HHBF533/HHBF561
软件移植开发手册*

Version 0.1

Meihui Fan
范 美 辉
<mhfan@ustc.edu>

2005年1月18日

*本文档大部分为BF533 和BF561 通用。事实上，它们用的是同一个分发包，只是配置选项的不同。
†http://www.hhcn.com

目录

List of Figures

List of Tables

1 前言

ADI[1] 公司雇佣了LG Soft India 公司做其Blackfin 系列DSP 处理器（主要是BF533）的μClinux 移植（μClinux for Blackfin 的主页：http://blackfin.uclinux.org），这包括GNU/Toolchain、U-Boot 和μClinux 三个开源项目的移植，并且所有代码都继承遵循GNU GPL 许可证协议。我们的HHBF533/HHBF561 开发板的μClinux 系统移植和应用开发以此为‘草稿’来进行。

1.1 移植开发的指导原则

鉴于当前官方的μClinux Blackfin 移植还很不稳定和成熟，μClinux 包括GNU/Toolchain 都存在诸多问题和BUG，我们自己的移植开发（包括μClinux 和GNU/Toolchain）均有必要采用与官方CVS 代码同步的方式，以保证及时更新最新的移植进展和BUG 修正。

所以，为了保证代码的清晰和便于与官方CVS 同步，移植的时候在编码和文件布局上采取以下策略：

1. 理解代码的本质，寻找最基本的不同，做最少和最通用的修改；并且保证所做的修改不负面影响原始的使用意向。
2. 对于在官方文件中做的修改一律做上自己的标记（`'mhfan'`）。
3. 对于相对独立的功能模块和驱动程序的开发一律建立以`'hhbf'` 开头的文件夹放置代码文件，不破坏原来的目录文件组织。
4. 对于相对独立的功能模块和驱动程序的开发尽量地反映在配置菜单选项中。
5. 充分利用预编译器的功能；常量尽量采用宏定义。
6. 每日与官方的μClinux CVS 内核代码比较（`diff`）并手动更新；对于其它部分：μClinux 应用程序代码、U-Boot、以及GNU/Toolchain 代码则隔一段时间（如一周）更新一次。
7. 镜像并定时更新官方μClinux CVS Repository 以便于将来多人协作开发。

2 U-Boot 移植

在U-Boot for Blackfin 的移植中，CPU 的初始化工作主要包括：

- 各种时钟频率的设置
- 异步内存接口（ABI[2]）的初始化设置

[1] Analog Device Inc，公司主页：http://www.analog.com
[2] 用于FLASH、内存映射IO 等

- SDRAM 控制器（SDC）的初始化设置
- 集成串口的初始化设置
- 中断向量表的初始化设置

此外，为了让U-Boot 支持以太网下载和FLASH 烧写，还必须添加以太网芯片和FLASH 芯片的驱动程序。

在官方的U-Boot 移植中，实现上述CPU 初始化的代码大部分借用了Linux 内核移植的成果；板级配置文件（'u-boot/include/configs/*.h'）充当了内核配置文件的作用（尤其在最新的U-Boot 代码中，这一点更为明显）。所以在我们的移植中，即使是给BF561 的移植也可以不需要修改太多的代码。

由于设置和修改串口波特率的代码是通过执行期动态计算得出相关寄存器（UART_DL）所需要的值[3]，也就是说只要设置好了CPU 各种时钟频率，总能将串口设置在特定的波特率工作；所以我们不需要修改串口初始化设置的代码。另外，由于U-Boot 中基本没有使用外部中断，于是中断向量表就显得很机械也很简单，因而也就不需要修改了。

对于SDC 的初始化设置，也可以由一些'先验知识'计算出来（通过宏定义由预编译器计算），只需要显式地设置内存大小（HHBF533–32M, HHBF561–64M）和列地址位宽（9bit）即可。

剩下的问题就是CPU 时钟频率相关寄存器的设置了。其中核心时钟倍频由'PCC_CTL'寄存器中的14-9 位决定，而系统时钟分频倍率由'PLL_DIV'寄存器中的3-0 位决定。当然，我们同样用宏定义抽象我们关心的概念（如外频、倍频、系统时钟等），让CPP 为我们计算确切的寄存器值。关于这些寄存器位的详细描述请参考BF561 的CPU 硬件参考手册**??**。

