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.

Using the Blackfin® Processor SPORT to Emulate a SPI Interface

Contributed by J.Galindo and Joe B.

Rev 1 – November 10, 2006

Introduction

The Blackfin® family of embedded processors supports numerous methods of booting application code, including serial booting from an SPI flash memory device. The Blackfin SPI interface is limited to a maximum clock rate of one-quarter the system clock (`SCLK`) rate. If the `SCLK` is maximized to 133 MHz, this translates to an SPI performance limitation of ~33 MHz. However, there are serial flash devices capable of speeds higher than this, and it is possible to achieve serial clock speeds of up to ~66 MHz if one of the Blackfin serial ports (SPORTs) is used instead, as the maximum SPORT clock frequency is `SCLK`/2.

This EE-Note describes how to use the Blackfin SPORT to emulate an SPI interface and how to then use that emulated hardware to boot an application from an SPI memory device using the SPI boot mode of the Blackfin processor. To make the process work, one must first understand the boot process of Blackfin processors and have a fundamental understanding of the boot image (`.ldr` file) expected by the Blackfin boot ROM. These concepts are discussed in *ADSP-BF533 Blackfin Booting Process (EE-240)*[1].

This application was tested using the VisualDSP++ 4.5® development tools and the ADSP-BF537 EZ-KIT Lite® evaluation system, but the concepts discussed apply to all members of the Blackfin processor family.

Emulating the SPI Boot Process

If the intent is to boot from an SPI device whose operating frequency exceeds the maximum Blackfin SPI frequency of `SCLK`/4, the SPORT interface may be an attractive alternative, as it can run twice as fast as the SPI. However, since SPORT booting is not supported by the Blackfin processor boot ROM, the SPORT must be configured/connected such that it can successfully communicate with the SPI device and have the ability to execute the boot process normally automated by the boot ROM for the SPI interface. This requires specific use of hardware and configuration software, as well as modification to the boot process itself.

Configuring/Connecting the Hardware

The master SPI boot mode is used for this application. The master mode SPI interface requires four signals: data in (`MISO`), data out (`MOSI`), clock (`SCK`), and chip-select (`/SPISS`). Three of these four signals have evident correlation to SPORT pins:

- `MOSI` is the transmit data (`DTxPRI`)
- `MISO` is the receive data (`DRxPRI`)
- `SCK` is the clock (`TSCLKx`/`RSCLKx`)

The Blackfin SPORTs support both primary and secondary transmit/receive data channels. The primary pins (denoted by the `PRI` suffix) are utilized when no secondary channel is required.

The Blackfin SPORTs also support dedicated clock sources for both the transmitter and receiver. In this application, the SPORT's transmit and receive logic are interfaced to the same SPI memory device; therefore, the transmitter and receiver clocks are shared, and are thus connected together externally.

The pin in the SPI interface that has no equivalent pin on the SPORT is the chip-select signal, `/SPISS`. The `/SPISS` is the gating factor for the `SCK`. As a slave SPI device, action is only taken on `SCK` edges detected while the `/SPISS` signal is being held active low by the bus master. Most master SPI devices supply the `SCK` signal only when they are placing data on the `MOSI` line, but extraneous `SCK` transitions are ignored if the `/SPISS` signal is transitioned to inactive high before the extra `SCK` pulses are active. This built-in behavior is helpful because the SPORT employs a continuous clock once the hardware has been enabled and communications have begun. The challenge is to make the SPORT hardware handle the appropriate timing and control of the `/SPISS` signal. To address this need, the frame sync signals in the SPORT interface are used. Figure 1 depicts how the SPORT and SPI pins of the Blackfin processor should be connected to each other and to the SPI memory device.

