ADSP-BF533 Blackfin®加载过程

Hiren Desai 撰稿

Rev 4 – September 29, 2008

引言

本应用笔记说明ADSP-BF531，ADSP-BF532以及ADSP-BF533 Blackfin®处理器的加载过程，不同硅版本之间的差别也给出了解释。

本应用笔记讨论：

- 加载模式
- 加载文件头信息
- 初始化代码
- 多应用程序（多DXE）管理

加载过程

加载就是将存储在外部存储设备（或外部主机）中的应用程序代码/数据加载到Blackfin处理器内部或外部存储器的过程，该过程由位于Blackfin存储器地址0xEF00 0000到0xEF00 03FF的片上Boot ROM进行处理。图1说明了从源代码到最终生成目标独立系统的操作顺序。

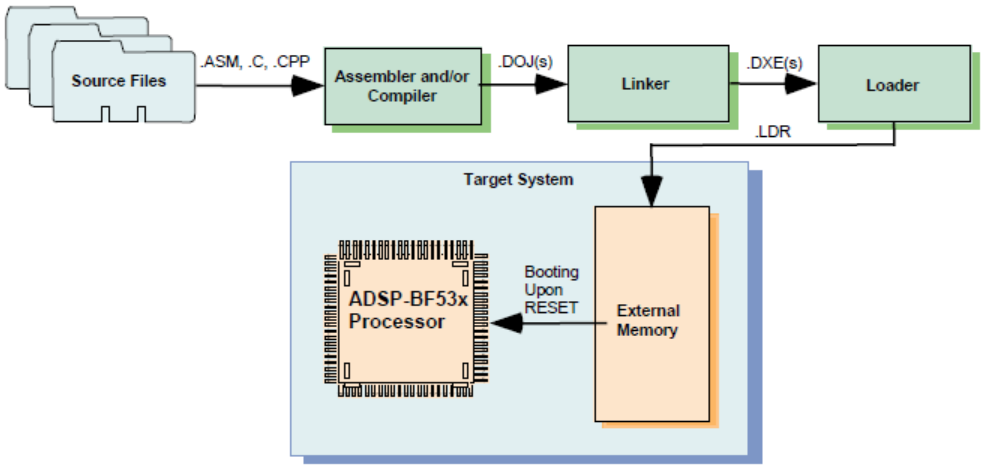


图1. ADSP-BF531/BF532/BF533 独立系统

加载模式（硅版本0.3）*

Blackfin处理器可以通过EBIU的异步存储区0由FLASH/PROM加载或通过SPI接口由SPI设备（存储器或主机）加载。表1列出了ADSP-BF531/BF532/BF533处理器的加载模式，当RESET信号无效后，通过BMODE引脚[1:0]的状态进行选择。

BMODE[1:0]	说明（也可见特定Blackfin加载模式）
00	从连接到ASYNC存储区0的外部16位存储器执行（旁路Boot ROM）
01	从8/16位FLASH/PROM加载
10	在SPI从模式下从SPI主机加载
11	在SPI主模式下，从8/16/24位可寻址SPI存储器加载，支持Atmel AT45DB041B，AT45DB081B，和AT45DB161B数据FLASH®设备

表1. Blackfin ADSP-BF531/BF532/BF533加载模式

**对于先前的硅版本，其支持的加载模式请参见附录：加载模式vs.硅版本*

如图1所示，加载器实用程序（elfloader.exe）对输入可执行文件（.DXE）进行语法分析，并生成由前面有头文件的多个块组成的加载文件（.LDR）*，该加载文件接着可以编程/烧写到外部存储器/设备。在加载过程中，片上Boot ROM读取头文件并进行语法分析。

**关于转换加载文件的信息请参见16位处理器VisualDSP++®加载手册[1]。*

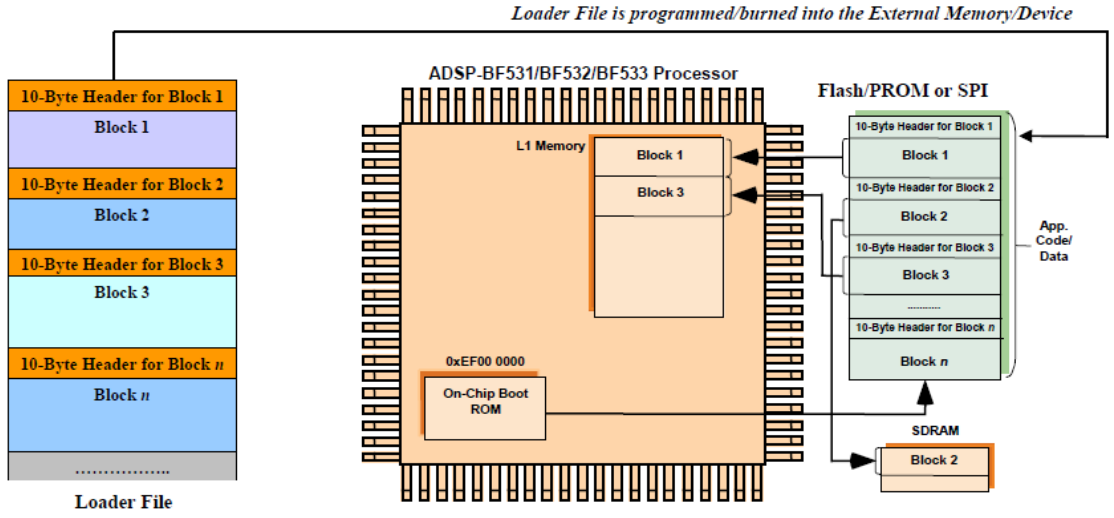


图2. ADSP-BF531/BF532/BF533加载过程

不支持加载到高速暂存（scratchpad）存储器（0xFFB0 0000 – 0xFFB0 0FFF），如果加载到了便笺式存储器，处理器就会悬停在片上Boot ROM中。

文件头信息

如图3所示，加载文件中每个10字节的头文件由一个4字节的ADDRESS（地址）位段，一个4字节的COUNT（字节数）位段和一个2字节FLAG（标志）位段组成。

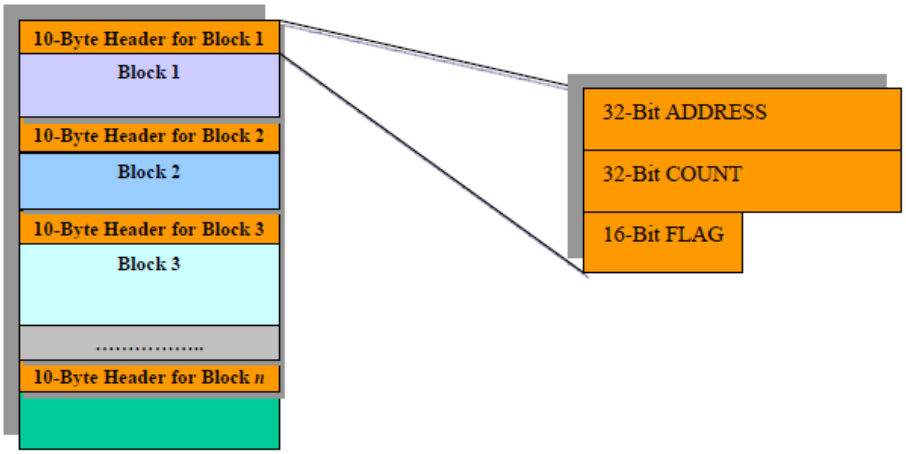


图3. 10字节文件头的内容

在加载文件的各个块之前的10字节的头文件包含以下信息，由片上Boot ROM在加载过程中使用：

- ADDRESS（4字节）—目标地址，数据块将加载到存储器中的地址
- COUNT（4字节）—块中的字节数
- FLAG（2字节）—块的类型和控制命令：

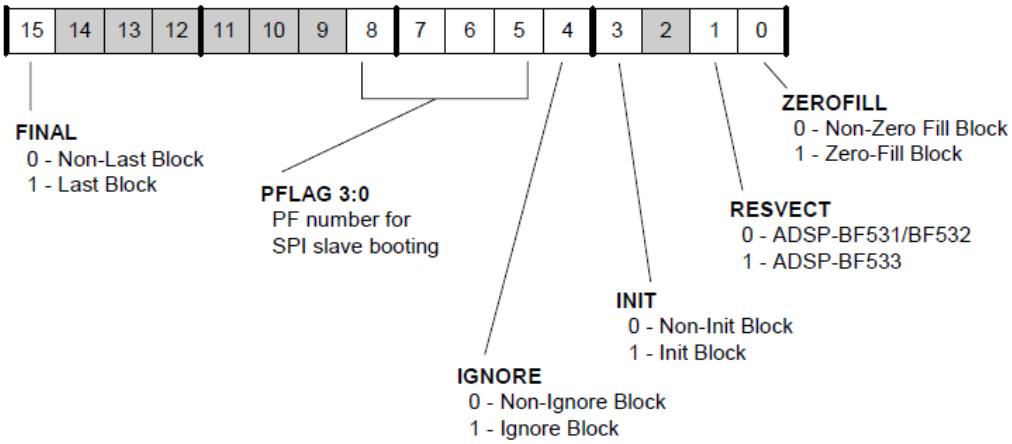


图4. FLAG字的各个控制位

各FLAG位包括：

- **位0：ZEROFILL**—标志该块是一个全0的缓冲区，ZEROFILL的块没有有效载荷数据，它们简单的命令片上Boot ROM从存储器ADDRESS地址开始载入COUNT字节的0。这就为应用程序形成了一个简要的加载文件，具有一个填充值为零的大缓冲区。这对于ANSI-C兼容的工程也有很大帮助，这种工程常常要求在加载过程中有填充值为0的大缓冲区。

- **位1：RESVECT**—指加载后的复位向量，所有ADSP-BF531/BF532/BF533衍生产品都使用相同的Boot ROM。对ADSP-BF531/BF532该位设置为0，而对ADSP-BF533设置为1。加载完成后，片上Boot ROM利用该位，让ADSP-BF533跳至地址0xFFA0 0000，让ADSP-BF531/532跳至地址0xFFA0 8000。

在硬件复位后，取决于RESVECT位，复位向量（存储于EVT1寄存器）设置为0xFFA0 0000或0xFFA0 8000。如果SYSCR寄存器的位4（没有软件复位启动）被设置且软件复位有效，则处理器就会导向EVT1寄存器中所设置的地址。该复位向量可以在运行时配置为另一地址，从而应用程序可以在软件复位后导向0xFFA0 0000或0xFFA0 8000以外的地址。如果复位向量在运行时被修改，应确保EVT1寄存器中的复位向量地址是有效的指令地址，该地址可以是内部指令存储器，SDRAM存储器，或异步存储器。EVT1寄存器没有默认值，该寄存器中的值在发出一次复位后将保持。当BMODE=00时，片上Boot ROM被旁路掉，在发布软件复位之前，必须先初始化EVT1寄存器。

- **位3：INIT**—初始化块（Init块）是一个程序代码块，该块在实际应用程序代码加载覆盖它之前执行。当片上Boot ROM检测到Init块时，它就将该块加载到内部存储器，并发出一个CALL给它（初始化代码必须在其末尾有RTS）。初始化代码执行后，通常会由应用程序代码覆盖，如图5。

- **位4：IGNORE**—指示一个没有加载到存储器的块。它指示Boot ROM跳过加载流COUNT字节。在主加载模式下，Boot ROM可以修改其源地址指针，在从加载模式下，Boot ROM必须积极清理有效载荷数据。目前的VisualDSP++®工具只支持全局文件头的IGNORE块（目前有4字节DXE计数，见下面的多应用程序（多DXE）管理部分内容）。

