct number of clock pulse, was offset to the transmitted data.

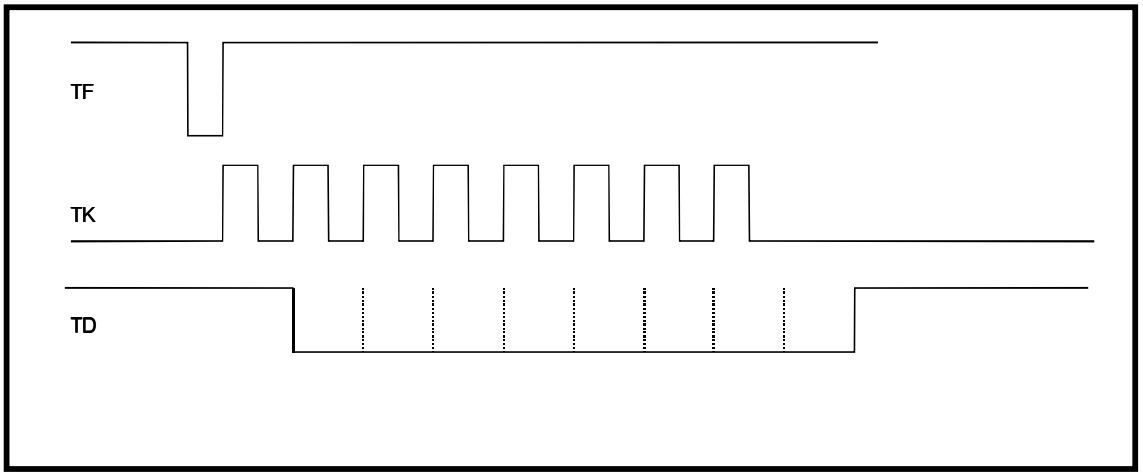


Figure 3-2 Transmission of 0x00

Figure 3-2 shows the result of transmitting a frame containing a single byte of value 0x00. There are 8 clock pulses but the first data bit is a '1'. After the final clock pulse the final data bit is sent on the line, but without a corresponding clock. This results in the receiver receiving the value 0x80 if the 8 clock pulses are used.

By allowing the TK to free-run (continuously sending clocks, even when no data is being transmitted) the receiver can correctly receive the data as long as it is programmed to start receiving one clock after the synch. This is the mode used for this application.

Due to the fact that the transmitter has to operate in periodic mode in order to generate the frame signal it also needed to be disabled after the frame transmission(s) have completed. It is important that the frame size is never defined less than 4 words in length in order for the disabling to take place once the frame has started operation – if less size is used the transmitter disabling take space before the first word has been completely copied to the output FIFO by DMA and so the frame is corrupted by the disable operation. Disabling the transmitter after a frame has started is otherwise not a problem since the frame will then be completely transmitted before the transmitter actually stops operation.

If the transmitter were not enabled only when transmitting a frame and disabled afterwards the TF signal would be continuously generated so that the remote receiver would see data all of the time.

If the periodic transmission were to take place continuously and data sent at arbitrary points in time, it would also be sent without any synchronisation to the frame sync signal. By enabling the transmitter at the start of only data transmissions the synchronisation is achieved.

Since the transmitter needs to be disabled by the transmit interrupt routine, delays in servicing this lead to additional empty frames (with content 0xffff) to be sent until the transmitter is finally disabled. The receiver must be able to handle this (recognise that there is no valid data available). This effect is more pronounced as the transmission speed increases.

4. Reception

The receiver is configured to require an input frame synchronisation to start frame reception. Once the synchronisation has occurred a single frame of data will be received and copied to the input buffer. The end of frame interrupt allows the driver to signal to the user that there is a frame full of data to be read. It also causes the next frame buffer to be prepared so that the SSC receiver can continue receiving further data. *It is advisable to always use double-buffered DMA operation*.

As long as the receiver interrupt can be serviced before a second complete frame has been received no data will be lost. At 5MHz and with no gap between reception frames, this give an interrupt rate of 51µs and so a maximum latency which must be achieved to ensure continuous reception within a burst of more than two consecutive 16 x 16 bit frames.

Since interrupts may be blocked for periods of several ms when internal FLASH to the SAM7X is written or deleted, loss of continuous SSC data may occur. This can be avoided by either not allowing such operations when the SSC is in operation or else allowing the SSC DMA complete interrupt to still be serviced, with the handling routine and all its sub-routines in internal SRAM (not FLASH).

Note that the receiver copies received frames directly to its input buffer by DMA using a DMA swap buffer (*double-buffered DMA operation*). The reception buffer should therefore be at least 2 frame lengths in size and should also be of a size equal to a multiple of the frame size; the buffer size is counted in words when configured, where the word width is configurable.