下表是典型的CPU 寄存器初始化设置，供调试时参考：

寄存器名	系统复位值	HHBF533 的设置	HHBF561(CORE A) 的设置
EBIU_AMBCTL0	0xffc2ffc2	0x7bb07bb0	0x7bb07bb0
EBIU_AMBCTL1	0xffc2ffc2	0x99b37bb0	0x99b37bb0
EBIU_AMGCTL	0x00f2	0x00f8	0x00f8
EBIU_SDBCTL	0x00000000	0x00000013	0x00000015
EBIU_SDRRC	0x081a	0x074a	0x074a
EBIU_SDGCTL	0xe0088849	0x0091998d	0x0091998d
UART_DLL	0x0000	0x0012	0x0012[a]
PLL_CTL	0x1400	0x1400	0x2800

以太网（硬件采用DM9000 芯片）驱动的原始代码来自Davicom 官方发布的最新V1.25 版给linux-2.4.x 的驱动；移植给U-Boot 主要有以下步骤：

1. 初始化异步内存（以太网片选地址－－0x2C000000－－的内存映射）访问控制接口，这在CPU 初始化过程中已经考虑并设置好了，所以在以太网驱动中无需重复了。

[3]可以通过读取CPU 时钟频率相关寄存器的值换算出系统时钟（SCLK），再通过公式：$BAUDRATE = SCLK/(16 \times UART\_DL)$ 计算出来。

2. U-Boot 中没有使用中断，也不需要以太网收发包的统计信息，所以注释掉相关的代码，中断处理改成计时查询。

3. 定义与内核兼容的数据结构一些接口（如inb，inw，inl，outb，outw，outl），这样可以保持原来的大部分代码结构。

4. 还有一些细微的代码需要调整，可以根据编译情况来处理。

5. 定义U-Boot 中以太网驱动的调用接口：eth_init，eth_send，eth_rx，eth_halt。

至于FLASH 驱动，HHBF533 所用的FLASH（AM29LV16008-90EC）与EZKIT533 的兼容，可以直接使用原来的代码。而在HHBF561 上所用的FLASH（TE28F128J3C150），其烧写访问的操作流程，比较简单，只需在原来的代码结构上换上芯片手册中给出的操作流程就行了，这里不详细罗列。另外一个需要设置的地方是FLASH 的大小、块数、块大小等。

2.1 调试记录

现象描述	原因分析	解决方法
用仿真器跑没有反应，挂起CPU 发现每次都停在同一条指令上（STI R2;）	软件设置了PLL 相关寄存器后让CPU IDLE 以等待CPU 频率稳定，但是连着仿真器的时候CPU 不能从IDLE 状态正确返回。估计根本原因是CPU 内部问题或者仿真器及Visual DSP++ 有BUG。（BF533 上没有这个问题）	幸运的是不带仿真器的时候，CPU 可以正确运行。
运行之后，minicom 中出现乱码	这通常是波特率设置不匹配。由于波特率是动态计算的，并且计算公式是确定的，所以不需要修改代码，只需修改波特率计算相关的设置。	波特率计算相关的设置包括：波特率值、晶振频率、倍频设置、系统时钟分频设置，这些都在板级配置文件里面。
TFTP 下载不成功	跟中发现以太网的初始化程序都没有调用	在以太网设备初始化函数里面加上相应的DM9000 初始化函数
U-Boot 启动的时候显示没有检测到以太网芯片	跟踪发现读到的DM9000 标志寄存器值不对	试探性的每次要读寄存器的时候连续读两次，解决了问题。

一般来说，如果足够的细心处理配置文件和一些头文件里面的内容，移植U-Boot 并不需要做太多工作。另外，有一个建议就是把编译的警告打到最高，编译器通常总是能够帮我们找到代码中的问题。

2.2 参考资料

- U-Boot 源码树中的‘README’ & ‘ReleaseNotes’。
- U-Boot 的BF533 移植的项目主页：http://blackfin.uclinux.org/projects/uboot533/；
  CVS Repository: ‘:pserver:anonymous@cvs.blackfin.uclinux.org:/cvsroot/uboot533’。
- U-Boot 原始的项目主页：http://u-boot.sourceforge.net。
- “ADSP-BF561 Blackfin Processor Hardware Reference”.
- “Intel 28F128J3A, 28F640J3A, 28F320J3A”——Intel 系列FLASH 芯片手册。

