什么是USB的枚举(Enumeration)?

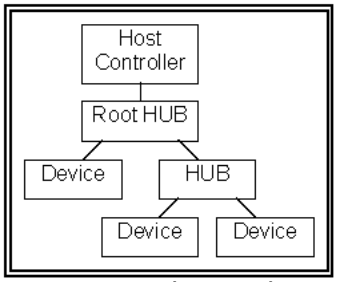
枚举是某个USB设备连接到系统并指派一个明确的地址码的过程，地址码用来访问个别设备。USB主机控制器查询设备属于哪一类的设备时也尝试为其绑定适当的驱动程序。
主机发往设备的一些基本命令：

- 设置地址—指示设备更改它的当前地址设置
- 取设备描述符—关于设备的全部信息（制造厂商，固件版本…）
- 取配置描述符—端口的使用方式
- 取界面描述符—设备可能使用的界面
- 取字符串描述符—制造厂商和产品名的Unicode格式字符串

每个USB设备都有这个基本的过程，如果没有它，设备将永远不能被操作系统使用。

枚举看起来象什么?

我相信要轻松地解释USB枚举过程的方法是展示它是怎样发生的。一个CamConnect演示固件，显而易见，包含的代码让它能够在任意的USB系统上枚举。通过从CamConnect 固件源代码取一小段使用CATC USB分析器进行跟踪，你可以跟踪和理解枚举过程。



USB Connection Topology

设备插入初期:

所有USB设备会插入集线器的某个端口。连接后，集线器检测设备是全速设备还是低速设备。对于全速设备，从D+线接一个1.5k上拉电阻到3.3v电源，从D-线接则表示低速设备。
一旦集线器检测到新设备连接，它开始让由主机每隔1毫秒产生的、发往设备的帧开始包（SOF）通过。主机控制器为了枚举新设备也开始向设备发出设置包。

当一个设备刚插入时，它总是使用默认的设备地址0进行通信。枚举过程中，主机控制器分配一个新的地址给设备使用。枚举过程通信总是使用设备的端口0。控制传输的规定。USB控制传输必须使用设备的端口0。

在主机接收到设备的全部描述符后，操作系统尝试查找适当的设备驱动程序与新设备相关联。

帧开始:

下面是主机每隔1毫秒发送的帧开始包的例子。这个帧开始包会发送给总线上的每一个设备，因此它们能够保持同步。总线上的每一个USB包都以一个同步块开始以使设备能够同步收发。大多数类型的包都包含一个CRC值。M16C USB硬件自动处理这些细节。

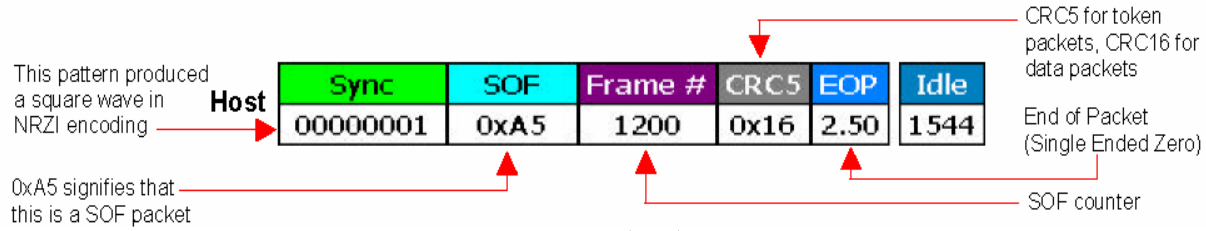


图1 帧开始包（SOF）

初始通信：

USB线上首先会发生什么呢，一些USB初学者对此有些混乱。，直到有人向我解释之前，我自己看到时也是相当困惑的。这依赖于USB主机控制器的实现，看到总线在交换信息时总会以为它是错误的，但实际上，这是USB主机控制器的必须的、标准的过程。

