Interfacing the Am188™EM Controller to the DSLAC™/QSLAC™ Devices Using the SSI

AMD

Application Note

The purpose of this application note is to show the user how to interface the Am188™EM microcontroller to the DSLAC™ and QSLAC™ devices using the Synchronous Serial Interface (SSI). These techniques are not restricted to the Am188EM microcontroller; other members of the Am186™ microcontroller family with SSI ports can be interfaced in a similar fashion.

BACKGROUND

Traditionally, line cards (if they had a processor at all) used a simple, inexpensive 8-bit microcontroller. However, as the number of lines per card increases, 16-bit controllers like the Am188™EM microcontroller become more attractive for several reasons:

- Over time 16-bit controller costs have decreased
- More peripheral functions are integrated, reducing external components counts
- Newer, smaller packaging options are available
- 16-bit controllers generally offer larger address spaces

These reasons, combined with the availability of superior, low-cost development tools like Microsoft & Borland C, reduce time-to-market and long-term maintenance costs.

The SLAC™ device connects to the host processor through a 3-pin serial interface. While this interface is used primarily to initialize the SLAC device, several critical functions of the SLIC™ device can be monitored through the serial interface; therefore, it may be necessary to make the interface as fast as possible. Using the SSI port of the Am188EM microcontroller reduces software overhead making the interface much faster.

The serial Microprocessor Interface (MPI) of the DSLAC™ and QSLAC™ devices pre-dates most—if not all—of today's industry standard serial interfaces ports, including the Synchronous Serial Interface (SSI) port of the Am188EM microcontroller. Because these two serial interfaces (MPI and SSI) were not designed to be compatible, it takes a little effort to make them work together. This application note attempts to show that it is worth the effort and explains how to do it.

In most line-card designs, the only cost effective alternative is to interface to the processor's PIOs and manipulate the MPI signal lines directly from software. While this is a perfectly acceptable approach—even desirable in some cases—the use of the SSI port can greatly reduce software overhead and code space.

FURTHER REFERENCES

The remainder of this application note assumes at least a passing familiarity with the chips involved; the Am188EM microcontroller, the Am79C02 DSLAC family, and the Am79Q02 QSLAC family. If additional details are needed, the following literature is available from AMD:

Am186™EM/EMLV and Am188™EM/EMLV Microcontrollers Data Sheet, order #19168

Am186™EM and Am188™EM Microcontrollers User's Manual, order #19713

Am79C02/03/031(A) DSLAC™ Device Data Sheet, order #18503

Am79Q02/021/031 QSLAC™ Device Data Sheet, Available through your local AMD sales office

AMD's complete line of line card devices are found in the *Linecard Products for the Public Infrastructure Market Data Book*, Publication #18503.

MPI HARDWARE OVERVIEW

The QSLAC and DSLAC devices have very similar MPIs; both are serial, master/slave-type interfaces. Differences between the two devices are described in the following paragraphs. A system or line card microprocessor is the master and the interface is designed so that multiple slaves (i.e. SLAC devices) can be attached to a single master's MPI bus.

The MPI signals, like most digital buses, consist of three types of signals:

- Clock/Control
- Address
- Data

The data line (DIO) is a bidirectional, three-state serial bus. The Am79C02 has separate data in (Din) and data out (Dout) pins that can be strapped together to look

Publication# 21728 Rev: A Amendment/0
Issue Date: May 1997

like the other SLAC device's single DIO pin. The data on this line consists of eight-bit bytes transmitted most significant bit (MSB, D7) first, regardless of direction. The master initiates all transfers by sending a command byte to the SLAC device. Each command has a predetermined length (number of bytes) and direction (read or write). For example, if the master microprocessor sends out command number 25 (read GX filter coefficients) to the DSLAC device, the DSLAC device knows to transmit two bytes. Because the command determines what is transmitting, master or slave; it is important to make sure the software drivers are correct to prevent bus contention, which could damage the devices. Also, in the case of a read, the SLAC device will not accept a new command until the old one is finished (i.e. all of the data is clocked out). Software verification is critical.

