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.

Putting ADSP-BF54x Blackfin® Processor Booting into Practice

Contributed by Andreas Pellkofer

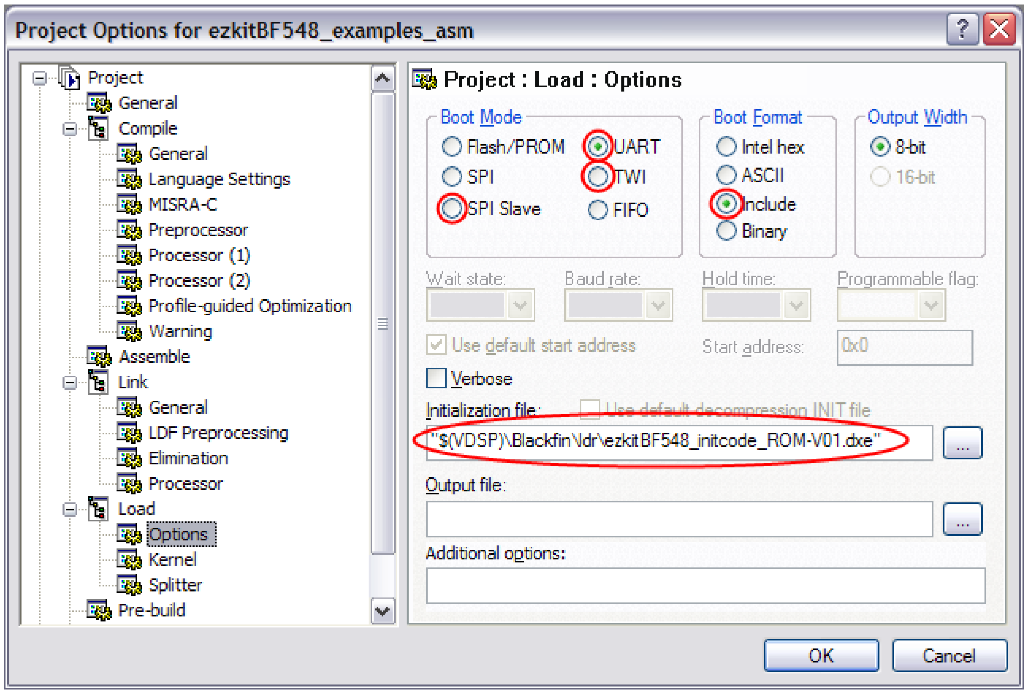
Rev 1 – June 25, 2008

Introduction

ADSP-BF54x Blackfin® processors provide multiple ways to boot the processor. Some are known from older Blackfin derivatives, some have updated features, and some are completely new.

For conceptual information and descriptions of booting modes, refer to the Booting chapter in the processor's *Hardware Reference [1]*. This document provides the following references and helps you get started:

- Slave boot modes like SPI slave, TWI slave, and UART slave
- A software example for the host device (which is another ADSP-BF548 processor for the purpose of this demonstration), including loader files
- Records of the booting process in `ala` format. See *[4]* for the required software.
- Description of special functions in the initialization code examples *[5]*



Project : Load : Options

Boot Mode: Flash/PROM, SPI, SPI Slave, [✓ UART], TWI, FIFO

Boot Format: Intel hex, ASCII, [✓ Include], Binary

Output Width: [✓ 8-bit], 16-bit

Wait state:
Baud rate:
Hold time:
Programmable flag:
[✓ Use default start address]
Start address: 0x0
[ ] Verbose
Initialization file: [ ] Use default decompression INIT file
"$(VDSP)\Blackfin\ldr\ezkitBF548_initcode_ROM-V01.dxe"
Output file:
Additional options:

Figure 1. VisualDSP++ project options (Project -> Project Options -> Load -> Options)

Boot Host Wait (HWAIT) Signal

The boot host wait (HWAIT) signal is asserted by the boot kernel when the processor is not ready to receive data. It is available in all boot modes, by default. This signal is a GPIO, which is handled very conservatively by the boot code without encountering the receive fill status of a peripheral buffers. This means that even if there is free space in the peripheral's buffer to receive data, this signal is asserted when the boot kernel is busy.

The HWAIT GPIO signal must be pulled to its active state by a resistor to prevent the host device from starting the boot process while the Blackfin processor is still held in reset. The resistor also defines the polarity of the signal.

If a Blackfin processor is the host device, the transfer should be done by the core only as the Blackfin DMA architecture does not enable the core to pause a running DMA transfer. The HWAIT signal can be evaluated every single word or an interrupt can be triggered if the signal is asserted. Once asserted, the host must pause immediately after having sent the current word.

