What is USB Enumeration?

Enumeration is the process by which a USB device is attached to a system and is assigned a specific numerical address that will be used to access that particular device. It is also the time at which the USB host controller queries the device in order to decide what type of device it is in order to attempt to assign an appropriate driver for it.

Some of the basic commands issued by the host to the device are:

- Set Address – Instructs the device change it's current address settings
- Get Device Descriptor – Overall information about the device (manufacture, firmware version …)
- Get Configuration Descriptor – How the endpoints will be used
- Get Interface Descriptor – Various different interface that the device may use
- Get String Descriptor – Unicode strings for Manufacture and Product

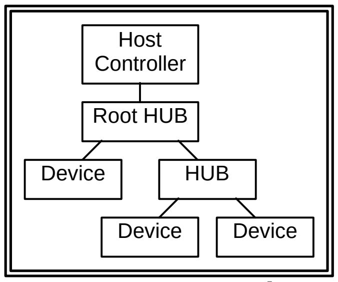
This process is a fundamental step for every USB device, fore without it, the device would never be able to be used by the OS.

What does enumeration look like?

I believe the easiest way to explain the USB enumeration process is to show it happing. The CamConnect demo firmware, for obvious reasons, contains code that will allow it to enumerate on any USB system. By using traces taken using a CATC USB Analyzer and snippets of code take from the CamConnect firmware source code, you should be able to following and understand the enumeration process.

Initially plugging in of the device:

All USB devices are plugged into a hub of some sort. When this is done, the hub detects whether the device is a full speed or low speed device. This is signified by the device pulling the D+ line to a 3.3v volt supply through a 1.5k pull-up for a full speed device, or the D- line for a low speed device.



USB Connection Topology

Once the hub has detected the connection of that new device, it will start passing Start Of Frame (SOF) packets produced by the host down to the device at 1ms intervals. The host controller will also start issuing setup packets to the device in order to enumerate the new device.

When a device is initially plugged in, it always uses the default device address 0 for communication. During the enumeration process, the host controller will assign a new numerical address for that device to use. Communication for the enumeration process always uses endpoint 0 on the device. These are considered to be Control Transfers. All USB Control transfers must use that device's endpoint 0.

After the host receives all the descriptors for the device, the OS will attempt to find an appropriate device driver to be associated with that new device.

Start of Frame:

The following is an example of a SOF packet sent by the host at 1ms intervals. This SOF packet is sent to every device on the bus so that they may all be synched up. Every USB packet sent over the bus begins with a Sync pattern in order to allow all devices to synchronize their transceivers. A CRC calculation for that packet is also included for most packet types. The M16C USB hardware automatically takes care of these details.

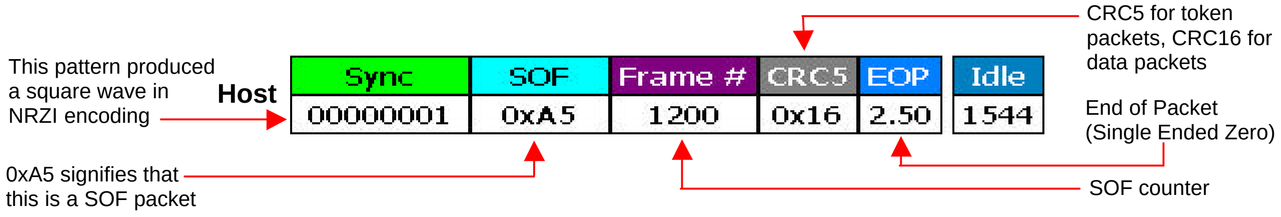

This pattern produced a square wave in NRZI encoding
Host
Sync
00000001
SOF
0xA5
Frame #
1200
CRC5
0x16
EOP
2.50
Idle
1544
CRC5 for token packets, CRC16 for data packets
End of Packet (Single Ended Zero)
0xA5 signifies that this is a SOF packet
SOF counter