The application must read data from its input buffer before the buffer fills up with the maximum number of receive frames that it can store. By allocating larger input buffer sizes (`tInterfaceParameters.Rx_tx_sizes.RxQueueSize`) the application reaction time to reading received data becomes less critical since the driver can queue more frames.

Notes:

- Internal loop-back mode causes an additional frame with 0xffff content to be received even though no additional sync is seen.
- HW loop back (TF -> RF; TK -> RK; TD -> RD) doesn't have the same problem.
- HW loop back at 5MHz has an additional frame of 0xffff content at end due to the fact that the transmitted doesn't disable fast enough to stop this.

5. Configuring the SSC Interface

The SSC is configured in a similar way to the UART, using the `fnopen()` command.

```
static void fnInitI2S(void)
{
    SSCTABLE tInterfaceParameters;   // table for passing information to driver
    tInterfaceParameters.Channel = OUR_SSC_CHANNEL; // set I2S channel for use
    tInterfaceParameters.usSpeed = ((MASTER_CLOCK/2)/(5000000));// data rate 5MHz
    tInterfaceParameters.Task_to_wake = OWN_TASK;// wake self when frames received
    tInterfaceParameters.ucTxDataShift = 1;
                                  // bit shift from sync to start of transmit data
    tInterfaceParameters.ucRxDataShift = 2;
                                 // bit shift from sync to start of receive data
    tInterfaceParameters.usConfig = (TX_CLOCK_INVERTED | TX_MSB_FIRST |
    TX_NEGATIVE_FRAME_PULSE | RX_MSB_FIRST | RX_CLOCK_INVERTED | RX_SYNC_FALLING);
    tInterfaceParameters.ucWordSize = 16;
                       // transmit and receive data treated as words of this width
    tInterfaceParameters.ucFrameLength = 16;     // the number of words in a frame
    tInterfaceParameters.Rx_tx_sizes.RxQueueSize = 256;
                                                 // input buffer size (in words)
    tInterfaceParameters.Rx_tx_sizes.TxQueueSize = 128;
                                                 // output buffer size (in words)
        #ifdef SSC_SUPPORT_DMA
    tInterfaceParameters.ucDMAConfig = (UART_TX_DMA | UART_RX_DMA);
                                    // activate DMA on transmission and reception
        #endif
    if ((I2S_handle = fnOpen(TYPE_SSC, FOR_I_O, &tInterfaceParameters)) != 0) {
                                     // open the channel with defined parameters
        fnDriver(I2S_handle, (TX_ON | RX_ON), 0); // enable rx and tx
    }
}
```

Since user data can be of different word sizes (1 to 4), the transmit and receive buffers used to hold the user data are aligned to appropriate boundaries in heap. This also ensures that the DMA controller used to transmit and receive data operates always aligned.

The word length of the input and output buffers must be chosen to be multiples of the frame length. At least 4 x frame length is suggested to ensure efficient DMA operation.

6. Loop-Back Test Configuration Example

6.1. Transmitting Data

Data should normally be transmitted in frame block sizes only. It is possible to transmit only one block or a multiple of blocks as shows in the following examples, whereby the data is prepared in an array of elements of the defined SSC word size (16 bits in the example).

```
static unsigned short usTest[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
fnWrite(I2S_handle, (unsigned char *)usTest, 16);
```

The first example shows a 16 x 16 bit array being transmitted as single frame

```
static unsigned short usTest1[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
static unsigned short usTest2[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};

fnWrite(I2S_handle, (unsigned char *)usTest, 16);
fnWrite(I2S_handle, (unsigned char *)usTest, 16);
```

The second example shows 2 x 16 x 16 bit arrays being transmitted. The two writes cause the first frame to be started and the second frame to be queued. Once the first frame transmission has terminated, the second frame will immediately by sent.

Note that the transmitter has to be deactivated by the transmitter interrupt routine once the DMA controller has completed its transfer. Should the interrupt be delayed until after the first frame transmission has also physically been completed (next frame sync will be generated) the second block will not necessarily be synchronised to the frame and errors may occur at the remote receiver. It is therefore recommended that this method is not used, but rather either the SW waits until the initial transfer has completed (eg. by limiting the rate of calls), or else the third technique be used as follows:

```
static unsigned short usTest3[32] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
                           17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32};

fnWrite(I2S_handle, (unsigned char *)usTest, (16*2));
```

The third example shows 2 x 16 x 16 bit arrays being transmitted started by a single write. The single write ensures that the DMA controller transfers multiple frames without any processor intervention, which guaranties that the two frames are consecutive with no de-synchronisation.

