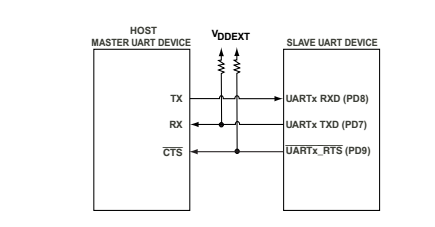
Uart流控制功能

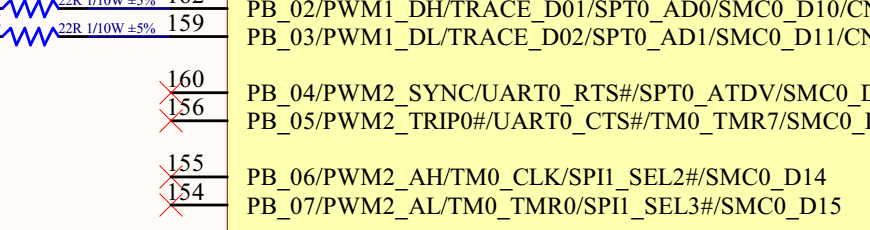
驱动组 王彬 2016.08.17

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 时间 | 作者 | 说明 |
| V1.0 | 2016.8.17 | 王彬 | 第一版本draft |
|  |  |  |  |

# 1:硬件连接



UART Slave Boot Mode使用的是uart0。现在板上UART0\_RTS没有接，需要飞线。上图中没有考虑线驱动和电平的转化。因为ARM那边是TTL电平的，应该可以直接和FPGA直连。



其它的TXD和RXD都引出来了。

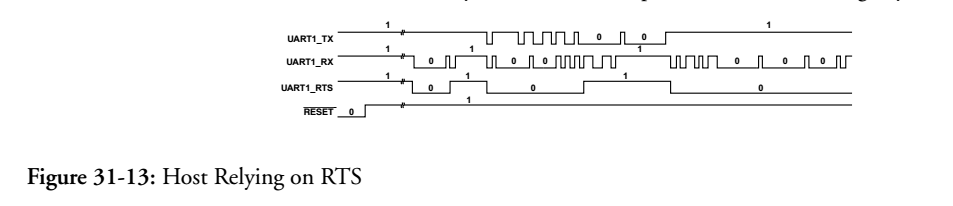
# 2：操作流程

## 2.1：简介

Uart0初始化时通过自动波特率进行监测。也可以用boot routine API做一些定制。

Boot期间，主设备依赖从设备的RTS信号，但是从设备不需要主设备的RTS信号，因为当从设备reset或者是执行pre-boot的时候主设备的RTS都是高阻态。

当slave uart使能后，slave uart的RTS立刻拉低，通过此信号来通知host发送boot stream数据。当是半双工连接的时候，host需要等待它从slave接收到4bytes之后才发送数据。当处理fill或者是initcode的时候需要更多的时间去延迟host发送数据。Uart控制寄存器中的RFRT位在uart slave boot的时候被清除，根据手册上描述，RFRT位为零，那么当rxbuffer已经有四个words，第五个startbit被监测到的时候，UART\_RTS 变为无效，（该信号低电平有效，所以变为高电平。）当rxbuffer中不再含有多于4个words的时候，又变得重新有效，也就是变为低电平。



这个图想表达的意思是？怎么看出host是接收4byte之后才发送？

RTS怎么进行流控？

## 2.2：自动波特率检测

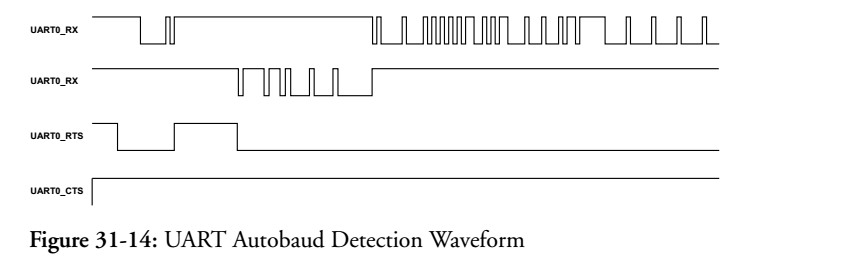
1：host发送，在slave的rxd上有an '@' character (0x40, eight bits data, one start bit, one stop bit, no parity bit)