3 μClinux 移植

关于CPU 初始化的主题在上一节“U-Boot 移植”2 中已经详细阐述过，此不赘述。

这里说明一下BF561 移植的一些要点。虽然BF561 是对称双核的结构，但由于我们当前的移植都是把BF561 当作单核来使用，它的每一个核都与BF533 兼容，所以我们采用以下策略：

1. 在BF561 系统内存映射寄存器（System MMRs）的宏定义头文件（‘include/asm/bfinnommu/board/{bf561.h,bf561_irq.h, defBF561.h,cdefBF561.h}’）中定义与BF533 兼容的宏，即所有与BF533 兼容的寄存器都再定义一个与BF533 的定义相同的宏名。
2. 只一个头文件‘include/asm/bfinnommu/blackfin.h’ 中封装CPU 的信息，即在该文件中判断从内核配置菜单得到的CPU 信息，据此包含相应的头文件。
3. 在系统其它模块和驱动程序中所有需要CPU 相关信息的文件都只包含‘blackfin.h’，这样官方的BF533 μClinux 内核移植中的许多驱动可以基本不需要做什么修改就可以使用。

另外，从BF533 到BF561 的移植中一个非常关键的地方在于BF561 每一个核的中断资源和中断相关寄存器（SIC_IWR，SIC_IAR，SIC_ISR，SIC_IMASK）都是BF533 的两倍，所以要对BF533 的中断处理相关函数和接口做一些技巧性的改动。所有的这些改动都在‘arch/bfinnommu/mach-bf533/ints-priority.c’ 文件中。

关于中断还有一个要注意的地方是串口驱动的中断处理，官方驱动中是用明文数值来判断是否串口中断，而BF533 与BF561 中串口中断位并不一致，所以需要修改。我们已经定义了一个‘IRQ_UART_MASK’ 的宏来做判断。

3.1 调试记录

现象描述	原因分析	解决方法

分散于后面各章节。

3.2 参考资料

- μClinux 源码树中的‘README’ & ‘ReleaseNotes’ 和‘Documentation/{Adding-Platforms-HOWTO,Adding-User-Apps-HOWTO,FAQ}’。
- Linux 源码树中的内核文档。
- GNU/Toolchain 的Blackfin 移植的项目主页：http://blackfin.uclinux.org/projects/gcc3/；CVS Repository: ‘:pserver:anonymous@cvs.blackfin.uclinux.org:/cvsroot/gcc3’。
- μClinux 的BF533 移植的项目主页：http://blackfin.uclinux.org/projects/uclinux533/；CVS Repository: ‘:pserver:anonymous@cvs.blackfin.uclinux.org:/cvsroot/uclinux533’。
- “ADSP-BF561 Blackfin Processor Hardware Reference”.

4 以太网驱动和应用

以下是移植DM9000X 以太网驱动所需要做的工作：

1. 初始化异步内存（以太网片选地址－－0x2C000000 on BF561, 0x20300000 on BF533－－的内存映射）访问控制接口，这在CPU 初始化过程中已经考虑并设置好了，所以在以太网驱动中无需重复了。
2. 由于片上SROM 中并未写入以太网地址，所以在以太网检测 / 初始化例程中为DM9000 芯片分配合法物理地址（‘random_ether_addr’）。
3. 使用‘PF10’ 做以太网中断线，初始化该引脚（使之工作在上升沿边沿触发输入模式），并在中断处理例程中清该中断引线。
4. 将中断处理例程中的自旋锁函数换成‘spin_lock_irq’（原来为‘spin_lock’）。
5. 将中断处理中的底半部处理策略（由‘tasklet’实现）改成即时处理。
6. 为了少修改原始代码，我们用内核模块初始化宏定义‘module_init’ 来加载DM9000X 驱动。
7. 为内存映射的IO 访问加上DCACHE 的管理（有必要给‘asm-bfinnommu/io.h’打个补丁，以杜绝其它IO 驱动的隐患）。

另外，由于原始代码是为linux-2.4.x 的内核写的，而linux-2.6.x 在以太网驱动的接口上有一些变动（如没有了‘init_etherdev’函数，代之以‘alloc_etherdev’），所以在一些细节上还要做一些修改。