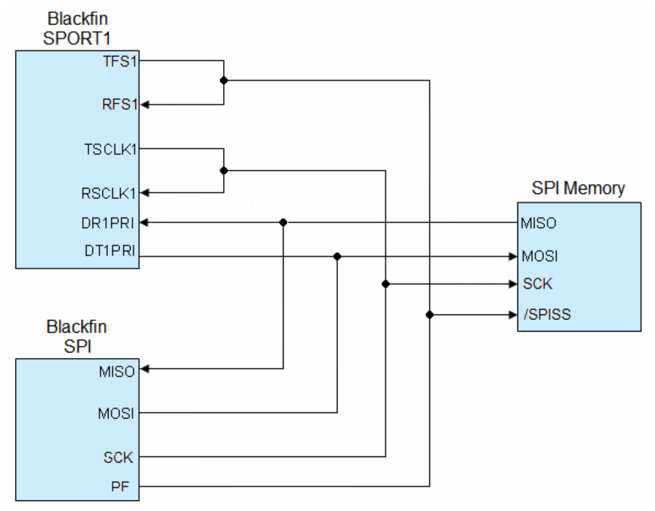


Figure 1. SPORT/SPI Hardware Connections

If the SPORT is configured to generate an active low, late frame sync (as is required for every word transmitted), the behavior of the transmit frame sync pin (`TFSx`) is comparable to the `/SPISS` signal. For internally generated, active low, late framing, the `TFS` signal is asserted low in the same SPORT clock cycle as data is placed on `DTxPRI`, and the signal is held low for the duration of the word being transmitted. `TFSx` is then de-asserted unless new data is ready to be transmitted, in which case it is held in the active low state. This is precisely the way the `/SPISS` signal would be managed by an SPI master device. The following assembly source code can be used to configure the SPORT:

```
P1.H  = HI(SPORT1_TCR1);
P1.L  = LO(SPORT1_TCR1);
R3.L  =
TCKFE|LATFS|LTFS|TFSR|ITFS|ITCLK;
W[P1] = R3;
```

The equivalent C code would be:

```
*pSPORT1_TCR1 =
TCKFE|LATFS|LTFS|TFSR|ITFS|ITCLK;
```

Booting Via the SPORT Interface

As stated previously, there is no native support built into the boot ROM for booting over the SPORT. Achieving this functionality requires the boot sequence to begin booting over SPI, as configured by the `BMODE` pins, and then transfer control to the SPORT to continue the boot process over the faster SPORT interface.

To do this, there is a need for a secondary piece of software, called a *second-stage loader* (SSL). An SSL is simply a kernel that the boot ROM loads and executes in place to complete the boot process. This causes the boot ROM to load the SSL code over SPI, at which point the boot ROM is exited and execution starts at the beginning of the SSL code section in memory. In the `.ldr` file, the SSL resides between any needed initialization code to set up external memory and the actual application code, as shown in Figure 2.

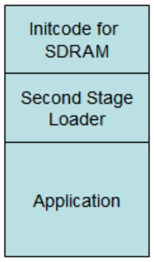


Figure 2. Loader File (.LDR) Content

In this particular application, the SSL consists of the required initialization code for the SPORT interface, as well as the SPORT version of the SPI boot code found in the boot ROM, which is required to proceed with booting the actual application that is still out in the SPI memory at the time the SSL is executed.

Ideally, this can be handled by placing all SPORT initialization code into the SSL and tagging it as an init block in the boot stream (as described in EE-240), such that it boots over SPI, executes in place, and then completes the boot process via the SPORT while executing the SSL on-chip. However, this is a problem because the boot process simply moves instructions/data via DMA from blocks of SPI memory to blocks of Blackfin memory, be it internal instruction/data RAM or external SDRAM.

When the boot stream gets to the input block destined for the top of on-chip Blackfin L1 instruction memory, the SSL that is currently executing this boot code will be overwritten. This behavior is usually not a factor because the on-chip boot ROM is in a protected memory region that is not writeable by the DMA sequence used to move the application code into on-chip memory. However, when executing init block code out of unprotected on-chip memory, this overwrite will compromise the boot process and result in an invalid processor state because the code that is booted in via DMA replaces the currently executing SSL code, which has not completed executing yet. This will cause undesired execution of newly booted instructions before the boot itself has completed.