是什么这么神秘呢？长话短说，USB主机控制器在做任何事之前会先查询设备的设备描述符。有趣的是设备描述符长18字节，但是主机并不关心这些，只需要前8个字节。在接收之后，主机并不会接着查询设备的其余数据。此外，主机复位该线并开始发送USB枚举命令。

主机这样做的原因是因为设备描述符包含供控制传输端口0充许的最大有效荷载尺寸。这个值包含在发送回主机的设备描述符内的第8字节。所以主机首先使用GetDescriptor命令询问设备就是为了得到这个值。一旦主机检测到这个数，它复位USB线并开始枚举过程。

下面的包跟踪展示第一件事是发送设置标志到设备的端口0（EP0），指示返回设备描述符。

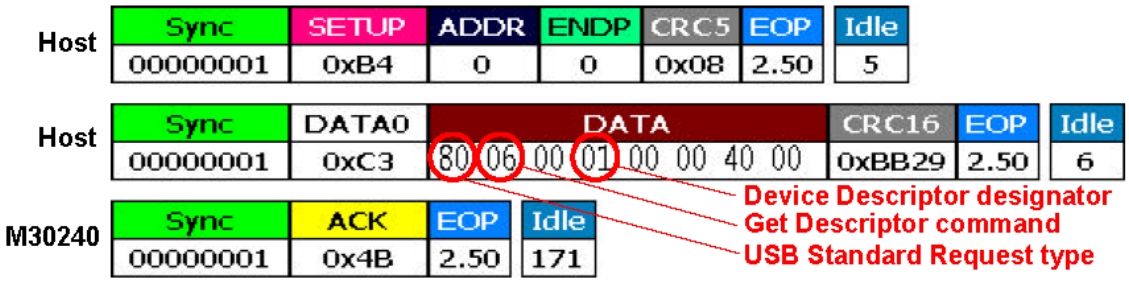


图2 从主机启动取设备描述符

下面可以看到设备返回18字节长的描述符的首8个字节，主机接着发出另一个输入包，因此我们可以开始传送另一个8字节的数据包，它发出一个输出包，并后跟一个无数据的包。我们用ACK回应这个传送，接着主机控制器复位USB线。

