第三章 深入浅出话 Bootloader

当我面对一个有一定规模、稍显复杂的嵌入式项目时，我通常并不会直接专注于主要功能的实现，而是会作一些磨刀不误砍柴工的工作--设计一个 Bootloader(以下简称 BL)以及构建一个 Shell 框架。可能有人会觉得它们很高深，实则不难，正所谓“会者不难，难者不会”。本章就针对 BL 进行详细的讲解，希望让大家可以体会到它的重要性。

1、烧录方式的更新迭代

1.1 古老的烧录方式

单片机诞生于 20 世纪 80 年代，以 51 为代表开始广泛应用于工业控制、家电等很多行业中。起初对于单片机的烧录，即将可执行的程序写入到其内部的 ROM 中，不是一件容易的事情，而且成本不低，因为需要依赖于专门的烧录设备。而且受到半导体技术与工艺的限制，对于 ROM 的烧写大多需要高压。这种境况一直持续到 2000 年左右(我上大学的时候还曾用过这种专门的烧录器)，图 3.1 所示。

图 3.1 单片机烧录器

1.2 ISP 与 ICP 烧录方式

随着低压电可擦写 ROM 的成熟，单片机开始集成可通过数字电平直接读写的存储介质。其最大的优势在于可实现在系统或在电路直接烧录程序，而无需像以前一样把单片机芯片从电路中拿出来，放到编程器上，这种烧录方式就是 ISP(In System Programming)或 ICP(In Circuit Programming)，如图 3.2 所示。

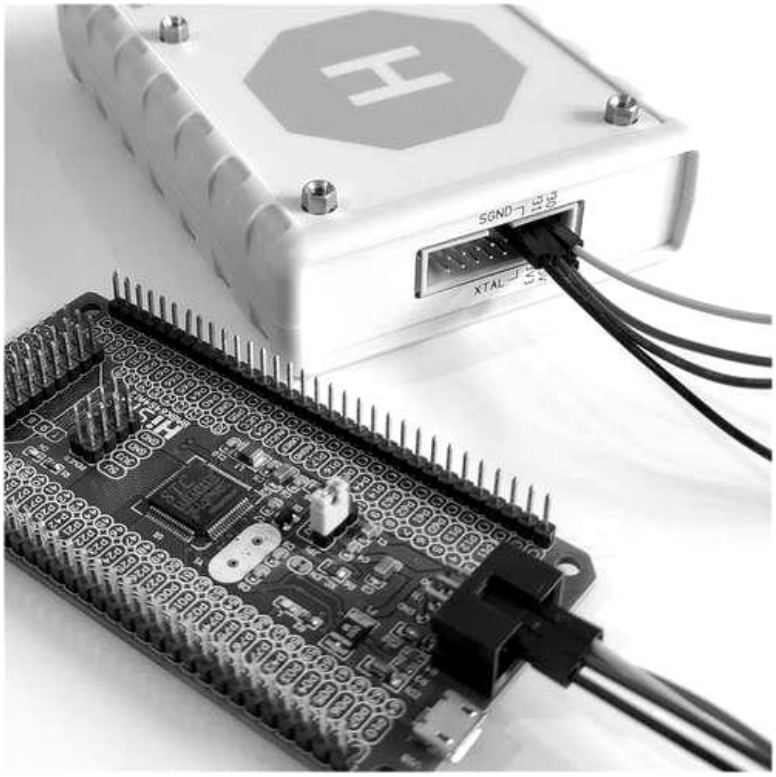

图 3.2 单片机的 ISP 烧录

有人问过这样一个问题：“ISP 和 ICP 我都听说过，都说是可以在电路板上直接烧录程序，而无需拿下芯片，那 ISP 和 ICP 有什么区别？”从广义上来说，两者没有区别，平时我们把其意义混淆也毫无问题。非要刨根问底的话，那可以这样来理解：ISP 要求单片机中驻留有专门的程序，用以与上位机进行通信，接收固件数据并烧录到自身的 ROM 中，很显然 ISP 的单片机是需要可运行的，即要具备基本的最小系统电路(时钟和复位)；而 ICP 可以理解为 MCU 就是一块可供外部读写的存储电路，它不需要预置任何程序，也不需要单片机芯片处于可运行的状态。

支持 ISP 或 ICP 的芯片，以 AT89S51 最为经典，当时从 AT89C51 换成 S51，多少人因不再依赖烧录器而大呼爽哉。这种并口下载线非常流行，如图 3.3，网上还有各种 ISP 小软件，可以说它降低了很多人入门单片机的门槛，让单片机变得喜闻乐见。一台电脑、一个 S51 最小系统板、一条并口 ISP 下载线，齐了！

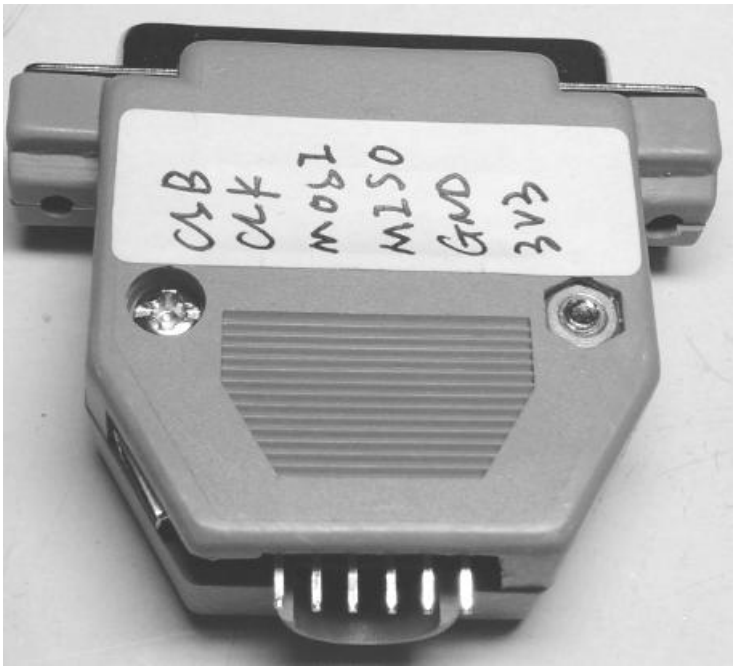

图 3.3 用于 AT89S51 的并口 ISP 下载线

1.3 更方便的 ISP 烧录方式

1.3.1 串口 ISP

但是后来我们发现带有并口的电脑越来越少。那是在 2005 年前后，STC 单片机开始大量出现，在功能上其实与 S51 相差无几，甚至比同期的一些高端 51 单片机还要逊色。但是它凭借一个优势让人们对它爱不释手，进一步降低了单片机的学习门槛。这个优势就是--串口 ISP，这是真正意义上的 ISP，如图 3.4 与图 3.5 所示。

