TEXAS INSTRUMENTS

TUSB2136/TUSB3210/ TUSB5052 USB Firmware Programming Flow

8052 Embedded USB Microcontroller

User's Guide

February 2004

MSDS Bus Solutions

SLLU020A

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Contents

Figures

Tables

Chapter 1

Introduction

This chapter discusses the features of the 8052 embedded microcontroller.

Topic **Page**

1.1 8052 Embedded Microcontroller Features

The 8052 embedded microcontroller is a high-performance 8-bit microcontroller. It is binary/instruction-execute-time compatible with the industry standard 8052AH and 8752BH discrete devices. Several features are listed below and Table 1−1 illustrates the features of TUSB2136/3210/5052 devices in normal operation.

- 256-byte internal data memory
- Three 16-bit timer/counters
- Full-duplex serial port
- Power-down mode
- I^2C support
- Up to 48 MHz operation at 12 clocks per instruction cycle
- Supports SUSP and PUR pins

Table 1−1. Device Feature

Features/Device	TUSB2136	TUSB3210	TUSB5052
8052 clock	12/48 MHz	12/48 MHz	12/24 MHz
USB interrupt	INT3	INT3	INT3
Code memory size (bytes)	8K	8K	16K
Serial port (RS232)	1	1	3§
USB downstream port	2	0	5
Available I/O pins†	32	32	28‡
External interrupt pins	8	8	1
External interrupt source	P2[7:0], P3.3	P2[7:0], P3.3	Wake-up

† P0 to P3 are embedded in 8052 while all others are in memory-mapped register (MMR).
‡ Some are output or input only.
§ Including 8052 embedded serial port

Chapter 2

USB Transfers

This chapter provides details of the three types of supported USB transfers.

Topic **Page**

2.1 USB Transfers

Four types of USB transfer are: control, interrupt, bulk, and isochronous transfer. Control transfer allows the host to send USB requests to the device. Interrupt transfer allows the device to report its status back to the host in a specified period. Bulk transfer allows the host and the device to send/receive data when data arrival time is not important, while isochronous transfer ensures the data packet is delivered to the destination within a predefined period. Any USB device is capable of supporting at least a control transfer since standard USB requests use control transfers.

Only interrupt programming flow is discussed, even though polling programming flow is feasible, but not sufficient, to utilize microcontroller processing power.

Application firmware sets up control endpoint 0 before it connects to USB. The following registers are interrelated to control endpoint 0.