2：清除EDBO和UART\_CLK寄存器。怎么清除？CM408自己清除

3：slave回复四个字节4 bytes: 0xBF, UART\_CLK [15:8], UART\_CLK [7:0], 0x00 可以不考虑这个返回数据是否正确

4：直到接到slave回复的四个字节之前，host不能发任何数据。

5：当00收到后就可以发送整个数据了。还是解析的hex的数据吗？数据格式是什么？等待回复



这个图又是什么鬼？两个RX。怎么看？

## 2.3：run-time API

客户定制功能

adi\_rom\_Boot() dBootCommand parameter.在哪里调用？什么时候调用？

## 2.4：Boot ROM Programming Model

包含booting 函数，API ，数据结构。

在运行时或者是initcode或者在callback中，bootcode 暴露了几个函数可以调用。

# 3：boot

## 3.1：boot简介

当处理器上电或者是进入硬复位的时候，会进入的一系列过程称为boot

当reset的时候，处理器开始从内部ROM取指令。Bootcode就包含在ROM中用来加载应用程序，bootcode可以自动的初始化某些外设根据bootmode，然后加载应用程序。开发工具会打包应用到bootstream中，这样bootcode就可以将这个代码数据加载到对应的位置。

Bootstream是被bootloader tool分割成block的应用或者数据。每一个block都有一个16byte的header用来告诉bootcode怎么处理这一部分。处理器基于header中的flag执行某些boot功能。

除了支持加载bootstream从源到内部的memory执行，SPImasterboot还支持通过SPI2在spi flash上执行。

内部flash的4kb sector0保留用于安全boot的配置信息。Boot过程默认是从sector1开始的。

API包含很多功能。例如拷贝内存，比较内存等，除了API，bootcode还提供能力定制bootmode。这可以用于列表中没有的bootmode，对于不支持的boot外设，它允许二阶bootloader来从该外设启动。

假如使用二阶boot的方式，将能够用网络烧写的函数放在二阶boot的代码中，上电先执行二阶boot的代码， 但是需要等待上位机命令或者延迟，如果延迟到没有要烧写的命令来，那么就执行应用代码，否则就进行烧写。这种方式的好处是应用程序的烧写不会改变二阶boot的功能，所以可能烧写失败了以后还能够恢复。只要二阶bootcode没问题。但是就是需要按个按钮触发或者上位机发个命令触发。不是很使用，倒是比较适合开发板。

其实也可以将延迟时间设短，比如2s，上电后按个按钮，再进行烧写。因为烧写也不是经常要进行的。所以要烧写的时候这样做也不是不可以。

那问题就是二阶boot用什么烧？还是要用uartboot，uartboot会将程序烧到内部flash中吗？还是只运行在Ram中，掉电就没有了。按我的理解它掉电就没有了。因此要是真的要用二阶boot还是要用uartboot起来，再将二阶boot烧到flash，然后再将应用程序烧到flash。所以还是要用uartboot，并且这种方式对于工厂来说很方便。