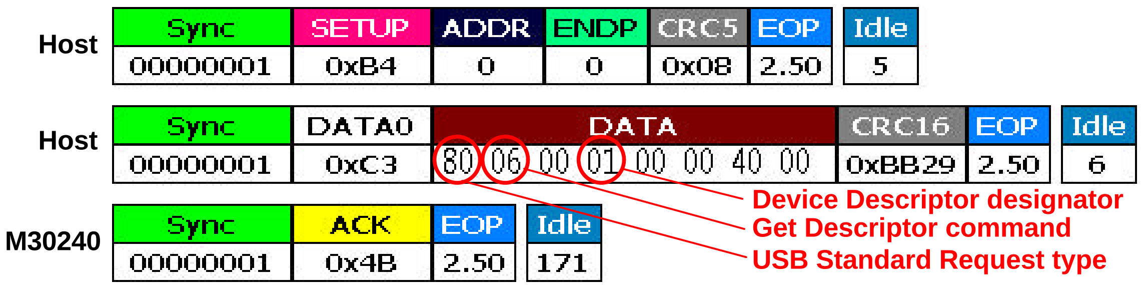
Figure 1. Start of Frame (SOF) Packet

Initial communication:
What first happens on the USB lines can be somewhat confusing to some new to USB. I myself was quite confused at what I was seeing until someone explained it to me. Depending on how the USB host controller was implemented, you might see bus traffic that you would normally assume to be incorrect, but in actuality, it is a standard and necessary process for a USB host controller.

So what is this mystery? More or less, the USB Host Controller first ask the device before it does anything else for it's Device Descriptor. The interesting part is that the device descriptor is 18-byte long, but the host could care less about that and may only want the first 8 bytes of it. After it receives those, it will not even ask for the rest of the data from the device. Further more, the host will issue a reset for that line after which it will then start to send commands for USB enumeration.

The reason the Host does this is because the Device Descriptor contains the maximum payload size that is allowed for an endpoint 0 Control Transfer. This value is contained in the 8th byte of the Device Descriptor that we must send back to the Host. So the host first queries the device with a GetDescriptor command in order to just get this one piece of information. Once the host has determined that number, it resets the USB lines and starts the enumeration process.

The packet traces below are showing that a setup token was the first thing given to our device on EP0 that instructed to return the Device Descriptor.



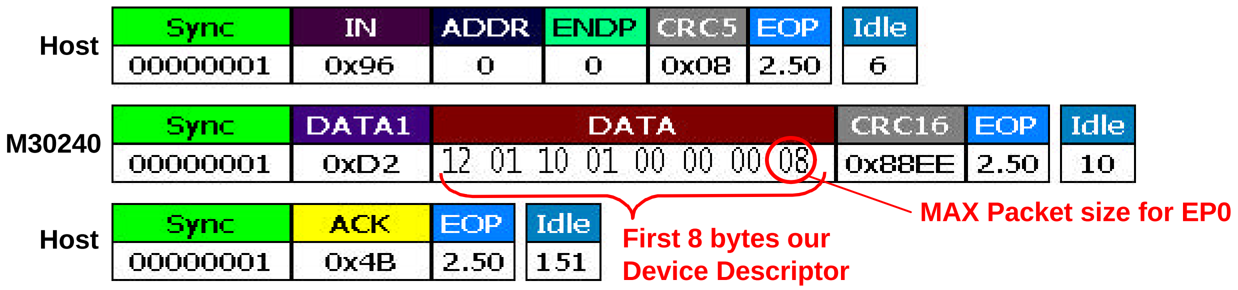
	Sync	SETUP	ADDR	ENDP	CRC5	EOP	Idle
Host	00000001	0xB4	0	0	0x08	2.50	5

	Sync	DATA0	DATA	CRC16	EOP	Idle
Host	00000001	0xC3	80 06 00 01 00 00 40 00	0xBB29	2.50	6

	Sync	ACK	EOP	Idle
M30240	00000001	0x4B	2.50	171

Figure 2. Initial Get Descriptor Request from Host

Below you will see that our device returned the first 8 byte of the 18-byte descriptor, but instead of the host issuing us another IN packet so that we could transfer another 8-byte data packet, it issues an OUT packet followed by a NULL data packet. We ACK this transition, then the host controller resets the USB lines.