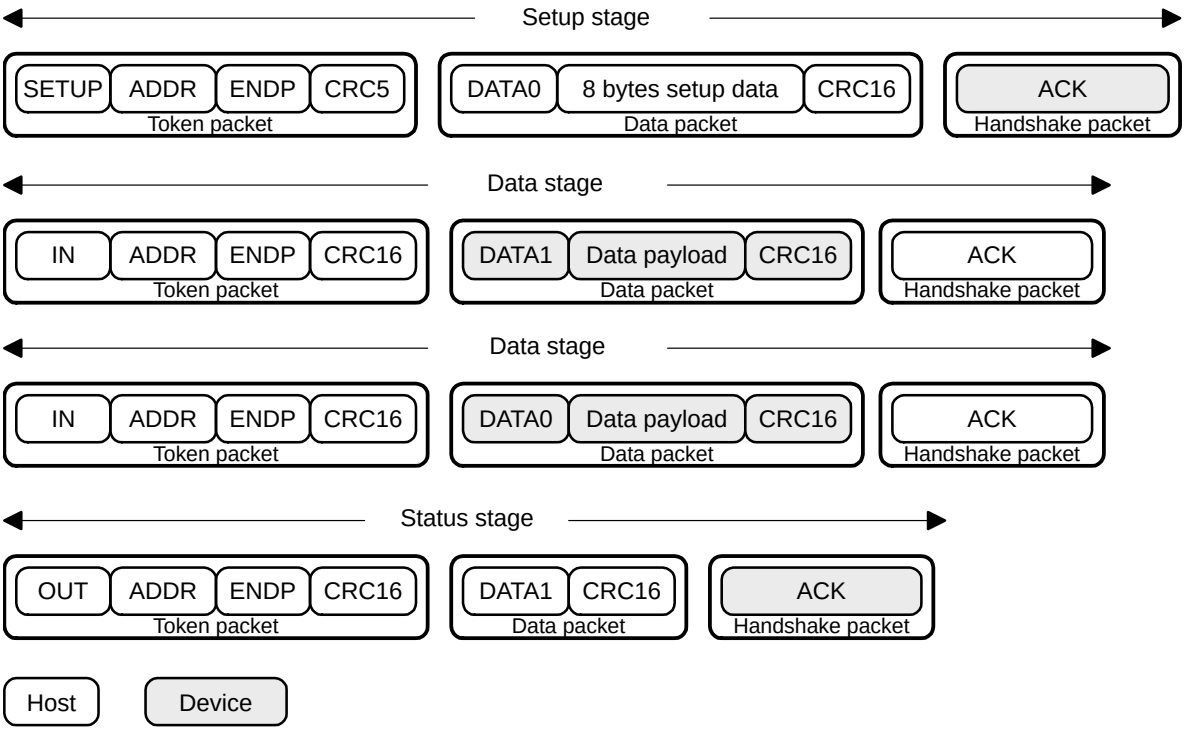
- **FUNADR: Function Address Register**
  This register resets to zero before the device connects to the bus.
- **IEPCNFG_0: Input Endpoint-0 Configuration Register**
  USBIE and UBME are set to 1 to enable the endpoint and corresponding interrupt. The STALL bit is clear in normal operation. If the STALL bit is set, the device stalls the IN transaction. Therefore, the device does not send any data back to the host.
- **IEPBCNT_0: Input Endpoint-0 Byte Count Register**
  Default value is zero with the NAK bit set. If application firmware sends 0x05 bytes of data back to the host, the firmware sets the register to 0x05 with the NAK bit clear. The maximum is 8 bytes per IN transaction.
- **OEPCNFG_0: Output Endpoint-0 Configuration Register**
  USBIE and UBME are set to 1 in order to enable the endpoint and corresponding interrupt. The STALL bit is clear in normal operation. If STALL is set, the device stalls the OUT transaction. Therefore, the device does not receive any data from the host.
- **OEPBCNT_0: Output Endpoint-0 Byte Count Register**
  Default value is zero with the NAK bit clear. If the device receives 0x05 bytes from the host, it sets the byte count to 0x05 and clears the NAK. If the interrupt is enabled, it generates an interrupt to notify application firmware that an OUT transaction is completed and data is in the buffer.
- **USBMSK: USB Interrupt Mask Register**
  STPOW and SETUP are set to 1. If there is a setup packet coming from the host, the device generates a SETUP interrupt to the application firmware and sets the SETUP bit in the USBSTA register. If more than one setup packet is received before the application firmware is able to respond to the command, the STPOW bit is set.
- **USBCTL: USB Control Register**
  CONT is set to 1 when the device is ready.
- **EA: Global Interrupt Enable Bit**
  This is a global interrupt in the 8052 microcontroller. This bit is set if any interrupt is used.

- **EX0: Internal INT 3**
  This bit is set to 1. Application firmware uses this interrupt to handle all internal interrupts including a USB interrupt.

2.2 Control Read Transfer

Control read transfer has three stages: setup, data and status stage. Each stage represents a single USB transaction. As shown in Figure 2−1, the host sends out a setup packet to the device. The device returns an ACK to show that it received the setup packet. Once the host receives an ACK from the device, it then sends an IN token to get data from the device. If the device is ready, it sends data back to the host (the first data in the data stage is always DATA1). The host sends an ACK showing it received the data packet. If the device is not ready or still processing the setup request, NAK runs until the data is ready. Once the host receives enough data from the device, it sends an OUT token indicating it is ready to finish (or terminate, if all the data is not yet sent back to the host) the current transfer. This is a zero-length out packet. The device returns an ACK to finish or terminate the current transfer.