以上工作可以让以太网正常工作了，但是IPV6 的支持还有一个问题。Linux 内核中关于IPV6 地址校验（‘csum_ipv6_magic’）有两套实现：一是体系结构相关的汇编实现，在‘include/asm/checksum.h’ 文件中，另一是体系结构无关的C 语言实现，在‘include/net/ipv6_checksum.h’ 文件中（它们靠宏定义：‘_HAVE_ARCH_IPV6_CSUM’

来切换）。在官方的BF533 移植中体系相关的实现是以m68k 的实现为模板，估计没有经过调试和验证，所以导致IPV6 的通信过程中地址验证总是失败。但是C 语言的实现可以很好地工作。

4.1 调试记录

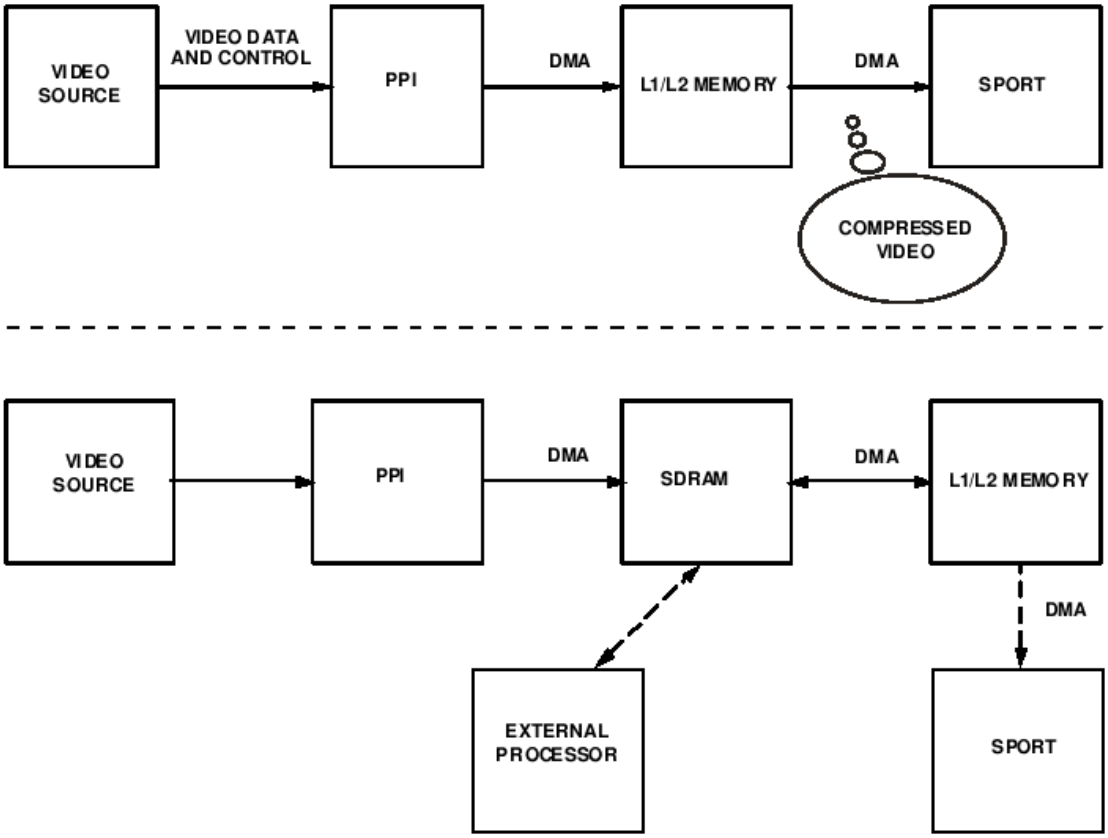
现象描述	原因分析	解决方法
某一个内核版本打开ICACHE 后以太网没有反应	跟踪没有发现任何问题板子收到真确的包，也正确地调用了发包函数，但问题肯定与CACHE 有关	查阅内核文档发现有一些与CACHE 有关的函数，试探性地使用CACHE LOCK 保护以太网发包函数，于是以太网通了。
以太网不能正常收发，minicom 中显示包长度不匹配	跟踪发现读包长度的时候错了位	在读包长度之前再发一次基地址，于是解决了问题
测试IPV6 不通，minicom 中显示‘‘4 printk messages suppressed’’	自底而上跟中以太网包的处理发现最终原因是IPV6 的地址验证失败，校对PC 发的以太网包和板子收的以太网包发现它们是一致的，看来是校验函数有问题。	注释掉体系结构相关的校验函数实现，改用IPV6 协议中的C 语言实现，于是解决了问题。
打开DCACHE 和ICACHE 系统运行一段时间（同时做ls -R 和flood ping）后触发硬件中断（IRQ 0x05）并且显示‘‘System MMRs Error’’	这是指令错误，原因可能是CACHE 不一致或者硬件不稳定导致	给以太网的IO 访问函数加上CACHE CLEAN 和CACHE FLUSH 暂时解决了问题。