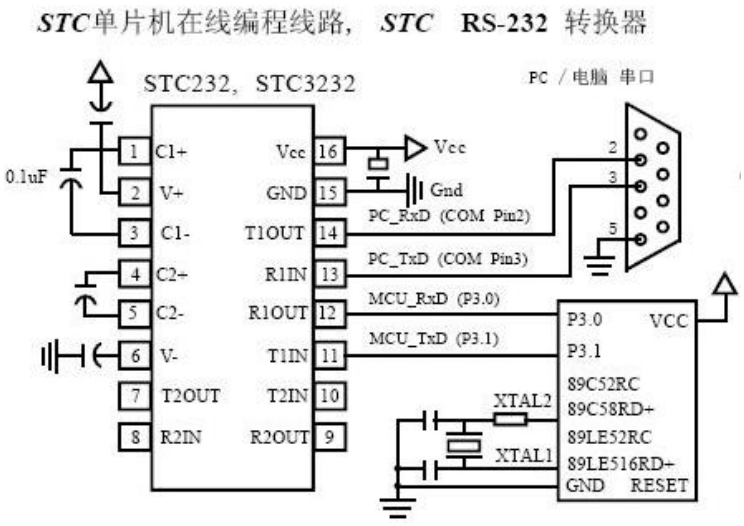

图 3.4 STC 单片机的串口 ISP 线路示意

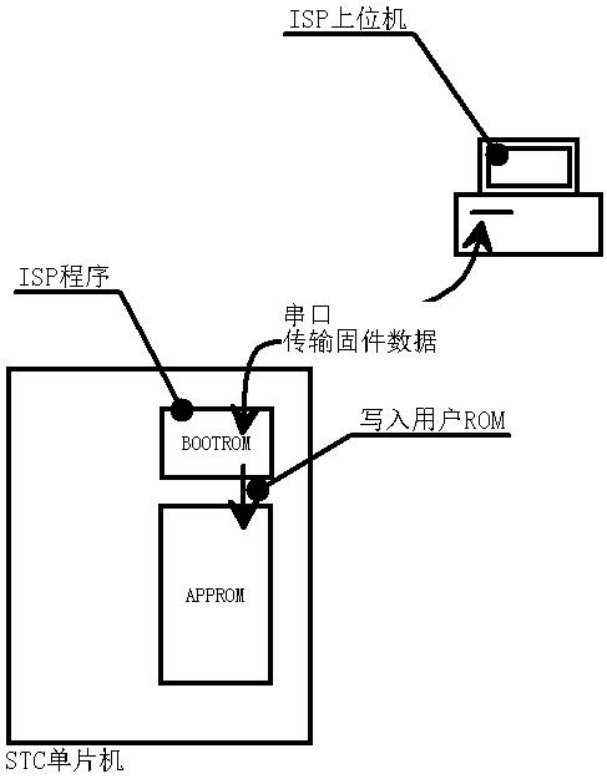

图 3.4 STC 单片机的串口 ISP 原理示意

再后来，9 针串口都很少见了，只有 USB。促使一个烧录和调试神器炙手可热--USB TTL 串口。这下 232 转换芯片省掉了，直接通过 USB 进行烧录。这种方式造福了无数的单片机学习者和工程师。我本人虽然已经搞了近 20 年单片机和嵌入式，USB 串口依然是不可或缺的调试工具。

多年来，在串口与单片机的交互上，我动了很多脑筋，这也是我乐于开发 Bootloader 的一个原因。我希望“USB 串口在手，一切全有！”

STC 并不是第一个使用串口 ISP 烧录程序的，但它是最成功和最深入人心的。与之同期的很多单片机，包括时至今日仍然应用最广泛的 STM32 全系列也都支持了串口 ISP，它成为了一种标配的、非常普遍的程序烧录手段。

1.3.2 各种 USB ISP

串口 ISP 固然方便，但是下载速度是它的硬伤，当固件体积比较大的时候，比如一些大型嵌入式项目的固件动辄几百 K，甚至几 M，再用串口 ISP 就未免太慢了。所以一些单片机配有专门的 USB ISP 下载器。以下列举几种比较主流单片机及其 USB ISP 下载器。

1) AVR

AVR 单片机曾经盛极一时，但经历了 2016 年的缺芯风波之后，加之 STM32 的冲击，开始变得一蹶不振，鲜有人用了。与之配套的 USB ISP 下载器非常多样，有些是官方发布的，更多的是爱好者开源项目的成果，如图 3.5 所示。

图 3.5 AVR 多样的 USB ISP 下载工具(AVRISP MKII 与 USBASP)

2) C8051F

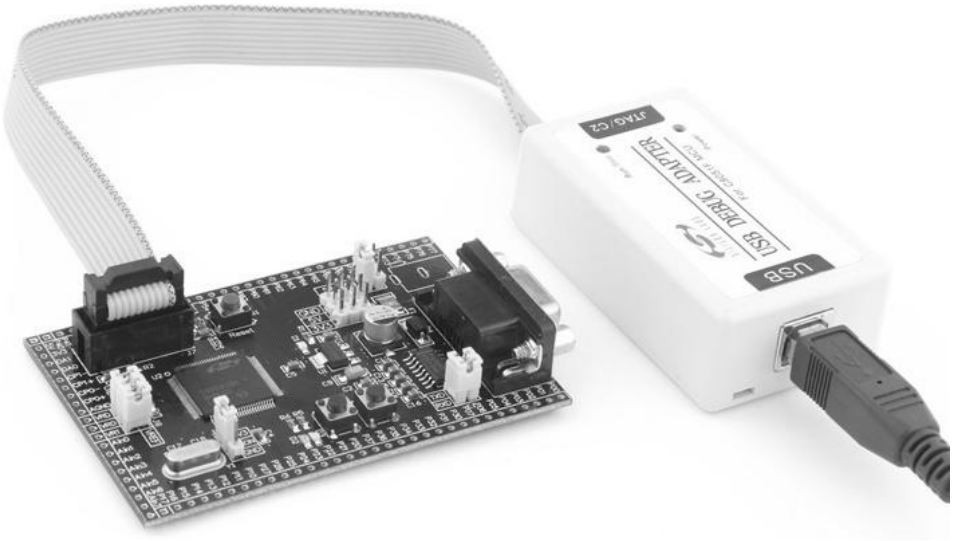

图 3.6 C8051F 的 USB ISP 下载器(EC6)

3) MSP430

图 3.7 MSP430 的 USB ISP 下载器

我们会发现，一个具体良好生态的主流单片机，一定有配套的高效便捷的烧录下载工具。可见一种好的烧录方式，对单片机开发是多么重要。

不论是串口 ISP 还是各种专用的 ISP 下载器，都有一些共同的弊端。

1、依赖于专门的上位机或下载器硬件，不能作到统型

2、下载器价格仍然比较高，尤其是原厂的，这也是为什么有些单片机催生出很多第三方的下载器，比如 AVR

3、下载的时候通常需要附加额外的操作，比如 STC 要重新上电、STM32 需要设置 BOOT 引脚电平等。这些额外的操作都增加了烧录的复杂性。尤其是在产品形态下要去重新烧录程序，比如嵌入式升级，就要打开外壳，或将附加信号引出到壳外。这都是非常不高效，不友好的作法。

如果有一种烧录方法,对于任何一种单片机 1、通信方式统一(比如一律都用串口) 2、提供一个友好的操作界面(比如命令行方式) 3、高效快速，没有附加操作，最好一键自动化烧录 4、另外再增加一些嵌入式固件管理的功能(比如固件版本管理)，那这一定会让我们事半功倍。