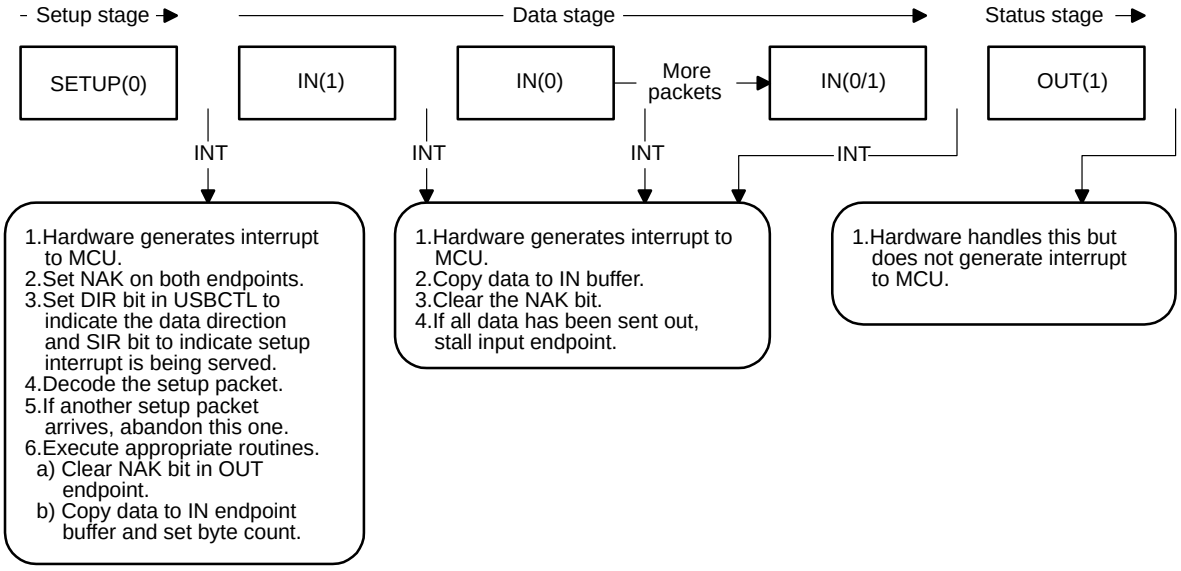
Figure 2−1. Control Read Transfer



An interrupt is generated at the end of each transaction as shown in Figure 2−1. Interrupt 0x03 is captured in the application firmware. Once the routine takes control from the hardware, it disables the global interrupt (EA = 0) and then processes the interrupt vector. Once it knows the interrupt source, it clears the source before it notifies the respective interrupt handling routine. After the service routine is finished, it enables the global interrupt (EA = 1) then releases control to the interrupted routine. Some real-time systems require that global interrupt always be enabled. In this case, disable global interrupt is removed. However, interrupt routines check for other setup-packet arrivals

before processing the current one. If there is another setup packet and the current one is still being processed, application firmware abandons the current one and processes the next setup packet.

Figure 2−2. Interrupts in Control Read Transfer



In some cases, early termination takes place if the host does not get any more data from the device. Therefore, interrupt routine handing input/output endpoint keeps tracking if the current stage is the data or status stage. If the transfer reaches the status stage, firmware stalls the IN endpoint, so any further IN tokens become stalled. A new setup packet arrival forces internal logic to clear the STALL bit on both endpoints.

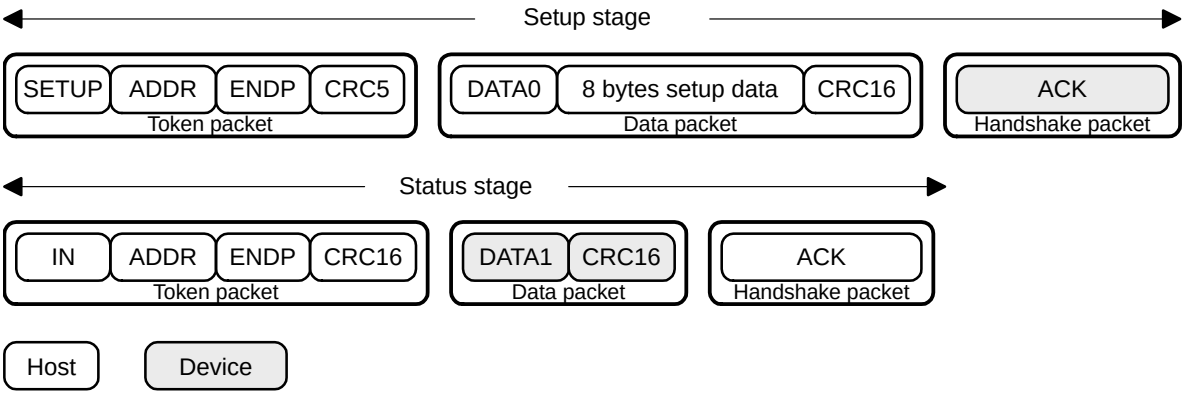
Additional registers related to this transfer are:

- **VECINT: Vector Interrupt Register**
  A 0x32 value represents a SETUP packet received. A 0x30 represents a STPOW packet received.
- **USBCTL: USB Control Register**
  The DIR bit in the USBCTL register is set to 1 for control read transfer.
- **USBSTA: USB Status Register**
  The SETUP/STPOW bit is set when setup(s) is/are received. Application firmware writes a 1 to clear this bit.
- **IEPCNFG_0: Input Endpoint-0 Configuration Register**
  The STALL bit in the IEPCNFG register is set to 1 at the end of data and status stages. This prevents application firmware from receiving an irregular request from the host.
- **OEPCNFG_0: Output Endpoint-0 Configuration Register**
  The STALL bit in the OEPCNFG register is set to 1 at the end of data and status stages. This prevents application firmware from receiving an irregular request from the host.

2.3 Control Write Transfer Without Data

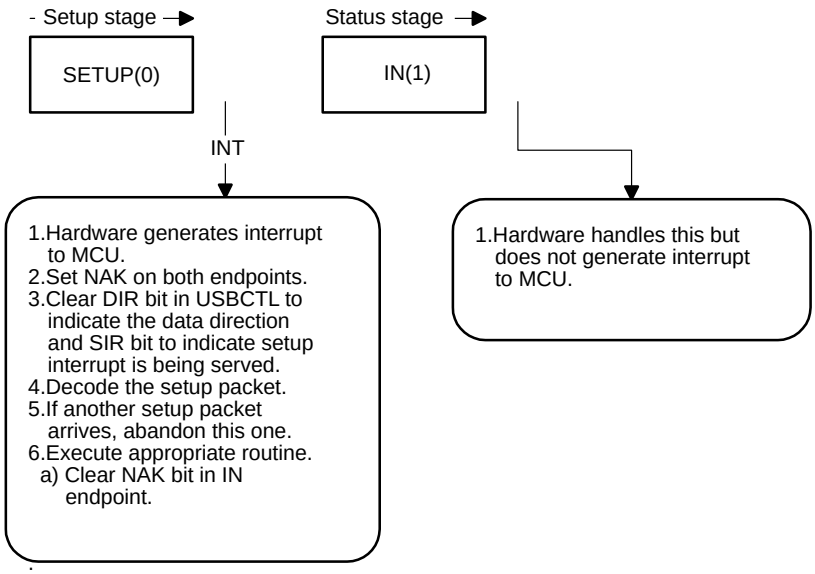
The control write transfer without data has only two stages: setup and status. In this type of transfer, application firmware processes the request and clears NAK bit for the status stage.

Figure 2−3. Control Write Transfer Without Data



As shown in Figure 2−4, application firmware retains a note in the setup stage. At the end of status stage, the application firmware processes the request.