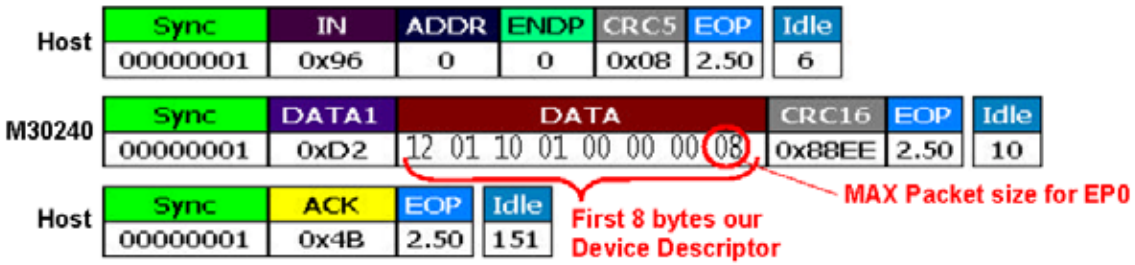


图3 设备回应初始的取描述符请求

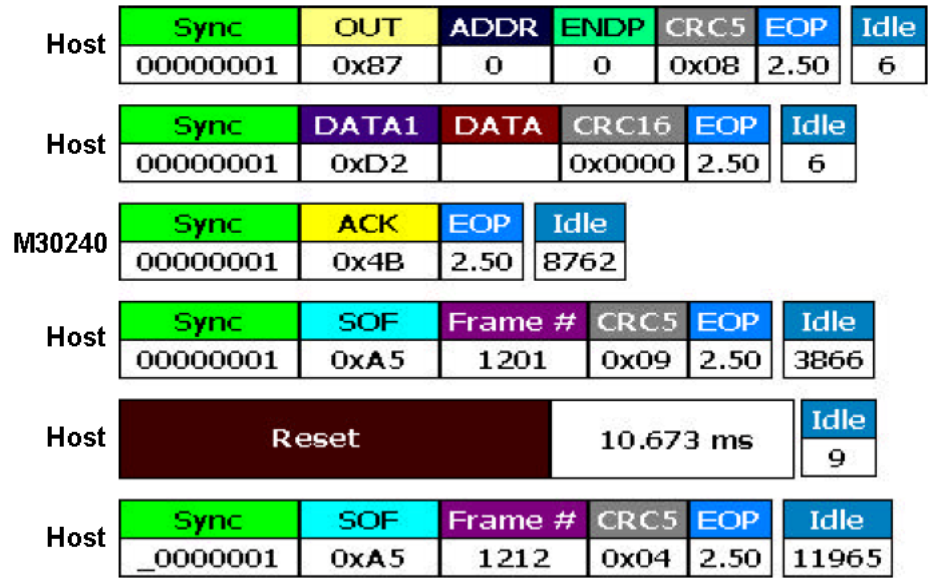


图4 主机复位USB线并开始枚举

设置地址：in order to provide,

对于枚举过程，传送到设备的第一个命令是设置地址命令。在讨论之前，总线上的新设备临时使用地址0,这是主机提供的通信方式。接着主机指派一个地址号码给设备使用，在总线上这个地址号码是唯一的。

下面，你可以看到一个设置包发送到设备0的0端口，8字节数据是用来判断将要发送的设置包的类型，及需要赋些什么值。M16C USB硬件有一个寄存器用于维护当前设备地址。复位后，寄存器默认值为0，可以在任意时间内写入以更改USB硬件应答地址。数据接收无错时M16C USB硬件自动向主机回应一个ACK握手包。

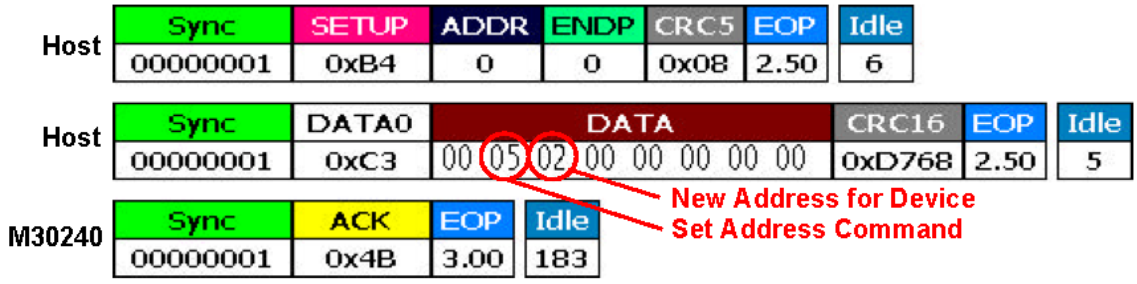


图5．发送到设备的设置地址命令

现在，让我们来看一下CamConnect的固件是怎样解码这个包的。

（The M16C’s USB hardware will automatically accept and respond to device requests addressed to its current device address.）。枚举开始时，地址寄存器的值为0，设备会回应所有包地址到设备0。

任何时候，USB包发送的设备地址与我们的当前设备地址相匹配时，M16C的USB硬件会产生一个中断（USBF-USB功能中断）。中断控制通过查询USB寄存器来找到引起中断的原因并作回应。

下面是一个USBF中断的USB中断服务例程。

```
void USB_Int_Handler() {
/* Save and clear the current EP interrupts */
USB_IntReg1 = usbis1; /* USB Interrupt Status Register 1 */
USB_IntReg2 = usbis2; /* USB Interrupt Status Register 2 */
```

```
/* Write this value back in order to clear those interrupts */
usbis1 = USB_IntReg1;
usbis2 = USB_IntReg2;
/* We will use the mirrored variables for checking endpoint interrupts*/
/* == Check for EP0 Interrupt Status Flag ==*/
if( USB_IntReg1 & 01) {
ParseEP0Packet(); /* Service EP0 Request */
USB_IntReg1 &= 0xFE; /* Clear USBINT0 bit in mirror */
}
. . . .
. . . .
}
```