- **位8:5：PFLAG**—这些位用于SPI从模式加载（BMODE=10）。PFLAG指用于从Blackfin处理器到主SPI主机的主等待（HWAIT）信号的PFx数目。对于ADSP-BF531/BF532/BF533处理器，该数值在1-15（0x1-0xF）之间。关于该PF选通的用法的更多详细信息请参见下面的SPI从模式加载和主主机模式（BMODE=10）部分内容。

- **位15：FINAL**—指示这个块后的加载过程已完成。处理完FINAL块后，片上Boot ROM跳至EVT1寄存器存储的复位向量地址。当处理器跳转到L1存储器执行代码时，它仍处于supervisor模式，且是优先权最低的中断（IVG15）。

与ADSP-BF535处理器不同，ADSP-BF531/BF532/BF533处理器不要求二级加载。10字节文件头的FLAG位段为ADSP-BF531/BF532/BF533处理器提供执行不需要二级加载的单级加载过程所需的所有信息。

初始化代码（Init code）

在实际应用程序加载之前,Init code允许执行一段程序代码，该程序代码可以服务于多种用途，包括初始化SDRAM控制器，或修改PLL设置，SPI波特率，或EBIU等待状态以提高启动速率等等。Init code通过elf加载文件-Init Init_Code.DXE命令行转换，加载到加载文件流的始端，这里的Init_Code.DXE是指用户提供的自定义可执行的初始化代码。

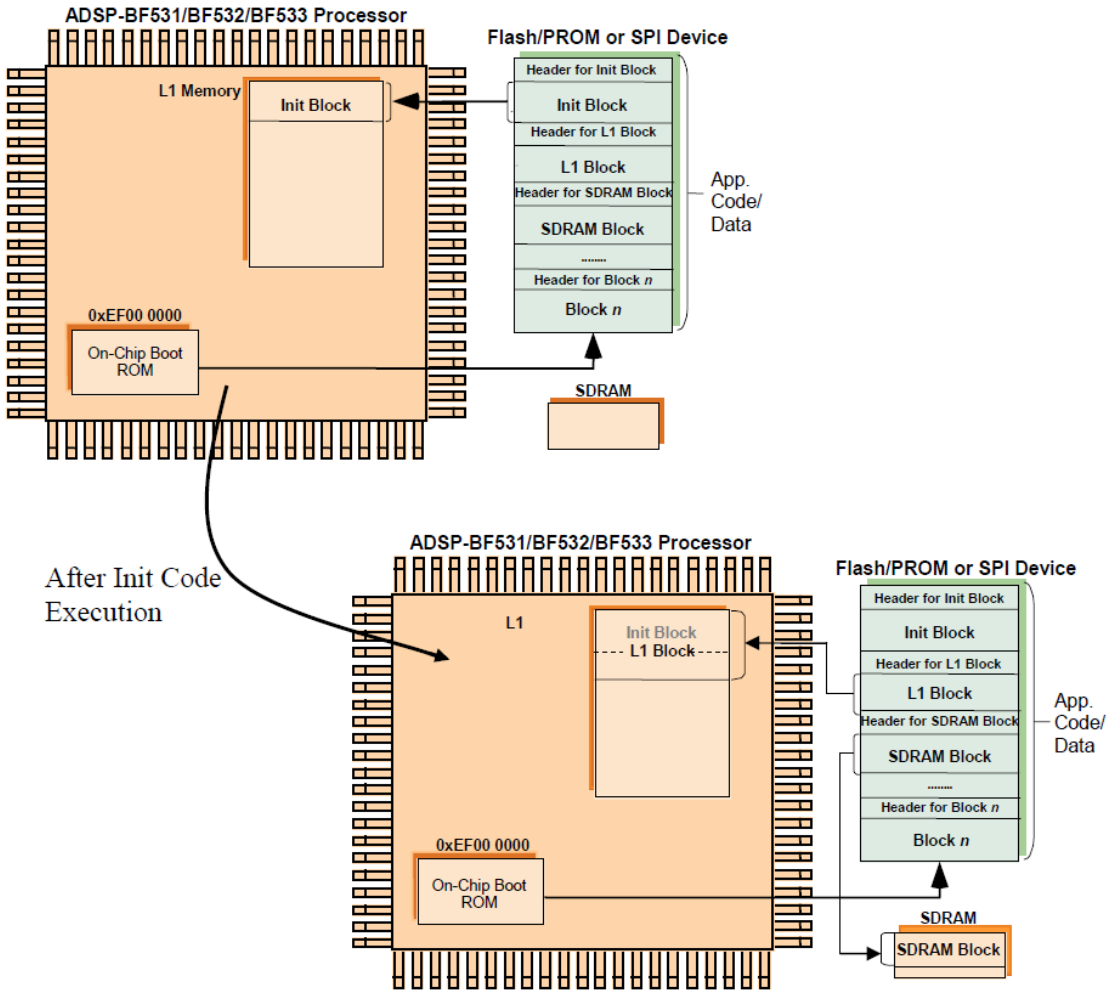


图5. 初始化代码执行/加载

当片上Boot ROM检测到INIT位已置位的块时，它将首先将其载入Blackfin存储器，然后发出一个CALL给其目标地址来执行该块。因此，为了完成余下程序的加载过程，用户必须利用RTS指令来终止Init code以返回到片上Boot ROM。

用户有必要保存所有被Init code修改的处理器寄存器，并在Init code返回之前恢复。至少，每个Init code都应保存ASTAT，RETS，及所有Rx和Px寄存器的值。Blackfin处理器在高速暂存器（0xFFB0 0000 – 0xFFB0 0FFF）中提供了足够的堆栈空间，Init code可以通过堆栈指针SP可实现出栈和压栈操作。列表1说明了Init code文件应用实例，该实例用于建立SDRAM控制器。

```
#include <defBF532.h>
.section program;
/*********************************************************************/
    [--SP] = ASTAT;          // Save registers onto Stack
    [--SP] = RETS;
    [--SP] = (R7:0);
    [--SP] = (P5:0);
/*********************************************************************/
/*******Init Code Section*********************************************/
/*******SDRAM Setup************/
Setup_SDRAM:
    P0.L = lo(EBIU_SDRRC);
    P0.H = hi(EBIU_SDRRC);   // SDRAM Refresh Rate Control Register
    R0 = 0x074A(Z);
    W[P0] = R0;
    SSYNC;

    P0.L = lo(EBIU_SDBCTL);
    P0.H = hi(EBIU_SDBCTL);  // SDRAM Memory Bank Control Register
    R0 = 0x0001(Z);
    W[P0] = R0;
    SSYNC;

    P0.L = lo(EBIU_SDGCTL);
    P0.H = hi(EBIU_SDGCTL);  // SDRAM Memory Global Control Register
    R0.H = 0x0091;
    R0.L = 0x998D;
    [P0] = R0;
    SSYNC;
/*********************************************************************/
    (P5:0) = [SP++];         // Restore registers from Stack
    (R7:0) = [SP++];
    RETS = [SP++];
    ASTAT = [SP++];
/*********************************************************************/
    RTS;
```

列表1. Init code实例

通常，Init code由单个段组成或由加载流中的单个块来表示，当然，该块也有INIT位置位。但是，一个初始化块也可由多个段组成，这样，由多个块表示加载流中的Init code，而只有最后一块的INIT位被置位。elf加载实用程序能确保这最后一块指向Init code的入口地址。如果elf加载实用程序无法实现，则它将保持INIT位清零，即使对最后一个块也如此。并在之后发出一个附加块，而将该附加块的INIT位置位，但不提供任何有效载荷数据（COUNT=0），只是为了片上Boot ROM向给定的ADDRESS执行一次CALL指令。

虽然Init Code.DXE文件通过各自的VisualDSP++工程编译得到，但它们与标准工程也存在差别。Init code只提供一个可调用的子函数，因此，它们看起来更像是一个库函数，而不是一个应用程序。Init code总是普通应用程序代码的文件头，因此，不管Init code是由一个还是多个块组成，它都不能由FINAL位指示器终止，否则将导致Boot ROM终止加载过程。

多应用程序（多DXE）管理

除了用于预加载初始化外，Init code特性还可用于加载管理。如果多个可执行文件（.DXE文件）排列在elf加载器命令行中，则加载文件（.LDR）可以存储多个应用程序。Elf加载器利用可执行文件创建多个加载流，而只有一个独立的可执行文件生成的带Init code DXE位于起始位置的启动流（如图6）。

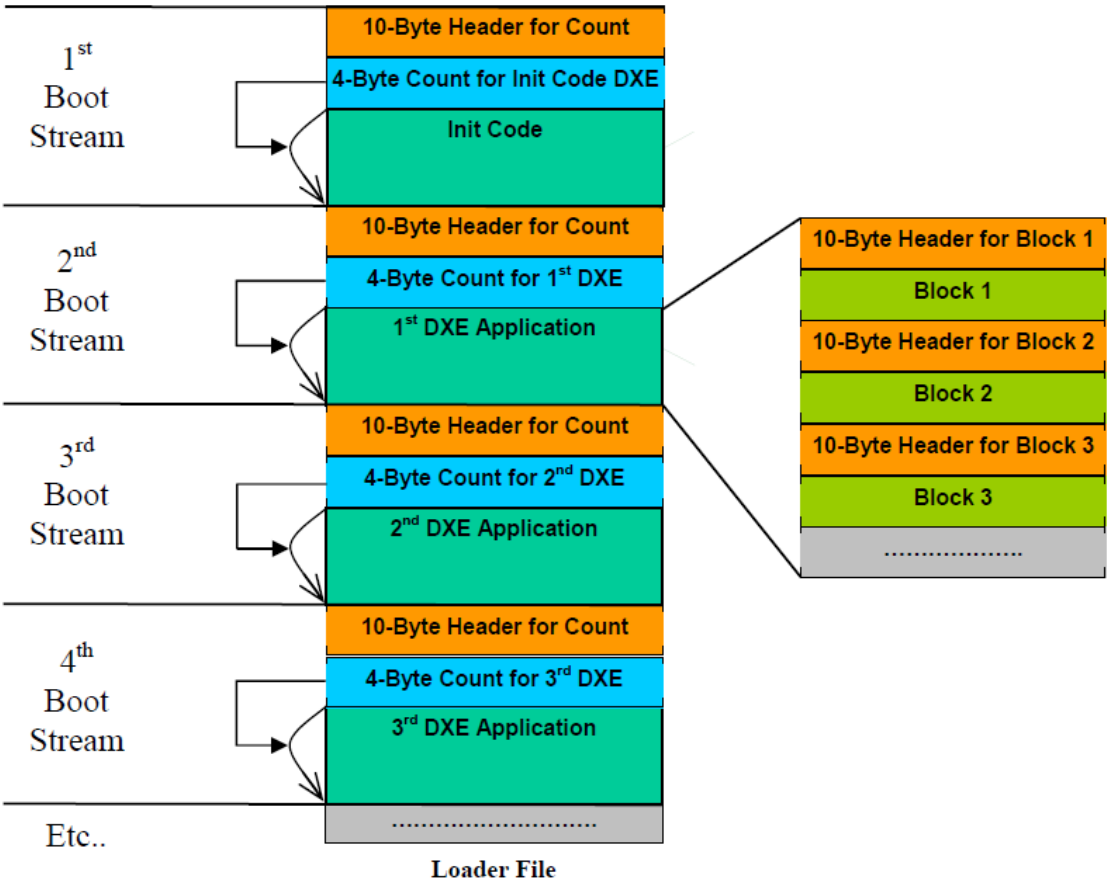


图6. 多DXE加载文件的内容

ADSP-BF531/BF532/BF533加载文件（.LDR）的结构允许用户确定存储于外部存储器中的可执行文件（DXE应用程序）的边界，因此，也具有加载指定DXE应用程序的能力，将被分析并放置在.LDR文件中的.DXE文件置于一个IGNORE块之前。目前，该IGNORE块包括一个4字节的计数值，即是包括文件头在内的DXE应用程序所包含的字节数，换言之，它就是下一个DXE应用程序的偏移量。以后，IGNORE块除了用于保存4字节的计数值外，还可用于保存其它信息。需要注意的是，每个IGNORE块都以一个10字节的文件头作为起始。