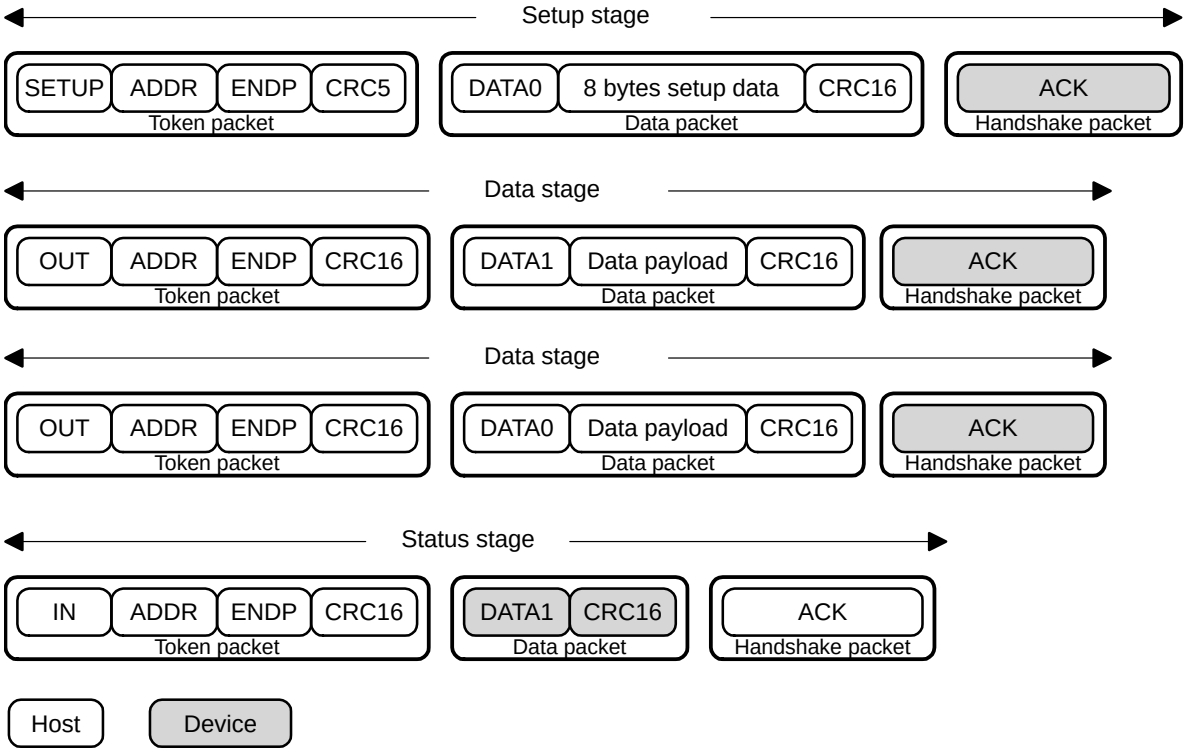
Figure 2−4. Interrupt in Control Write Transfer Without Data



2.4 Control Write Transfer With Data

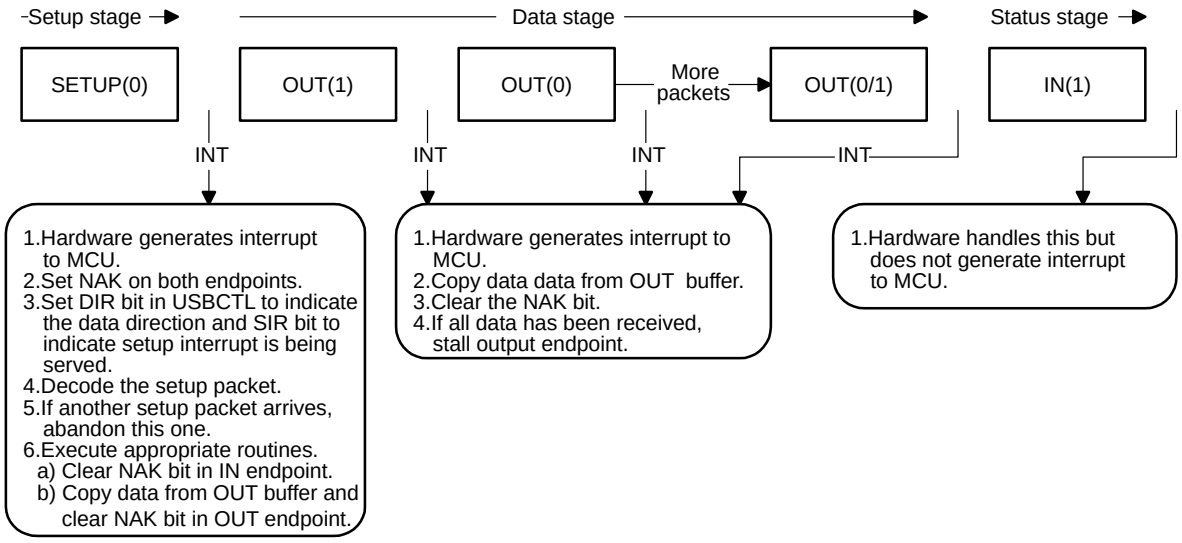
Control write transfer with data has the following three stages: setup, data, and status. As shown in Figure 2−5, the host sends a setup packet along with a data packet to the device. The device returns an ACK showing it received the setup packet. Once the host receives an ACK from the device, it transmits an OUT token and data to the device. If the device is ready to receive data, it returns an ACK. If the device is not ready or still processing the setup request, it continues to NAK until it is ready to receive data. Once the host sends enough data to the device, it transmits an IN token indicating it is set to finish (terminate if all the data has not been transmitted to the device) the current transfer. The device returns a zero-length packet, concluding the transfer.