Bootloader 就能实现上述的这一切！

2、关于 Bootloader

2.1 Bootloader 的基本形态

直接看图 3.8 所示。

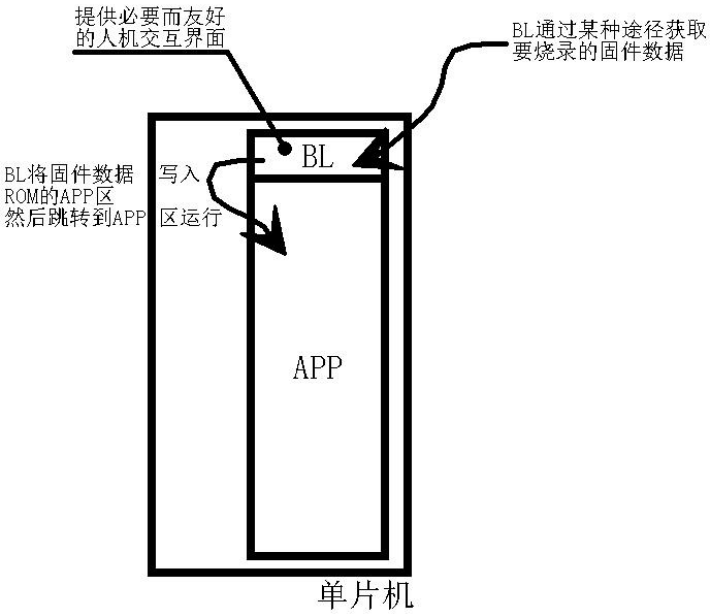

图 3.8 Bootloader 的基本形态

可以看到 BL 就是一段存储在 ROM 中的程序，它主要实现 4 个功能。

1、通过某种途径获取要烧录的固件数据

2、将固件数据写入到 ROM 的 APP 区中

3、跳转到 APP 区运行，将烧录进去的用户程序引导起来

4、在此过程中，提供必要而友好的人机交互界面

这么说可能不好理解，我们还是通过实例来进行讲解。

2.2 Bootloader 的两个设计实例

下面的两个实例，用于说明 BL 的实际应用形态，不涉及具体的实现细节，旨在让大家了解 BL 的实际是如何运行。

1)带 Shell 命令行的串口 BL

基本的操作逻辑如下：

1、通过超级终端、SecureCRT 或 Xshell 之类的串口终端输入命令 program

2、BL 接收到命令后，开始等待接收固件文件数据

3、串口终端通过某种文件数据传输协议(大家可以参见本书相应章节对 X/Y/Zmodem 协议的介绍)将固件数据传给 BL

4、BL 将固件数据依次写入到 ROM 的 APP 区中

5、BL 将 APP 区中的程序运行起来

更具体的示意如图 3.9 所示。

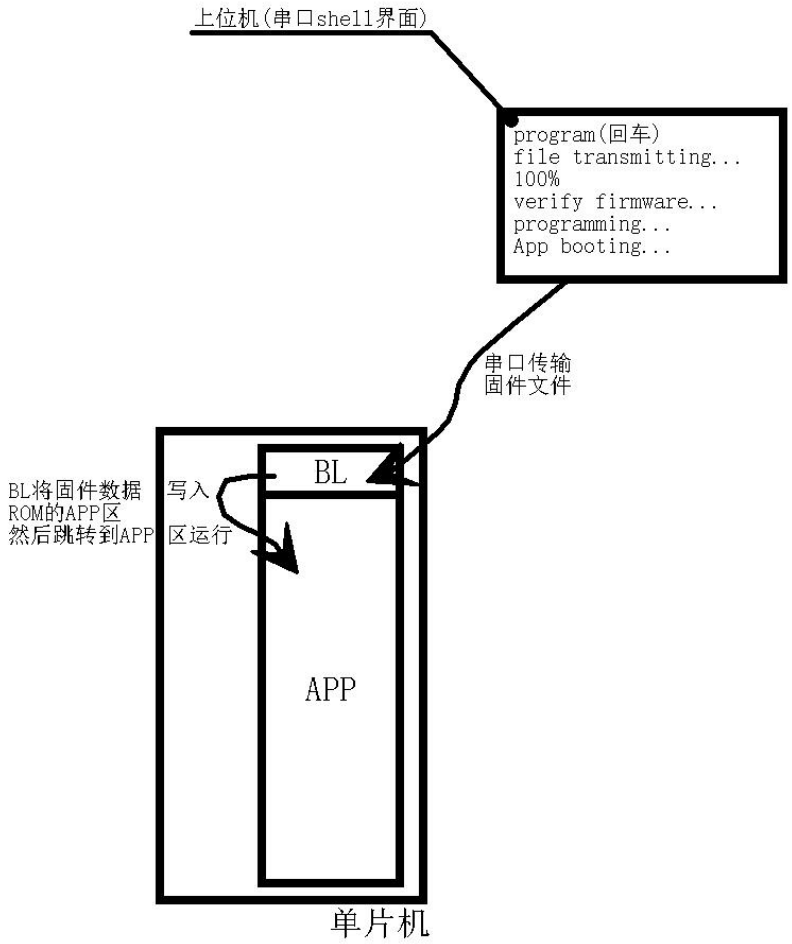

图 3.9 带 Shell 命令行的串口 BL 逻辑示意

这里把操作逻辑说得很简单，实际实现起来却并不简单，我们放在后面去细究其具体实现。

2) 插 SD 卡即烧录的 BL

基本的操作逻辑如下：

1、将待烧录的固件拷贝到 SD 卡中

2、将 SD 卡插入到卡槽中

3、BL 检测到 SD 卡插入，搜索卡中 BIN 文件

4、将 BIN 文件数据读出写入到 ROM 的 APP 区中

5、BL 将 APP 区中的程序运行起来

如图 3.10 所示。

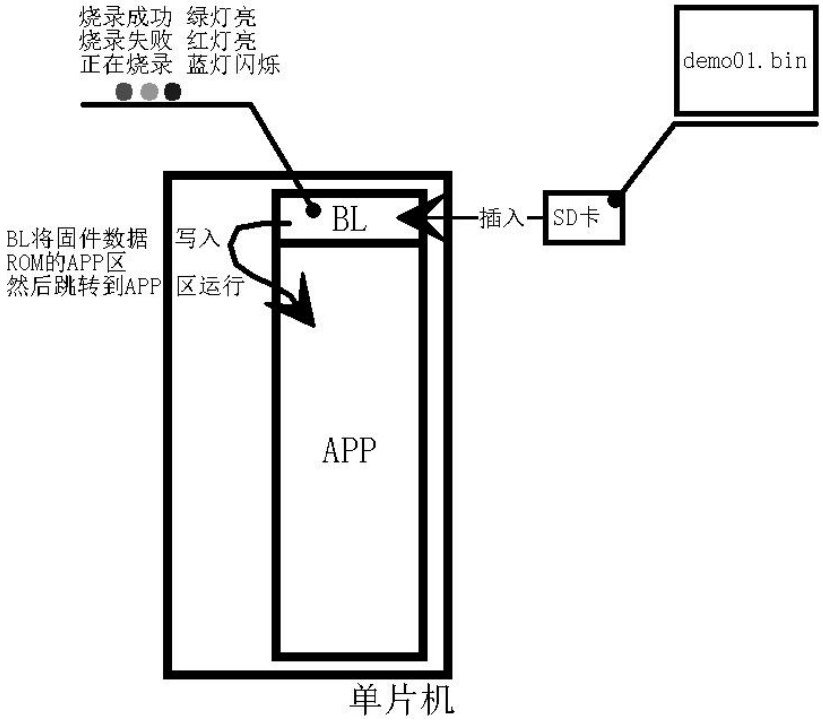

图 3.10 插 SD 卡即烧录的 BL 逻辑示意

通过这两个设计实例，大家已经了解 BL 是什么了吧。是不是感受到 BL 是比 ISP 烧录器更通用、更灵活、更友好、功能更强大的固件烧录和管理手段了。有人可能知道 Linux 下的 Uboot，它就是一个强大的 BL，它提供非常强大的刷机(烧录操作系统镜像)的功能和完备而灵活的 Shell 界面，如图 3.11 所示。其实我们电脑的 BIOS 也是一种广义的 BL。

```
#####    Boot for Nor Flash Main Menu   #####
#####     EmbedSky TFTP download mode     #####

[1] Download u-boot.bin to Nand Flash
[2] Download Eboot (eboot.nb0) to Nand Flash
[3] Download Linux Kernel (zImage.bin) to Nand Flash
[4] Download stepldr.nb1 to Nand Flash
[5] Set TFTP parameters(PC IP,TQ2440 IP,Mask IP...)
[6] Download YAFFS image (root.bin) to Nand Flash
[7] Download Program (uCOS-II or TQ2440_Test) to SDRAM and Run it
[8] Boot the system
[9] Format the Nand Flash
[0] Set the boot parameters
[a] Download User Program (eg: uCOS-II or TQ2440_Test)
[b] Download LOGO Picture (.bin) to Nand  Flash
[l] Set LCD Parameters
[o] Download u-boot to Nor Flash
[p] Test network (TQ2440 Ping PC's IP)
[r] Reboot u-boot
[t] Test Linux Image (zImage)
[q] Return main Menu
Enter your selection:
```

