单片机自编程及 Bootloader 设计

Bootloader 是在单片机上电启动时执行的一小段程序。也称作固件，通过这段程序，可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用应用程序准备好正确的环境。

Boot 代码由 MCU 启动时执行的指令组成。这里的 loader 指向 MCU 的 Flash 中写入新的应用程序。因此，Bootloader 是依赖于特定的硬件而实现的，因此，在众多嵌入式产品中目前还不可能实现通用 Bootloader。

Bootloader 的最大优点是：在不需要外部编程器的情况下，对嵌入式产品的应用代码进行更新升级。它使得通过局域网或者 Intemet 远程更新程序成为可能。例如，如果有 5 000 个基于 MCU 的电能表应用程序需要更新，电能表制造商的技术人员就可以避免从事对每一个电能表重新编程的巨大工作量，通过使用 Bootloader 的功能，由控制中心通过电能表抄表系统网络，远程对 5 000 个电表重新编程。可见，Bootloader 功能对于嵌入式系统的广泛应用具有十分重要的意义。

1 78K0/Fx2 系列单片机简介

78K0/Fx2 系列是带 CAN 控制器的 8 位单片机，该系列单片机广泛应用于汽车电子，智能仪表等领域。其内置 POC（可编程上电清零电路）/LVI（可编程低电压指示器），单电压自编程闪存，引导交换功能（闪存安全保护），具有低功耗、宽电压范围、超高抗干扰等性能。

78K0 系列单片机支持自编程（Self-programming）。所谓自编程，是指用 Flash 存储器中的驻留的软件或程序对 Flash 存储器进行擦除/编程的方法。通过单片机的自编程功能，可以设计 Bootloader 程序，通过串口等通信接口实现对产品重新编程、在线升级的功能。

以 μ PD78F0881 为例。μ PD78F0881 为 78K0/Fx2 系列中的一款 44 管脚单片机，内置 32 KB Flash ROM，2 KB RAM，自带 2 个串行通信接口。其内部 Flash 结构如图 1 所示。为了方便实现擦除和编程，人为地将整个 Flash 分成若干个 block，每个 block 大小为 1 KB。block 为自编程库函数中空白检测、擦除、校验的最小单位。block0 从地址 0000H 开始，程序都从 0000H 开始执行。block0～block3 共 4 KB 存储空间为 Bootloader 程序存储区域。block4～block31 为应用程序存储区域。