5 视频驱动

我们的视频处理流程见图1。在软件上，一般来说，视频驱动包括四层：

1. I2C 驱动，用于与视频ADC/DAC 芯片通讯。
2. 视频ADC/DAC 芯片本身的驱动。
3. PPI 和DMA 的驱动，用于视频数据流的输入输出。
4. 为应用层提供通用的规范接口。

首先是I2C 驱动，我们利用官方Linux 内核中官方I2C 通讯协议的实现结构，只在最底层实现板级硬件相关的I2C 位操作例程。（关于Linux 内核中I2C 驱动的结构请参考内


VIDEO SOURCE
VIDEO DATA AND CONTROL
PPI
DMA
L1/L2 MEMORY
DMA
SPORT
COMPRESSED VIDEO
VIDEO SOURCE
PPI
DMA
SDRAM
DMA
L1/L2 MEMORY
DMA
EXTERNAL PROCESSOR
SPORT

第 1 图: 视频处理流程

核目录树中的‘Documentation/i2c/*’。）由于Blackfin 系列处理器中没有I2C 模块，所以我们的板子上使用两个GPIO（‘PF0 & PF1’）来模拟I2C 总线。

我们的视频芯片用的是ADV7171 和SAA7113H，前者的驱动在内核源码中有其兼容芯片ADV7170 的实现，而后者的驱动在开源项目RivaTV 中也有实现，由于我们使用了内核的I2C 驱动，所以这一层可以直接挪用这两个驱动的实现。这两个驱动实现都定义了一套寄存器初始化表，只有修改这些寄存器值芯片就能正常工作了。

PPI 和DMA 驱动，我们都只是用几个简单的函数来实现，而没有做成一个规范的驱动。（最新的BF533 内核移植中已经包含了统一的DMA 驱动，我们虽然也把它移植到BF561 上，但到现在为止还没有充分利用它。）其中视频输入使用PPI0，视频输出使用PPI1。

最后是应用层接口，主要有两种框架：Frame Buffer 框架和Video 4 Linux V2 框架。

5.1 Frame Buffer 驱动

Frame Buffer 驱动本身很简单，因为内核代码中已经有一个框架代码了，只需将那些数据结构、函数接口填充一下就可以了。关键的部分在于视频数据的转换：Frame Buffer 的应用程序使用的是RGB 裸位图数据，而它的驱动使用的是ITU-656 数字电视的视频格式。转换程序借鉴了RivaTV 的优化实现。

5.2 Video 4 Linux V2 驱动

Implemention in progress.

5.3 调试记录

现象描述	原因分析	解决方法

驱动未完善，有待总结。

6 视频编解码联调

6.1 网络传输的应用程序

6.2 设计思想

为了调试和演示视频编解码的方便，我们需要实现两套网络传输的应用程序：其一把开发板采集编码后的码流传送到PC 机上存成文件，其二为开发板提供视频解码的码流。在这里，出于简单实现的考虑，我们不准备采用太复杂的协议，而只用socket /TCP 连接作为网络通信的方式。我们假设网络状况足够好，以至于能够满足实时视频码流传送的要求。

6.3 程序流程

关于socket 通信的细节请参考**??**，另外网上也有很多示例代码，此不赘述。

6.3.1 视频编码传输

[vsvr.c]	[hvclt.c]
初始化码流缓冲区	
监听socket 端口等待连接	
协商连接	建立socket/TCP 连接
启动CORE B（参考9.1）	
查询码流缓冲区并发送数据	接收数据存成文件
传送完毕，挂起CORE B	关闭文件和socket 连接

6.3.2 视频解码传输

[vclt.c]	[hvsvr.c]
初始化码流缓冲区	监听socket 端口等待连接
建立socket/TCP 连接	协商连接
启动CORE B（参考9.1）	读取码流文件并发送数据
接收数据并提交给CORE B	
接收完毕，挂起CORE B	关闭文件和socket 连接