有了DXE计数值信息，本质上，用户就可以在.LDR文件中在整个应用程序中“跳转”，直到所选的需加载的DXE应用程序。列表2给出了一段Init code，说明如何从8位FLASH形成多个DXE应用程序加载流。设.LDR文件包含一个Init code DXE和两个DXE应用程序，Init code跳过.LDR文件，加载第二个DXE应用程序。

```
#include <defbf533.h>
.section program;
    [--SP] = ASTAT;          // save registers onto Stack
    [--SP] = RETS;
    [--SP] = (r7:0);
    [--SP] = (p5:0);
    [--SP] = LC0;
    [--SP] = LT0;
    [--SP] = LB0;
/*****************************/
BOOT_DXE:
    R0.H = 0x2000;           // R0 = start of ASYNC Bank 0
    R0.L = 0x0000;
    P1 = 2;                  // Number of DXEs to jump over (CANNOT BE ZERO!!)
                             // After first iteration, R0 will point to DXE1
                             // After second iteration, R0 will point to DXE2
    LSETUP(ADD_DXE_COUNT_BEGIN, ADD_DXE_COUNT_END) LC0 = P1;
ADD_DXE_COUNT_BEGIN:
    R1 = 0xA;                // Skip over 10 bytes for the 1st 10-byte header
    R1 = R1 << 1;            // Multiply by 2 since we are booting from a 16-bit
                             // flash (compensate for zero padding)
    R0 = R0 + R1;
    P0 = R0;                 // P0 points to 4-Byte DXE COUNT
    R0 = W[P0++](Z);         // R0 = xx | Bits[7:0] of DXE COUNT
    R1 = W[P0++](Z);         // R1 = xx | Bits[15:8] of DXE COUNT
    R1 = R1 << 8;
    R2 = W[P0++](Z);         // R2 = xx | Bits[23:16] of DXE COUNT
    R2 = R2 << 16;
    R3 = W[P0++](Z);         // R3 = xx | Bits[31:24] of DXE COUNT
    R3 = R3 << 24;
    R0 = R0 | R1;            // R0 = Bits[15:0] of DXE COUNT
    R2 = R2 | R3;            // R2 = Bits[31:16] of DXE COUNT
    R3 = R0 | R2;            // R3 = DXE COUNT
    R0 = P0;
    R0 = R0 + R3;            // Modify pointer by the DXE COUNT so now R0 points
    P0 = R0;                 // to next DXE
ADD_DXE_COUNT_END:
    NOP;
/*****************************/
DONE:
    LB0 = [SP++];            // Restore Regs
    LT0 = [SP++];
    LC0 = [SP++];
    (p5:0) = [SP++];
    (r7:1) = [SP++];         //----->DO NOT RESTORE R0<-------
    RETS = [SP++];           // Modify SP by one for R0 case
    RETS = [SP++];           // Pop off the real value of RETS
    ASTAT = [SP++];
    RTS;
```

列表2. 多DXE启动的Init code实例

需要注意的是约定寄存器R0，是FLASH/PROM启动（BMODE=01）的外部指针，在Init code最后也不会恢复。当处理器执行RTS指令后回到片上Boot ROM时，片上Boot ROM将继续从存储于R0中的位置开始加载。类似的，如果加载模式设置为SPI加载（BMODE=11），外部指针则存储于R3，因此，对于SPI启动多DXE应用程序，也不要在Init code中恢复R3。

本应用笔记还附带一个基于ADSP-BF533 EZ-KIT Lite®板的多DXE加载实例（BF533 Ez Kit Multiple DXE Boot.zip）。ZIP文件包含两个闪烁应用程序工程和一个Init code工程。RESET时，片上Boot ROM将载入Init code，然后Init code将一直等待，直到PF8（SW4按钮）或PF9（SW5按钮）有效。如果PF8有效，则加载DXE1并执行，如果PF9有效，则加载DXE2并执行。DXE1是一个应用程序，它切换闪烁板上的LED，而DXE2则是闪烁板上所有的LED。

具体的Blackfin加载模式

现在已经对ADSP-BF531，ADSP-BF532，和ADSP-BF533的加载过程有了一个大概的了解，本应用笔记的剩余部分将讨论与每种加载模式相关的信息，如硬件接口，加载文件结构，以及预期的引脚动作。“*Running Programs from Flash on ADSP-BF533 Blackfin Processors (EE-239)*” [3]中讨论了如何从外部16位存储器（BMODE=00）执行。

以下部分内容，将用到带有Init code的基于ADSP-BF533 EZ-KIT Lite板的ASM闪烁指示灯程序实例，该实例程序附带在本应用笔记中。

需要注意的是，以下所列的每种加载模式，必须保留地址0xFF80 7FF0 – 0xFF80 7FFF（L1数据存储区A的最后16个字节），片上Boot ROM将该存储空间用于存储加载文件中每个块的文件头信息。加载完成后，应用程序在运行时可使用该存储空间。硅修订版本0.1和0.2见附录。

8位FLASH/PROM加载（BMODE=01）

因为Blackfin处理器上的EBIU为16位宽（因此没有ADDR[0]），所以8位FLASH/PROM只会占用数据总线的低8位（D[7:0]）。图7说明了Blackfin处理器和8位FLASH/PROM间的引脚连接：

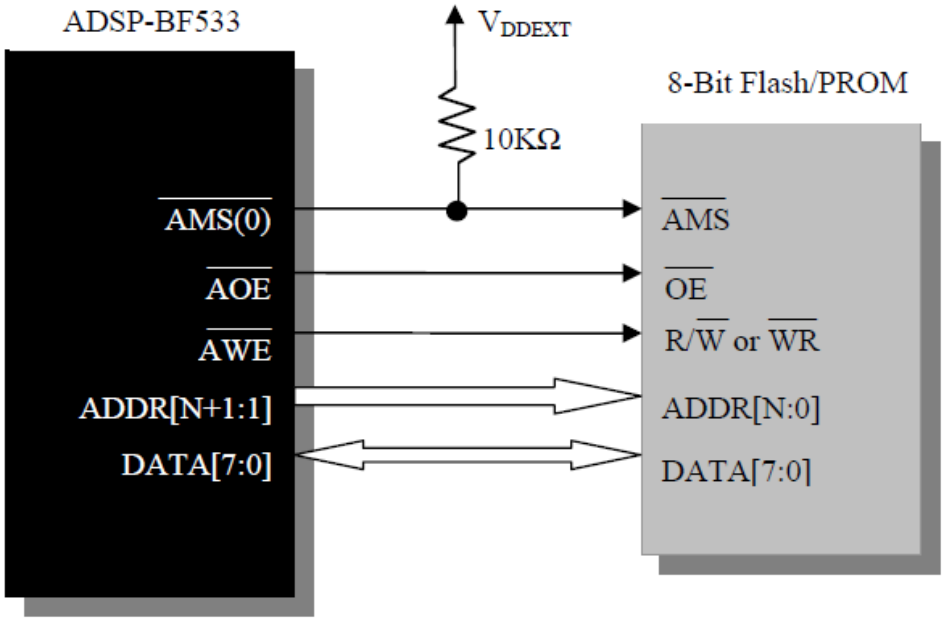
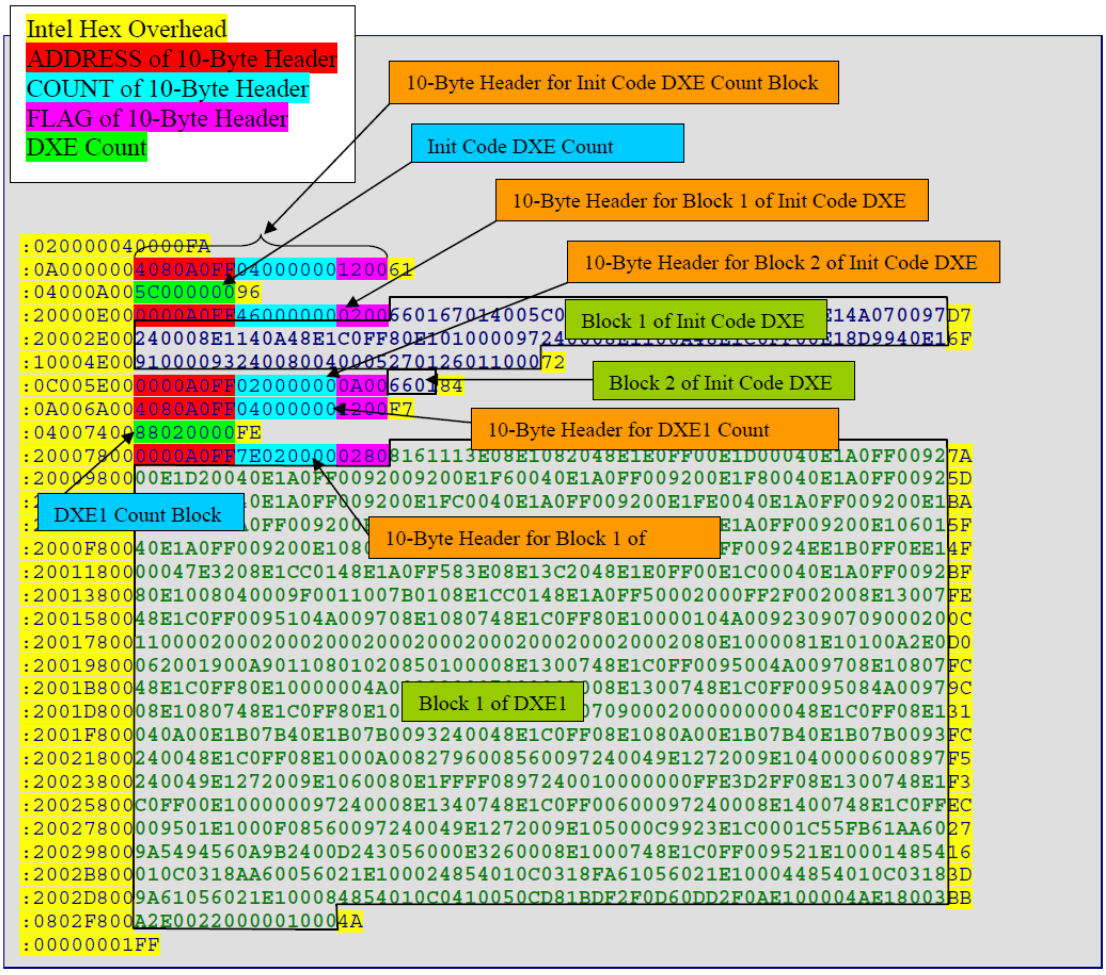


图7. Blackfin处理器和8位FLASH/PROM间的引脚连接

列表3给出了以Intel十六进制格式生成的8位FLASH/PROM的加载文件（见本应用笔记附带的example.zip），为了说明加载文件的结构，将其分解为几个不同的部分。