就象所看到的那样，我们首先保存中断状态寄存器，接着把值写回以清除任何已设置的位。然后使用这些信息来判断是哪个端口引起中断。其后是一个设置包，它使用端口0。接着调用适当的函数来处理，ParseEP0Packet()，在下面列出。

```
void ParseEP0Packet() {
if(ep0csr0 !=1 ) /* Check Out Packet Ready flag for EP0 set */
return; /* Fifo not ready to be read to return */
/* Read the out 8 byte header from EP0 FIFO*/
EP0_Header.bmRequestType = ep0;
EP0_Header.bRequest = ep0;
EP0_Header.wValueLow = ep0;
EP0_Header.wValueHigh = ep0;
EP0_Header.wIndexLow = ep0;
EP0_Header.wIndexHigh = ep0;
EP0_Header.wLengthLow = ep0;
EP0_Header.wLengthHigh = ep0;
/* Mask out all but request type ( 01100000 ) */
tmp_byte = EP0_Header.bmRequestType;
tmp_byte &= 0x60;
switch( tmp_byte ) {
case 0: ProcessStandardReq(); // USB Chapter 9 stuff
break;
case 0x20: ProcessClassReq(); // Specific Class stuff
break;
case 0x40: ProcessVenderReq(); // Custom stuff
}
}
```

从上面可以看到8字节数据从端口0读出到一个结构变量。注意发送的数据中每一个设置包都有相同的8字节格式。一旦读出数据，就能对它进行分析。可以看到我们关注第一字节的第5、6位，它在USB里表示为bmRequestType，用于确定请求类型的形成。所有USB枚举请求都通过标准请求来产生。

函数ProcessStandardRequest()在下面列出，每个调用进行更深层次的数据包解码。

```
void ProcessStandardReq() {
/* Determine what is being requested */
switch( EP0_Header.bRequest ) {
case 0: CmdGetStatus();
```

```
break;
case 1: CmdClearFeature();
break;
case 3: CmdSetFeature();
break;
case 5: CmdSetAddress();
break;
case 6: CmdGetDescriptor();
break;
case 7: CmdSetDescriptor();
break;
case 8: CmdGetConfiguration();
break;
case 9: CmdSetConfiguration();
break;
case 10: CmdGetInterface();
break;
case 11: CmdSetInterface();
break;
case 12: CmdSynchFrame();
break;
default: ep0csr = 0x44; // Clear out pky ready with send stall
ep0csr2 = 1; // Stall all subsequent transactions
asm("nop");
asm("nop");
}
}
```

USB主机发送的数据包的第二个字节的使用，在USB里它表示为bRequest，我们能够确定正执行的设置命令的类型。上面列出了USB规范里所有可能的标准USB请求。

注意早期展示的包图表中，那里是一个5，与设置地址命令相关。接着调用适当的函数来处理这个请求，见下面的CmdSetAddress()函数。

```
void CmdSetAddress() {
/* Load our new Device address */
usba = EP0_Header.wValueLow;
/* Set DATA_END and OUT_PKT_RDY bit for EP0 */
ep0csr = 0x48;
}
```

这时候，判断给我们的设置请求包属于哪一类。看到这是一个设置地址命令，所需要做的是指示USB硬件用新地址开始接收数据并向主机回应已理解这个请求，任务完成。

变量 usba 事实上是M16C的USB设备地址寄存器的符号连接。在写新值到这个寄存器后，USB硬件仅对新设备地址进行自动应答。这个值是从主机来的wValue字的低字节（8字节数据流的第3个字节）传入的。在图4的设置包引用这个数据。可以看到新设备地址是2。

