



让U盘固件开发成为过去[文章纪念]

作者：frank_wang

当你准备开发 U 盘的固件时，是否心里面仍然不踏实，让这些成为过去吧。

今年年初的时候，在驱动开发上发贴《在 ARM7 上使用 D12 开发 U 盘的详尽技术报告》。其中详尽地介绍了我在 ARM7 上用 D12 开发 U 盘的过程，以及所涉及到的方方面面的知识。得到了一些朋友的肯定和鼓励。从中我也体会到了帮助别人的快乐。后来忙于别的，不经常上驱动了，但偶尔看一下仍然有许多人遇到 U 盘开发中的各种各样的问题，心里觉得十分不快。因为看到还有许多人仍然在为已经不存在什么问题的内容所折磨，看到我们的技术开发者仍然处于一种单干的状态，技术交流的模式和通道仍然不畅.....别人做过的工作，我们完全可以用一种合理的方式拿过来为己所用，而把精力放在更多未知的问题上的。

后来再做 U 盘文件系统（即在固件中创建文件，在 PC 上可以读取）USB Host（可以读写 U 盘）的过程中，也碰到过一些问题，在向别人征询时，有一些朋友很爽快，愿意交换彼此的源码，有一些网友则是有所保留，可能每个人都忙，别人也没有时间解答所有问题。感觉就是得到的帮助往往是模糊的，往往还隔层纸。我们一段时间将问题整理一次，放到开发网上，作为大家的参考。我也可以考虑把 Bulk Only 和 SCSI 命令集响应部分的源码以合理的方式提供给大家，并乐意解答其中的问题。

总之，我的愿望是，所有中国人，想做 U 盘的，让这个都不再是问题，让我们把精力放在其他更多更重要的事情上，让我们的整体水平更上层楼。我愿意为我这个愿望做些什么。同时，我也愿意跟大家交流我曾经做过的一些项目内容，比如文件系统读写，Usb - Host 控制 U 盘等等。

USB 固件编程之一：固件编程的工作内容

USB 固件编程可以用以下语句来精练地进行描述：

Device 的固件编程，要搞定的是那几个端点。端点多少和配置情况受所用的 Device 芯片决定，具体可以看芯片资料。芯片一般提供一个中断信号，与单片机接口时，只要端点接受到数据，或发送数据成功后，便后产生中断，在固件里面，只要对些中断进行响应即可。

当 Device 接收到数据时，对这些数据进行分析处理（端点 0 遵守标准的数据格式，其他端点受端点类型的不同，有不同的数据格式），一般来说，这些数据是主机对设备发出的请求，设备只要响应主机的这些请求即可。Device 芯片发送完数据后也会产生中断，这个中断信号告诉与之接口的单片机，可以继续发送后续的数据。

USB 固件，好比一个有“妻管炎”的男人，而主机，则好是一个女管家。一般来说，主机让干啥就干啥，所以，USB 固件程序的结构一般是基于中断处理的。平时，主程序做完必要的初始化工作后，就什么事也不做了，等待 USB 中断的产生，中断产生后，根据中断状态对相应的端点读取数据，或是向相应的端点发送数据。这一点是 USB 通讯协议的主 - 从模



式先天决定的。但让人不可思议的是，这还有点象是母系氏族时期，因为，一个 USB 总线上，只能有一个主机，可以有多个设备，整个 USB 总线上通讯的协议和处理，发起与中止，基本上是主机控制的。因此，固件编程中，只要满足了主机的要求，就万事大吉了，可以确保自己的氏族中的应有地位。

U 盘固件编程之二:固件编程的几个主要部分

在整个 U 盘固件中,程序从功能模块上分成两个部分:USB 协议的处理和对 Flash 的读写。

USB 协议的处理又分成两个方面。

一是端点 0 的配置过程。所有 USB 设备在插入 USB 端口时，主机都通过地址 0 与设备的端口 0 进行通讯。在这个过程中,主机发出一系列得到描述符的请求,通过这些请求,主机得到所有感兴趣的设备的描述符,从而知道设备的情况,然后通过 Set Address 为设备设置一个唯一的地址,配置过程完成以后,主机就通过为设备所设定的地址与设备通讯,而不再是使用默认地址 0。配置地址后,可能还要获取一次描述符,然后设置配置(Set Configuration)，之后便完成了对新插入 USB 总线的设备的配置过程。