The clock signal (DCLK) can free run or be active only during data transfers and is an input to the SLAC device. The DCLK maximum frequency is 4.096 MHz for both SLAC devices. Data is clocked into the SLAC device on the rising edge of DCLK, but data is sent out on the falling edge of DCLK. This common technique makes it easier to satisfy setup and hold time requirements. DCLK can be stopped indefinitely in either the High or Low state if the chip select input is held High.

Each of the individual SLAC devices on the MPI bus is addressed (i.e. selected) by pulling one of the chip select inputs Low. The QSLAC device has a single chip select for all four channels while the DSLAC device has a separate chip select for each channel (CS1 and CS2). The rising edge of the chip select marks or frames the end of each byte; therefore, the chip select line *must* go High for at least the minimum off period before the next byte is read or written. The DSLAC device's minimum off period is 5 µs while the QSLAC device's minimum is 2.5 µs.

Finally, the QSLAC device does have an interrupt pin as part of the microprocessor interface. A description of this pin is available in the *Am79Q02/03/031 QSLAC™ Device Data Sheet*.

SSI HARDWARE OVERVIEW

The Synchronous Serial Interface (SSI) on the Am188EM microcontroller was designed to provide a low pin-count interface to application-specific integrated circuits (ASICs). Fortunately, although not by design, it is similar to the SLAC device's MPI. Like the MPI, the SSI is a synchronous, master/slave serial bus protocol that allows multiple slaves on the bus. The maximum clock rate can be as high as 20 MHz.

The SSI bus consists of four signals:

- SCLK
- SDATA
- SDEN0
- SDEN1

Each of these signals is on a separate pin. All of the pins are shared (i.e. multiplexed) with one of the Am188EM microcontroller's 32 PIOs. This allows the SSI pins to be used as PIOs if their normal SSI function is not needed.

The SDATA pin, like DIO, is a bidirectional, three-state serial bus. Unlike DIO, a weak pull-up or pull-down resistor keeps the last value on the bus for systems that cannot tolerate three-state inputs. The data on this pin consists of eight-bit bytes transmitted least significant bit (LSB, D0) first. The master/slave protocol is controlled entirely with software.

The clock signal (SCLK) is only active during byte transfers and is an output. The frequency is derived from the processor's internal clock by dividing it by 2, 4, 8, or 16. In the case of a 40-MHz device, this allows for speeds up to 20 MHz as mentioned above. Like the SLAC device, data is clocked out on the falling edge and clocked in on the rising edge.

The two enable pins (SDEN0 and SDEN1) are outputs and unlike the chip selects of the MPI, they are high-level active. While the state of the SDEN pins is controlled by software somewhat like a PIO, the pin must be high for the interface to transmit or receive.

COMPARING MPI TO SSI

Table 1 summarizes the similarities and differences between the two interfaces.

Table 1. MPI and SSI Comparison

Feature	MPI	SSI	Comment
Word size	8	8	OK
Bit order	MSB first	LSB first	Bit order is reversed
Master/Slave	yes	yes	OK
Max frequency	4.096 MHz	uP	OK
Transmit clock edge	falling	falling	OK
Receive clock edge	rising	rising	OK
Clock inactive state	either	high	OK
Synchronous	yes	yes	OK
CS framed	yes	yes and no	Yes, with software
Enable state	low	high	CS and SDEN are the wrong polarity

SOLVING THE REVERSED-BIT-ORDER PROBLEM

The reversed-bit-order problem is not as severe as it might seem. Generally, there are two types of data sent over the serial interface: coefficients and bit flags.

In most cases, coefficients are generated by AmSLAC™ software and stored as hex values in a table in the microprocessor's non-volatile storage. It should not be difficult to rearrange the bit order prior to placing them in the table, preventing the need for the microprocessor to do it.

The bit flags are used to monitor or set the state of the SLAC devices' I/O pins in real time, but because the bit flags are independent, the host microprocessor's code only needs to know where they are in the byte. Order is not important. Bit reversal should not be significant as it is taken into account by software.