图 3.11 Uboot Shell 界面

那如何实现一个 BL 呢？别急，要实现 BL 是需要满足一些基本要求的。

2.3 BL 实现的要点

首先要说，并不是任何一个单片机都可以实现 BL 的，要满足几个要点。

1)体系架构支持

来看图 3.12。

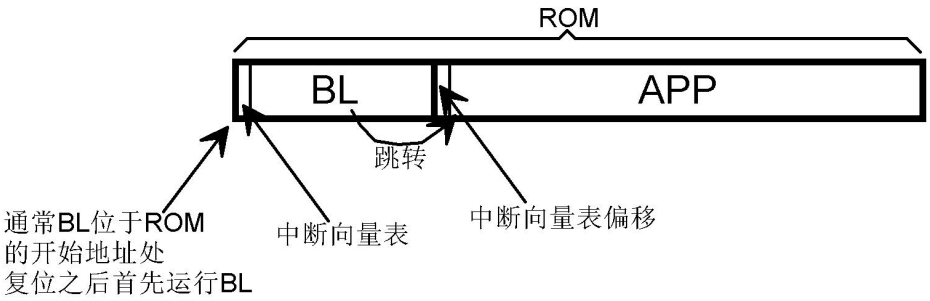

图 3.12 BL 的实现需要单片机支持中断向量表偏移(重定向)

我们知道单片机程序的最开头是中断向量表，包含了程序栈顶地址以及 Reset 程序入口，通过它才能把程序运行起来。很显然在从 BL 向 APP 跳转的时候，APP 程序必须有自己的中断向量表。而且单片机体系架构上要允许中断向量表的重定向。

传统 51 单片机的中断向量表只允许放到 ROM 开头，而不能有偏移量，所以传统 51 单片机是不能支持 BL 的。有人要问“你这不是自相矛盾吗？你前面说 STC 的 51 单片机是支持串口 ISP 的，那它应该内置有 ISP 程序，我理解它应该和 BL 是一个道理。”没错，它内置的 ISP 程序就是一种 BL。STC 之所以可以实现 BL 功能，是因为宏晶半导体公司对它的硬件架构进行了改进，请看图 3.13。

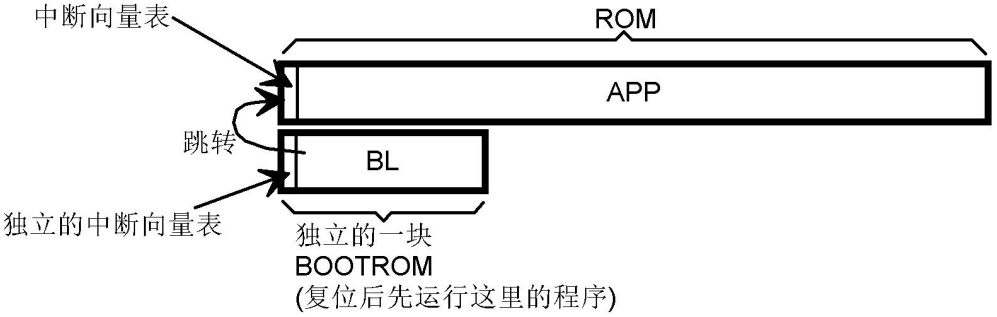

图 3.12 STC 对传统 51 单片机硬件架构上的改进

可以看到，STC51 单片机多出一块专门存放 BL 的 ROM，称为 BOOTROM。

网上有一位叫 shaoziyang 的网友为 AVR 单片机写了一个 BL，还配套开发了一款叫 AVRUBD 的上位机，如图 3.13 (AVRUBD 是很有用的，本章后面会介绍，它可以让我们实现隔空烧录)，实现了 AVR 单片机的串口烧录，让很多人摆脱了对 USBISP 之类 ISP 下载器的依赖(虽然 ISP 下载器已经很方便了，但它毕竟还需要银子嘛)。