With the new hardware flow control mechanism introduced with the ADSP-BF54x UART module, evaluating the HWAIT signal is optional when the host device properly deals with the RTS/CTS handshake. Refer to the Booting chapter in *[1]* for more information.

SPI Slave Boot Mode (BMODE = 0100)

For SPI slave mode boot, the Blackfin processor receives boot data from an external host device connected over SPI. Refer to *[1]* for more details on the general setup and configuration.

Figure 2 shows the start of a booting sequence using the SPI interface. After the first bytes, some calculations on the Blackfin SPI slave device hold up the device from receiving more data. That's why the HWAIT signal is asserted. The SPI master device does not stop sending data immediately, but finishes the current data word. After HWAIT is de-asserted, the booting process continues (for demonstration purposes, a delay loop implemented in the initialization code is executed during the booting process).

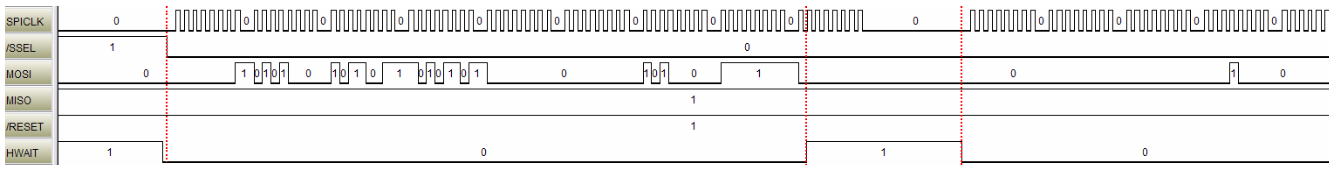


Figure 2. Beginning of SPI slave booting

TWI Slave Boot Mode (BMODE = 0110)

For TWI slave mode boot, the Blackfin processor receives boot data from an external host device connected over TWI. Refer to *[1]* for more details on the general setup and configuration.

The Blackfin TWI slave device behaves similarly to the SPI device. Figure 3 shows the beginning of the TWI slave booting sequence, whereas Figure 4 shows how the TWI slave device can stop the host to send more data (for demonstration purposes, a delay loop is implemented in the initialization code executed during the booting process).

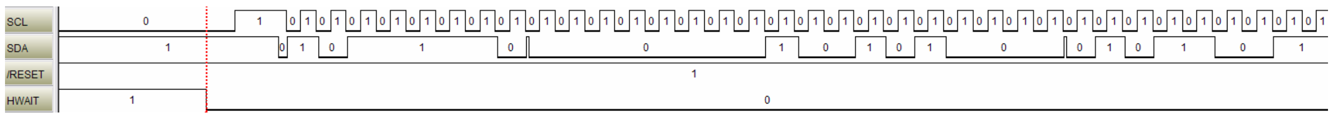


Figure 3. Beginning of TWI slave booting

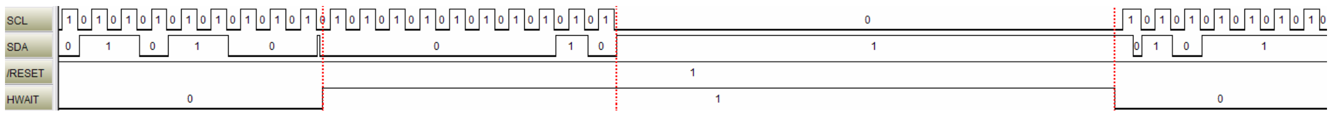


Figure 4. Initialization code execution with delay

UART Slave Boot Mode (BMODE = 0111)

For UART slave mode boot, the Blackfin processor receives boot data from an external host device connected over UART. Refer to *[1]* for more details on the general setup and configuration. During the booting UART host booting process, a few critical moments must be handled properly.

Bit Rate: Handling System Clock Changes

The initial action taken is the autobaud sequence that sets the UART bit rate in relation to the system clock speed. Unlike other slave boot modes (e.g., SPI or TWI), the slave device requires the correct setting for the UART bit rate. Figure 5 shows the autobaud detection sequence at the very beginning of a transfer. Once the `@` (0x40) autobaud detection byte has been received, the boot kernel returns a `0xBF`, `DLL`, `DLH`, `0x00` sequence on its transmit output. Finally, `HWAIT` is released.

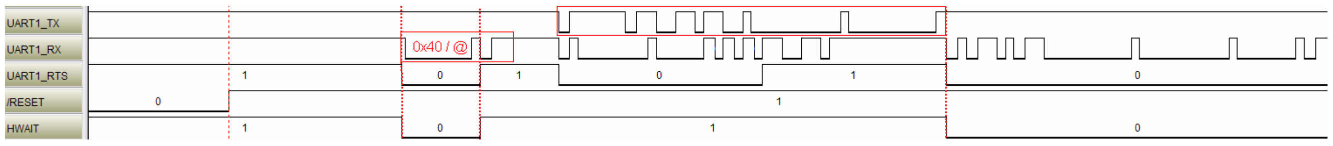


Figure 5. UART: Autobaud detection

Flow Control

The UART slave can signal the host to pause any on-going transfer in two different ways: `HWAIT` and UART hardware flow control (`UART RTS`).