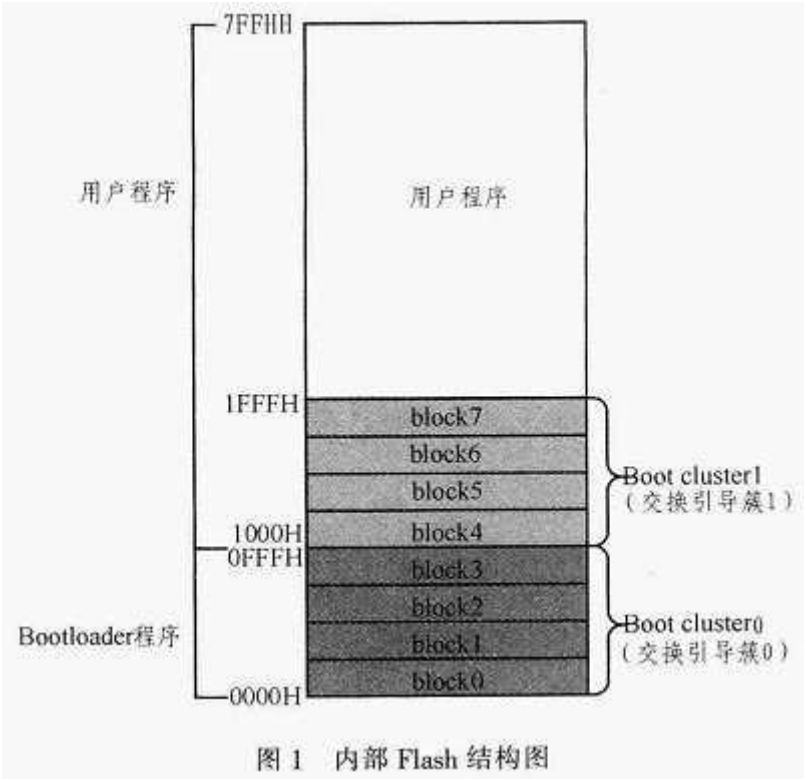

图 1　内部 Flash 结构图

为了防止 Bootloader 自身的升级失败，设计了引导交换功能。该功能定义 2 个簇，即 Boot cluster0 和 Boot cluster1。Boot clustee0 为 block0～block3 的 4 KB 存储空间，Boot cluster1 为 block4～block7 的 4 KB 存储空间。因此，实际运用过程中，一般把应用程序的开始定义在 2000H，也就是从 block8 开始。

Flash 地址为 0000H～FFFFH。7FFFFH～FFFFH 存储空间为保留区域以及特殊功能寄存器区域等，用户无法对其进行编程。

2 自编程

2.1 自编程环境

2.1.1 硬件环境

FLMD0 引脚是 78K0/Fx2 系列单片机为 Flash 编程模式设置的，用于控制 MCU 进入编程模式。在通常操作情况下，FLMD0 引脚下拉到地。要进入自编程模式，必须使 FLMD0 引脚置成高电平。因此，通过一个普通 I/O 接口控制 FLMD0 引脚的电平。如图 2 所示。

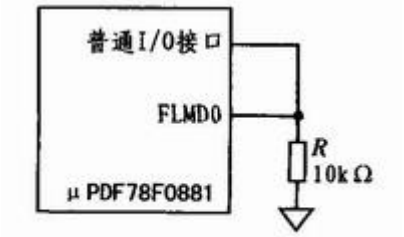

图 2　FLMD0 引脚连接示意图

2.1.2 软件环境

1）使用通用寄存器 bank3，自编程库函数，需要调用通用寄存器 bank3。因此，在自编程时，不能对通用寄存器 bank3 操作。

2）使用 100 B RAM（入口 RAM）作为隐藏 ROM 中函数的工作区，入口 RAM，是 Flash 存储器自编程样例库所使用的 RAM 区域。用户程序需要保留着块区域，当调用库时，需要指定这片区域的起始地址。入口 RAM 地址可以指定在 FB00h～FE20h 之间。

3）4～256 B RAM 作为数据缓冲区，必须是 FE20H～FE83H 以外的内部高速 RAM 区域。

4）最大 39 B RAM 作为隐藏 ROM 函数的堆栈。

5）隐藏 ROM 中的函数被 0000H～7FFFH 中的应用程序调用。

2.2 自编程流程

自编程功能利用自编程软件库完成用户程序对 Flash 内容的重新编程。如果在自编程的过程中有中断发生，那么自编程将暂停来响应中断。中断结束，自编程模式恢复后，自编程过程将继续进行。采用汇编语言编写 78K0/Fx2 自编程软件库，如表 1 所示。

表 1　78K0/Fx2 自编程库
Tab. 1　78K0/Fx2 self-programming library

库函数名称	调用函数	说明
自编程启动函数	CALL！_FlashStart	声明自编程的开始
初始化函数	CALL！_FlashEnv	入口 RAM 的初始化、FLMD0 引脚置高电平
模式检测函数	CALL！_CheckFLMD	确认 FLMD0 引脚电平
Block 空白检测函数	CALL！_FlashBlockCheck	确认制定 Block 是否为空白
Block 擦除函数	CALL！_FlashBlockErase	擦除指定 Block
字写入函数	CALL！_FlashWordWrite	向指定地址写入 1~64 个字数据
Block 验证函数	CALL！_FlashBlockVerify	-
自编程结束函数	CALL！_FlashEnd	声明自编程结束
读取信息函数	CALL！_FlashGetInfo	读取 Flash 信息
改变信息设置函数	CALL！_FlashSetInfo	改变 Flash 信息设置