二是其他端点的数据通讯过程。在配置阶段中，主机已经知道了设备的端点的使用情况，以后，便可以通过这些端点来进行特定传输方式的通讯了。对于 U 盘来说，有两种传输方式，BULK ONLY 和 CBI 方式，一般使用 Bulk Only 较多。这种传输方式要使用特定的 Bulk 端点，然后还要为其选择一种命令集。比如 UFI 或 SCSI，因为 Bulk 端点的数据没有特定的数据格式，因此，需要使用某种命令集，来约定所传数据的格式。对于 U 盘固件编程来讲，就是要处理 BULK 端点的各种数据通讯。

除了对各个相关的端点的 USB 协议的处理，剩下的就是 FLASH 的读写问题。这里存在两个层面的问题。

一是解决 Flash 读写的问题，就是说你使用的 Flash，先要实现成功的读写和擦除，这部分内容，是比较成熟的，一般都使用三星公司出的 K9FXX08 系列的，有 16M (2808)，32M (5608)，64M (1208)，138M (1G08)。它们的封装一致，只需要软件编程中稍做修改，便可以进行适应于另一种容量的存储器。

第二个层面的问题，就是在 U 盘通讯过程中的问题了。NAND 型的 Flash 有个特点，不可随机存取，擦除操作一次对 16K 的内容进行。所以，在 U 盘响应过程中，不可避免要对数据进行缓存。如果你的 U 盘方案中有较宽裕的 RAM (超过 16K)，这个问题变得简单，只需要开一个 16K 的数组，把数据存到这 16K 中，最后再写入 Flash 即可。否则，在缓冲上面是要花一些功夫的。最基本的思路是用 Flash 的另外一个 Block 做缓冲空间。

但这种方式会引发下列问题：1、速度；2、Flash 的那个用来做缓存的块将比别的块使用频率大幅上升，磨损最严重，最先坏，这影响整个 Flash 的寿命。在实际处理中，引入了一系列的折中方案，比如，对 Write 命令所写的 Block 号进行判断，如果是整个的数据 Block，则直接擦除原有内容，将数据写入。再如，对于非整个 Block 的数据进行缓冲，而对于整个



Block 的读写，不缓冲直接写入。再就是对于前面文件分配表、目录项所在的 Block 进行缓冲，等等。经过这些处理，可以尽可能地提高 Write 的速度。

U 盘固件编程之三:合理的 USB 通讯调试方法和思路是成功的关键

在介绍更多细节内容之前，我不得不谈谈我对 USB 调试方法的理解。

USB 通讯过程是一个动态的过程，是不太好使用硬件仿真器来设断点调试的，因为每一次 USB 的传输过程，都有时效要求，等待时间过长，通讯过程也就中止了。但也不排除可以巧妙地使用断点仿真的方法进行调试。但个人认为，使用串口辅助编程过程，却是一种经济有效的方法。

所谓用串口辅助调试过程，也就是在固件代码中加入类似于 Printf 的语句，向串口输出一些信息。这些信息可以是几个字符（如 A、B、C），或是某个变量或寄存器的值。程序运行到此处时，便会输出这些信息，借此，便可以知道：1、程序是否运行到此处；2、运行到此处时相应变量或寄存器值。这不就是硬件仿真调试的功能么？

如果想使用这种方式来调试，在硬件上必须增加一个 RS232 串口电平转换芯片，而且所使用的 MCU 得要有串口，并且，一般要自己编写 Printf 函数的实现方式。这个翻翻串口控制方面的书籍，很容易就可以搞定。

串口调试的方法，还可以推广到其他的单片机应用中，在简单系统中，它基本可以替代掉硬件仿真器，降低开发者的门槛和投资。

在 USB 通讯过程中，有两个阶段，一是通过端点 0 完成对设备的配置，在此阶段，把从 USB 端点得到的数据输出到串口，就对通讯所处的阶段一目了然了。一旦完成配置阶段，Bus Hound 便可以粉墨登场了，因为此时，Bus Hound 中已经可以看到设备了，看到设备后，便可以选择设备，对主机与此设备间的通讯数据进行分析 and 监视。