列表3. Intel十六进制加载文件实例

列表3中的加载文件是为硅版本0.3编译生成的。硅版本0.2及以下版本所编译生成的加载文件的结构还略有不同。

当该加载文件编程到与Blackfin处理器ASYNC存储区0连接的8位FLASH中时，从Blackfin角度看则存储器的内容（从地址0x2000 0000开始）如图8所示。

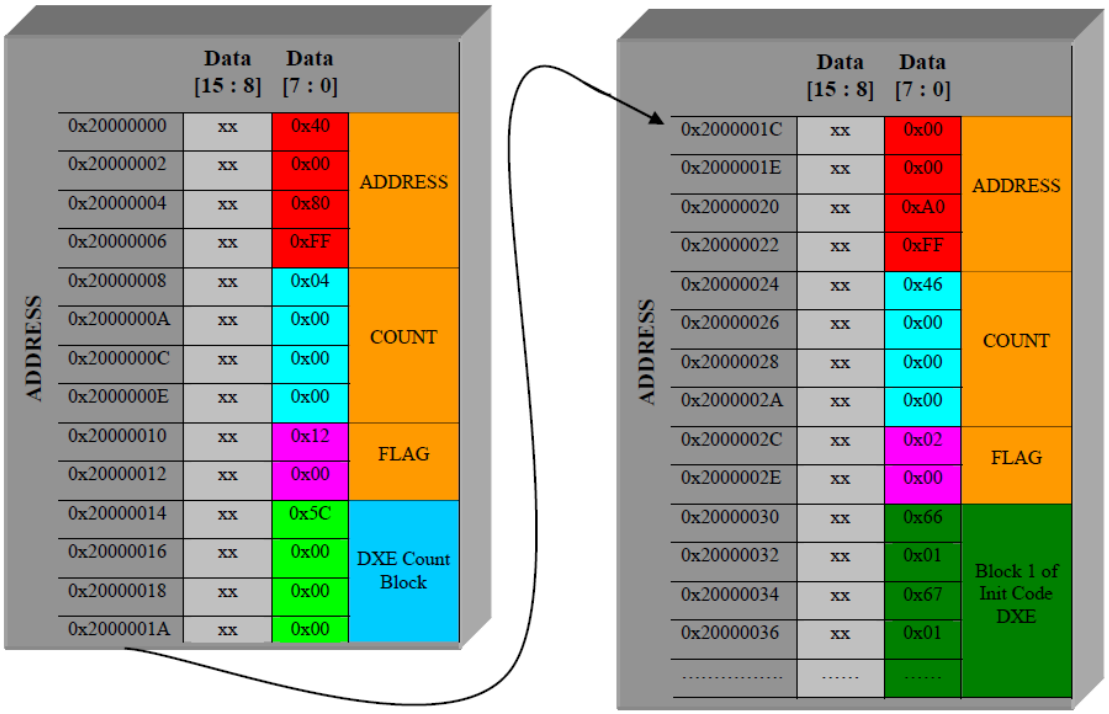


图8. 从Blackfin存储器窗口看8位FLASH/PROM存储器的内容

图9说明8位FLASH/PROM加载的加载序列的起点。

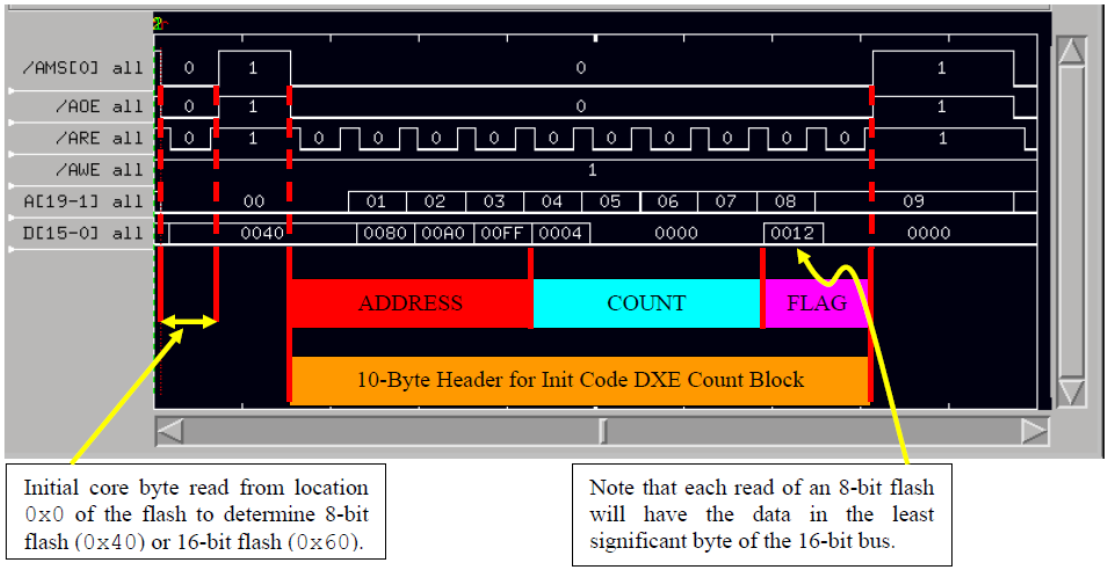


图9. 8位FLASH加载序列的时序图

16位FLASH/PROM启动（BMODE=01）

图10说明了Blackfin处理器与16位FLASH/PROM间的引脚连接：

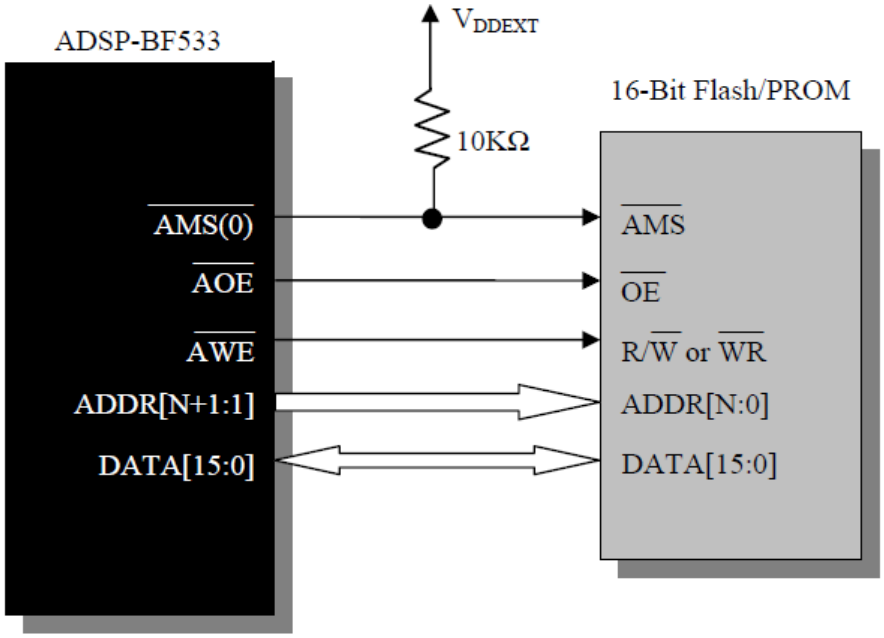


图10. Blackfin处理器与16位FLASH/PROM间的连接

如果依照本应用笔记附带的实例为16位FLASH/PROM生成加载文件，基本和以上列表3说明的相同，除了DXE计数块的10字节文件头的ADDRESS将为0xFF80 0060，而不是0xFF80 0040以外，这将使加载文件的第一个字节为0x60而不是0x40。片上Boot ROM也利用该首字节确定连接的是8位还是16位FLASH/PROM，如果首字节为0x60，则采用的是16位FLASH/PROM，如果首字节为0x40，则采用8位FLASH/PROM。

当该加载文件编程到与Blackfin处理器ASYNC存储区0连接的16位FLASH中时，从Blackfin角度看，其存储器内容（从地址0x2000 0000开始）如图11所示。

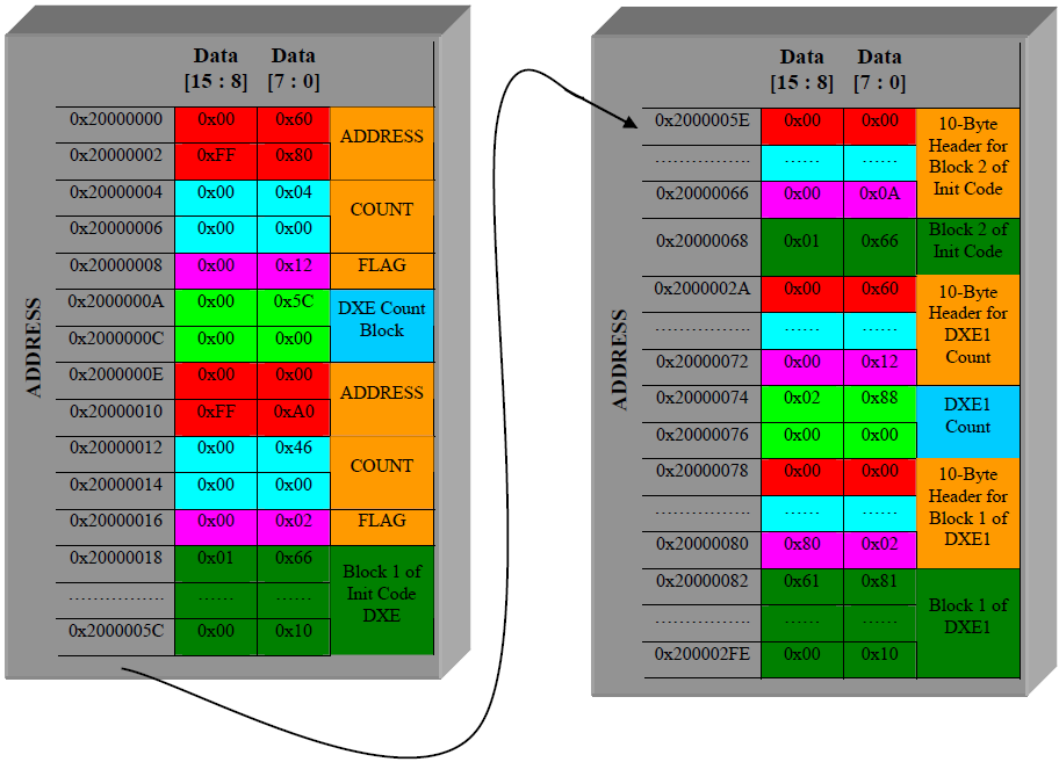


图11. 从Blackfin存储器窗口看16位FLASH/PROM存储器的内容

只有硅版本0.3及以上版本支持16位FLASH/PROM的加载文件结构（图11）。ADSP-BF531/BF532/BF533的硅版本0.2及以下版本仅支持8位加载。因此，如果为硅版本0.2及以下版本选择宽度16，则输出的加载文件将补零，以“模拟”从16位FLASH/PROM的8位加载。该加载文件编程到FLASH/PROM存储器中时，就像图8中数据总线的高8位（DATA[15:8]）始终为0。