In both cases, it should not be necessary to do bit reversal in the microprocessor while in use.

SOLVING THE ENABLE-POLARITY PROBLEM

The obvious solution is to use inverters on the SDEN outputs, but with a little bit of extra software, it is possible to do without the glue logic. Because the DSLAC device requires two chip selects and the QSLAC device requires one chip select, if inverters are used, this technique can only handle one DSLAC device or two QSLAC devices. If more SLAC devices are required, the easiest solution is to use general PIOs on the Am188EM microcontroller to drive the chip selects of the SLAC devices.

The alternate solution is to use the Am188EM microcontroller's PIO pins instead of SDEN to drive the chip selects. As discussed above, if multiple SLAC devices are on the bus and more than two chip selects are needed, PIOs are required anyway. Always using PIO pins maintains consistency and is the better solution. Software will still need to control the DE0 and DE1 bits as if SDEN0 and SDEN1 were used. However, because their pins are not used, they can be programmed as PIOs and used to drive the chip selects. No pins are wasted.

Using software to drive the SLAC devices' chip selects through PIOs also provides a solution for the polarity problem and CS framing issue. Because software is controlling the state of the PIOs directly, it is trivial to invert the sense of the PIOs relative to SDEN0 or SDEN1. Correctly controlling the chip selects further requires that CS go high after each byte is transmitted or received. The multiple writes and reads illustrated in the *Am186™EM and Am188™EM Microcontrollers User's Manual* in figures 11-5 and 11-6 are not allowed.

Figure 1 shows the resulting connections for a DSLAC device. The QSLAC device is similar, but has only one chip select.

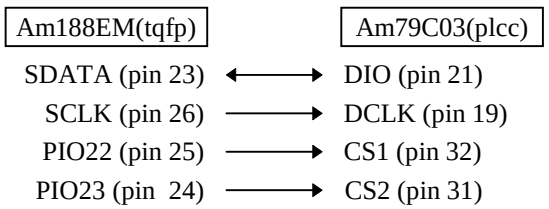

Figure 1. Interface Connections

TIMING CONSIDERATIONS

The following tables compare the timing requirements for the worst case. For each case in the table, the worst case is determined by looking at the most stringent requirement to see if the other end of the interface can meet the requirement. There is only one speed grade for the DSLAC device and the QSLAC device, but there are multiple speed grades of the Am188EM microcontroller, so each case has been looked at with the worst possibility in mind.

For example, in Table 2, the date setup time for the case where the Am188EM microcontroller is driving

data out to the DSLAC device is calculated as described in the following paragraph.

The DSLAC device requirement is read directly from the data sheet parameter #10 (t_{IDS} - Input Data Setup Time). This value is 30 ns for both the DSLAC and QSLAC devices. Determining what the Am188EM microcontroller provides takes a little calculation. First, decide what the minimum low period is for SCLK. This is really determined by the DSLAC device, where the minimum DCLK low period is determined by parameter #3 (t_{DCL} - Data Clock Low Pulse Width), which is 97 ns. SDATA is guaranteed to be stable no more than 25 ns after the falling edge of SCLK by parameter #78 (t_{SLDV} - SCLK Low to Data Valid) for the slowest processor (20 MHz). Subtracting 25 from 97 leaves 72 ns worst-case setup time before the rising edge of DCLK.

The previous example assumes worst-case duty cycle for DCLK. The fastest clock allowed has a period of 244 ns, but if the clock is not perfectly symmetric, either the Low or High period can be as short as 97 ns rather than one half of the clock rate (122 ns). In the case of the Am188EM microcontroller this is probably over-design. Because SCLK's frequency is related to one half CLKOUTA, it should always be close to a 50% duty cycle. In this case, the data setup time provided by the Am188EM microcontroller is closer to 97 ns.

As the tables illustrate, there are generous margins even in the worst case.