However it should be noted that, if the interrupt at the end of frame transmission is delayed it is possible for the SSC transmitter to continue with the next frame, resulting in an empty frame (the frame synch is generated and the corresponding number of frame clocks are generated, but no data is sent – the data line remains at '1'). In this situation the remote receiver will receive 16 x 0xffff and may need to be able to recognise this as a non-valid frame.

The following code shows a test of transmitting 8 x 16 x 16 frames periodically (called subsequently on the `E_NEXT_SSC_TEST` event) with a recognisable pattern.

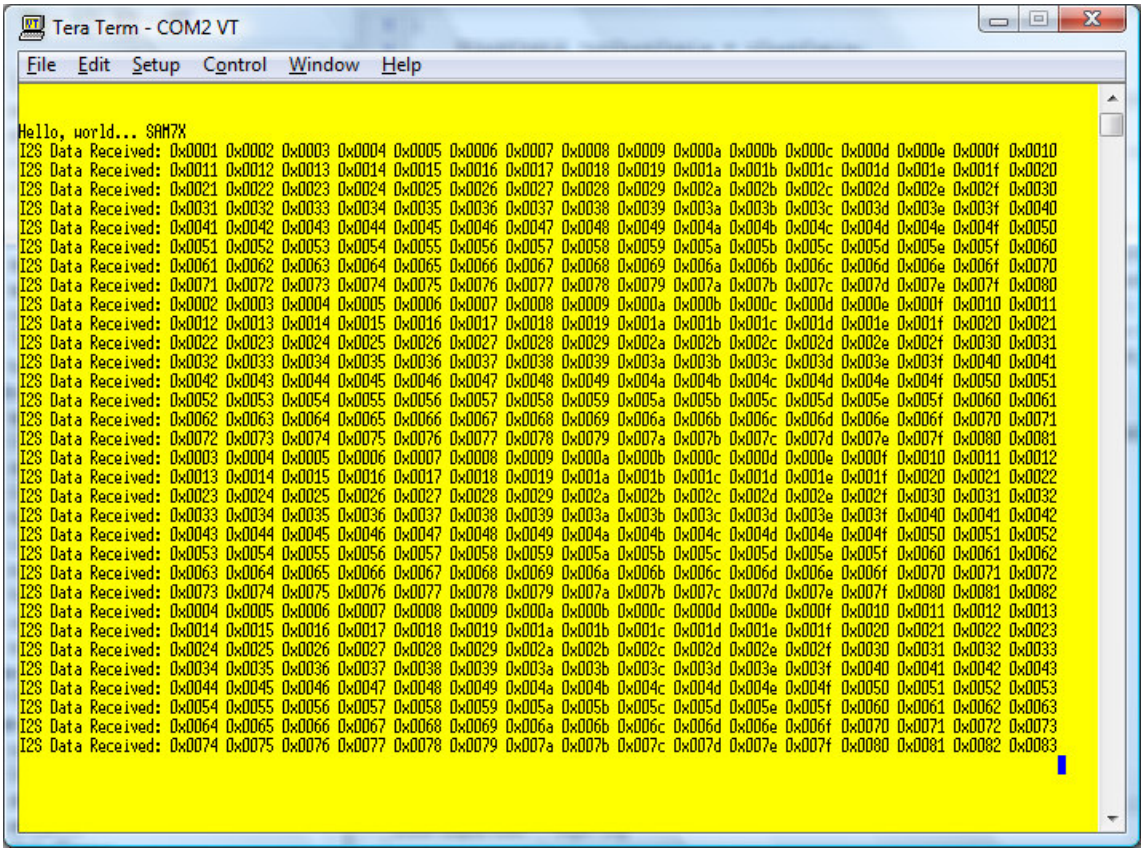
```
static void fnSendSSC_Test(void)
{
    static unsigned short usTest[128] = {0};
    if (usTest[0] == 0x0000) {
        int i = 0;
        while (i < 128) {
            usTest[i] = (i + 1);
            i++;
        }
    }
    else {
        int i = 0;
        while (i < 128) {
            usTest[i] += 1;
            i++;
        }
    }
    fnWrite(I2S_handle, (unsigned char *)usTest, 128);          // transmit test
    uTaskerMonoTimer(OWN_TASK, (DELAY_LIMIT)(2.0*SEC), E_NEXT_SSC_TEST);
}
```