6.3.3 码流缓冲区的管理

为了避免与CORE B 竞争访问同一个缓冲区，码流缓冲区采用双缓冲区切换的管理策略：开发板上的vsvr 和vclt 进程都是在一开始就分配两个单元的缓冲区，并串成循环缓冲区；每次要发送数据之前依次查询下一个缓冲区直至它存满数据（flag 为‘1’）。

以下是缓冲区的结构定义：

```
typedef struct BufDesc {
        struct BufDesc* next;
        unsigned char*  base;
        int             flag;
}   Buf_Desc;
```

6.4 调试记录

现象描述	原因分析	解决方法

见吴卅建的调试文档。

6.5 参考资料

- “ADSP-BF561 Blackfin Processor Hardware Reference”.
- Linux 源码树中相应模块 / 驱动（I2C, FB, V4L）的内核文档。
- Frame Buffer FAQ: http://www.sunhelp.org/faq/FrameBuffer.html
- RivaTV Project: http://rivatv.sourceforge.net/
- Video 4 Linux 2: http://linux.bytesex.org/v4l2/
- V4L2 Resource: http://www.exploits.org/v4l/

7 音频驱动

与视频驱动类似，音频驱动也分为几个层面：

1. SPI 驱动，用于实现设备通讯。
2. 音频AD/DA 芯片（ADI1836A）的驱动。
3. SPORT 和DMA 驱动实现音频输入输出数据的传输。

音频驱动采用Advanced Linux Sound Architecture(ALSA)[5] 的框架。由于得到了一份非官方的EZKIT-BF533 开发板的音频驱动，而且跟原作者反馈建议的结果使得这份驱动更具有可移植性，所以只做了很少的移植工作（包括DMA、IRQ，以及一些System MMRs 地址），于是对驱动的实现细节也没有做太多的学习。

7.1 调试记录

现象描述	原因分析	解决方法
调试音频播放的时候发现一调用`ioctl` 的时候就死机	跟踪发现音频驱动中`32` 位乘法的实现总是出错，主要是一个`64`位的函数参数传递有误，导致其后的某一个参数为零，而这个参数是做为除数来用的。	这个问题有点莫名奇妙，估计是编译器有`BUG`，换了一个新的编译器就解决了问题。

[5] ALSA 是linux-2.6.x 的标准音频驱动框架。

8 CACHE 问题

BF533/BF561 的CACHE 配置取决于32 对CPLB[6] 寄存器，其中DCACHE 和ICACHE 各16对。每一对寄存器分别指定了CACHE 映射的起始地址和缓存策略，后者包括映射的存储空间大小、访问权限、缓存方式、是否锁定，以及是否缓存和是否有效等。（详细的细节请参考CPU 硬件参考手册“Memory”一章。）

如果要使用CACHE 必须保证CPLB 寄存器的配置覆盖了所访问的所有内存区域，即使在某些区域并不需要缓存（通过置相应CPLB 配置寄存器中相应位来指定是否缓存），也必须有相应的CPLB 配置。

以上所说是硬件级的静态CACHE 管理。由于CPLB 寄存器有限，很可能不能覆盖所有的内存区域，所有需要加入OS 级的软件动态CACHE 管理。在这种管理策略中，同样有两张相应于CPLB 配置寄存器对的页面描述表（Page Description Table, PDT），里面可以填上任意项，这样当硬件的CACHE 管理找不到所访问的内存区域就触发一个异常，然后在异常处理中从PDT 中找到相应的表项来替换某个CPLB 配置寄存器对。

在访问存储空间的时候，如果所访问的存储空间是可缓存的，并且有潜在的不经过当前CPU 的存储内容访问，则当前CPU 在读取该存储空间内容前必须执行CACHE CLEAN，在写该存储空间内容后必须执行CACHE FLUSH。这主要有三种情况：

1. 另一个核访问了该存储空间
2. 某一个DMA 通道访问了该存储空间
3. 该存储空间是某个外设的IO 内存映射

对于BF561 的CACHE 配置，我们采取以下原则：

- L1 SRAM、L2 SRAM 配置成不缓存、不可替换；
- SDRAM 缓存32M，其中内核所有内存区域配置成不可替换；
- 在页面表述表中缓存所有SDRAM 空间，FLASH 空间和网络芯片的地址空间。

[6]