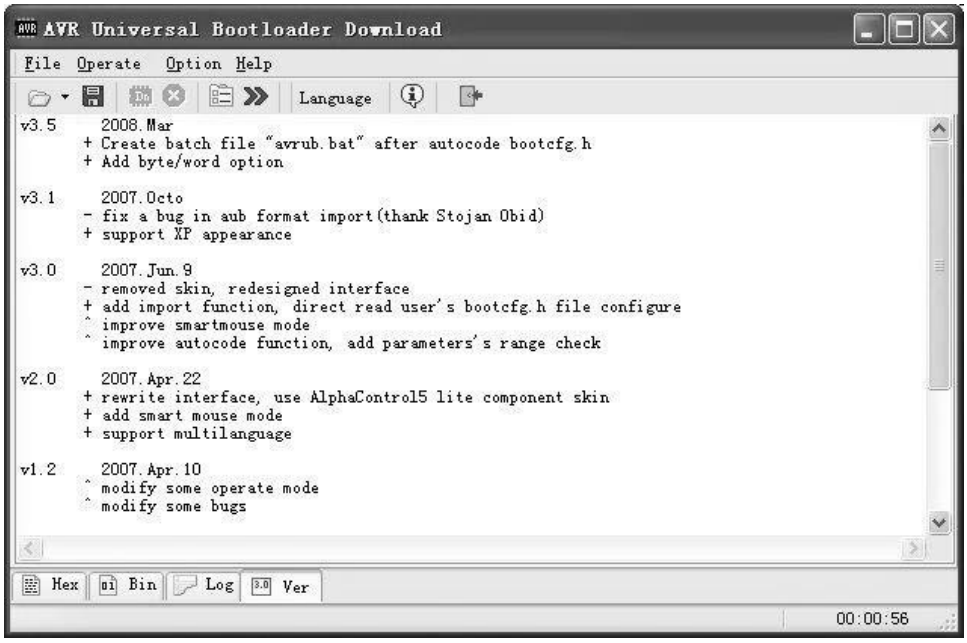

图 3.13 shaoziyang 的 AVR BL 配套上位机软件 AVRUBD

AVR 在硬件架构上与 STC51 是一个套路，如图 3.13 所示。

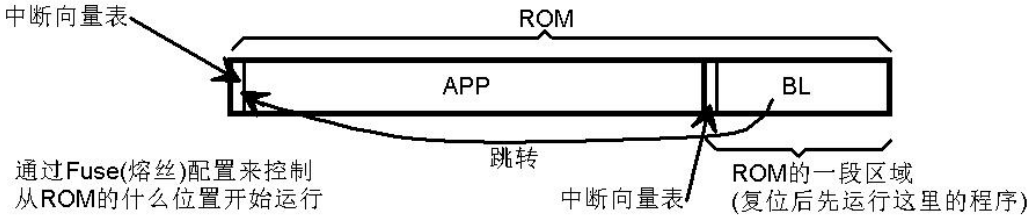

图 3.14 AVR 单片机硬件架构上对 BL 的支持

通过配置 AVR 的熔丝位可以控制复位入口地址以及 BOOT 区的大小和开始地址，如图 3.14 所示。

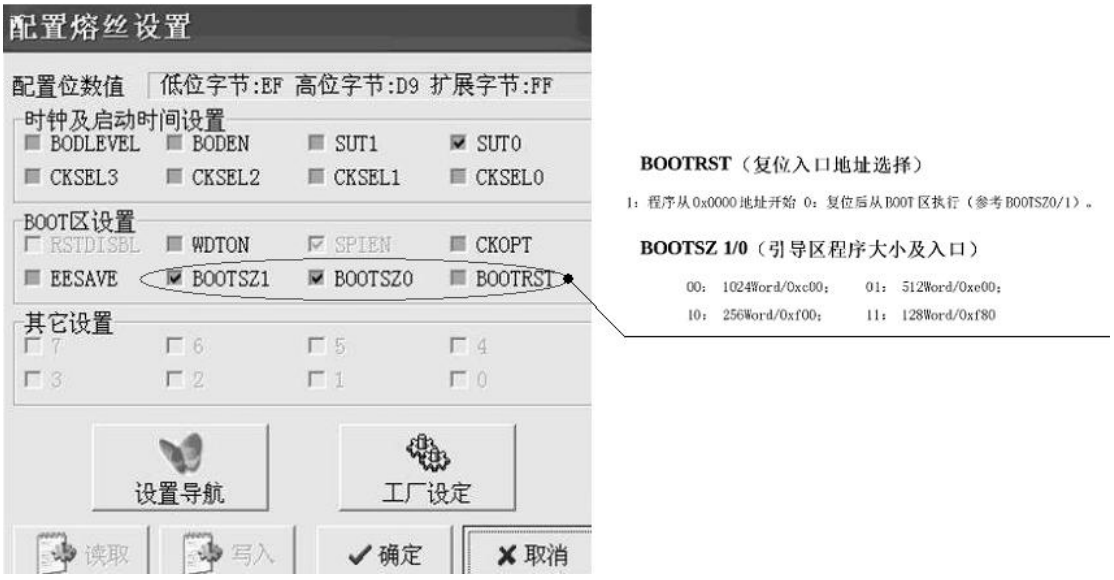

图 3.14 AVR 单片机硬件架构上对 BL 的支持

讲到这里，有人会说：“那有没有一种单片机，程序放在 ROM 的任何位置都可以运行起来，也就是中断向量表可以重定位？”当然有，这种单片机还很多，其中最典型的就是 STM32。它的程序之所以可以放之任地皆可运行，是因为在它的 NVIC 控制器中提供了中断向量表偏移量的相关配置，这个后面我们再详细说。

2）ROM 可 IAP

这也是需要单片机硬件支持的。很好理解，在 BL 获取到固件数据之后，需要将它写入到 ROM 的 APP 区中，所以说单片机需要支持 IAP 操作，所谓 IAP 就是 In Application Programming，即在应用烧录。也就是在程序运行过程中，可以对自身 ROM 进行擦除和编程操作。

大家仔细想想是不是这样？似乎支持串口 ISP 的单片机都支持 IAP 功能。STC 还把这一功能包装成了它的一大特色，可以用内部 ROM 来充当 EEPROM 的功能，可以在运行时记录一些掉电不丢失的参数信息。

STM32 的 ROM 擦写在配套的固件库(标准库或 HAL 库)中已经有实现，大家可以参考或直接使用。

3)APP 程序的配套修改

为了让 BL 可以顺利的将 APP 程序引导运行起来，APP 程序在开发的时候需要配合 BL 作出相应的修改。最重要的就是 APP 程序的开始地址(即中断向量表的开始地址)以及对中断控制器的相应配置。

对于 51、AVR 这类单片机 APP 程序不用修改，具体原因大家应该明白。这里主要对 STM32 APP 程序如何修改进行详细讲解。

我们依然是结合实例，请看图 3.15 所示。

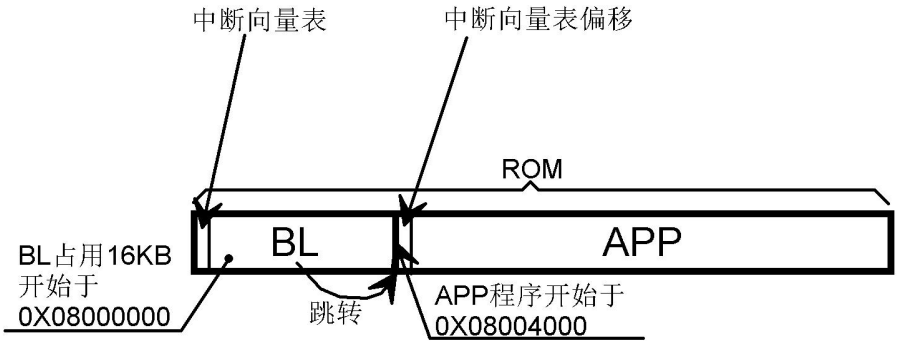

图 3.15 STM32 ROM 划分实例

假设我们所使用的 STM32 的 ROM 总大小为 128KB，BL 程序的体积是 16K，APP 程序紧邻 BL，那么 APP 区的开始地址为 0X08004000，亦即 APP 程序的中断向量表偏移地址为 0X4000。

如果我们使用 MDK 作为开发环境的话，需要修改这里，如图 3.16 所示。

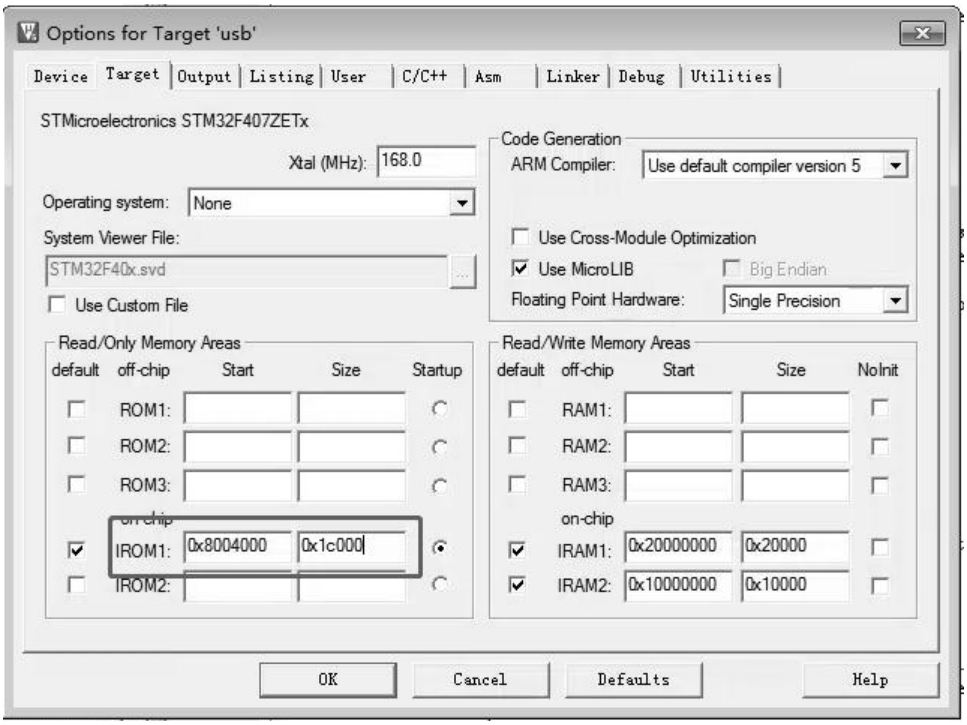

图 3.16 MDK 开发环境中对程序存储器开始地址与大小进行配置

而如果我们使用的是 gcc 的话，则需要对 link.lds 链接文件进行修改。

```
/* Program Entry, set to mark it as "used" and avoid gc */
MEMORY
{
    ROM (rx) : ORIGIN = 0x08004000, LENGTH =  112k /* 112K flash */
    RAM (rw) : ORIGIN = 0x20000000, LENGTH =  20k /* 20K sram */
}
ENTRY(Reset_Handler)
_system_stack_size = 0x400;

SECTIONS
{
    .text :
    {
        . = ALIGN(4);
        _stext = .;
        KEEP(*(.isr_vector))            /* Startup code */
```