串口调试和 Bus Hound 这两种手段配合使用，可以使 USB 通讯过程的调试更加容易。比如，刚开始时，端点 0 的数据量本来就少，因此，使用串口调试比较方便。而对于 Bulk 端点的数据传送过程，再使用串口就不太方便了，因为数据量大，串口输出的数据太多，延时会比较严重，影响 USB 通讯过程，所以改用 BUS HOUND 来监视 USB 总线上的数据。这个时候很有趣的一件事情是，Bus Hound 在 PC 机上，而串口实际上在单片机端。所以，利用这两种手段，里应外合，有助于我们确定一方发时另一方收的数据是否正确。比如，单片机上发出的一组数，将其输出到串口，然后看看 Bus Hound 上是否收到的是这些数，如果正确，则说明硬件通讯过程没有问题，如果不正确，则说明通讯的某一方有问题，进一步可以定位此问题，排除之。

正确的调试思路，将使调试过程事半功倍。

比如，在调试端点 0 的配置过程时，可以先用 Switch...Case 语句建立对于端点 0 的数据的分支响应，对照标准请求的数据格式，可以得到什么情况下是 Get Device Descriptor，什么时候是 Get Configuration Descriptor，每个分支处理时对应一个函数，在这个函数里向串口标志信息。这个工作完成以后，便一个一个地来处理请求，处理完一个后，主机自动进入下



一个阶段，这时，通过串口可以看到相应的状态，按部就班地一个一个处理余下的请求，即可完成端点 0 的设备配置过程的固件程序的编写。

对于 Bulk 端点也是一样，先建立程序框架，然后再一个一个地处理请求。这种自上而下，逐步求精的思路并不稀奇，在 USB 调试过程中十分受用。最忌一上来就处理请求，不讲求结构，不讲求代码的条理性，最后可能弄得自己都是一头雾水。

U 盘固件编程之四:玩转你的端点

接上面我来谈谈端点的问题。

前面提到过，端点是由 USB 设备端的接口芯片决定的。你选择了什么样的芯片，那么端点的配置情况属性就已经决定了，你只能使用将就这些特定的情况。这些端点的配置，具体要参考你所使用的接口芯片的芯片资料，比如说，端点 0 当然都为控制端点，其 MaxPacketSize 可能为 8, 16, 32, 64；端点 1 可能是 Bulk - In 端点，2 是 Bulk - Out 端点，其字长也有可能是 8, 16, 32, 64，但一般是 64。其他端点可能有同步端点，或者同一端点既可被配置成同步传输方式，也可以工作在 Bulk 传输方式下，等等，不一而足。

USB 协议精妙之处就在于枚举过程。主机最初发过来的包，一定是 8 个字符长的。所以，你的端点的 MaxPacketSize 至少必须是 8，能满足与主机之间最基本的通讯过程。对于主机的第一个请求 Get Device Descriptor，你也只用回复 8 个字符就 OK 了，因为主机在第一次只对这 8 个字符感兴趣，在后面逐渐的获取描述符的过程中，主机逐渐得到设备使用那些端点，每个端点的最大字长（这些内容在 Endpoint Descriptor 中，通过 Configuration Descriptor 提供）是多少，等等，总之，通过枚举，主机便知道你的端点的情况了，以后就会用这些端点来与设备进行通讯。

对于 Hewx 的问题，我想是你在 Endpoint Descriptor 中没有正确进行端点的设置，因为，如果进行了正确的端点配置，主机是会自动通过 Bulk 端点来发 Inquiry 命令的，而不会从你说的 Endpoint1（16B）来发送这一信息。而且，主机会自动对要发送的信息进行分割，每次以不高于相应端点的 MaxPacketSize 长度来发送。

除了描述符中要给出正确的端点描述符的描述，有些时候在芯片中也需要设备相应的控制位，在决定你要使用哪些及如何使用这些端点，这个也得根据具体的芯片资料来设置。

U 盘固件编程之四:玩转你的端点（增补）

对于某种设备来说，需要使用到的端点是固定的。比如说，Mass Storage 设备吧，就只需要用到一个 Bulk - In 端点和一个 Bulk - Out 端点。而不需要几个此类端点。至于到底需要几个端点，完全需要根据有关协议中的说明进行，描述符也据此进行提供，而不是没有根据地在描述符中提供许多端点？