8.1 调试记录

现象描述	原因分析	解决方法
内核启动的时候显示‘‘kernel panic: No CPLB Address match’’	通常是用了CACHE 而没有覆盖所访问的存储空间	添加相应的CPLB 项和PDT 表项即可
系统运行不稳定，经常进入硬件错误中断(IRQ 0x05) 并显示：‘‘External Memory Addressing Error’’	可能是EBIU 设置有问题或者编译器指令生成有误或者硬件不稳定，不知道确切原因	最新的内核＋最新的编译器已没有出现这一问题。

9 其它的一些驱动和模块

9.1 二级引导器

实现这个驱动是为了方便在应用程序中加载CORE B 代码，并唤醒CORE B 执行代码。因为Blackfin 处理器具有不完全的MMU 和指令操作模式的限制。考虑到接口简单易实现的因素，我们使用proc 文件系统作为驱动的应用层接口。

这个驱动包括两个部分：二级引导加载器（用于加载CORE B 的运行代码）和SICA_SYSCR 寄存器的操作（用于唤醒或挂起CORE B）。

二级引导加载器挪用自ADI Visual DSP++ 中给的示例，只作了小的修改；驱动的实现本身很简单这里不详述。

9.2 片内SRAM 的利用

这个驱动的实现是为了演示如何利用BF533/BF561 中闲置的片内SRAM。驱动通过在C 代码中添加编译器指示（compiler directive, GCC 里通过‘__attribute__’来实现）并结合链接脚本来指示编译器把代码 / 数据放到特定的内存位置。另外，在驱动中还实现了一个proc 文件系统接口，用以演示调用片内SRAM 中代码的执行和数据的操作。

以下是测试代码：

On-chip SRAM Utilization

```
1 static int  __attribute__((__section__(".data.l2"))) l2_var = 0;
2
3 static int  __attribute__((__section__(".data.l1_a"))) l1_a_var = 1;
4 static int  __attribute__((__section__(".data.l1_b"))) l1_b_var = 2;
5
6 void __attribute__((__section__(".text.l1")))   l1_sram_func_test(void)
7 {
```

```
8       volatile int i=12345;
9
10      printk( "\033[1;31m0x%p:␣%s\n"
11              "\033[1;34m0x%p:␣%d(stack␣variable)\n"
12              "\033[1;32m0x%p:␣%d(l1_a_var)\n"
13              "\033[1;33m0x%p:␣%d(l1_b_var)\033[0m\n"
14              "\033[1;36m0x%p:␣%d(l2_var)\033[0m\n",
15              l1_sram_func_test, __FUNCTION__, &i, ++i,
16              &l1_a_var, ++l1_a_var,
17              &l1_b_var, ++l1_b_var,
18              &l2_var, ++l2_var);
19
20      asm("nop;");
21      return;
22  }
23
24  void __attribute__((__section__(".text.l2"))) l2_sram_func_test(void)
25  {
26      volatile int i=67890;
27
28      printk( "\033[1;31m0x%p:␣%s\n"
29              "\033[1;34m0x%p:␣%d(stack␣variable)\n"
30              "\033[1;32m0x%p:␣%d(l1_a_var)\n"
31              "\033[1;33m0x%p:␣%d(l1_b_var)\033[0m\n"
32              "\033[1;36m0x%p:␣%d(l2_var)\033[0m\n",
33              l2_sram_func_test, __FUNCTION__, ++i,
34              &l1_a_var, ++l1_a_var,
35              &l1_b_var, ++l1_b_var,
36              &l2_var, ++l2_var);
37
38      asm("nop;");
39      return;
40  }
```

需要说明的是，由于要利用片内SRAM 就必须使用ELF 格式的内核映像了（默认的内核映像是无格式的裸二进制映像）。

9.3 调试记录

现象描述	原因分析	解决方法
测试发现不能正常使用片内SRAM ，系统死机	用仿真器跟踪发现相应的代码没有正常加载到相应的位置	修改U-Boot 中的memcopy 函数解决问题。
测试L2 SRAM 时发现系统总是跑飞，并导致死机	反汇编发现L2 中的函数调用地址有误，只有地址的24 位值；估计是编译器做符号解析的时候只取绝对地址的24 位，或者相对地址的16 位导致	仍未解决。