自编程操作流程如图 3 所示，当单片机收到自编程执行信号时，开始进入自编程模式。将 FLMD0 引脚设置成高电平，初始化入口 RAM，为自编程库函数开辟空间。当确认 FLMD0 为自编程状态时，开始检查需要编程区域是否为空白区域。当被编程区域不是空白区域时，先将其擦除，然后在此区域进行编程。编程结束后进行校验。若校验无误，则将 FLMD0 引脚设置成低电平，退出自编程模式。

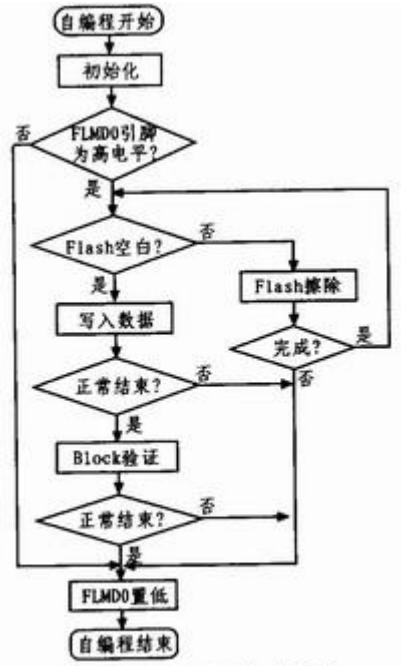

图 3　自编程操作流程图

3 引导交换（boot swap）

产品程序的升级包括应用程序的升级和引导程序（Bootloader 自身）的升级。为了防止引导程序在升级的过程中发生错误，从而导致 MCU 无法启动，设计了引导交换功能。以图 4 说明引导交换的实现过程。