图 3.17 gcc 编译环境下对 link.lds 文件的修改

然后我们还需要对 NVIC 的中断向量表相关参数进行配置，主要是中断向量表的偏移量，如下代码。

#define VECT_TAB_OFFSET 0x4000

OK，经过修改后的程序，我们把它放到 ROM 的 0X08004000 开始位置上，然后再让 BL 跳转到这个地址上，我们的程序就能运行起来了。

有人又会问：“BL 中的跳转代码怎么写？”别急，这是我们要讲的下一个要点。

4)BL 中跳转代码

跳转代码是 BL 要点中的关键，直接关系到 APP 程序能否正常运行。

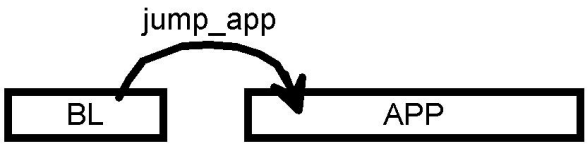

图 3.18 BL 向 APP 的跳转示意

我直接给出 STM32 下 jump_app 函数的代码。

```
typedef void (*iapfun)(void);
iapfun jump2app;

void MSR_MSP(u32 addr)
{
 __ASM volatile("MSR MSP, r0");       //set Main Stack value
 __ASM volatile("BX r14");
}

void load_app(u32 appxaddr)
{
 if(((*(vu32*)appxaddr)&0x2FFE0000)==0x20000000)//检查栈顶地址合法
 {
  //用户代码区第二个字为程序开始地址(复位地址)
  jump2app=(iapfun)*(vu32*)(appxaddr+4);
  //初始化 APP 堆栈指针(用户代码区的第一个字用于存放栈顶地址)
  MSR_MSP(*(vu32*)appxaddr);
  jump2app();     //跳转到 APP.
 }
}
```

这段代码大家自己研究，如果展开讲就属于赘述了。

到这里 BL 相关的要点就介绍完了，大家应该有能力去完成一个简单的 BL 了。我基于 STM32 设计了一个小实验，大家有兴趣可以小试牛刀一下。

图 3.19 BL 功能验证实验

我们将 BL 程序用 Jlink 烧录到 0X08000000 开始位置，而把 APP 程序烧录到 0X08002000 开始位置，然后复位，如果串口打印了 hello world 或流水灯亮起来了，就说明我们的 BL 成功了。

3、花样百出的 BL

上面我所讲的都是 BL 最基础的一些内容，是我们实现 BL 所必须了解的。BL 真正的亮点在于多种多样的固件数据获取方式。

3.1 BL(串口传输固件)的实现与延伸

前面我讲到过两个 BL 应用的实例，一个是串口传输固件文件，一个是 SD 卡拷贝固件文件。它们是在实际工程中经常被用到的两种 BL 形式。这里着重对前一个实例的实现细节进行讲解剖析，因为它非常具有典型意义，如图 3.20。

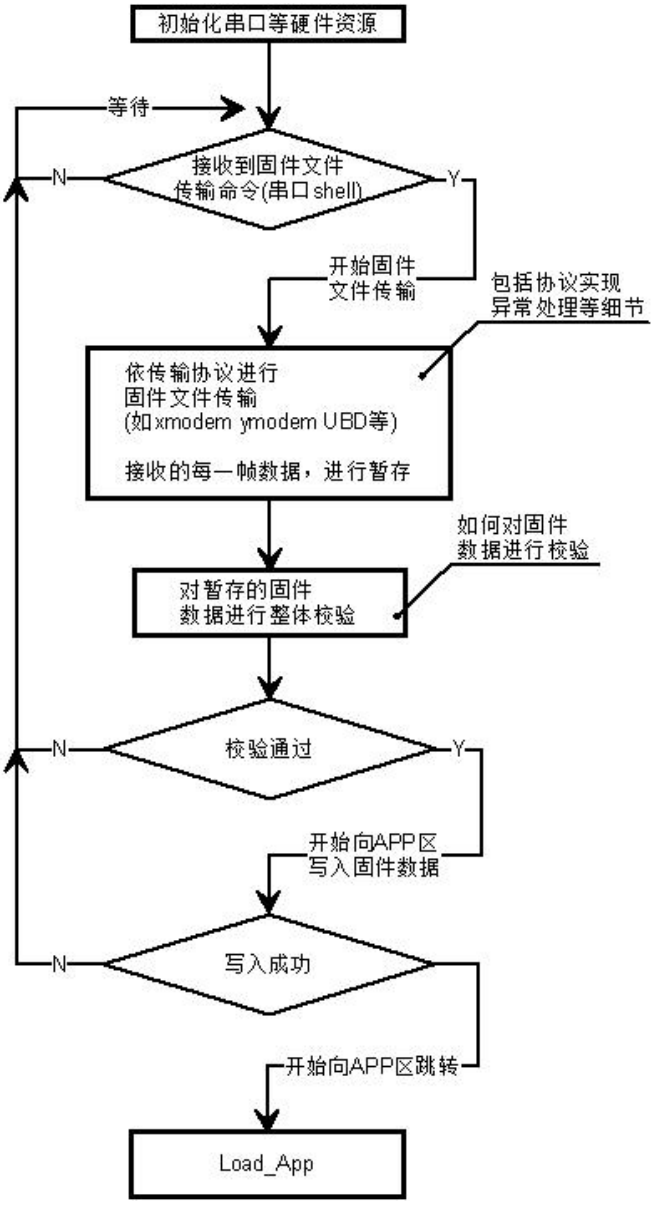

图 3.20 BL(串口传输固件)的实现流程图

这个流程图提出了 3 个问题：

1、串口通信协议是如何实现的？

2、为什么获取到上位机传来的固件数据，不是直接写入到 APP 区，而是先暂存，还要校验？

3、对固件数据是如何实现校验的？

串口通信协议以及文件传输实现的相关内容略显繁杂，在本书相应章节会专门进行讲解。

第二个问题：经过串口传输最终由单片机接收到的固件数据是可能出现差错的，而有错误的固件冒然直接写入到 APP 区，是一定运行不起来的。所以，我们要对数据由帧进行暂存，等全部传输完成后，对其进行整体校验，以保证固件数据的绝对正确。

针对第三个问题，我们要着重探讨一下。

一个文件从发送方传输到接收方，如何确定它是否存在错误？通常的作法在文件中加入校验码，接收方对数据按照相同的校验码计算方法计算得到校验码，将之与文件中的校验码进行对比，一致则说明传输无误。

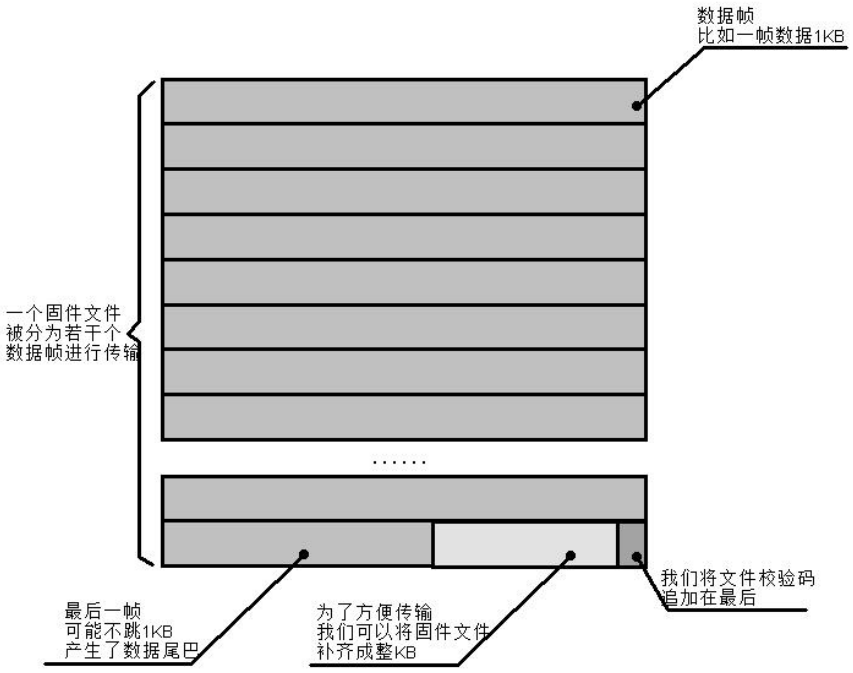

图 3.21 对固件文件进行补齐并追加校验码

图 3.21 是对固件文件的补齐以及追加校验码的示意。为什么要对文件补齐？嵌入式程序经过交叉编译生成的可烧录文件，比如 BIN，多数情况下都不是 128、256、512 或 1024 的整数倍。这就会导致在传输的时候，最后一帧数据的长度不足整帧，即会产生一个数据尾巴。取整补齐是解决数据尾巴最直接的方法。这一操作是在上位机(PC 端)上完成的，通常是编写一个小软件来实现。这个小软件同时会将校验码追加到固件文件末尾。这个校验码可以使用校验和(Checksum)或者 CRC，一般是 16 位或 32 位。