## 3.2：boot loader stream

### 3.2.1：简介

Loader stream是一个格式化的block的集合，其中包含对bootkernel的指令，要烧写的应用程序和数据等。

block的头会告诉boot rom这段数据是干什么的，一般来说有两个用途

1：初始化外设

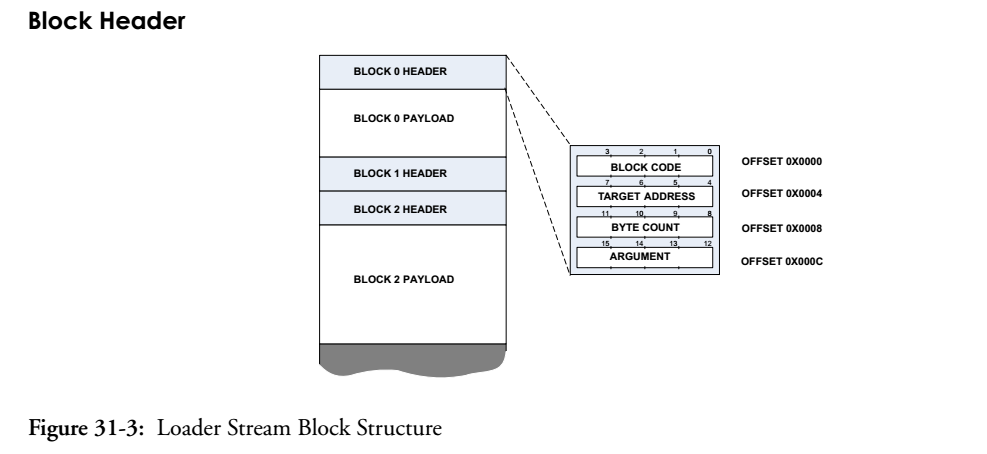
2：将数据放到具体的位置。

Bootstream起作用通过两种方式

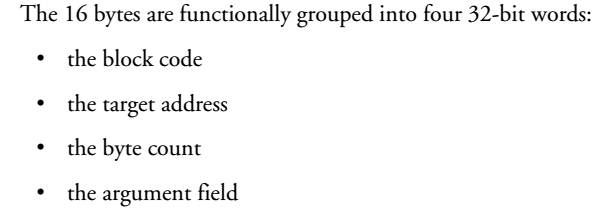
1：编程或者烧写到flash

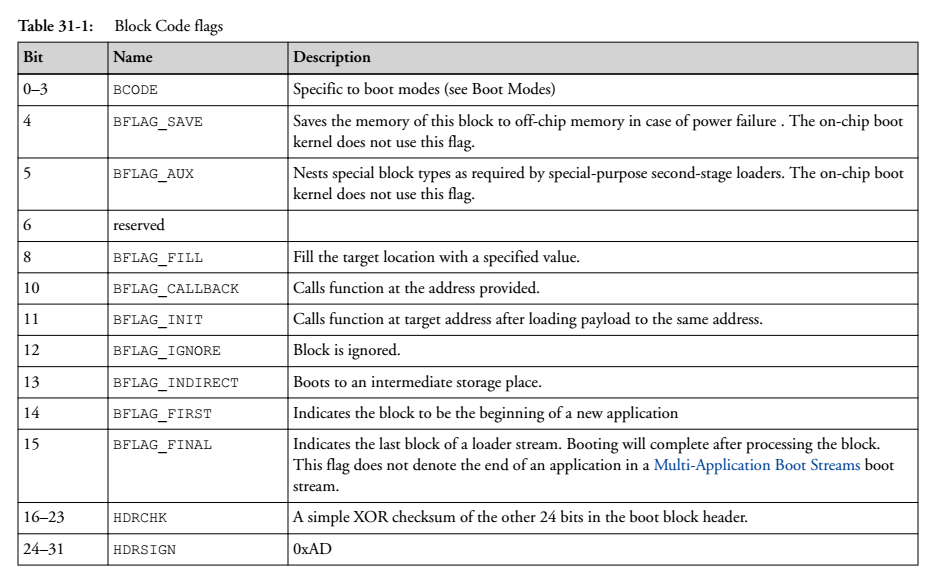
2：在boot的时候通过外设发送给处理器。

### 3.2.2：header



分为四个部分





其中目标地址必须被4整除，目标地址必须指向有效的地址。

根据block code的不同，数据会有不同意义。

### 3.2.3：block type

1：normal block主要功能是加载数据到具体的地址。

2：first block 指示stream的开始，他使用address来决定应用的入口地址

3：final block 标示stream中的最后一个block，处理完之后bootrom跳转到应用程序开始的地方，通过使用 initcode或者是API，boot kernel可以直接从boot routine返回而不是进入应用程序。在跳转到应用程序之前，它需要做一些家务，例如复位一些寄存器等。