图12说明了16位FLASH/PROM加载的加载序列的起点

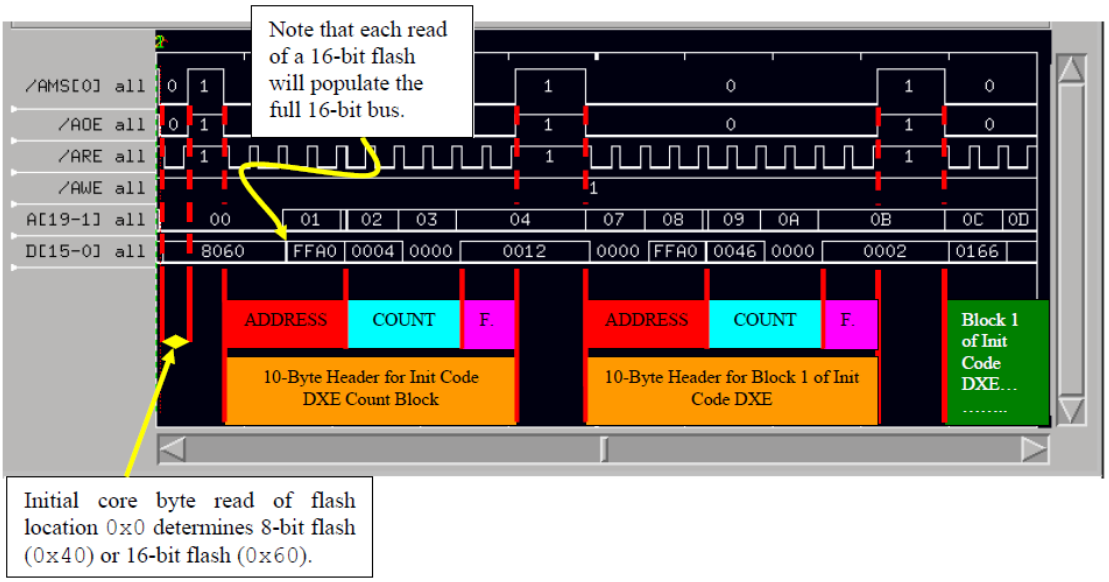


图12. 16位FLASH加载序列的时序图

处理器将执行一次从FLASH 0x0地址的处理器内核字节读操作，以确定FLASH的存储宽度。当从FIFO启动时，首字节（加载文件中的第一个10字节头文件的一部分）必须发送两次：一次用于初始处理器内核读操作，另一次用于实际加载序列。

通过主机SPI从模式加载（BMODE=10）

SPI从模式加载中，将ADSP-BF531/BF532/BF533配置为一个SPI从设备，并利用另外的主机加载处理器。

ADSPBF531/BF532/BF533处理器的硅版本0.2及以下版本不支持该加载模式。

图13说明该模式下所需的引脚到引脚间的连接，主机不需要知道加载文件流的任何信息就可加载Blackfin处理器，但必须将其配置为每次从加载文件（ASCII格式）发送一个字节。在该过程中，PFx为从Blackfin处理器到主机设备的等待信号，必要时Blackfin处理器可利用该信号在加载过程中使主机处于“保持”。当PFx有效（为高电平）时，主机设备必须中断向Blackfin处理器发送字节，当PFx无效（为低电平）时，主机设备将重新开始从上次停止处继续发送字节。由于PFx引脚是在处理完第一个块后才由从设备驱动，所以应考虑在HWAIT信号上加一个下拉电阻。

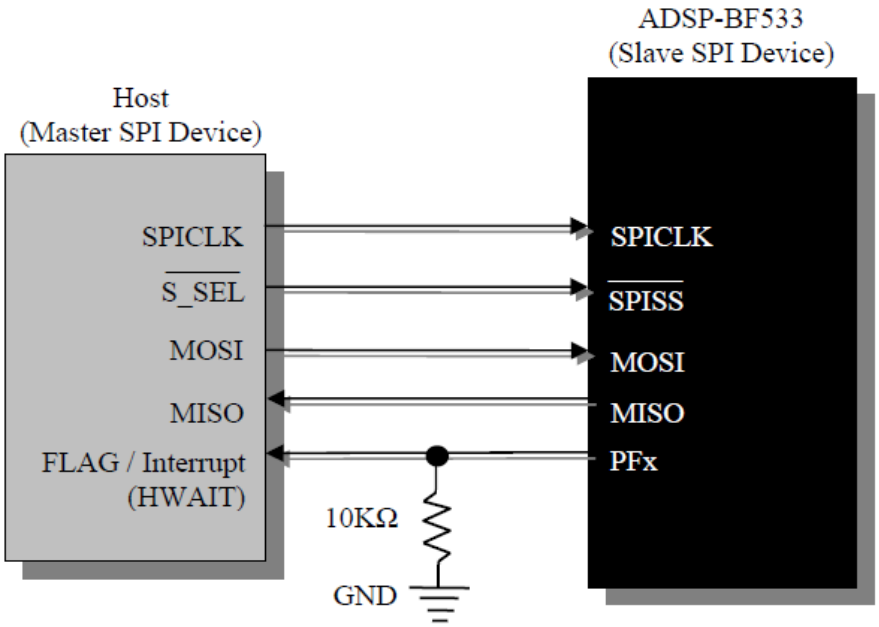


图13. 主机（SPI主机）和Blackfin处理器（SPI从设备）间的连接

主机必须确保在向Blackfin发送任何字节前，处理器已退出复位，否则Blackfin处理器复位前发送给它的任何字节都将丢失，加载过程将失败。

PFx序号可由用户定义，并可嵌入到加载文件中。elf加载器将该序号嵌入到每个10字节文件头中的PFLAG位段（FLAG字的[8:5]位），可使用–pflag number命令行转换实现这一过程，其中number为Blackfin从设备使用的PFx标志，其值为1到15之间。

如果未用到–pflag number转换，FLAG的位8:5的默认值为0，即将PF0默认为到主机的HWAIT信号。但由于PF0与/SPISS引脚是复用的，这对于要实现成功的SPI从启动非常重要，因此，应始终使用–pflag转换，而不是使用默认的0值。

在Rev0.3 Boot ROM中存在一个异常，SPI控制和DMA5配置寄存器在加载完成后没有恢复默认值。

在从设备CCLK = 333 MHz，SCLK = 66.6 MHz的系统中，经过测试的主机最大波特率为1MHz。

本应用笔记附有当ADSP-BF532处理器用作主机时的主机代码实例（Host_Code.zip）。

以下的SPI从模式加载时序图，以ADSP-BF532处理器作为主机，ADSP-BF533处理器作为从SPI设备。在主机端，PF4作为/CS信号连接到从ADSP-BF533的/SPISS。SPI从设备（ADSP-BF533处理器）的PF13连接到主机（ADSP-BF532处理器）的PF15，该连接用作HWAIT信号以使主机“保持”。图中的所有时序都是SPI从设备角度看到的。

除了增加两个块来说明该加载模式的功能外，所用加载文件（SPI_Slave_HostFile.ldr）都与列表3中的完全相同。这两个块分别为：字节数为0x4000的全0块，地址为0xFFA0 0300；一个值为0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF，和 0x19的数据块，地址为0xFFA0 4300。

图14给出了完整的SPI从模式加载过程。

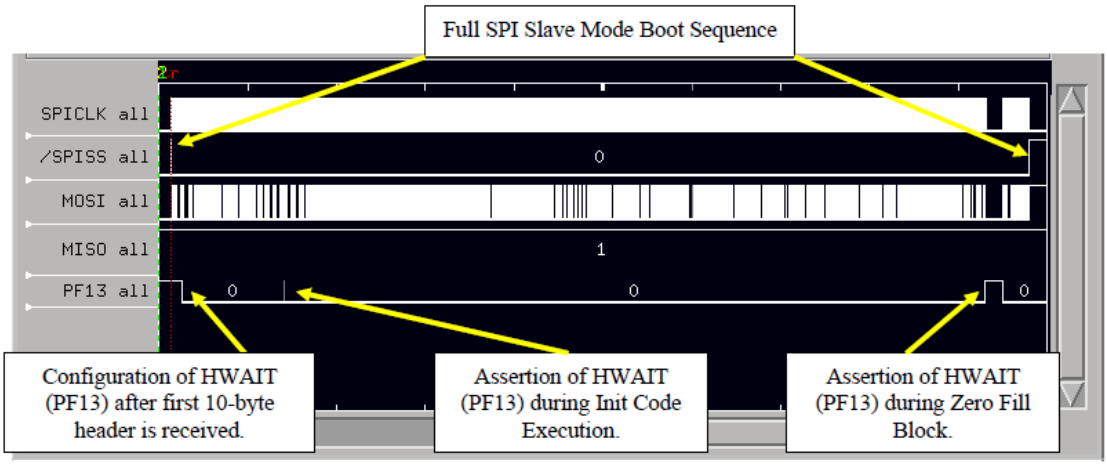


图14. SPI从模式加载过程时序图

SPI从设备接收到来自主机的第一个10字节文件头后，就知道要将哪个PFx标志配置为HWAIT信号。在本例中，使用PF13。需要注意的是，图15中，对FLAG位段的位8:5处理后，PF13变为无效。

为了达到调试的目的，利用有上拉电阻的PF13捕获得到图15。对于正常操作，需要下拉电阻，这样PF13就始终保持低电平。

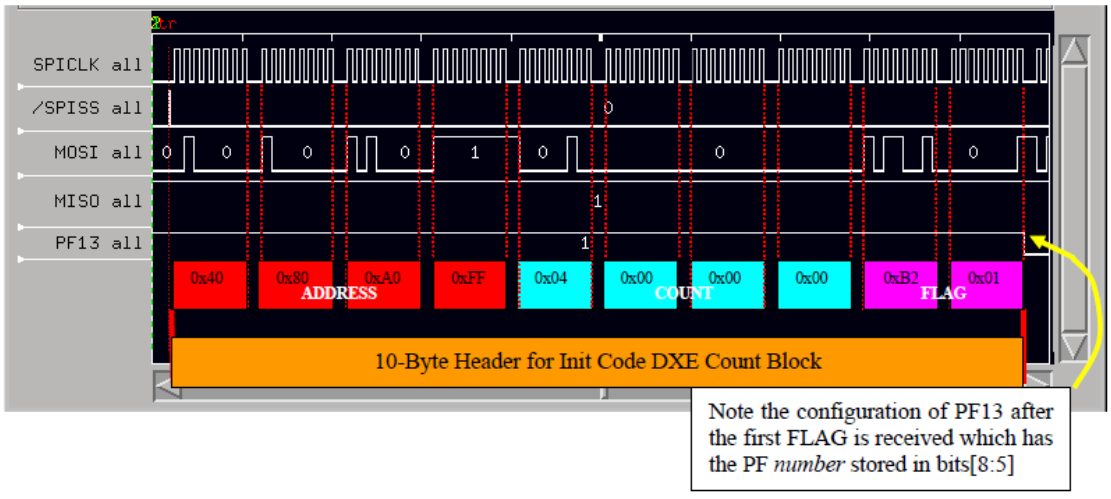


图15. SPI从模式加载过程：加载过程起点

在这之后，主机发出4字节Init Code DXE计数块，然后是Init Code DXE块1的10字节文件头，接着是块1本身。

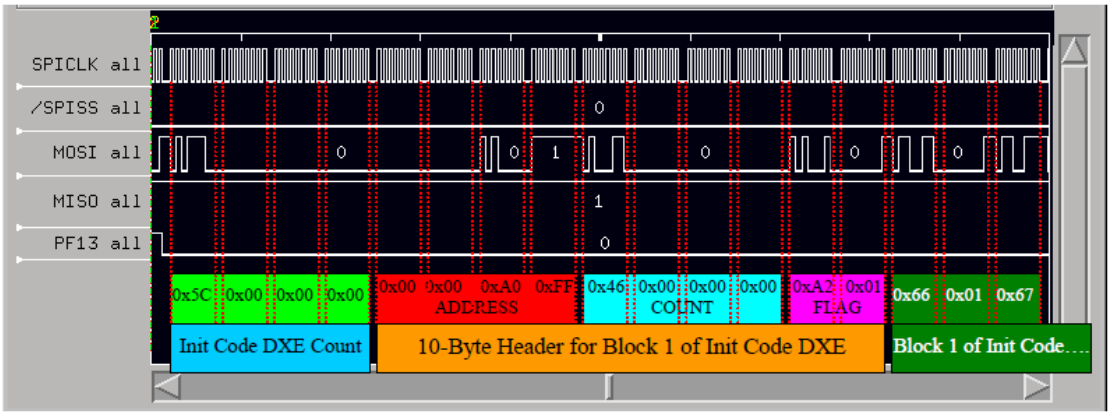


图16. SPI从模式加载过程：Init Code DXE的加载块1

当完整的Init Code DXE加载到Blackfin存储器后，从SPI Blackfin处理器将会使HWAIT信号，PF13有效，指示主机不要在Init Code执行过程中再发送任何字节。由于Blackfin处理器的内核速度比SPI接口快得多，所以将以比主机发送字节速率快得多的速率执行Init Code。