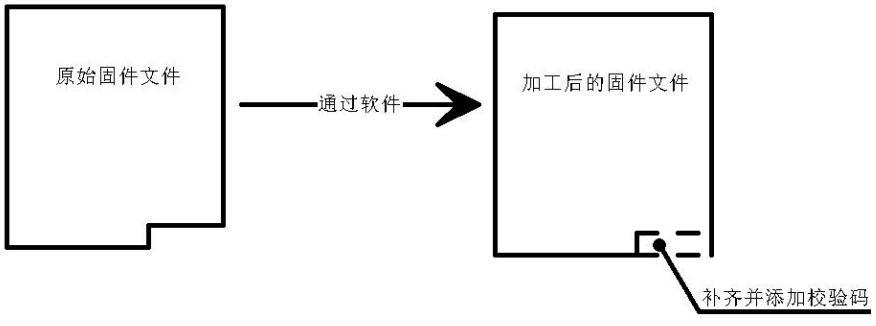

图 3.22 通过一个小软件实现对固件文件补齐和添加校验码

又有人会问：“要把整个固件暂存下来，再作校验，那得需要额外的存储空间吧，外扩 ROM(FlashROM 或 EEPROM)？”是的。如果想节省成本，我们也可以不暂存，传输时直接烧写到 APP 区。这是有风险的，但是一般来说问题不大(STC 和 STM32 的串口 ISP 其实也都是实时烧写，并不暂存)。因为在传输的过程中，传输协议对数据的正常性是有一定保障的，它会对每一帧数据进行校验，失败的话会有重传，连续失败可能会直接终止传输。所以说，一般只要传输能够完成，基本上数据正确性不会有问题。但是仍然建议对固件进行整体校验，在成本允许的情况下适当扩大 ROM 容量。同时，固件暂存还有一个另外的好处，在 APP 区中的固件受到损坏的时候，比如固件意外丢失或 IAP 时不小心擦除了 APP 区，此时我们还可以从暂存固件恢复回来(完备的 BL 会包含固件恢复的功能)。

其实也不必非要外扩 ROM，如果固件体积比较小的话，我们可以把单片机的片上 ROM 砍成两半来用，用后一半来作固件暂存。

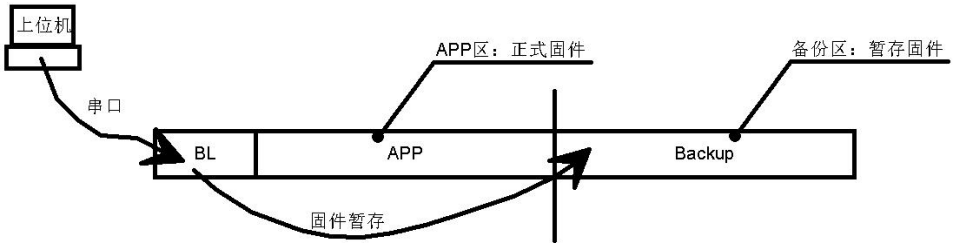

图 3.23 将片上 ROM 划分为 3 部分

如图 3.23，我们将片上 ROM 划分为 3 部分，分别用于存储 BL、APP 固件以及暂存固件。比如我们使用 STM32F103RBT6，它一共有 128KB 的 ROM，可以划分为 16K/56K/56K。

有些产品对成本极为敏感。我就有过这样的开发经历，当时使用的单片机是 STM32F103C8T6，片上 ROM 总容量为 64K，固件大小为 48K，BL 为 12K。在通过 BL 进行固件烧写时根本没有多余的 ROM 进行固件暂存。我使用了一招“狗尾续貂”，如图 3.24。

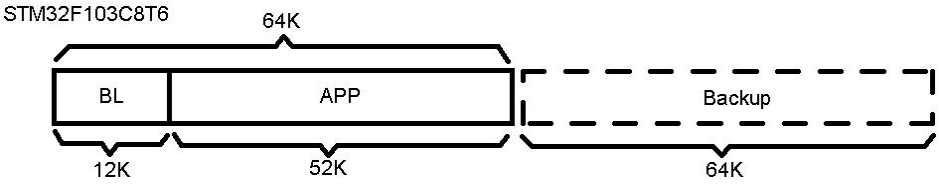

图 3.24 STM32F103C8T6 后 64K 也可用

我无意中了解到 STM32F103C8T6 与 RBT6 的晶元是同一个。只是因为有些芯片后 64KB 的 ROM 性能不佳或有瑕疵，而被限制使用了。我实际测试了一下，确实如此。但是后 64K ROM 的使用是有前提的，即需要事先对其好坏进行验证。如果是好的，则暂存校验，再写入 APP 区；而如果是坏的，那么就直接在固件传输时实时写入 APP 区(这个办法我屡试不爽，还没有发现后 64K 有坏的)。

以上振南所介绍的是一种骚操作，根本上还是有一定的风险的，ST 官方有声明过，对后 64K ROM 的质量不作保证，所以还是要慎用。

3.2 10 米之内隔空烧录的实现

这个“隔空烧录”源于我的一个 IoT 项目。它是对空调的外机进行工况监测，而大家知道，空调外机的安装那可不是一般人能干的，它要不就在楼顶，要不就在悬窗上。这给硬件升级嵌入式程序带来很大的困难。所以，我实现了“隔空烧录”的功能，其实

它就是串口 BL 应用的一个延伸，如图 3.25 所示。

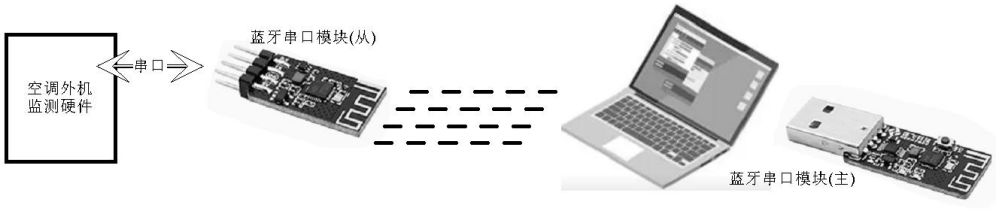

图 3.25 通过蓝牙串口模块实现“隔空烧录”

“隔空烧录确实牛，但是总要抱着一个电脑，这不太方便。”确实是！还记得前面我提过的 AVRUBD 通信协议吗？它的上位机软件是有手机版的。这样我们只要有手机，就能“隔空烧录”了，如图 3.26。

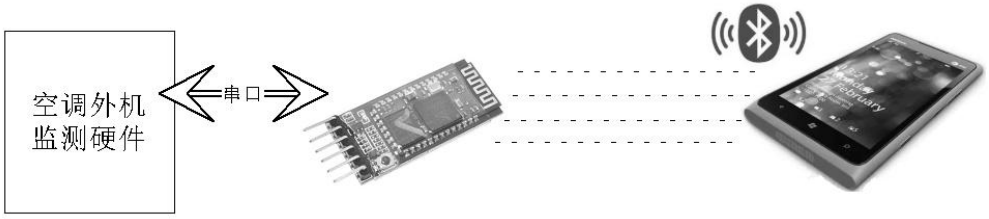

图 3.26 手机连接蓝牙串口模块实现“手机隔空烧录”