4：indirect block 被拷贝到目标地址之前先load到一个地址。适用于加密代码的场合，先放到一个地方加密处理后才写到目标地址。

5：ignore block：例如包含固件版本等，这样的block不需要处理

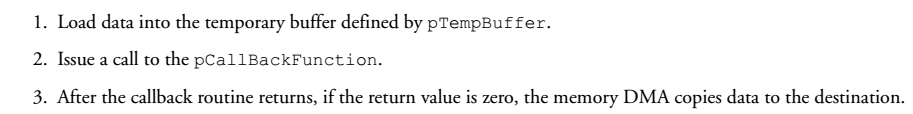
When BFLAG\_IGNORE is set along with BFLAG\_FIRST, only the payload associated with the first block is ignored. The application entry point retrieved from the first block is always processed.

6：fill block: 当应用中有大的数组初始化等的时候用于减小bootstream的size.

7: init block 指示bootkernel在整个block被load之后去做一些功能调用，假如这个block在之前load，那么这个block被认为是零大小并且没有效有数据。Initcode主要用来设置pll，外部memory等加速boot时间。一些情况下在last block中设置flag，用来指示bootkernel去调用这个函数。

8：callback block 目的是对block中的数据进行一些处理，通过initcode注册，先于load block。一般来说callback包含加解密等。一般和init block一同使用。

9：callback和indirect block一同使用



10：quit boot block

从睡眠中唤醒的时候可能不需要加载一些地方。不懂？

11： save block：当应用load以后，这个标志标示应用想把一些数据存储到flash或者从flash把数据导出。

## 3.3:crc保护

需要kernel initcode

## 3.4：sram需求

Boot过程中需要L1 SRAM的资源作为堆栈使用，所有boot过程中用到的资源都在stack上创建。离开reset以后堆栈指针是0x2005FFFC，这个指针被boot rom使用。当应用程序load以后堆栈指针就复位到之前boot时使用的空间。

It is recommended that users reserve 0xC00 bytes at the top of the L2 data SRAM space for the stack use and thus not boot content to this region of memory when booting a boot stream

## 3.5: preboot

当硬件reset或者是软件reset，（怎么软件reset，软件reset可以重新调用bootrom去boot吗？）preboot默认都会被执行。负责检测处理器状态和采取适当的措施。包括检测boot引脚。

TAPC\_RCMSG register 影响preboot的处理

在boot起来之前，PLL都是被旁路的。

Please note that upon power on reset, hard reset and software triggered system reset events the processor is running by default in PLL Bypass mode. The entire boot sequence runs with the PLL in this configuration until user software reconfigures the clocking requirements.

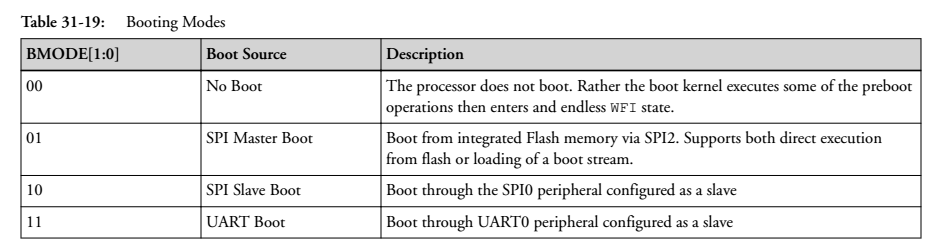
L1 code Cache默认被打开，加速从flash读取数据

The default configuration enabled during preboot sets the M4P0\_CACHE\_CFG register to 0x0000FC10.

## 3.6：boot mode

Slave方式下host控制复位信号

In *slave* boot modes, the processor functions as a slave to any host device. In these modes, the host device usually controls the processor SYS\_HWRST input.