最后我们为端口0设置M16C的USB寄存器的DATA_END位和OUT_PKT_RDY位，这将使M16C的USB硬件向主机回应一个长度为0的数据包（也称空数据包）。这表示同意改变地址请求。在下面列出。

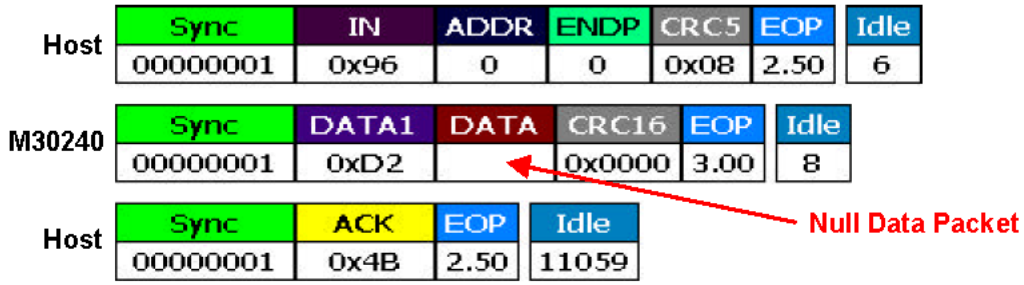


图6．改变地址请求的设备确认

为了接收结构，主机向设备发送输入请求包（设备依然使用地址0）。可以看到，回答空数据包表示正确。回答空数据包的原因是依照USB规范，设备可以不确认（NAK）主机的输入（IN）标识，想多久都行。主机保持发送输入（IN）标识到那个设备直到收到应答。在这里，主机使用输入（IN）标识表示：“准备好开始在新地址接收数据了吗？”。

设置适当的USB寄存器后，M16C的USB硬件会自动发回NAK包直到固件完成任务。因此，空数据包好象在说：“是的，我现在准备好了，你可以继续。”主机回应一个ACK表示回答已正确接收。

取描述符：

与枚举过程的其余部分类似。就象追踪从连接新设备的USB主机控制器来的系统查询、命令。当主机获得足够的信息来搜索适用驱动程序时，它将停止发送标准请求的设置包。这时候，设备被枚举。在枚举过程中，GetDescription 是另一个重要的设置命令。不象开始时从主机发出第一个取描述符（GetDescriptor）那样，这时候我们希望向主机传送全部的描述符。设置包在下面列出。

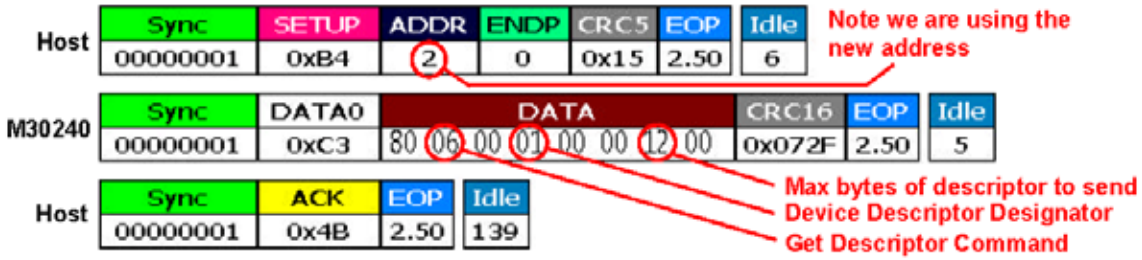


图7．从主机取设备描述符命令

固件会以下列步骤…

```
void USB_Int_Handler() // USB Interrupt Sub Routine
void ParseEP0Packet() // EP0 Control packet parsing function
void ProcessStandardReq() // USB Standard Request
void CmdGetDescriptor() // Service GetDescriptor command
```