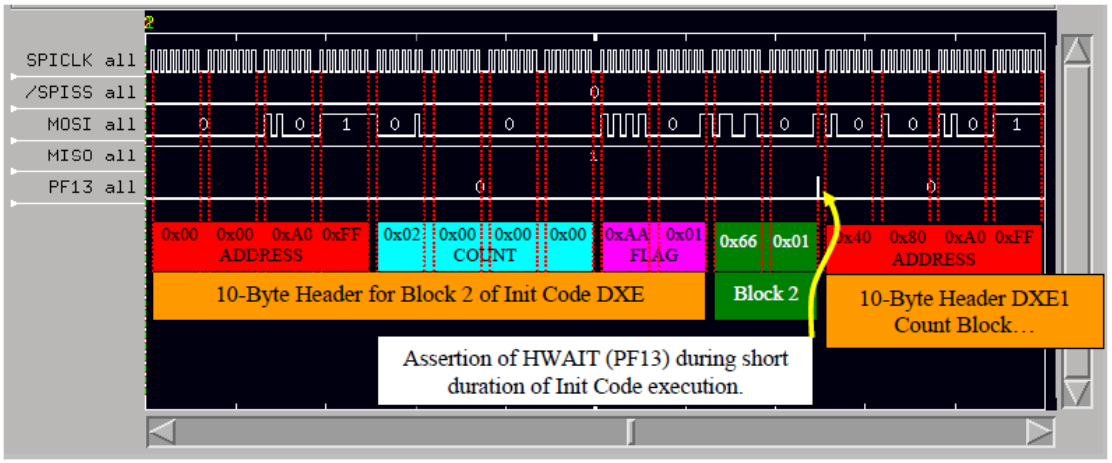


图17. SPI从模式加载过程：Init Code DXE加载块2

图18给出了该加载模式的一个全0块的处理过程。当片上Boot ROM遇到全0块时，它就使HWAIT，PF13有效，使主机保持，不再发送字节，这时，它通过存储器DMA向地址0xFFA0 0300-0xFFA0 4300发送0x4000个0。完成后，片上Boot ROM使HWAIT失效，而主机则继续发送剩下的加载字节（块3的10字节文件头和块3本身）。

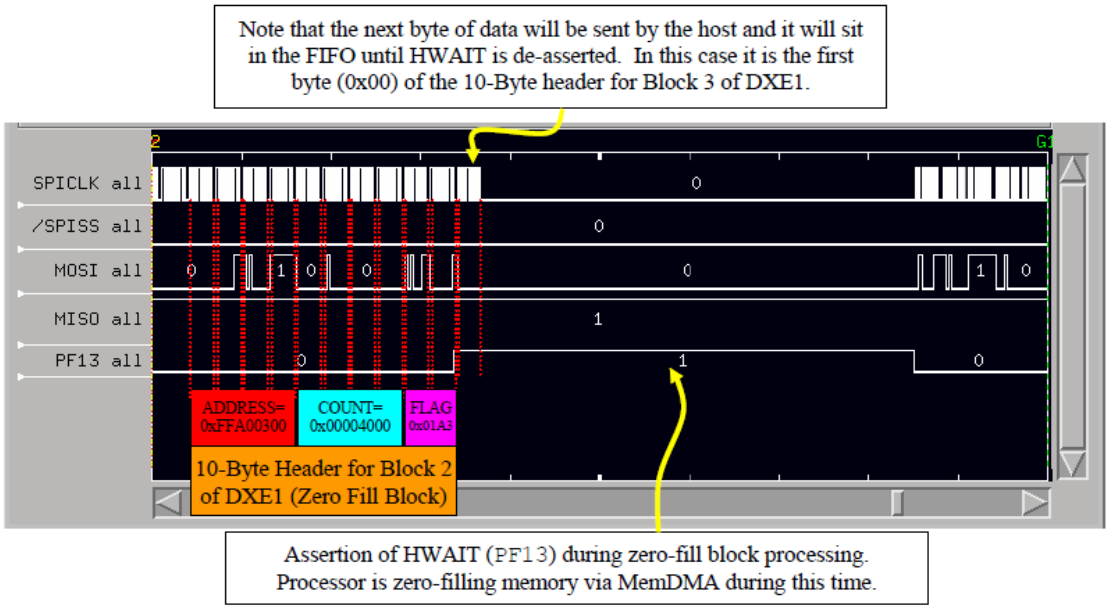


图18. SPI从模式加载过程：加载全0块（DXE1的块2）

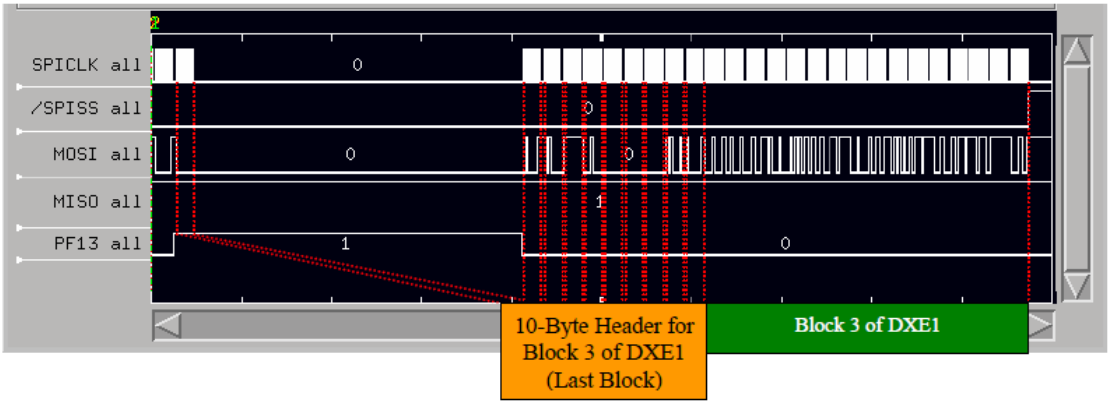


图19. SPI从模式加载过程：DXE1的加载块3（最后一个块）

通过一个SPI存储器的SPI主模式启动（BMODE=11）

对于SPI主模式加载，ADSP-BF531/BF532/BF533处理器配置为连接一个SPI存储器的SPI主设备。下面给出了该模式下所需的引脚间连接。

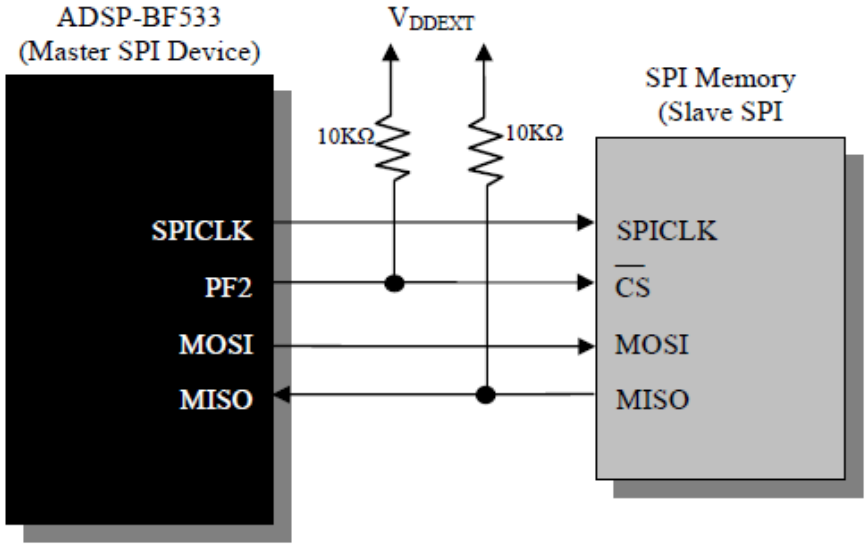


图20. Blackfin-SPI存储器引脚连接

为了正常工作，该加载模式需要在MISO加上拉电阻。否则，ADSP-BF531/BF532/BF533处理器将从MISO引脚读取到0xFF（即SPI存储器没有写任何数据到MISO引脚），不是希望的响应值。

不仅MISO线上的上拉电阻是必要的，额外的上下拉电阻还用于：1）上拉PF2信号，确保SPI存储器在Blackfin处理器复位状态下未激活。2）在SPICLK上用下拉电阻，使显示画图更加清晰。

在硅版本0.2及更低版本，SPI控制（SPICTL）寄存器中的CPHA和CPOL位同时设置为1（有关这些位的详细信息请参见*硬件参考手册*[2]），因此SPI存储器从三态恢复时可能检测到错误的时钟信号上升沿。如果加载过程因此而失败，SPICLK信号上的一个上拉电阻会缓解该问题。在硅版本0.3上，通过设置SPI控制寄存器中的CPHA=CPOL=0可完成上述过程。硅版本0.3相对于SPICLK上的上拉电阻更鲁棒，对于各种版本的电路板都可以安全地上拉SPICLK。然而，需要注意，从显示器上观察，在硅版本0.3上，当PF2无效时SPICLK会异常变高。

该接口所支持的SPI存储器为标准8/16/24位可寻址SPI存储器（读取顺序下面再作介绍），以及Atmel SPI数据FLASH设备：AT45DB041B，AT45DB081B，AT45DB161B*。

**本应用笔记附有利用ADSp-BF532处理器对Atmel数据FLASH设备进行设计的代码实例（见Program_Atmel.zip）。*

标准的8/16/24位可寻址SPI存储器先获取读命令字节0x03，接着是一个地址字节（对8位可寻址SPI存储器），或两个地址字节（对16位可寻址SPI存储器），或三个地址字节（对24位可寻址SPI存储器）。当发送完正确的读命令和地址后，存储器中指定存储单元的数据将移出到MISO引脚，根据连续的时钟脉冲，顺序送出该地址的数据。ADI公司已对以下标准SPI存储设备进行了测试：

- 8位可寻址SPI存储器：Microchip公司的25LC040
- 16位可寻址SPI存储器：Microchip公司25LC640
- 24位可寻址SPI存储器：STMicroelectronics公司的M25P80

SPI存储器检测程序

因为BMODE=11支持各种SPI存储器加载，片上Boot ROM将负责检测连接的是什么类型的存储器。为了确定所连接的存储器类型，片上Boot ROM将向SPI存储器发送以下字节序列，直到存储器响应为止。SPI存储器也只有被正确寻址后才会给出响应。片上Boot ROM执行下面操作。