Figure 6 shows the behavior when initialization code delays the boot process by transmitting status words back to the host device, which, in this case, relies on the `HWAIT` signal. Once `HWAIT` asserts, the host finishes the current data word and then waits before sending more data (see `UART1_TX`). That is similar to the other slave boot modes described here. Figure 5 also shows some activity on the transmit line before `HWAIT` releases and the host processor resumes operation.

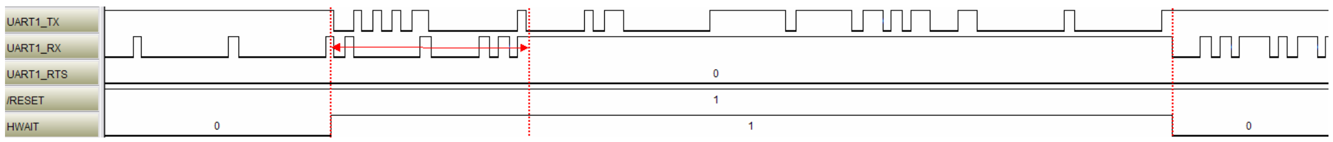


Figure 6. UART: HWAIT signalling

Another way to pause an on-going transfer is via the UART hardware flow control mechanism. This feature is implemented on the ADSP-BF54x UART for the first time and is not available on other Blackfin derivatives at this time. It uses the `UART RTS` signal. If the host device supports this feature, no additional signal need be connected. The behavior on the host side is similar; if `RTS` is asserted, again the host transmits the current word only and pauses immediately after the transmission has been completed. Figure 7 shows the behavior: the host stops to send (see `UART1_TX`) more data after `UART1_RTS` has been de-asserted (`RTS` is active-low logic). This occurs when a specific fill status in the buffer has been reached.

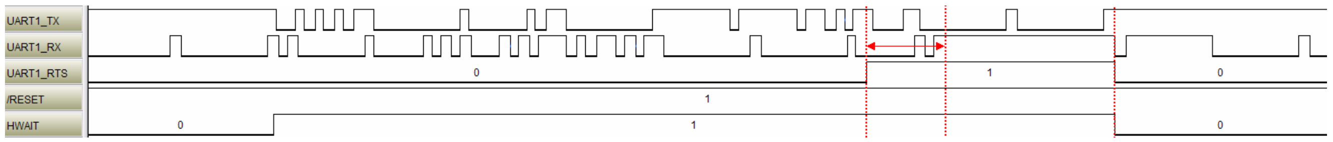


Figure 7. UART: RTS signalling

Figure 6 and Figure 7 show that the slave device is sending data words to the host. For full-duplex connections, the time overlap is not an issue. For half-duplex connections, where the data line is shared, the programmer must take care of this (see *VisualDSP++® initcode Examples*).

To properly resolve a reset situation, a pull-up resistor on the `RTS` output is recommended.

VisualDSP++® initcode Examples

The initcode examples that ship with the VisualDSP++ 5.0 (Update 3) *[5]* installation changes the PLL frequency on the fly and pays close attention to the following two challenges:

- Updating the UART bit rate divider when the system clock has changed during booting process
- Half-duplex transfer requirements by taking control of `UART RTS` signaling

If the PLL is changed during the booting process (e.g., initialization code), special care must be taken by the programmer; otherwise, the boot process may fail. The UART controller requires that the correct bit rate be set in the `UARTx_DLL` and `UARTx_DLL` registers to interpret the incoming data properly. If the system clock changes, the bit rate dividers must be adjusted. The initcode examples that ship with VisualDSP++ 5.0 (Update 3) development tools and later take care of this with two functions. The functions, which are called `u32 uart1_get_bitrate(void)` and `void uart1_set_bitrate(u32)`, are executed before and after the PLL change sequence, respectively. They are called automatically if `BMODE`=0111 is detected. These functions save (`u32 uart1_get_bitrate(void)`) the current UART bit rate, calculate the new UART divider value that fit the saved bit rate, and store it in the `UARTx_DLL` and `UARTx_DLH` registers (`void uart1_set_bitrate(u32)`).

Function `void uart1_set_bitrate(u32)` provides feedback to the host by sending back some bytes (like the boot kernel does after autobaud detection):

- `0xAD`
- `UART1_DLL`
- `UART1_DLH`
- `0x00`

You can see this on the `UART1_TX` signal in Figure 6 to Figure 9.