在CmdGetDescriptor() 例程中，将18字节的描述符拆分为8字节数据回应。在例程中，仅当主机发送一个输入标识时才会向主机回应数据。我们只需简单地等待下一个到主机的输入包，填充到端口0的FIFO，将M16C的其中一个USB寄存器用于EP0并设置IN_PKT_RDY位，硬件会将它传送到主机。

下面展示这些动作的总线跟踪。

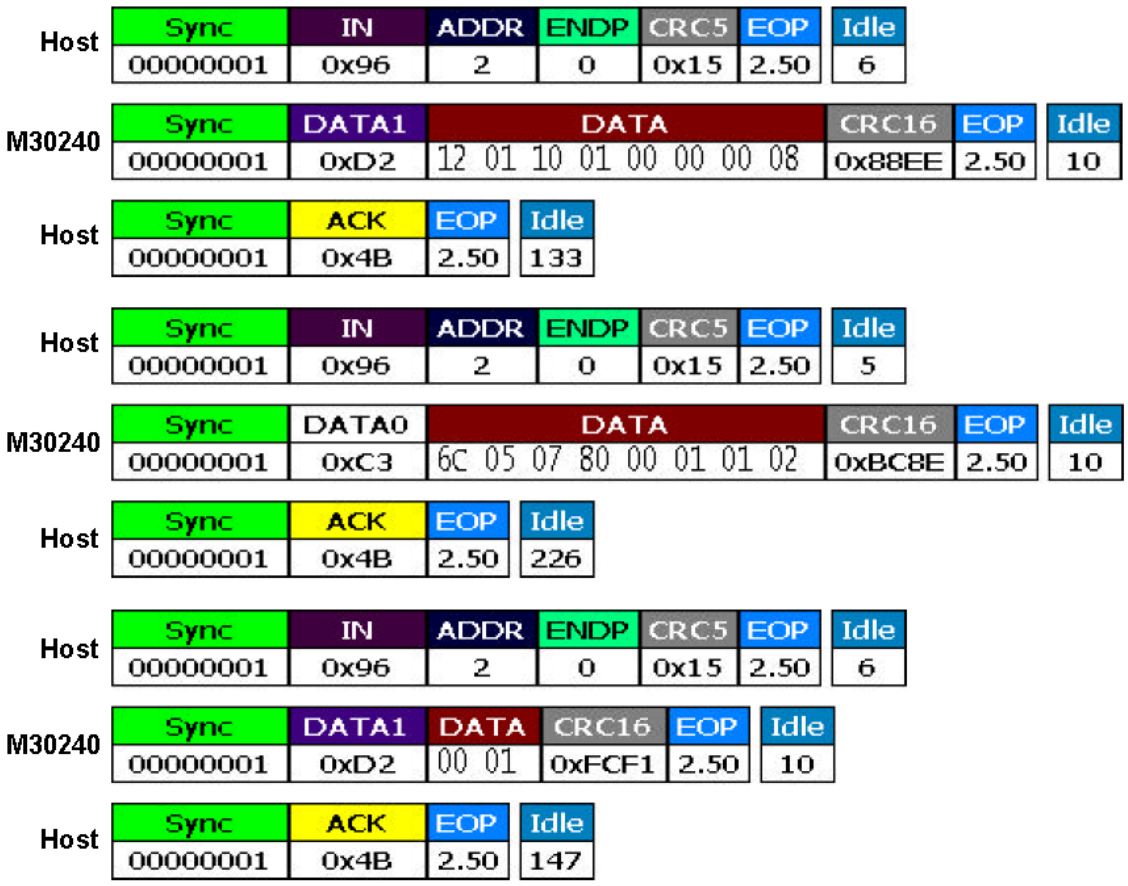


图8．M30240 设备发送18字节的设备描述符，每次8字节

包被拆分为8字节或少于8字节，在使用各种其它的USB命令时，象取设备描述符、字符串描述符、界面描述符等也一样需要拆分。这是因为开始时我们告诉USB主机控制器，包的最大尺寸是8字节。