1. 在MOSI引脚发送读命令0x03，然后对MOSI引脚进行虚拟读操作。
2. 在MOSI引脚发送地址字节0x00，然后对MOSI引脚进行虚拟读操作。
3. 在MOSI引脚发送另一字节0x00，并检查到达MISO引脚的字节是否为0xFF之外的其他值（上拉电阻的值；参考下面的NOTE）。返回的接收字节不是0xFF意味着SPI存储器在一个地址字节后已经响应，处理器已经与8位可寻址SPI存储设备连接了。
4. 如果接收字节还是0xFF，片上Boot ROM在MOSI引脚再发送另一字节0x00，并检查MOSI引脚上的接收字节是否为0xFF外的其他值。若不是0xFF，意味着SPI存储器在两个地址字节后响应，处理器与已经和16位可寻址SPI存储设备连接了。
5. 如果接收字节仍为0xFF，则片上Boot ROM在MOSI引脚发送另一字节0x00，并检查MOSI引脚上的接收字节是否为0xFF外的其他值。若不是0xFF，意味着SPI存储器在三个地址字节后进行了响应，处理器和一个24位可寻址SPI存储设备连接了。
6. 如果接收字节仍为0xFF（意味着没有设备响应），则片上Boot ROM假定连接了以下Atmel Data Flash设备中的一个：AT45DB041B，AT45DB081B，或AT45DB161B。这些Data Flash设备的读序列与上面所讲的标准SPI存储器有所不同。若需更多信息，请参见这些设备的数据手册[4]，[5]和[6]。片上Boot ROM通过读状态寄存器就可以判定究竟是连接了以上Atmel Data Flash存储器中的哪一个。以上所列的Data Flash间的主要区别在于每个页面的字节数不同。AT45DB041B和AT45DB081B为264字节/页，而AT45DB161B为528字节/页，要确定究竟哪个与Blackfin连接，片上Boot ROM读取Data Flash的状态寄存器即可，它包含了设备密度位。如果设备的密度位=1011（二进制），则片上Boot ROM认为连接了AT45DB161B且地址分配正确，否则，则认为连接了AT45DB041B或AT45DB081B且地址合适。自硅版本0.3片上Boot ROM代码生成后，Atmel又引入了其他更多Data Flash派生产品，如果用户计划使用这些派生产品，必须确保该设备每个页面有264字节，否则加载过程就会失败。

以上解释的SPI存储器检测程序中，硅版本0.2及更低版本中的片上Boot ROM检查MISO引脚上的接收数据是否为0x00（加载文件的第一个字节）。硅版本0.3中的片上Boot ROM则检查MISO引脚上的接收数据是否为0xFF。因此，为硅版本0.2及更低版本建立的SPI加载文件的首字节必须为0x00。对于硅版本0.3，加载文件的首字节设置为0x40。

SPI波特率寄存器设置为133，当系统时钟为54MHz时，可得到波特率为54MHz/(2*133)=203kHz。在ADSP-BF533 EZ-KIT Lite板上，默认的系统时钟频率为54MHz。

下图说明了16位可寻址SPI存储器（Microchip 公司25LC640）实现的SPI主模式加载的加载过程，所用的加载文件与列表3相同，所有图像都是硅版本0.3上捕获的。

图21给出了完整的SPI主模式加载过程。

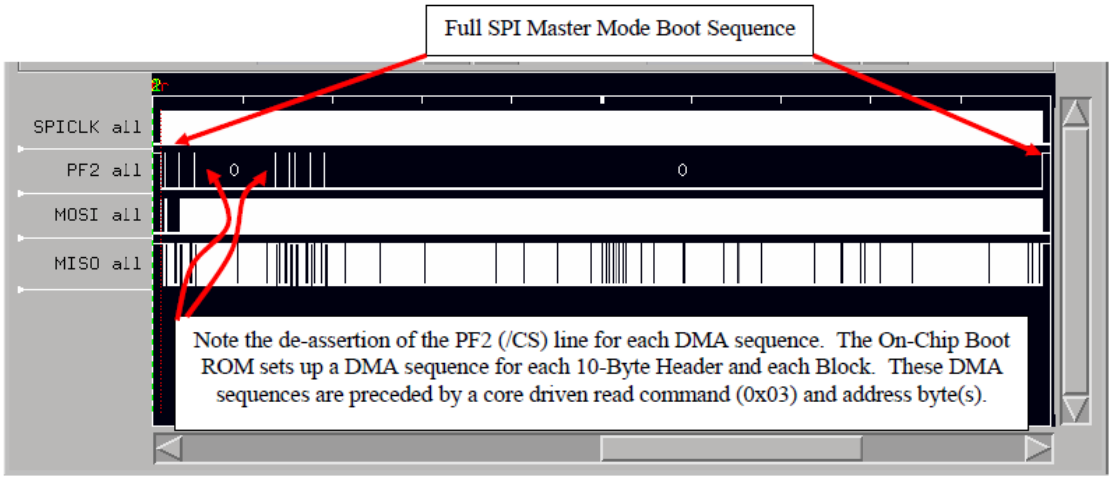


图21. SPI主模式加载过程的时序图

开始，片上Boot ROM确定连接的SPI存储器类型：8/16/24位可寻址或Atmel Data Flash。

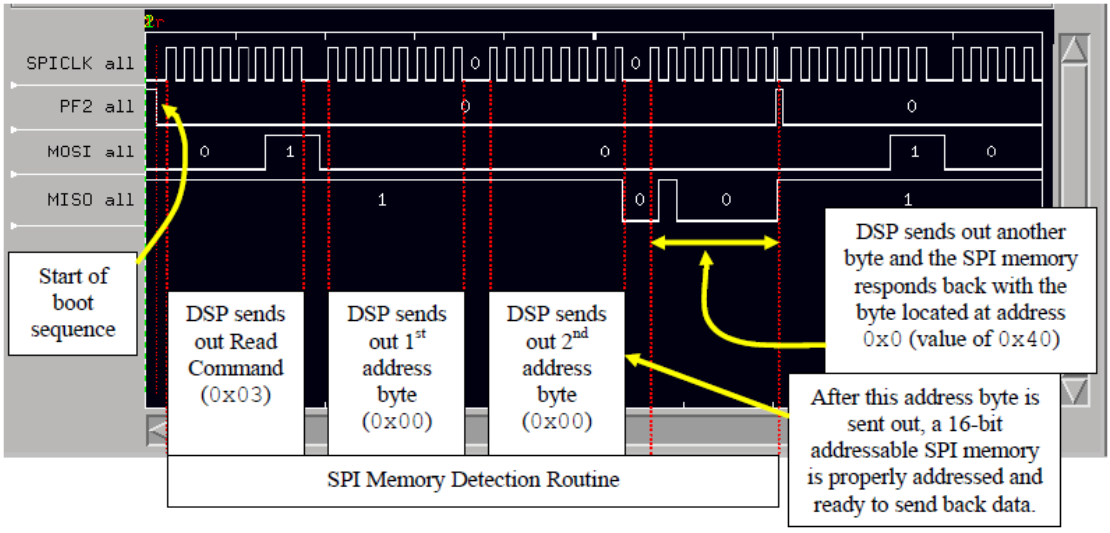


图22. SPI主模式加载过程：SPI存储器检测过程

需要注意的是，图22是用硅版本0.3捕获的。在硅版本0.2上，SPICLK在发送之前和发送过程中都为高电平。

在该点片上Boot ROM已检测到连接了16位可寻址SPI存储器，然后，它发出读命令并发出地址0x0000，以读取Init Code DXE计数块的前10字节文件头。

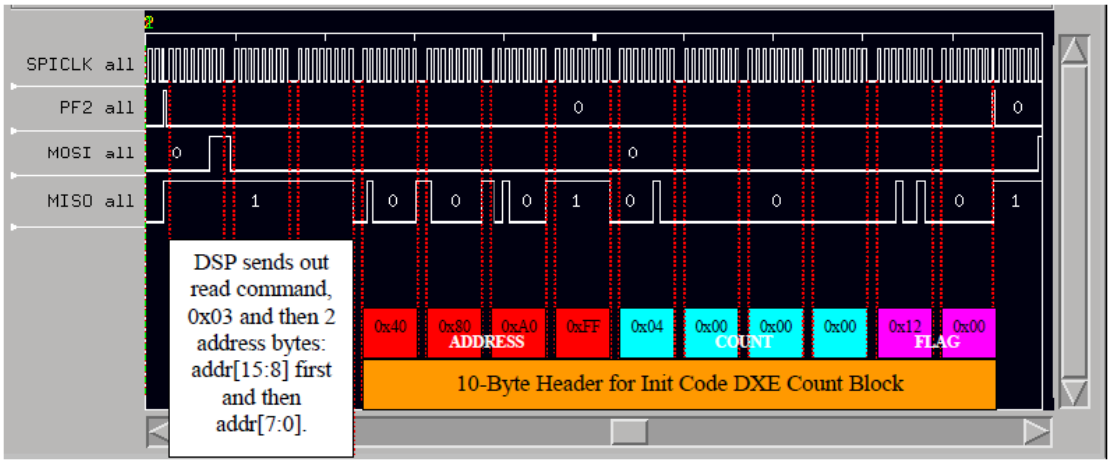


图23. SPI主模式加载过程：加载Init Code DXE块的10字节文件头

由于Init Code DXE计数块为4字节可忽略块，片上Boot ROM将接着发出读命令和两字节地址0x000E，以读取Init Code DXE块1的10字节文件头。当读取到该文件头后，片上Boot ROM就知道块1驻留在存储器的什么位置以及要从该地址加载多少字节。