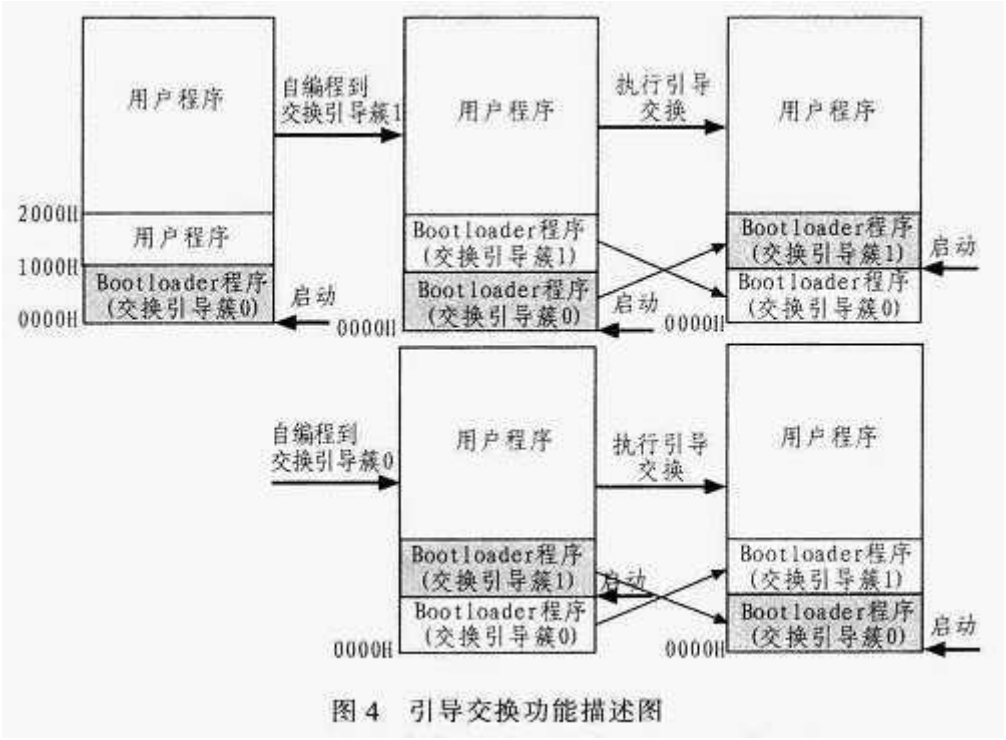

图 4　引导交换功能描述图

1）旧的 Boot 程序首先将新的 Boot 程序编程到交换引导簇 1（Boot cluster 1），然后设置启动交换标志位，并强迫看门狗复位。

2）复位启动后，MCU 看到交换标志位，便从交换引导簇 1 处开始启动。交换引导簇 1 处的新 Boot 程序将检查交换标志位。如果交换标志位被置 1，则新的 Boot 程序将擦除交换引导簇 0（Boot cluster 0）区域，并将自身复制到交换引导簇 0，然后将交换标志位清零，强迫看门狗复位。

3）复位启动后，MCU 看到交换标志位被清零。又从交换引导簇 0 处开始执行。这样就完成了 boot 程序自身的升级。即使在升级过程中遇到断电等异常情况，在重新上电后也能重新完成 Boot 程序升级。有效地防止在升级过程中出现断电等等异常情况而导致升级失败，MCU 无法启动的问题，使 Boot 程序的升级变得安全可靠。

4 Bootloadler 设计

4.1 简单的 Bootloader

一个简单的 Bootload 包括 5 个元素。

1）启动 Bootloader 的信号 Bootloader 程序是在执行应用程序之前所执行的一小段程序，当 Bootloader 程序把控制权转交给应用程序后，在 MCU 复位前，Bootloader 程序将不再执行。因此，需要产生一个信号触发 MCU 开始 Bootloader

程序。该信号可以是中断，也可以通过串口传送的一条指令，或者是别的程序触发的信号。

2）执行 Bootloader 的信号 单片机程序启动时，MCU 是装载新的应用程序还是执行已经存在的程序取决于外部信号。该信号可以是上电时的一个端口信号，用来控制 MCU 装载新程序还是执行旧程序，也可以是从串口接收到的指令等。

3）将新的代码传送给 MCU 通过 RS485、I2C、CAN 或者 USB 传送新的应用程序数据。因为要传送的代码一般会超过 MCU 的 RAM 容量，因此需要一些控制数据流量的措施。一般使用 XON/XOFF 软件握手协议，传送代码的格式一般选择 Intel hex 格式。

4）Flash 新代码的自动编程 每次 MCU 接收到一批新的数据，就要将其编程到正确的 Flash 地址。如果该地址非空白，MCU 在编程前必须先擦除。一般在编程中或者编程后还需要检查存储器的内容。

5）将控制权转移给有效的应用程序 在接收和编程了新的代码后，Bootloader 写一个校验和或者其他唯一字节序列到一个固定的存储单元。Bootloader 检测该值，如果该值存在，Bootloader 就将控制权传给应用程序。

4.2 Intelhex 格式

在线升级的程序代码采用编译器输出的 Intel hex 格式文件。Intel hex 文件常用来保存单片机或其他微处理器的程序代码。它保存物理程序存储区中的目标代码映象。一般的编程器都支持这种格式。Intel hex 文件记录中的数字都是十六进制格式。在 InteI hex 文件中，每一行包含一个 HEX 记录。Intel hex 文件通常用于传输将被存于 Flash 或者 EEPROM 中的程序和数据。Intel hex 由任意数量的十六进制记录组成。每个记录包含 5 个域，它们按照图 5 所示格式排列。