# 4：uart流控

在RS232中本来CTS与RTS有明确的意义，但自从贺氏(HAYES)推出了聪明猫(SmartModem)后就有点混淆了。在RS232中RTS与CTS是用来半双工模式下的方向切换；HAYES Modem中的RTS，CTS是用来进行硬件流控的。通常UART的RTC、CTS的含义指后者，即用来做硬流控的。

硬流控的RTS、CTS：RTS（Require To Send，发送请求）为输出信号，用于指示本设备准备好可接收；CTS（Clear To Send，发送清除）为输入信号，有效时停止发送。假定A、B两设备通信，A设备的RTS连接B设备的CTS；A设备的CTS连接B设备的RTS。前一路信号控制B设备的发送，后一路信号控制A设备的发送。对B设备的发送（A设备接收）来说，如果A设备接收缓冲快满的时发出RTS信号（意思通知B设备停止发送），B设备通过CTS检测到该信号，停止发送；一段时间后A设备接收缓冲有了空余，发出RTS信号，指示B设备开始发送数据。A设备发（B设备接收）类似。上述功能也能在数据流中插入Xoff（特殊字符）和Xon（另一个特殊字符）信号来实现。A设备一旦接收到B设备发送过来的Xoff，立刻停止发送；反之，如接收到B设备发送过来的Xon，则恢复发送数据给B设备。同理，B设备也类似，从而实现收发双方的速度匹配。

半双工的方向切换：RS232中使用DTR（Date Terminal Ready，数据终端准备）与DSR（Data Set Ready ，数据设备准备好）进行主流控，类似上述的RTS与CTS。对半双工的通信的DTE（Date Terminal Equipment，数据终端设备）与DCE（Data circuit Equipment ）来说，默认的方向是DTE接收，DCE发送。如果DTE要发送数据，必须发出RTS信号，请求发送数据。DCE收到后如果空闲则发出CTS回应RTS信号，表示响应请求，这样通信方向就变为DTE->TCE，同时RTS与CTS信号必须一直保持。从这里可以看出，CTS，TRS虽然也有点流控的意思（如CTS没有发出，DTE也不能发送数据），但主要是用来进行方向切换的。

如果UART只有RX、TX两个信号，要流控的话只能是软流控；如果有RX，TX，CTS，RTS四个信号，则多半是支持硬流控的UART；如果有RX，TX，CTS，RTS，DTR，DSR六个信号的话，RS232标准的可能性比较大。

顺便提一下：

DCD（ Data Carrier Detect， 数据载波检测）：DCE向DTE指示，线路上检测到载波。

RI（Ring Indicator，振铃指示）：DCE向DTE指示，有呼叫接入。

**RTS （Require ToSend，发送请求）**为输出信号，用于指示本设备准备好可接收数据，低电平有效，低电平说明本设备可以接收数据。

**CTS （Clear ToSend，发送允许）**为输入信号，用于判断是否可以向对方发送数据，低电平有效，低电平说明本设备可以向对方发送数据。

此处有人将CTS翻译为发送允许，我感觉的确比翻译为清除发送好。因为CTS是对方的RTS控制己方的CTS是否允许发送的功能。

UART CTS/RTS用于硬流控  
假设两颗芯片A和B，其线路连接如下   
 A B  
 TX --> RX  
 RX <-- TX  
 RTS ---> CTS  
 CTS <--- RTS  
当A要向B传输数据时，通过其的TX向B的RX进行发送，B将收入的数据先导入缓存，等累计到一定数据LEN\_MAX后进行处理。假设此时B由于出现临时事件需要去处理，来不及处理这LEN\_MAX个数据时，便希望A暂停发送等其处理完临时事件，再回来处理掉这LEN\_MAX个数据再清空缓存，此时B便需将其RTS拉高，待A在其RTS上检测到[高电平](https://www.baidu.com/s?wd=%E9%AB%98%E7%94%B5%E5%B9%B3&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHcvrjTdrH00T1d9PW7BnAcvPj-buWuhm1640ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EP1DzrHm3rjnkrjTLPjTYnjfd)后，A便会暂停发送数据; B恢复正常后便将RTS重新拉低，待A在其RTS检测到后，便又继续开始的发送数据。