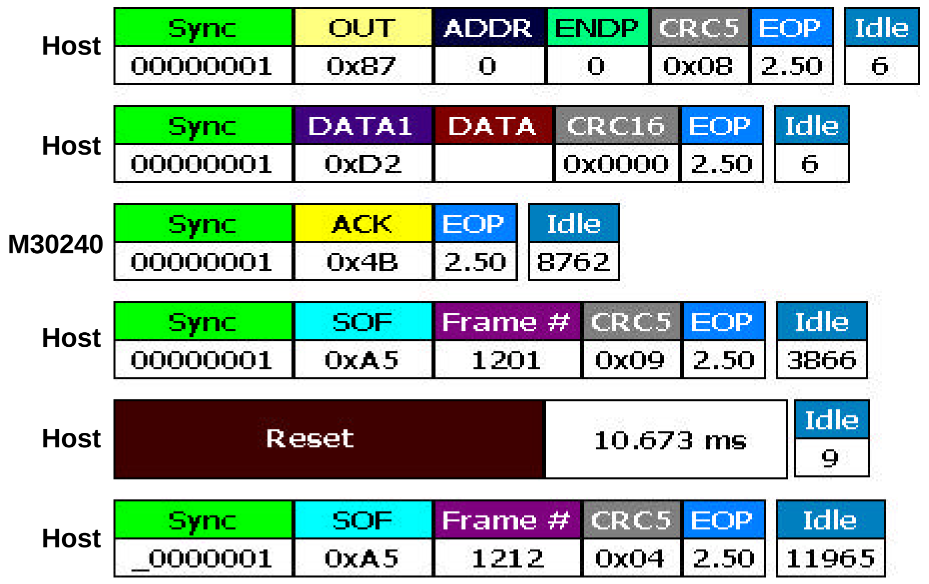


	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle
Host	00000001	0x96	0	0	0x08	2.50	6

	Sync	DATA1	DATA	CRC16	EOP	Idle
M30240	00000001	0xD2	12 01 10 01 00 00 00 08	0x88EE	2.50	10

	Sync	ACK	EOP	Idle
Host	00000001	0x4B	2.50	151

Figure 3. Our Device Answering the Initial Get Descriptor Request



Host

Sync	OUT	ADDR	ENDP	CRC5	EOP	Idle
00000001	0x87	0	0	0x08	2.50	6

Host

Sync	DATA1	DATA	CRC16	EOP	Idle
00000001	0xD2		0x0000	2.50	6

M30240

Sync	ACK	EOP	Idle
00000001	0x4B	2.50	8762

Host

Sync	SOF	Frame #	CRC5	EOP	Idle
00000001	0xA5	1201	0x09	2.50	3866

Host

Reset		Idle
	10.673 ms	9

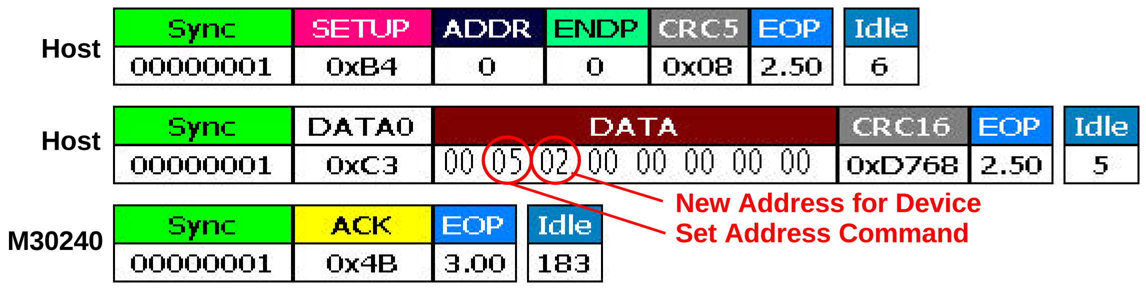
Host

Sync	SOF	Frame #	CRC5	EOP	Idle
_0000001	0xA5	1212	0x04	2.50	11965

Figure 4. Host Resetting USB lines to begin enumeration

Set Address:
For this enumeration process, the first command that was passed to the device was the Set Address command. As mentioned before, a new USB device on the bus temporarily uses a device address of 0 (zero) in order to provide a means of communication with the host. The host will then assign a specific numerical address for that device to use so that it will contain a unique identity on the USB bus.