A solution to this problem is to place the SSL in external SDRAM and to change the address where the boot kernel jumps to after completion from the top of L1 memory to the address of the SSL code itself. The nature of the boot ROM is to set a default reset address in the Event Vector Table location for the reset vector, which is stored in the `EVT1` register. For some processors, this address is 0xFFA00000. For the smaller memory derivative processors, the reset address is 0xFFA08000. If the application wants to set an explicit address, an overwrite of the `EVT1` register can be done in the code that is located in the init block. The code sequence to overwrite the `EVT1` register is as follows, and it can be found in the `SDRAM_InitCode.asm` file in the associated `.ZIP` archive[2]:

```
p0.h = hi(EVT1);
p0.l = lo(EVT1);
r0.h = START_OF_SSL_H;
r0.l = START_OF_SSL_L;
[p0] = r0;
ssync;
```

Using this scheme, the init block boots over SPI and executes on-chip, setting up the SDRAM interface timing appropriate for the application and reconfiguring the reset address to be the beginning of the SSL code. The SSL code is then booted and resolved to SDRAM, which spares the application from being forced to reserve valuable on-chip memory. After the init block and the SSL code are loaded, the boot kernel finishes and jumps into SDRAM, where the rest of the boot process is completed via the SPORT pins, as controlled by the SSL.

This boot process will not be corrupted as long as the actual application does not overwrite the section of SDRAM that the SSL is resolved to.

The SSL code is 925 bytes in size, and should reside in a portion of SDRAM that is guaranteed to be unused by the application being booted. An easy way to do this is by making a small change to the application's `.ldf` file to reserve a small block of SDRAM for the SSL in the memory definition section, and then map nothing to that segment of memory. Using the default ADSP-BF537 `.ldf` file as an example:

```
MEMORY
{
   MEM_SDRAM0_BANK0     { START(0x00000004) END(0x00FFFFFF) TYPE(RAM) WIDTH(8) }
   MEM_SDRAM0_BANK1     { START(0x01000000) END(0x01FFFFFF) TYPE(RAM) WIDTH(8) }
   MEM_SDRAM0_BANK2     { START(0x02000000) END(0x02FFFFFF) TYPE(RAM) WIDTH(8) }
// Comment out the original MEM_SDRAM0_BANK3 definition below
// MEM_SDRAM0_BANK3     { START(0x03000000) END(0x03FFFFFF) TYPE(RAM) WIDTH(8) }
// Replace it with a newly defined BANK3 with a reserved 925 bytes at the end for the SSL
   MEM_SDRAM0_BANK3     { START(0x03000000) END(0x03FFFC5F) TYPE(RAM) WIDTH(8) }
// MEM_RESERVE_SSL      { START(0x03FFFC60) END(0x03FFFFFF) TYPE(RAM) WIDTH(8) }
} /* MEMORY */
```

Figure 3. Modifications to Default ADSP-BF537 .LDF File to Reserve Space for SSL Code

As shown in Figure 3, these modifications to the `.ldf` file free the upper 925 bytes of SDRAM for the SSL to occupy during the boot sequence. If the memory is not defined in the `.ldf` file, it is unusable to the application and will therefore be protected memory. To change where the SSL code is resolved in SDRAM, the above example can be modified appropriately to reserve the block of memory, and then the SSL project itself would need to be adjusted to resolve the SSL to the desired memory range.

First, select the region of memory that the SSL will map to and reserve it in the application's `.ldf` file (`ADSP-BF537.ldf`), as shown in Figure 3. Next, the SSL's `.ldf` file (`SSL_Linker_Description_File.ldf`) must be modified to resolve the SSL code to the same memory region by changing the `START` and `END` addresses of the `SEG_LDR` segment to be the range defined in `ADSP-BF537.ldf` as the memory region to protect. For example, if the application uses a smaller SDRAM, the end of SDRAM may reside at 0x01FFFFFF, as shown in Figure 4:

```
MEMORY
{
    JMP_LDR      { TYPE(RAM) START(0xFFA00000)
                   END(0xFFA0000F) WIDTH(8) }

    SEG_LDR      { TYPE(RAM) START(0x01FFFC60)
                   END(0x01FFFFFF) WIDTH(8) }
}
```

Figure 4. SSL Linker Description File

Finally, change the reset address in the `EVT1` register, as shown in `SDRAM_InitCode.asm`:

```
#define     START_OF_SSL_L    0xFC60
#define     START_OF_SSL_H    0x03FF
/*******SSL Execution Setup**********/
     p0.h = hi(EVT1);
     p0.l = lo(EVT1);
     r0.h = START_OF_SSL_H;
     r0.l = START_OF_SSL_L;
     [p0] = r0;
     ssync;
```

Contents of Associated .ZIP File

In the associated `.ZIP` file, there are four executable files:

- `C_Talkthrough_I2S.dxe`
- `SDRAM_InitCode.dxe`
- `ssl.dxe`
- `SPIDriver.dxe`

C_Talkthrough_I2S.dxe

The `C_Talkthrough_I2S.dxe` file is the executable file for the actual application code. It is the audio talkthrough application that is supplied with the ADSP-BF537 EZ-KIT Lite, which simply uses SPORT DMA to take in an audio stream via the DAC, copies the data to an output buffer, and then uses SPORT DMA to send the output buffer through the ADC to the audio output on the board.