In the case of the data hold time in Table 2, there is no specification given in the Am188EM microcontroller data sheet for how quickly SDATA can change after the clock goes Low. It is assumed that the worst case is that SDATA will instantaneously change as soon as SCLK goes Low. This means that the hold time provided by the Am188EM microcontroller is the same as the minimum SCLK High period specified by the DSLAC device as 97 ns.

The CS setup and hold time times are given for the case where SDEN is driven from inverters as discussed above. Even though SDEN is driven by the SSI interface, it is still controlled by software by writing a one or zero to the DE0 or DE1 bits in the synchronous serial control (SSC) register. Because they are controlled by software, the actual delays will be much longer than specified. The same will hold true if PIOs are used to drive the SLAC device chip selects.

Table 3 gives the usable options for each of the available speed grades and the resultant data transfer rate. To achieve the maximum DCLK rate of 4 MHz, the Am188EM microcontroller's internal frequency must be 8, 16, or 32 MHz (÷2, 4, or 8).

Table 2. Microprocessor Output (Data Write)

	DSLAC Device Requires (ns, min)	Am188EM Microcontroller Provides (ns, min)	Comments
Data setup time	30	72	OK
Data hold time	30	97	OK
CS setup time	70	219	OK
CS hold time	0	122	OK

Table 3. Microprocessor Input (Data Read)

	Am188EM Microcontroller Requires (ns, min)	DSLAC Device Provides (ns, min)	Comments
Data setup time	10	47	OK
Data hold time	3	97	OK

SOFTWARE CONSIDERATIONS

The basics of using the SSI port from software can be illustrated with two subroutines; the first subroutine writes a byte to the SLAC device and the second reads a single byte. These two subroutines, along with initialization, form the core of the necessary drivers.

The SSI port appears as five registers in the Am188EM microcontroller's peripheral control block. This 256-byte block can be located in either memory or I/O space at the location pointed to by the Peripheral Control Block Relocation Register. Because the base location of the block can be moved, the location of individual registers is specified as an offset from the Peripheral Control Block Relocation Register rather than an absolute address. The PIO ports and control registers are also located in this block of addresses. At reset, the block is located at 0FF00h in the I/O space.

Table 4. SSI Port Registers

Offset from PCB	Register Mnemonic	Register Name
10h	SSS	Synchronous Serial Status
12h	SSC	Synchronous Serial Control
14h	SSD1	Synchronous Serial Transmit 1
16h	SSD0	Synchronous Serial Transmit 0
18h	SSR	Synchronous Serial Receive

The bit-level definitions from the SSI Port registers from Table 4 are shown in Figure 2.

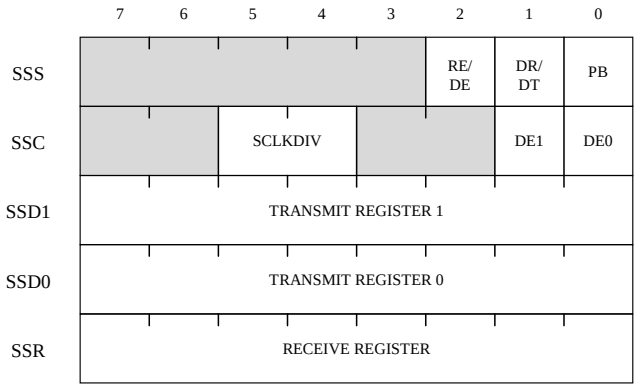

Figure 2. Bit-Level Definition of SSI Port Registers

The Port Busy (PB) bit in the status register goes High when a transmit or receive operation is in progress. The two SDEN enables (DE0 and DE1) control the state of SDEN and enable transmission or reception. A write to the transmit register or a read of the receive register initiates the transfer. For a more complete functional description of these registers, including the features not used here, refer to the *Am186™EM and Am188™EM Microcontrollers User's Manual*.

Assuming the connections in Figure 1, the following steps are required to execute the Read Revision Code Number Command (#23) of the DSLAC device.

