SHARC® SPI Slave Booting

Contributed by Matt Walsh and Brian Mitchell

Rev 3 – January 19, 2007

Introduction

Upon power-up, SHARC® processors can be boot-loaded as an SPI slave device. Although the hardware interface is simple, a complication arises when using this boot scheme. The SPI protocol does not provide an obvious way for the booting SPI slave to "pause" the boot-stream being transmitted by the SPI host device. This causes a problem when SPI booting because when performing "zero-initializations", the processor executes code rather than emptying the `SPIRX` buffer of the slave processor.

To save space in the `.LDR` file, sections of memory with a large number of contiguous zeros are truncated: Instead of storing these zeros in the `.LDR` file, the loader only includes the destination address and the number of zeros required. (Consecutive zeros often occur because of unitialized data arrays or several (5) consecutive `NOP` instructions, the opcode for which is `0x000000000000`.) Upon identifying one of these zero-init sections, the kernel simply runs a small loop to write zeros to each specified address. If the core stays in this loop too long, the SPI host device continues to transmit, then the 2-deep SPI FIFO may overflow, and data may be lost. This EE-Note describes three ways to avert this potential problem.

This document applies to all SHARC processors that support SPI slave boot: ADSP-2116x, ADSP-2126x, ADSP-2136x, and ADSP-2137x.

The hardware and the software solutions discussed throughout this EE-Note are for the ADSP-21161 processor, however, the same techniques can be used for ADSP-2116x, ADSP-2126x, ADSP-2136x, and ADSP-2137x devices. Example code for the ADSP-21161, ADSP-21266, ADSP-21365, and ADSP-21369 processors are provided in the associated `.ZIP` file.

Hardware Solution

There are two ways to pause the SPI host device by implementing a hardware feedback scheme: using a fifth pin as a `FLAG` signal, or using the `MISO` pin. In both scenarios, the processor drives a hand-shake signal back to the SPI host. This signal should be connected to a flag or interrupt pin on the SPI host. The SPI host would monitor this flag or interrupt pin, and would pause transmission when necessary. Similarly, the feedback signal may also be interpreted by the host as a "start transmission" signal. Regardless of the protocol, implementing a solution in hardware requires a small change to the loader kernel, `161_SPI.asm`, as well.

Described below is the former case, where the feedback signal indicates that the SPI host should "pause" transmission. In the loader kernel, the small sections of code that perform the zero initializations are simply modified to drive an output flag pin low until it finishes executing the zero-initialization loop. The boot kernel code before and after modification can be seen in Listing 1 and Listing 2.

```
DM_zero:
R0=R0-R0, I0=R3; //r0=0& I0=address
                 //as held in R3.
LCNTR=R2, DO dm_zero.end UNTIL LCE;
dm_zero.end:
DM(I0,M6)=R0;
JUMP read_boot_info;
```

Listing 1. Standard zero-init code from SPI boot-loader kernel - 161_SPI.asm

```
#include <def21161.h>
R0=R0-R0,I0=R3; //r0=0 & I0=address
                //as held in R3.
USTAT1=DM(IOFLAG);
BIT SET USTAT1 FLG4O; //make output
dm(IOFLAG)=USTAT1;
BIT CLR USTAT1 FLG4;  //flag4 = low
dm(IOFLAG)=USTAT1;

LCNTR=R2, DO dm_zero.end UNTIL LCE;
dm_zero.end:
DM(I0,M6)=R0;
BIT SET USTAT1 FLG4; //flag4 = high
dm(IOFLAG)=USTAT1;
JUMP read_boot_info;
```

Listing 2. Zero-init code with hardware feedback added using flag 4

This EE-Note includes a version of the kernel modified in the manner described in this section and a project to be used on an ADSP-21161 host processor to boot a slave in this manner. Note that when the host is “paused”, ensure that the transmission stops only after an entire word has been sent. In the case of ADSP-21161 processors, this causes the host to resend the last two words before the pause occurred.

Alternatively, as shown in Figure 1, the slave processor’s MISO pin may be connected to the flag/interrupt pin of the SPI host. The processor’s SPI port may then transmit all 1s or all 0s, depending on the implemented protocol. The SENDZ bit of the SPICTL register determines what is sent when the processor's transmit buffer is not refilled. If SENDZ is cleared, the SPI port will continue to retransmit the last word written to the SPITX buffer. If SENDZ is set, the MISO pin will be driven low (sending zeros) until a new word is written to the SPITX buffer.

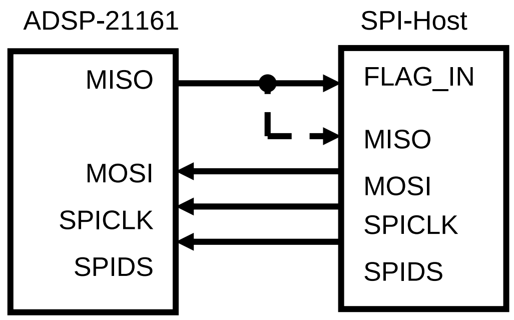

Figure 1. Hardware feedback mechanism using the processor's MISO pin

The modifications to the loader kernel would be similar to those in Listing 2. The actual code depends on the host and whether logic high or logic low means “pause” or “resume”. In either case, the code writes a data-word to the SPITX register, rather than toggling a flag. This word may be all ones, all zeros, or some other mix to shape the waveform accordingly.