Below you will see that a setup packet is sent to Device 0 , Endpoint 0, followed by 8 bytes of data that will be used to determine what type of setup packet is being sent, and what values need to be assigned. The M16C USB hardware contains a register that maintains the current device address. The register defaults to 0 after RESET, but can be written to at any time to change what address the USB hardware will respond to. The M16C USB hardware automatically sends an ACK back to the host saying that the data was received without error.



Host

Sync	SETUP	ADDR	ENDP	CRC5	EOP	Idle
00000001	0xB4	0	0	0x08	2.50	6

Host

Sync	DATA0	DATA	CRC16	EOP	Idle
00000001	0xC3	00 05 02 00 00 00 00 00	0xD768	2.50	5

M30240

Sync	ACK	EOP	Idle
00000001	0x4B	3.00	183

Figure 4. A Set Address command is sent to the device

Now lets take a look at how this packet is decoded in CamConnect's firmware.

The M16C's USB hardware will automatically accept and respond to device requests addressed to its current device address. Since the address register is set to 0 at the beginning of enumeration, our device will respond to all packet addressed to device 0.

Any time a USB packet is sent to a device address that matches our current device address, the M16C USB hardware will generate an interrupt (USBF– USB Function Interrupt). The interrupt handler can then query the USB registers to find out what caused the interrupt and respond accordingly.

Below is the USB Interrupt Service Routine for a USBF Interrupt.

```
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 */
       }
       . . . .
       . . . .
}
```

As you can see, we first save the Interrupt status registers, and then write the values back in order to clear any bits that were set. We then use this information to decide what endpoint caused the interrupt. Since this is a setup packet, it would be Endpoint 0. We then call the appropriate function to handle this, ParseEP0Packet(), which is shown below.

```
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
       }
}
```

Above you can see that the 8-byte data packet that was sent was read out of endpoint 0 into a variable structure. Note that every setup packet has the same 8-byte format in which data is sent. Once the data is read out, it can be analyzed. You can see that we look at bits 5 and 6 of the first byte, otherwise known as the bmRequestType in USB lingo, in order to determine which type of request is being made. All USB enumeration requests are made via Standard Requests.

The function ProcessStandardRequest() that is shown below is then call which will then further decode the packet data.

```
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");
	}
}
```

By using the second byte in the data packet sent by the USB Host, otherwise known as the bRequest, we can determine what type of setup command is being administered. All the possible USB standard requests as noted in the USB specification are listed above as well.

You will notice that from the packet diagram shown earlier, that it is a 5, which correlated to being a Set Address command. We then call the appropriate function to service this request which will be CmdSetAddress() as seen below.

```
void CmdSetAddress() {

	/* Load our new Device address */
	usba = EP0_Header.wValueLow;

	/* Set DATA_END and OUT_PKT_RDY bit for EP0 */
	ep0csr = 0x48;
}
```

At this point, we have now determined what type of setup packet request has been administered to us. Seeing that it is a Set Address command, all we need to do is instruct our USB hardware to start accepting data for a new device address and respond back to the host that we understood the request and completed the task.

The variable usba is actually a symbolic link to the M16C's USB device address register. By writing a new value to that register, the USB hardware will automatically start responding only to that new device address. You can see that this value is passed from the host in the lower byte of the wValue word (the 3rd byte in the 8-byte data stream). Cross-referencing that information to the setup packet in Figure 4, we can see that our new device address will be 2.

Finally we set the DATA_END and OUT_PKT_RDY bits for endpoint 0 in the M16C's USB registers which will cause the M16C USB hardware to send back a 0-length data packet (also call a NULL Data Packet) back to the host. This will signify that we have satisfied its Change Address request. This can be seen below.