Additionally, a critical moment is the PLL reprogramming sequence. There is a time delay between the PLL change and the renewing of the UART divisor. You must ensure that during this reprogramming phase the UART host does not send any more data.

The function `u16 uart1_check_buffer(u32)` forces the UART to assert the `RTS` signal by disabling hardware flow control and setting `RTS` manually. This is done in the `UARTx_MCR` register. After this, the host may send the very last byte, for which you must wait. As this function is sending out a 4-byte feedback sequence, there is a quite safe time slot:

- `0xAA`
- `UART1_MSR`
- `UART1_LSR`
- `0x00`

After this the PLL reprogramming sequence is running and after setting the new UART divisor value, the control for the `RTS` signal is passed back to the UART controller (re-enable hardware flow control). The original settings of the `UARTx_MCR` are restored. You can also see that `HWAIT` is de-asserted at the same time.

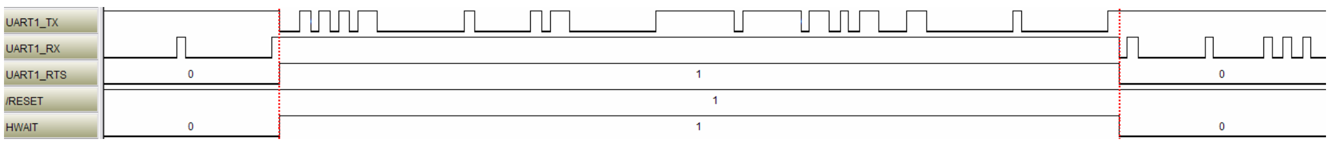


Figure 8. Manipulation of hardware flow control

For a half-duplex connection, there might still be an overlap. Therefore, you can insert a delay (e.g., a length of one data word). This should ensure that there will be no conflict on the line. Figure 9 shows the same sequence as Figure 8 but with the included delay.

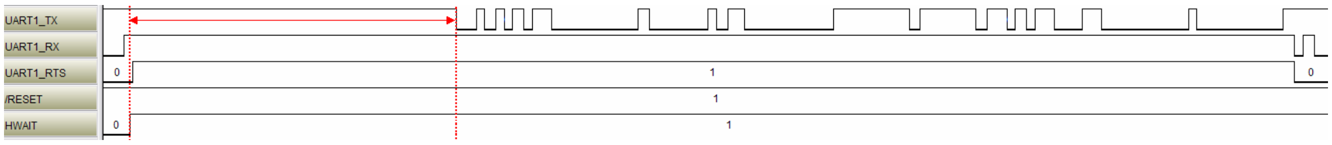


Figure 9. Manipulation of hardware flow control with additional delay

If the host sends the last word delayed, you must manipulate your booting process to fit all possible conditions.

Appendix A

A `ZIP` file is associated with this document. It contains a project with code examples for an ADSP-BF548 EZ-KIT Lite® board and supports the following boot modes as a host device:

- SPI slave boot
- TWI slave boot
- UART slave boot

The project is programmed and tested with VisualDSP++ 5.0 (Update 3) development tools. The basic configurations are set in `configurations.h`. Refer to `README.txt` for additional information.

Appendix B

A `ZIP` file is associated with this document. It contains some recordings of the specific booting processes. The files are for Agilent Technologies logic analyzer application software *[4]*. The code examples from Appendix A have been used. Refer to `README.txt` for more information.

References

[1] *ADSP-BF548 Blackfin Processor Hardware Reference.* Rev 0.3, May 2008. Analog Devices, Inc.

[2] *ADSP-BF548 Blackfin Processor Peripheral Hardware Reference Manual.* Rev 0.1, March 2007. Analog Devices, Inc.

[3] *ADSP-BF548 Blackfin Processor Evaluation System Manual.* Rev 1.2, April 2008. Analog Devices, Inc.

[4] Agilent Technologies 16900, 16800, and 1680/90 Series Application Software.

[5] VisualDSP++ 5.0 (Update 3): Initialization code examples (*<install_path>\Blackfin\ldr\init_code\*)

Readings

[6] *ADSP-BF53x/ADSP-BF56x Programming Reference.* Rev 1.2. February 2007. Analog Devices, Inc.

[7] *EE-240: ADSP-BF533 Blackfin® Booting Process.* Rev 3, January 2005. Analog Devices, Inc.

[8] *EE-331: UART Enhancements on ADSP-BF54x Blackfin® Processors.* Rev 1, November 2007. Analog Devices, Inc.

Document History

Revision	**Description**
Rev 1 – June 25, 2008 by Andreas Pellkofer	Initial release.