Send the command:

1. Enable CS1, set PIO 22 low [PDATA1 bit 6 = 0]
2. Enable transmit, set DE0 high [SSC bit 0 = 1]
3. Write the command, bit reversed [SSD0 = 0CEh]
4. Wait for PB to go low [SSS bit 0 = 0]
5. Disable transmit, set DE0 low [SSC bit 0 = 0]
6. Disable CS1, set PIO 22 high [PDATA1 bit 6 = 1]

After waiting 5 ms, receive the data:

1. Enable CS1, set PIO 22 low [PDATA1 bit 6 = 0]
2. Enable receive, set DE0 high [SSC bit 0 = 1]
3. Start reception [read SSR]
4. Wait for PB to go low [SSS bit 0 = 0]
5. Disable receive, set DE0 low [SSC bit 0 = 0]
6. Disable CS1, set PIO 22 high [PDATA1 bit 6 = 1]
7. Read revision number [read SSR]

Appendix A provides the listings for two general purpose read and write routines. They have been coded in assembly language to maximize speed. In most cases the natural flow of the software will guarantee that there is at least 5 ms between bytes. In the listing given, the print commands between writes and reads take much longer than 5 ms. If this is not the case, either a software delay or the Am188EM microcontroller's timer could provide the necessary wait.

INITIALIZATION

There are two parts to the initialization process. First, the on-board peripheral of the Am188EM microcontroller (PIO and SSI ports) must be set up for proper operation. This includes setting the mode and direction of the PIO pins as well as setting the pin itself to a known state. Second, now that the interface is operational, the SLAC device itself should be initialized. Each of the SLAC devices have a recommended power-up sequence that can be found in the data sheet. As an example, the DSLAC device's recommended sequence is as follows:

1. Select MCLK (command #6),
2. Software reset (command #2),
3. Program coefficients and parameters,
4. Activate (command #5)

The Am188EM microcontroller's PIO and SSI port should be initialized as soon as possible after reset to ensure the output pins are in the correct state, but 1 ms is needed after power is stable before commands can be sent to the SLAC device. Because the Am188EM microcontroller also requires 1 ms for PLL lock (parameter #61 - t_{LOCK}), most systems have an external power-up-reset monitor that provides this delay. If not, or if the systems have separate power supplies, software must wait before sending the first command. The QSLAC device has a power interruption flag (P1, command 23 bit 7) that should be checked after the delay.

SOFTWARE LISTING

The software listing in Appendix A is written in C and compiled with Microsoft's C/C++ compiler. This example code illustrates how to read the DSLAC device's Z filter coefficients. The software was tested on several of the evaluation boards available from AMD. An ASLAC™ Interface Board (ACIF) was used to load a known set of coefficients into a DSLAC Device Low Noise Board. An SD186EM demonstration board was then connected in place of the ACIF Board to read back the coefficients.

The main body of the program first initializes the various ports, then sends a "read Z filter" command. As noted previously, the command must be bit reversed. The "for" loop then reads back the 14 bytes of the Z filter coefficients. The subroutines SSI_read and SSI_write are written in assembly language for speed and clarity. They implement the code required to send or receive a single byte. Once again, the bytes coming back from the DSLAC device are bit reversed.

SUMMARY

While the two serial interfaces were not designed to be used together, they are surprisingly compatible. The two simple fixes required to make them work together are worth the effort to save code space and speed up operation.