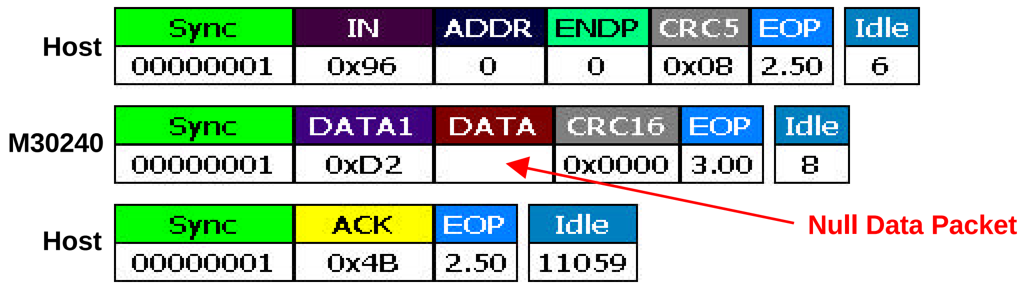


Figure 5. Device Acknowledgement of Change Address Request

The host sends an IN Packet Request to us (as we are still sitting at address 0) in order to receive conformation. As you can see, we respond correctly with a NULL data packet. The reason for the NULL data packet is that according to the USB spec, a device may NAK an IN token from the host as long as it wants. The host will simply keep sending IN tokens to that device until an answer is received. In this case, the IN token is being used by the host to say, "Are you ready to start accepting data at your new address?" The M16C USB hardware will automatically send NAK packets back for us until our firmware has completed that task and we have set the appropriate USB registers. So, the Null data packet is like saying, "Yes, now I am ready, you may continue."

An ACK is sent back from the host signifying that the response was received correctly.

Get Descriptor:

The rest of the enumeration process is similar to this. It is like a system of questions and commands from the USB Host Controller for the newly attached device to follow. When the host is satisfied that it has enough information to search for an appropriate driver, it will stop sending setup packets for Standard Requests. At this point, the device is considered enumerated.

The GetDescription is another important setup command in the enumeration process. Unlike the first GetDescriptor issued by the host at the very beginning, this time we are expected to pass the entire descriptor back to the host. The setup packets are shown below.