SDRAM_InitCode.dxe

The `SDRAM_InitCode.dxe` file is the executable file for the code required to configure the SDRAM that is present on the ADSP-BF537 EZ-KIT Lite, given the default PLL settings and the 25-MHz `CLKIN` used on the EZ-KIT Lite board. This code would need to be tailored to the target hardware in order to work properly. If this step is required, a new `.dxe` file will need to be generated from the modified source code.

Since the SPI, SPORT, and SDRAM interfaces are in the `SCLK` domain, the speed at which the application can boot will depend on how the hardware is configured. If the PLL registers that govern the `CCLK` and `SCLK` frequencies are modified as part of the SDRAM init block, faster SPI/SPORT/SDRAM speeds can be realized, and the boot speed can be optimized. For example, the code in Figure 5 can be added to the `SDRAM_InitCode.asm` source file to increase the `CLKIN` multiplier (`MSEL`) to 18 from the default setting of 10:

```
/************************************/
/*Initialize Phase Lock Loop for MAX*/
/*SPORT speed                      */
/************************************/
INIT_PLL:
    [--SP]  = RETS;
    [--SP]  = R3;

    R3 = 0x0001(Z);
    W[P1 + LO(SIC_IWR)]    = R3;
    // Change MSEL in PLL_CTL to
    // increase CCLK
    R3.L = 0x2400;
    W[P1 + LO(PLL_CTL)]    = R3;

    IDLE;

    R3   = [SP++];
    RETS = [SP++];
    RTS;
INIT_PLL.END:
```

Figure 5. SDRAM_InitCode .asm Code Snippet

Finally, the beginning address of the SSL code is also specified in the init code. This address must match the `SEG_LDR` start address of the second-stage loader.

ssl.dxe

The `ssl.dxe` file is the executable for the second-stage loader, which is required to transfer control of the boot process from the SPI to the SPORT. This code would only be modified if the user wished to resolve the SSL to an address other than 0x03FFFC60, or if a different SPI device other than the STMicroelectronics M25P32 is used. If either is desired, this executable would need to be regenerated using the modified `SecondStageLoader.asm` source code.

SPIDriver.dxe

The `SPIDriver.dxe` file is the flash programmer driver developed specifically for this application. This driver was developed for the STMicroelectronics M25P32 SPI flash memory device; however, it can be modified to work with any serial memory device.

Generating the Loader File

To utilize the contents of the associated `.ZIP` file, first save the contents to a working directory (e.g., `C:\Project_Name`).

Once the `C:\Project_Name` directory is populated, open the `ssl.dpj` VisualDSP++ project file. On the `Project` page of the `Project Options` dialog box (`Project->Project Options`), verify that the project target type is `Loader file`. This is the master project that will be utilized to generate the single cohesive `.ldr` file necessary for the application to boot as intended.

In the dialog box's tree control (left side), click `Options` (under `Load`); this opens the `Project:Load:Options` page. In `Boot Mode`, select `SPI`; in `Boot Format`, select `Intel hex`; in `Output Width`, select `8-bit`. These options properly configure the loader utility to create an image compatible with the SPI flash device chosen.

Populate the `Initialization file` field with the init code executable:

```
"C:\Project_Name\SDRAM Init Code
(ASM)\Debug\SDRAM_InitCode.dxe"
```

This instructs the loader utility to tag the `SDRAM_InitCode.dxe` executable as an init block in the `.ldr` file, which will allow it to be booted and executed in place prior to continuing the boot process.

In these dialog boxes, double-quotes are required around *all* path names that contain the space bar character.