Software Listing

```
#include <stdio.h>
#include "sys\types.h"
#include "sd186em.h" // defines register addresses

// function prototypes

  void  SSI_init(void);
  void  SSI_write(int);
  Uint8 SSI_read(void);

// useful constants

  #define   READ_Z    0xA1     // read z-filter (bit reversed)
  #define   P22_LOW   0xFFBF   // PIO 22 low bit mask
  #define   P22_HI    0x0040   // PIO 22 high bit mask
  #define   DE0_LOW   0xFFFE   // DE0 bit low bit mask
  #define   DE0_HI    0x0001   // DE1 bit high bit mask
  #define   PB_HI     0x0001   // PB bit high bit mask

void main() {

   Uint8   buf[256];
   int     i;

   printf("Start Program\n");

   SSI_init();
   printf("Finish SSI_init\n");

   for (i=0; i<256; i++) buf[i] = 0;
   printf("Finish buffer initialization\n");

   SSI_write(READ_Z);
   printf("Command Sent\n");
```

```
   for (i=0; i<14; i++) {
      buf[i] = SSI_read();
      printf("byte %d = %x \n",i,buf[i]);
      }  /* end for loop */

exit(0);
}
//****************************** END OF MAIN ******************************

void SSI_init(void)
{
    _asm{

     mov  dx,PIOMODE1  // point to mode register
     in   ax,dx        // read register (in IO space)
     or   ax,0x00C0    // set bits low (to make output)
     out  dx,ax        // write to register (in IO space)

     mov  dx,PDIR1     // point to PIO1 DIRECTION register
     in   ax,dx        // read
     and  ax,0xFF3F    // set PIO direction bit (to output)
     out  dx,ax        // write

     mov  dx,PDATA1    // point to PIO1 DATA register
     in   ax,dx        // read
     or   ax,0x00C0    // set PIO data bits (to make them high)
     out  dx,ax        // write

     mov  dx,SSC       // point to sync serial control register
     mov  ax,0x0030    // Set clk divisor to 16, enables low (inactive)
     out  dx,ax        // write

     } /* end _asm */


}

//============================ END OF SSI_INIT ============================
```

```
Uint8 SSI_read(void)
{

    int i;

    _asm{

     mov  dx,PDATA1    // STEP 1 - set CS1 low (PIO 22)
     in   ax,dx        //
     and  ax,P22_LOW   //
     out  dx,ax        //

     mov  dx,SSC       // STEP 2 - enable reception
     in   ax,dx        //          (i.e. set bit DE0 = 1)
     or   ax,DE0_HI    //
     out  dx,ax        //

     mov  dx,SSR       // STEP 3 - start reception
     in   ax,dx        //          (with dummy read of SSR)

     mov  dx,SSS       // STEP 4 - wait for data
h1:  in   ax,dx        //          (done when PB = 0)
     and  ax,PB_HI     //
     jnz  h1           //

     mov  dx,SSC       // STEP 5 - disable reception
     in   ax,dx        //          (set DE0 low)
     and  ax,DE0_LOW   //
     out  dx,ax        //

     mov  dx,PDATA1    // STEP 6 - set CS1 high (PIO 22)
     in   ax,dx        //
     or   ax,P22_HI    //
     out  dx,ax        //

     mov  dx,SSR       // STEP 7 - read the data
     in   ax,dx        //
```

```
     mov  i,ax          // move data to output variable

     } /* end _asm */

     return(i);

}

//============================ END OF SSI_READ ============================

static void SSI_write(i)

int i;
{


    _asm{

     mov  dx,PDATA1    // STEP 1 - set CS1 low (PIO 22)
     in   ax,dx        //
     and  ax,P22_LOW   //
     out  dx,ax        //

     mov  dx,SSC       // STEP 2 - enable transmission
     in   ax,dx        //          (i.e. set bit DE0 = 1)
     or   ax,DE0_HI    //
     out  dx,ax        //

     mov  dx,SSD0      // STEP 3 - transmit byte
     mov  ax,i         //
     out  dx,ax        //

     mov  dx,SSS       // STEP 4 - wait for completion
h1:  in   ax,dx        //          (done when PB = 0)
     and  ax,PB_HI     //
     jnz  h1           //
```

```
     mov  dx,SSC       // STEP 5 - disable transmission
     in   ax,dx        //          (set DE0 low)
     and  ax,DE0_LOW   //
     out  dx,ax        //

     mov  dx,PDATA1    // STEP 6 - set CS1 high (PIO 22)
     in   ax,dx        //
     or   ax,P22_HI    //
     out  dx,ax        //

     } /* end _asm */


}

//========================== END OF SSI_WRITE ==============================
```