“哪个 APP？快告诉我名字”，别急，蓝牙串口助手安卓版，图 3.27 是正在传输固件的界面。

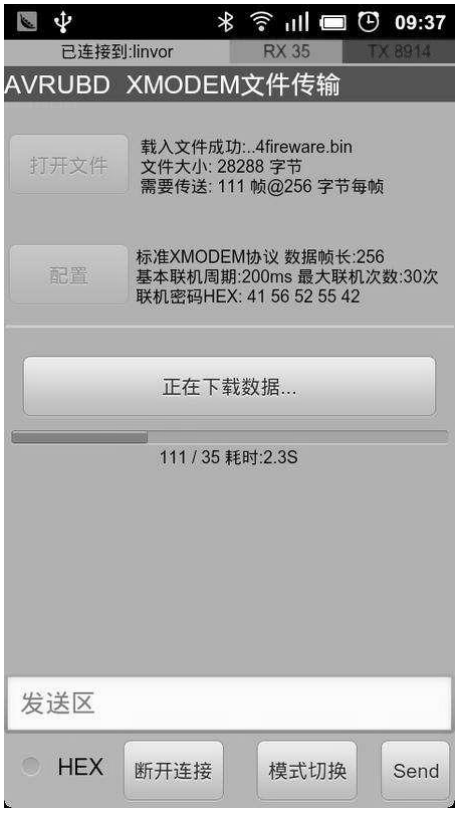

图 3.27 蓝牙串口助手传输固件文件的界面

AVRUBD 其实是对 Xmodem 协议的改进，这个我们放在专门的章节进行详细讲解。

3.2 BL 的分散烧录

我们知道了 BL 的核心功能其实就是程序烧录。那你有没有遇到过比较复杂的情况，如图 3.28 所示。

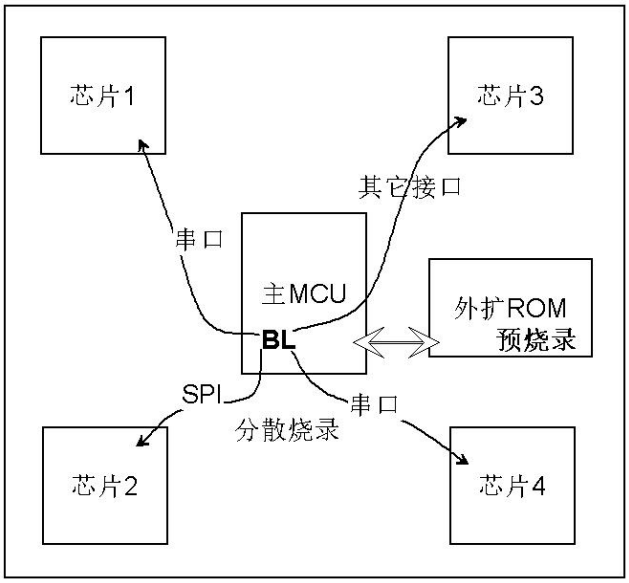

图 3.28 一个系统(产品)中有多个部件需要烧录

这种情况是有可能遇到的。主 MCU+CPLD+通信协处理+采集协处理就是典型的复杂系统架构。这种产品在批量生产阶段，烧录程序是非常繁琐的。首先需要维护多个固件，再就是需要一个个给每一个部件进行烧写，烧写方式可能还不尽相同。所以我引入了一个机制，叫“BL 的分散烧录”。

首先我们将所有的固件拼装成一个大固件(依次数据串接)，并将这个大固件预先批量烧录到外扩 ROM 中，比如 spiROM；再将主 MCU 预先烧录好 BL；然后进行 SMT 焊接。PCBA 生产出来之后，只要一上测试工装(上电)，BL 会去外扩 ROM 中读取大固件，并从中分离出各个小固件，分别以相应的接口烧录到各个部件中去。配合工装的测试命令，直接进行自检。这样作，批量化生产是非常高效的。当然，这个 BL 开发起来也会有一定难度，最大问题可能还是各个部件烧录接口的实现(有些部件的烧录协议是比较复杂的)。

OK，上面振南就对一些 BL 实例的实现和应用场景进行了介绍。还有一些实例没有介绍，比如通过 CAN 总线或 SPI 进行文件传输，这个我们还是放到专门的章节去详细讲解。当然，各位读者可以在此基础上衍生出更多有特色而又实用的 BL 来。BL 没有最好的，只有最适合自己的。通常来说，我们并不会把 BL 设计得非常复杂，原则上它应该尽量短小精炼，好为 APP 区节省出更多的 ROM 空间。毕竟不能喧宾夺主，APP 才是产品的主角。

4、不走寻常路的 BL

4.1 Bootpatcher

我来问大家一个问题：“Bootloader 在 ROM 中的位置一定是在 APP 区前面吗？”很显然不是，AVR 就是最好的例子。那如果我们限定是 STM32 呢？似乎是的。上电复

位一定是从 0X08000000 位置开始运行的，而且 BL 一定是先于 APP 运行的。

在某些特殊的情况下，如果 APP 必须要放在 0X08000000 位置上的话，请问还有办法实现 BL 串口烧录吗？要知道 APP 在运行的时候，是不能 IAP 自己的程序存储器的。请看图 3.29。

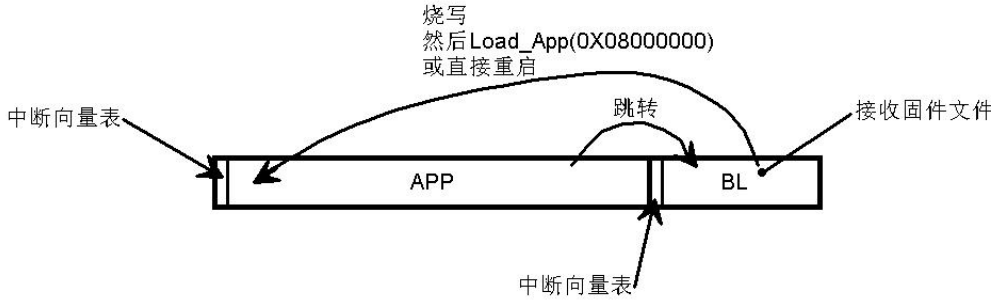

图 3.29 BL 位于 APP 之后称之为 Bootpatcher

APP 运行时，想要重新烧录自身，它可以直接跳转到后面的 BL 上，BL 运行起来之后开始接收固件文件，暂存校验 OK 之后，将固件写入到前面的 APP 区。然后跳转到 0X08000000，或者直接重启。这样新的 APP 就运行起来了。这个位于 APP 后面的 BL，我们称之为 Bootpatcher(意为启动补丁)。但是这种作法是有风险的，一旦 APP 区烧录失败，那产品就变砖了。所以这种方法一般不用。

4.2 APP 反烧 BL

前面我们都是在讲 BL 烧录 APP，那如果 BL 需要升级怎么办呢？用 JLINK。不错，不过有更直接的方法，如图 3.30 所示。

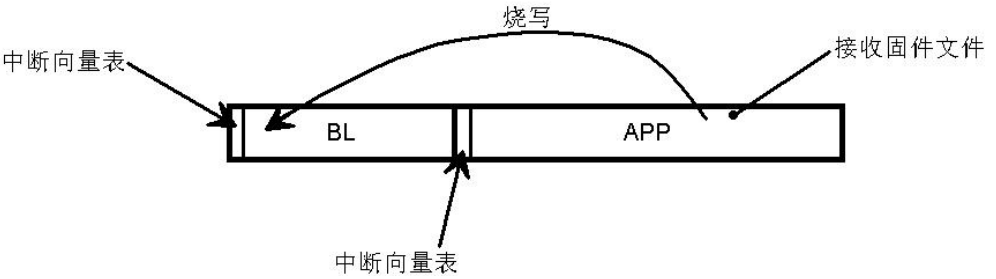

图 3.30 APP 烧录 BL 区

这是一种你想思维，我们在 APP 程序中也实现接收固件文件，暂存校验，然后将其烧录到 BL 区。这种作法与 Bootpatcher 同理，也是有一定风险的，但一般都没有问题。

OK，本章对 BL 进行了详尽的剖析讲解，应该作到了深入浅出，包含基本的原理，以及实例的实现，还有一些知识的发散。其中不乏振南的一些创新思想，希望能够对大家产生启发，在实际的工作中将这些知识付诸实践。