In the `Additional options` field, provide the application executable itself:

```
"C:\Project_Name\Audio Codec
Talkthrough (C)\Debug\
C_Talkthrough_I2S.dxe"
```

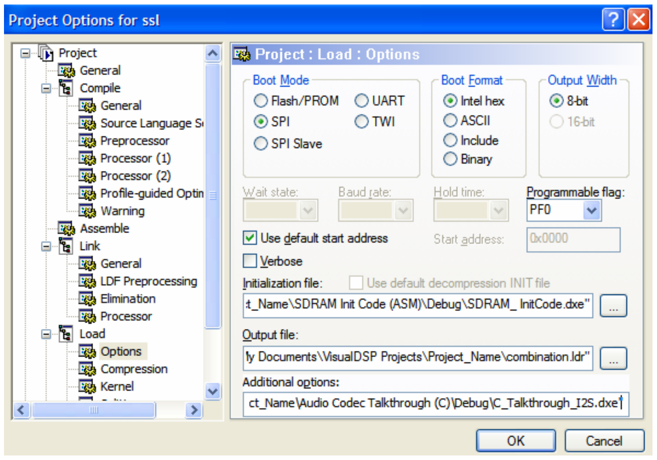
By placing additional executables in the `Additional options` box, the loader is being instructed to append additional `.dxe` files to the `.ldr` file in the order in which they appear in this box. The `.dxe` file for the project being built is placed in the `.ldr` file first, immediately after the init code block, and is then followed by any `.dxe` files specified here.

Finally, set `Output file` to:

```
"C:\Project_Name\combination.ldr"
```

This process is summarized in Figure 6 which shows the project options configured for the ssl project.

Click `OK` and build the project. The output file `combination.ldr` will appear in the `C:\Project_Name` directory.


Project Options for ssl

Project : Load : Options

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

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

Output Width: [✓ 8-bit], 16-bit

Wait state:
Baud rate:
Hold time:
Programmable flag: PF0

[✓] Use default start address
Start address: 0x0000

[ ] Verbose

Initialization file: [ ] Use default decompression INIT file

t_Name\SDRAM Init Code (ASM)\Debug\SDRAM_ InitCode.dxe"

Output file:

ly Documents\VisualDSP Projects\Project_Name\combination.ldr"

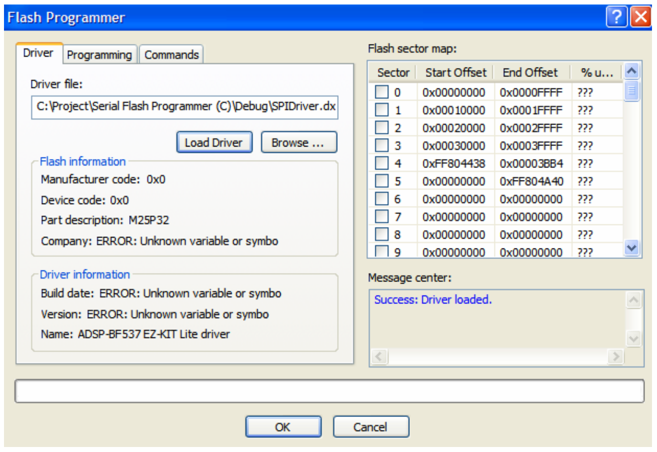
Additional options:

ct_Name\Audio Codec Talkthrough (C)\Debug\C_Talkthrough_I2S.dxe

Figure 6. Loader File Configuration

Programming the SPI Flash Device

Once the `.ldr` file has been properly generated, the next step is to program it into the SPI memory. Use the `SPIDriver.dxe` flash programmer driver in the associated `.ZIP` file by choosing `Tools->Flash Programmer` in VisualDSP++. Then browse for the driver by name, as shown in Figure 7.


Flash Programmer

Driver file:

C:\Project\Serial Flash Programmer (C)\Debug\SPIDriver.dx

Flash information

Manufacturer code: 0x0

Device code: 0x0

Part description: M25P32

Company: ERROR: Unknown variable or symbo

Driver information

Build date: ERROR: Unknown variable or symbo

Version: ERROR: Unknown variable or symbo

Name: ADSP-BF537 EZ-KIT Lite driver

Flash sector map:

Sector	Start Offset	End Offset	% u...
☐ 0	0x00000000	0x0000FFFF	???
☐ 1	0x00010000	0x0001FFFF	???
☐ 2	0x00020000	0x0002FFFF	???
☐ 3	0x00030000	0x0003FFFF	???
☐ 4	0xFF804438	0x00003BB4	???
☐ 5	0x00000000	0xFF804A40	???
☐ 6	0x00000000	0x00000000	???
☐ 7	0x00000000	0x00000000	???
☐ 8	0x00000000	0x00000000	???
☐ 9	0x00000000	0x00000000	???