6.2. Receiving Data

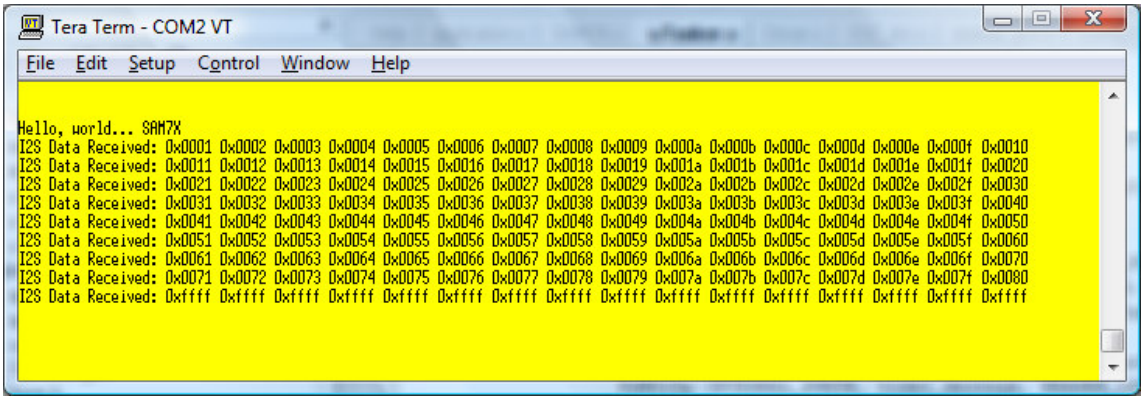
The receiver task is woken each time a frame of data has been received and can read its input buffer. The following shows the reception and display of frames to verify that the transmitted data is received correctly.

```
while (fnRead(I2S_handle, (unsigned char *)usI2C_data, 16) != 0) {
    int i = 0;
    fnDebugMsg("I2S Data Received:");
    while (i < 16) {
        fnDebugHex(usI2C_data[i++], (WITH_LEADIN | WITH_SPACE | 2));
    }
    fnDebugMsg("\n\r");
}
```

The following is the output data seen on the debug interface (after transmission of 5 blocks of 8 frames). The test is performed with the SSC Tx lines connected directly to the SSC Rx lines, where the configuration in section 5 was used. When connected to a remove device the Rx and TX configurations don't need to match and depend on the exact requirements of the remote device.



As discussed previously, the transmitter has to be disabled after a block of frames has been transmitted and cannot always achieve this at high data rates. The following shows a high data rate where a subsequent blank frame is transmitted:



7. SSC Simulation with the uTasker simulator

Data transmitted to the SSC interface is saved to the file SSCx.txt when the project-define `LOG_SSC0` is active, where x is the SSC port number. This allows the transmitted data to be verified.

More useful is generally a method of testing reception frames since software will usually need to be developed which reacts to the content of this data. This is achieved by playing in a simulation script, an example of which is shown below:

```
// Receive I2S test messages from DSP

+0    SSC-0      = 00 0c  01 02   03 04   05 06   07 08   09 0a   0b 0c   0d 0e   0f 10  11 12  13 14  15 16  17 18   19 1a   1b 1c   1d 1e
+100  SSC-0      = 00 0c  02 01   04 03   06 05   08 07   0a 09   0c 0b   0e 0d   10 0f  12 11  14 13  16 15  18 17   1a 19   1c 1b   1e 1d
+100  SSC-0      = 00 0c  03 03   04 05   06 05   08 07   0a 09   0c 0b   0e 0d   10 0f  12 11  14 13  16 15  18 17   1a 19   1c 1b   1e 1d
+100  SSC-0      = 00 0c  04 04   05 04   06 05   08 07   0a 09   0c 0b   0e 0d   10 0f  12 11  14 13  16 15  18 17   1a 19   1c 1b   1e 1d

// End of test
```

The script is loaded and started when the menu command "`Port Sim | Open Script`" or "`Port Sim | Repeat last script`" is executed. *Note that the last script is only saved when the simulator is executed normally and so new scripts do not become valid if the simulator is interrupted instead of closed*.

This example shows four frames of each 32 bytes being received on SSC0; the first frame arrives immediately and the following each with 100ms delay. The script commands can be mixed with other types (such a UART reception, port state changes, etc.) if required.

The result is that 4 frame receptions are executed, which allows the complete reception path (including DMA, the interrupt routine, driver layer and the application handling) to be exercised – by using various reference scripts, typical cases can be simply tested as required, or a complete test suit can be achieved by a single more complex script.

8. Conclusion

The SSC driver for the SAM7X allows the application to simply configure the interface and subsequently send and receive blocks of frame data.

The SAM7X has been found to not be optimally suited to the transmission of bursts of frame data in this format due to the fact that the SSC interface needs to operate in streaming mode. The solution to this problem was to enable the transmitter only when actually sending data, although it was found to be necessary to also allow the TK to run continuously.

There were no difficulties experienced with data reception and a high rate can be achieved by using the double-buffered DMA capabilities of the chip in conjunction with an adequate input buffer length.

Modifications:

V1.00 31.8.2009: First version
V1.02 19.02.2010: Add SCC simulation script