This second implementation (where the MISO pin is used as a handshake signal *and* as a data-line) requires software modifications on the SPI host side. This software would need to be configured to monitor the flag-in (handshake) pin in certain circumstances (during booting), and then ignore it in others (when receiving data over the MISO pin).

Software Solution

Alternatively, the SPI host device may insert pauses on its own, without any feedback from the booting slave device. The .LDR file, which is produced by the VisualDSP++® loader, is divided into multiple “initialization sections”. Each piece of contiguous, homogeneous data from the executable is grouped together and given a three-word descriptive header. The header and following data comprise an “initialization section”. The loader kernel uses the initialization header to correctly initialize the subsequent data. The initialization header can also be used by the SPI host to throttle the boot-stream when necessary. The header is made of three words: the word count, the destination address, and a tag identifying the data-type. Listing 3 shows the 27 different data-types.

```
0x00 FINAL_INIT
0x01 ZERO_DM16
0X02 ZERO_DM32
0x03 ZERO_DM40
0x04 INIT_DM16
0x05 INIT_DM32
0x06 INIT_DM40
0x07 ZERO_PM16
0x08 ZERO_PM32
0x09 ZERO_PM40
0x0A ZERO_PM48
0x0B INIT_PM16
0x0C INIT_PM32
0x0D INIT_PM40
0x0E INIT_PM48
0x0F ZERO_DM64
0x10 INIT_DM64
0x11 ZERO_PM64
0x12 INIT_PM64
0x13 INIT_PM8_EXT
0X14 INIT_PM16_EXT
0X15 INIT_PM32_EXT
0X16 INIT_PM48_EXT
0X17 ZERO_PM8_EXT
0X18 ZERO_PM16_EXT
0X19 ZERO_PM32_EXT
0X1A ZERO_PM48_EXT
```

Listing 3. Loader tag reference

Listing 4 shows a typical initialization header.

```
0x0000000e // tag
0x00000013 // count
0x00040100 // addr
<data0>
<data1>
```

Listing 4. Example initialization header

Interpreting this header, the kernel would recognize that there are 19 (`0x13`) instructions that are located at address `0x40100`. (The tag of `0x0000000E` signifies non-zero 48-bit PM data).

One way to implement a software solution requires the SPI host device to read this header. Instead of streaming the entire `.LDR` image in one shot, it sends it one initialization section at a time. It would read the init-header and send the appropriate number of words. The word count included in each init-header can be used by the SPI host to calculate how many words to “skip ahead” in the `.LDR` file to find the *next* init-header. When a header contains the tag for one of the zero-initialization sections, the host should pause after sending the header. Again, this is because these sections of contiguous zeros are going to be initialized in software by the loader kernel. The length of the required pause depends on the size of the initialization section (the number of zeros to be initialized) and the SPI baud rate. It takes one core cycle (10ns minimum) to for each zero initialization. If the SPI port runs at 10 MHz (100 ns), it would take 320 core cycles for a single 32-bit word to arrive over the SPI port. A `SPIRX` buffer overflow occurs when more than two SPI words are received.

Summary

This EE-Note discusses potential problems that may arise during SPI slave booting and provides hardware and software solutions for it. The example code is provided for the hardware solution. The modified boot kernels for the ADSP-21161, ADSP-21266, ADSP-21365, and ADSP-21369 processors and the SPI host project files are available in the associated `.ZIP` file.

References

[1] *ADSP-21161 SHARC Processor Hardware Reference.* Rev 4.0, February 2005. Analog Devices, Inc.

[2] *ADSP-2126x SHARC Processor Peripherals Manual.* Rev 3.0, December 2005. Analog Devices, Inc.

[3] *ADSP-2136x SHARC Processor Hardware Reference for the ADSP-21362/3/4/5/6 Processors.* Rev 1.0, October 2005. Analog Devices, Inc.

[4] *ADSP-21368 SHARC Processor Hardware Reference.* Rev 1.0, September 2006. Analog Devices, Inc.

[5] *VisualDSP++ 4.5 Loader and Utilities Manual, Revision 2.0, April 2006. Analog Devices, Inc.*

Document History

Revision	Description
Rev 3 – January 19, 2007 by Mallikarjun Reddy and Jeyanthi Jegadeesan	Made the EE-Note generic for the ADSP-2116x, ADSP-2126x, ADSP-2136x, and ADSP-2137x SHARC Processors. Updated code for VisualDSP++ 4.5.
Rev 2 – February 27, 2003 by Brian Mitchell	Included code for both the kernel and the host.
Rev 1 – November 5, 2002 by Matt Walsh	Initial Release.