Message center:

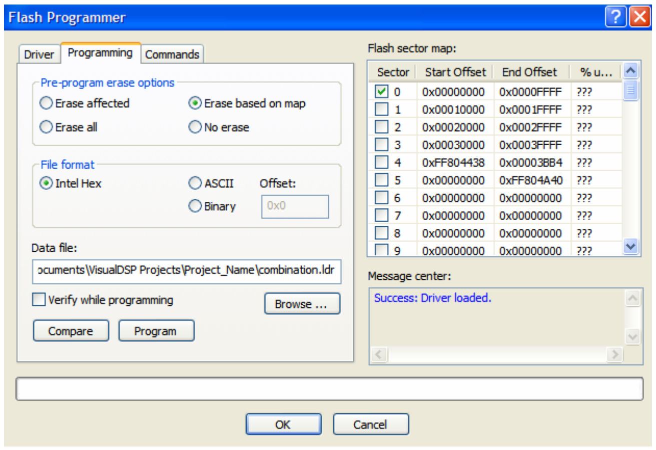
Success: Driver loaded.

Figure 7. Driver Page of Flash Programmer Window

Click the `Programming` tab. Then select `Erase affected` (under `Pre-program erase options`) and select `Intel Hex` in `File format`. Select the sectors that need to be erased (these will vary depending on the size of the `combination.ldr file`), and ensure that `Data file` specifies the path and file name of the `.ldr` file:

`C:\Project_Name\combination.ldr`

Click the `Programming` tab and verify that the `Message center` box indicates success, as shown in Figure 8.


Flash Programmer

Driver | Programming | Commands

Pre-program erase options: ○ Erase affected [✓ Erase based on map] ○ Erase all ○ No erase

File format: [✓ Intel Hex] ○ ASCII ○ Binary — Offset: 0x0

Data file: ocuments\VisualDSP Projects\Project_Name\combination.ldr

☐ Verify while programming

Flash sector map:

Sector	Start Offset	End Offset	% u...
✓ 0	0x00000000	0x0000FFFF	???
☐ 1	0x00010000	0x0001FFFF	???
☐ 2	0x00020000	0x0002FFFF	???
☐ 3	0x00030000	0x0003FFFF	???
☐ 4	0xFF804438	0x00003BB4	???
☐ 5	0x00000000	0xFF804A40	???
☐ 6	0x00000000	0x00000000	???
☐ 7	0x00000000	0x00000000	???
☐ 8	0x00000000	0x00000000	???
☐ 9	0x00000000	0x00000000	???

Message center:

```
Success: Driver loaded.
```

Figure 8. Flash Programmer Window

The application is now programmed into the SPI flash memory. If the emulator is disconnected, resetting the board will result in a successful boot of the application, in which the SPI port is utilized for the initialization process and the faster SPORT interface is employed for the rest of the boot sequence.

Conclusions

The SPORT interface of the Blackfin processor can be configured to act like a master SPI device. Because of this, booting over the SPORT can be achieved if proper care is given to the boot process and if the necessary precautions are taken with respect to hardware. Since the SPORT is capable of functioning at double the operating frequency of the Blackfin SPI port, the SPORT can be used to interface to faster SPI memory devices that normally would not be utilized to their capacity by the native SPI port, given the speed limitation of `SCLK`/4 that is resident on the SPI hardware.

References

[1] *ADSP-BF533 Blackfin Booting Process (EE-240)*. Rev 3, January 2005. Analog Devices, Inc.

[2] *Associated ZIP File*. Rev 1, October 2006. Analog Devices, Inc.

[3] *ADSP-BF537 Blackfin Processor Hardware Reference*. Rev 3.1, May 2005. Analog Devices, Inc.

Document History

Revision	Description
Rev 1 – November 10, 2006 by J. Galindo & Joe B.	Initial Release