9.4 多线程 / 进程和IPC 测试

虽然官方的发布手记中说仍未支持，但据我们的测试当前的μClinux on Blackfin 中已能支持POSIX thread, System V IPC 简单的测试代码，在内核中只对System V IPC 的共享内存部分做了很少的修改。测试代码全部在‘uClinux-dist/user/hhtech’ 目录下（其中大部分来自南京天朗电子的陶猛）。

测试很简单，无需记录。

10 系统性能

给μClinux 移植了lmbench（a system benchmarks project），已把它加到‘Customize Vendor/User Settings/Miscellaneous Applications’ 菜单中。

移植过程比较简单，主要是修改编译控制；另外，源代码中的一些文件路径也需要修改。

表1 是测试结果：

第 1 表: 系统性能——时延和带宽

测试项目	HHBF561[7] (us)
系统调用	10.0518/15.2089/14.5280//[8]
进程创建	411.4172/456.6911/28709.3947[9]
上下文切换	-
中断响应	-
网络延时	//[10]
信号安装 / 捕获	40.164/48.511[11]
网络带宽	//[12]
内存带宽	

10.1 调试记录

现象描述	原因分析	解决方法
测试进程创建的时候发现系统总是没有反应	估计是测试代码中使用fork 导致，检查代码之后发现果然如此	用一个宏定义把fork 替换成vfork
测试上下文切换的时候发现系统没有反应	跟踪代码发现，虽然fork 被替换成了vfork，但是vfork 之后没有创建新进程，而是仅仅调用父进程的一些函数；然而vfork 并没有复制父进程的代码，因而导致了问题	这个问题解决起来会有一些麻烦，暂时没有花时间去解决。

11 当前存在／未决的问题

系统设置 ：DCACHE & ICACHE 都打开。

测试过程 ：NFS mount 了一个PC 机根目录，同时做ls -R 和flood ping。

测试结果 ：系统正常的持续繁忙运行了40 多个小时（从周六上午到周一上午9:30）。

- GNU/Toochain 存在BUG，一些视频指令不能识别，更严重的是错误地解释一些指令；这些BUG 正在被逐步修正。
- Frame Buffer 驱动有待于完善和简单GUI 的测试。
- 音频驱动直接播放的效果不是很好，速度慢、频率高；还没有找到原因。
- L2 SRAM　不能正常使用，这是编译器的问题－－符号解析有误。
- V4L2 驱动的实现。

参考资料

- “*Analog Device Inc.*”, 公司主页：http://www.analog.com.
- “ADSP-BF561 Blackfin Processor Hardware Reference”.
- “*Advanced Programming in the UNIX Environment*”, W. Richard Stevens, 1992, Addison Welsley Publishing Company.
- “*Unix Netword Protramming*”, Volum1 Networking APIs: Sockets and XTI (Second Edition, W. Richard Stevens, 1998, Prentice Hall PTR.
- U-Boot 原始的项目主页：http://u-boot.sourceforge.net。
- U-Boot 的BF533 移植的项目主页：http://blackfin.uclinux.org/projects/uboot533/；
  CVS Repository: ‘:pserver:anonymous@cvs.blackfin.uclinux.org:/cvsroot/uboot533’。
- GNU/Toolchain 的Blackfin 移植的项目主页：http://blackfin.uclinux.org/projects/gcc3/；
  CVS Repository: ‘:pserver:anonymous@cvs.blackfin.uclinux.org:/cvsroot/gcc3’。
- μClinux 的BF533 移植的项目主页：http://blackfin.uclinux.org/projects/uclinux533/；
  CVS Repository: ‘:pserver:anonymous@cvs.blackfin.uclinux.org:/cvsroot/uclinux533’。

A BF533 vs. BF561

BF561 是双BF533 核心（$BF561 \lessapprox BF533 + BF533$）。下表比较两者的CPU 资源：

	BF533	BF561
L1 SSRAM	4K	4K
L1 ISRAM	16K + 64K	16K + 16K
L1 DSRAM	16K + 16K	16K + 16K
L2 SRAM	—	128K
DMA	8 DMA + 4 MDMA	三个独立通道（DMA1,DMA2,IMDMA） 2 × (12 DMA + 4 MDMA) + 4 IMDMA
IRQ	7 + 24 + 1 + 2	7 + 59 + 1 + 2 + 2
SPI	1	1
PPI	1	2
UART	1	1
SPORT	2	2
PFs	16	48
RTC	1	–
Timer	3	12
Watch Dog	1	1

第 2 表: BF533 vs. BF561