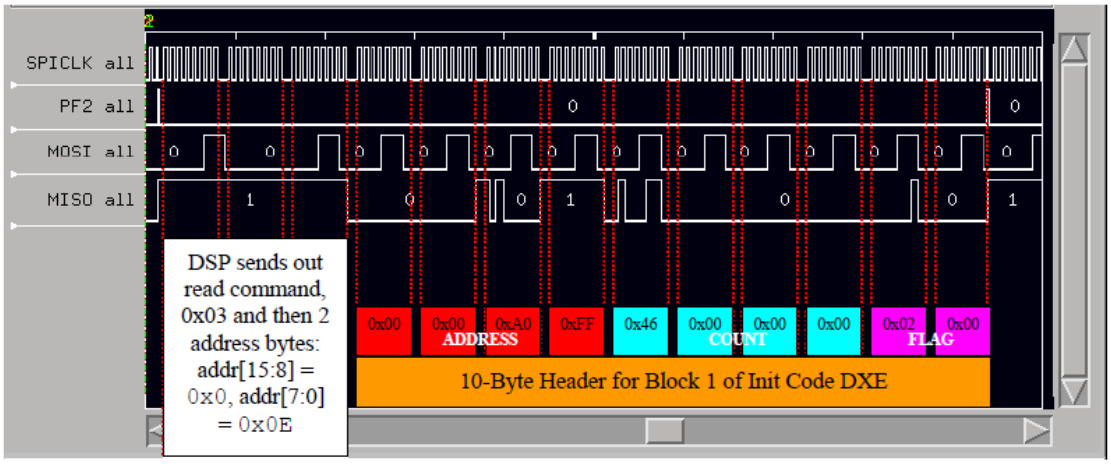


图24. SPI主模式加载过程：加载Init Code DXE块1的10字节文件头

一旦处理完这些信息，片上Boot ROM将再发出一个读命令，并发出地址0x0018来加载Init Code DXE块1。

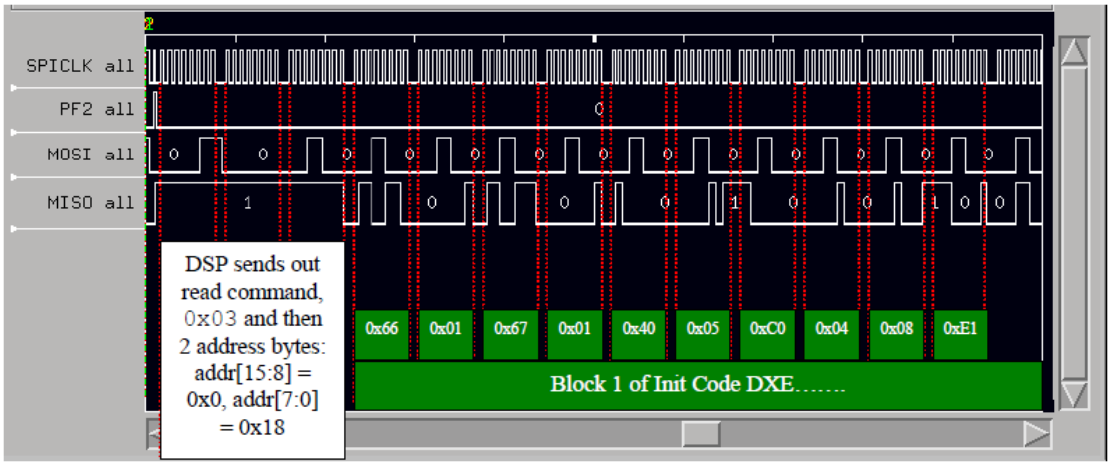


图25. SPI主模式加载过程：加载Init Code DXE的块1

附录：加载模式vs.硅版本

以下加载模式选项用于所有ADSP-BF531/BF532/BF533衍生产品。

硅版本0.1

BMODE[1:0]	说明
00	从连接到ASYNC存储区0的外部16位存储器执行（忽略Boot ROM）
01	从8/16位FLASH/PROM加载（仅支持8位加载）[1]
10	从SPI主模式下的8位可寻址SPI存储器加载
11	从SPI主模式下的16位可寻址SPI存储器加载

[1]仅支持8位加载。16位FLASH将被补零以“模拟”8位加载。

表2. 硅版本0.1加载模式

在硅版本0.1上，不支持IGNORE和INIT块。

在硅版本0.1上，地址0xFF90 0000 – 0xFF90 000F（L1数据存储区B的头16个字节）必须保留。该存储范围由片上Boot ROM用于存储加载文件中每个块的文件头信息，加载完成后，该存储范围可由应用程序在运行过程中使用。

硅版本0.2

硅版本0.2引入了24位SPI模式，SPI设备自动检测使能8/16/24位存储设备，这可被BMODE配置覆盖。更重要的是，硅版本0.2引入了INIT code功能。

BMODE[1:0]	说明
00	从连接ASYNC存储区0的外部16位存储器执行（旁路掉Boot ROM）
01	从8/16位FLASH/PROM加载（仅支持8位加载）[1]
10	保留
11	在SPI主模式下，从8/16/24位可寻址SPI存储器加载（MISO需要上拉）[2,3]

[1]仅支持8位加载。16位FLASH将补零以“模拟”8位加载。

[2]由于Boot ROM中出现错误，硅版本0.2中的SPI主模式加载不支持全0块。0必须包含在该模式的加载文件中。

[3]SPI可加载的文件的第一个字节必须为0x00。

表3. 硅版本0.2加载模式

需要注意的是对于硅版本0.2，地址0xFF80 7FE0 – 0xFF80 7FFF（L1数据存储区A的最后32字节）必须保留。该存储范围由片上Boot ROM用于存储加载文件中每个块的文件头信息。加载后，该存储范围可由应用程序在运行过程中使用。

以上[1,2,3]项只有elfloader实用程序应用在0.2模式（使用–si-revision 0.2命令行转换）下才要注意，这是当前VisualDSP++3.5工具的默认情况。然而，在硅版本0.3后的工具版本中这种情况会有所改变。

硅版本0.3

硅版本0.3引入了SPI从加载。SPI主模式在支持标准8/16/24位SPI存储器的同时还支持来自Atmel的DataFlash设备，它具有真正的16位FLASH/PROM模式特点，也清除了对软件复位的支持。

BMODE[1:0]	说明
00	从连接ASYNC存储区0的外部16位存储器执行（忽略Boot ROM）
01	从8/16位FLASH/PROM加载
10	在SPI从模式下从SPI主机加载
11	在SPI从模式下从8/16/24位可寻址SPI存储器加载，支持以下Atmel DataFlash设备：AT45DB041B，AT45DB081B和AT45DB161B

表4. 硅版本0.3加载模式

需要注意的是对于硅版本0.3，地址0xFF80 7FF0 – 0xFF80 7FFF（L1数据存储区A的最后16字节）必须保留。该存储范围由片上Boot ROM用于存储加载文件中每个块的文件头信息。加载完成后，该存储范围可由应用程序在运行过程中使用。

为了使能硅版本0.3的特性，elfloader实用程序必须在0.3模式（使用–si-revision 0.3命令行转换）下使用，一旦硅版本0.3可用，这将在以后VisualDSP++ 3.5工具升级版本中作为默认情况。当前的elfloader版本（早期-硅版本0.3）默认为0.2模式，且应由–si-revision 0.3命令行转换，以调用硅版本0.3的功能。

硅版本0.3的片上Boot ROM与硅版本0.2完全向下兼容。硅版本0.2和0.3用户应利用–si-revision 0.2命令行调用elfloader。

附录：Blackfin加载文件查看器

Blackfin加载文件查看器(LdrViewer)（见 http://www.blackfin.org/tools）是一个非常有用的实用程序，它可以将加载文件作为输入并拆分、分类为单个.DXE文件，显示为有文件头（ADDRESS，COUNT和FLAG）的单个块。很方便利实用程序帮助用户查看加载文件的内容。将列表3中的文件加载到LdrViewer中，GUI内容如下所示：

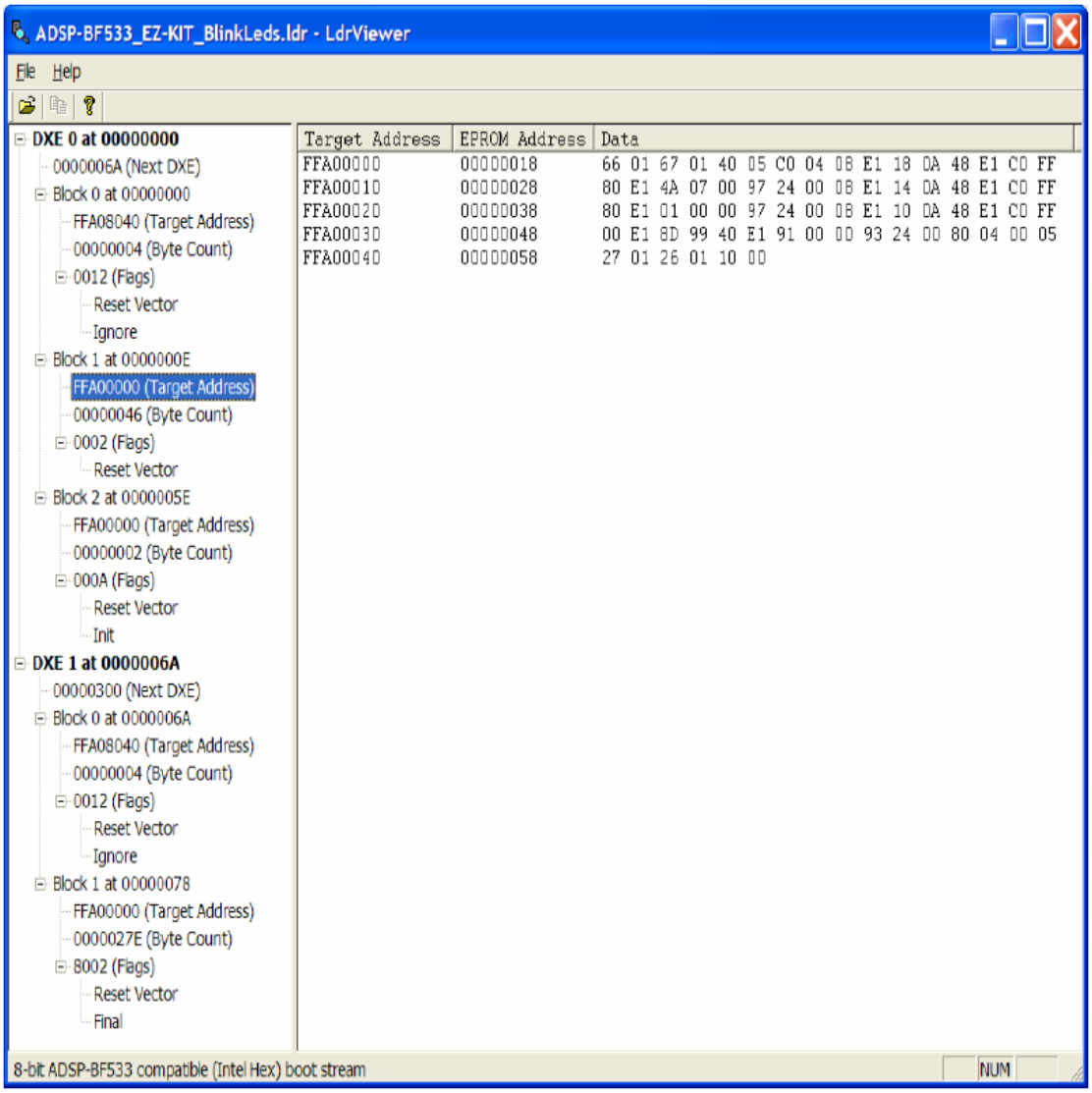


图26. Blackfin加载文件查看器实用程序

需要注意的是，LdrViewer实用程序不是标准VisualDSP++软件工具集的一部分。

参考文献

[1] *VisualDSP++ 3.5 Loader Manual for 16-bit Processors.* Rev 1.0, October 2003. Analog Devices, Inc.

[2] *ASDP-BF533 Blackfin Processor Hardware Reference.* Rev 3.3, September 2008. Analog Devices, Inc.

[3] *Running Programs from Flash on ADSP-BF533 Blackfin Processors (EE-239).* Rev 1, May 2004. Analog Devices Inc.

[4] *AT45DB041B DataFlash Datasheet.* April 2004. Atmel Inc.

[5] *AT45DB081B DataFlash Datasheet.* November 2003. Atmel Inc.

[6] *AT45DB161B DataFlash Datasheet.* November 2003. Atmel Inc.

文档记录

Revision	Description
Rev 4 – September 29, 2008 *by M. Kokaly-Bannourah*	• Added informational bullet for Figure 15.
Rev 3 – January 11, 2005 *by H. Desai*	• Added informational bullets for reserved memory regions
Rev 2 – December 17, 2004 *by H. Desai*	• Added pull-ups to the /AMS0 for all flash boot figures • Added a pull-up to PF2 in Figure 20 • Described pull-up or pull-down on SPICLK in Figure 20 • Specified maximum SPI Baud Rate for SPI Slave Mode Boot • Indicated that rev. 0.1 does not support IGNORE and INIT Blocks • Changed the term “feedback strobe” to host wait (HWAIT) signal • Added a discussion on Atmel DataFlash derivatives
Rev 1 – June 03, 2004 *by H. Desai*	Initial Release