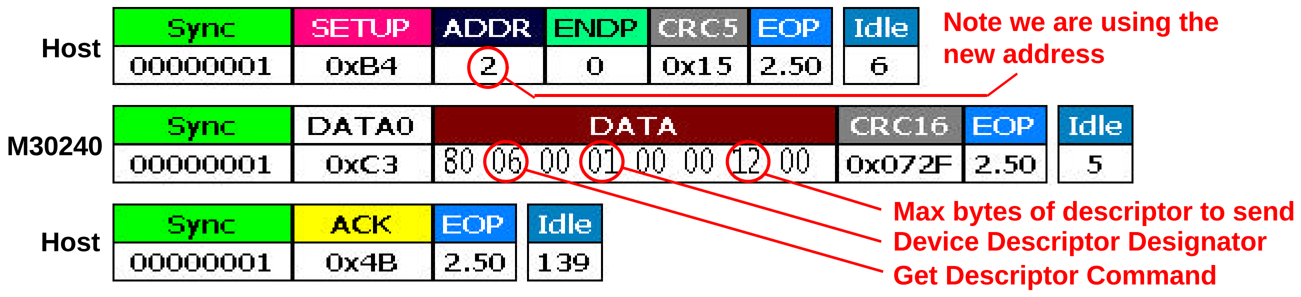


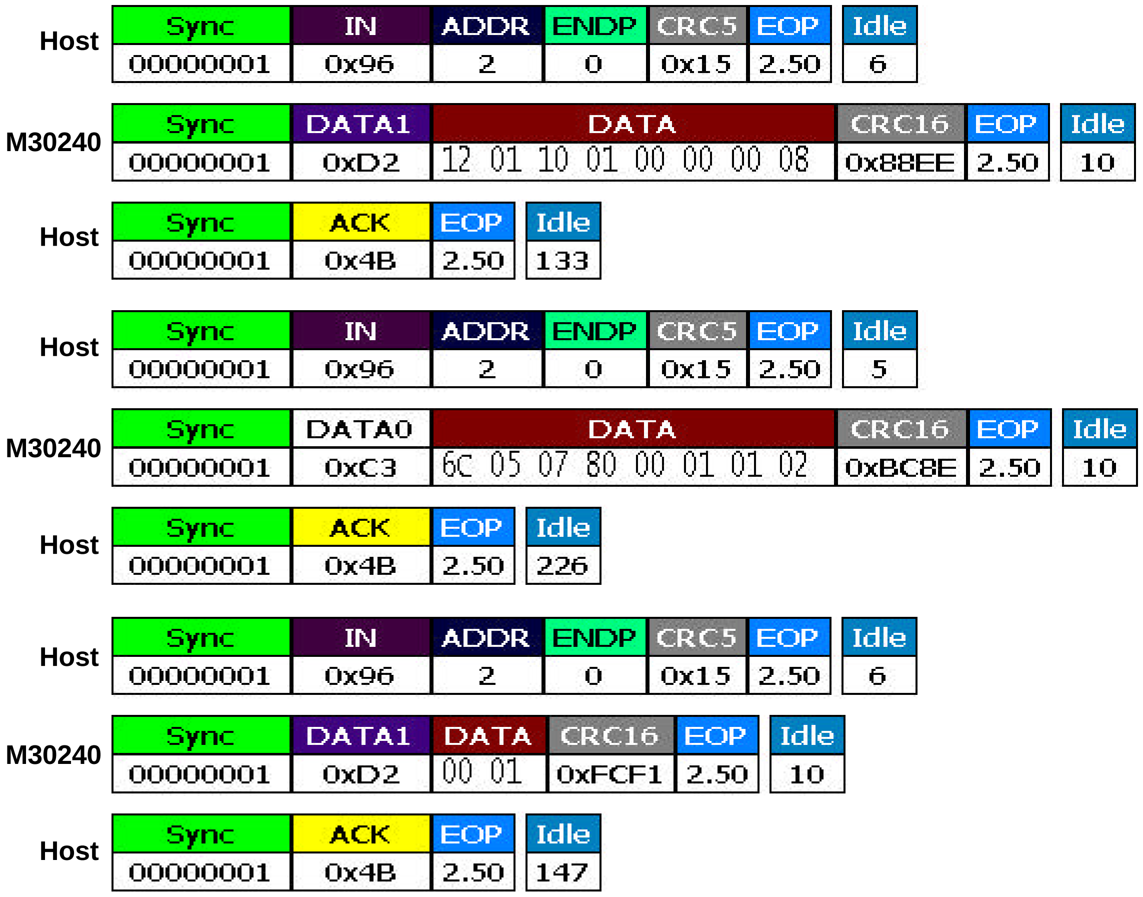
Figure 6. Get Device Descriptor Command from Host

Our firmware would then follow these steps…

```
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
```

In the CmdGetDescriptor() routine, we will then break up our 18-byte descriptor response into 8-byte data packets. Only when the host issues an IN token to us may we then transfer the data back up to the host in our firmware routine. We simply wait for the next IN packet to the host, fill up the endpoint 0 FIFO, then set the IN_PKT_RDY bit for EP0 in one of the M16C's USB registers, and the hardware then transfer it up to the host.

Shown below are the bus traces doing just that.



	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle
Host	00000001	0x96	2	0	0x15	2.50	6

	Sync	DATA1	DATA	CRC16	EOP	Idle
M30240	00000001	0xD2	12 01 10 01 00 00 00 08	0x88EE	2.50	10

	Sync	ACK	EOP	Idle
Host	00000001	0x4B	2.50	133

	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle
Host	00000001	0x96	2	0	0x15	2.50	5

	Sync	DATA0	DATA	CRC16	EOP	Idle
M30240	00000001	0xC3	6C 05 07 80 00 01 01 02	0xBC8E	2.50	10

	Sync	ACK	EOP	Idle
Host	00000001	0x4B	2.50	226

	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle
Host	00000001	0x96	2	0	0x15	2.50	6

	Sync	DATA1	DATA	CRC16	EOP	Idle
M30240	00000001	0xD2	00 01	0xFCF1	2.50	10

	Sync	ACK	EOP	Idle
Host	00000001	0x4B	2.50	147

Figure 7. The M30240 Device Sending the 18-byte Device Descriptor 8 bytes at a time

This same procedure of breaking up the packets into 8-byte or less payloads will be done for various other USB commands such as getting the Configuration Descriptor, String Descriptor, Interface Descriptors and so on. This is because we told the USB Host Controller at the beginning that are MAX Packet size would be 8 bytes.