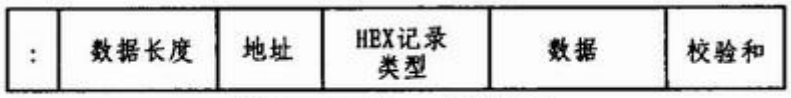

图 5 Intel hex 文件格式

每一个部分至少由 2 个十六进制编码字符组成。它们构成 1 个字节。每一个部分的意义如下所述：

1）每个 Intel hex 记录都由冒号开头，自编程的过程中以此判断一个 Intel hex 记录的开始。

2）数据长度代表当前记录中数据字节的数量。

3）地址代表当前记录中数据在存储区域中的起始地址。

4）HEX 记录类型有如下 4 种：00-数据记录;01-文件结束记录;02-扩展段地址记录;03-转移地址记录。NEC 编译器输出的 Intel hex 文件中。只包含数据类型 00 和 01。其中 01 作为自编程过程中数据结束的判定标志。

5）数据域分用于存储需要写入 Flash 中的内容，一个记录可以有许多数据字节。记录中的数据字节数量必须与数据长度中的值相符。

6）校验和是取记录中从数据长度到数据域最后一个字节的所有字节总和的 2 的补码。

根据以上说明，必须在程序中对接收到的 Inter hex 文件进行解码，获取数据以及数据地址，并对收到的数据进行校验，然后将接收正确的数据编程到 Flash 相应的地址上。

4.3 Bootloader 设计思路

单片机收到启动信号后，重新启动程序。启动的时候首先执行 Boot 代码，Boot 代码检查是否收到执行升级信号。如果需要升级程序，则通过串口或者其他通信接口接收新的应用程序，loader 程序向单片机 Flash 中写入新的应用程序代码。最后通过检查校验位检测程序是否有效。如果有效，则 Bootloader 将 CPUMCU 控制权交给应用程序。整个升级过程完成。Bootloader 执行过程如图 6 所示。

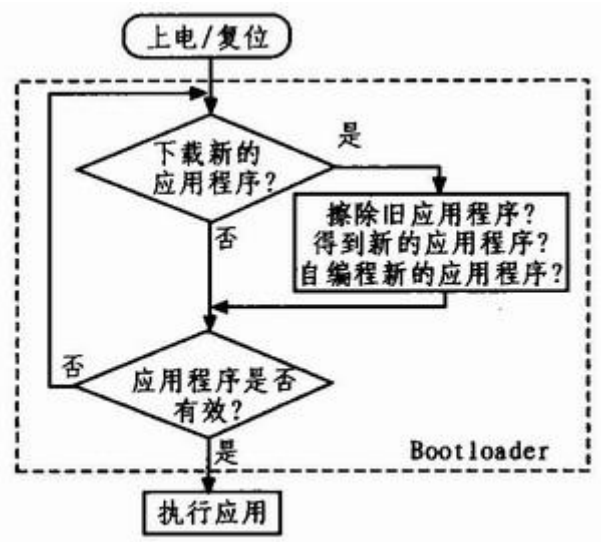

图 6　Bootloader 流程图

需要注意的是 Bootloader 自身的更新和应用程序的更新还需区别处理。通过辨别接收到数据的编程地址来判断是 Bootloader 更新还是应用程序更新。若编程地址从 0000H 开始，则为 Bootloader 更新。Bootloader 更新则需要执行引导交换（boot swap）功能;若为应用程序更新，自编程结束后，直接将 CPU 交给应用程序。

5 结束语

本文探讨了 78K0/FC2 系列μ PD78F0881 单片机的自编程功能以及 Bootloader 的设计方法。具体描述了通过单片机串口对相应的应用程序通过 Bootloader 进行升级。此版本的 Bootloader 使用晶振 20 MHz,通过串口 Uart60，设置波特率为 115 200，在μ PD78F0881 单片机上成功实现了用户应用程序的升级更新。在接下来的工作中，Bootloader 的设计应当面向更多的通信接口。例如，通过 CAN 总线接口升级，通过 USB 接口升级等等。