Figure 2−5. Control Write Transfer With Data



As shown in Figure 2−6, application firmware clears both NAKs in the IN/OUT endpoints for the data and status stages. The host is capable of terminating the current transfer by sending an IN token. If this happens, application firmware should terminate the current transfer and prepare for the next setup packet.

Figure 2−6. Interrupt in Control Write Transfer With Data



2.5 Irregular Control Transfers

As shown in Figure 2−7 and Figure 2−8, application firmware processes the first setup packet while the second setup packet is being received. If the second setup packet is received before the application firmware can respond to the setup packet, the application firmware abandons the first one and processes the second one.

As shown in Figure 2−9 and Figure 2−10, application firmware clears the data and status stages once it receives another setup in the previous transfer. This is a simple process for the application firmware, since it discards the data and serves a new setup request.

Figure 2−7. Back-to-Back Setup

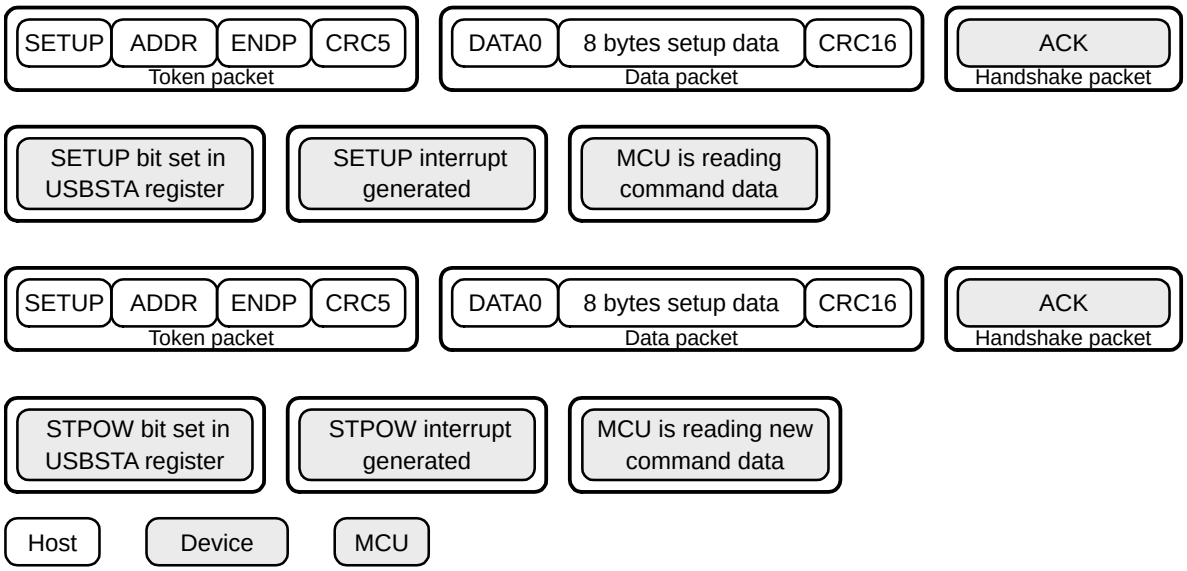


Figure 2−8. Interrupt Back-to-Back Setup

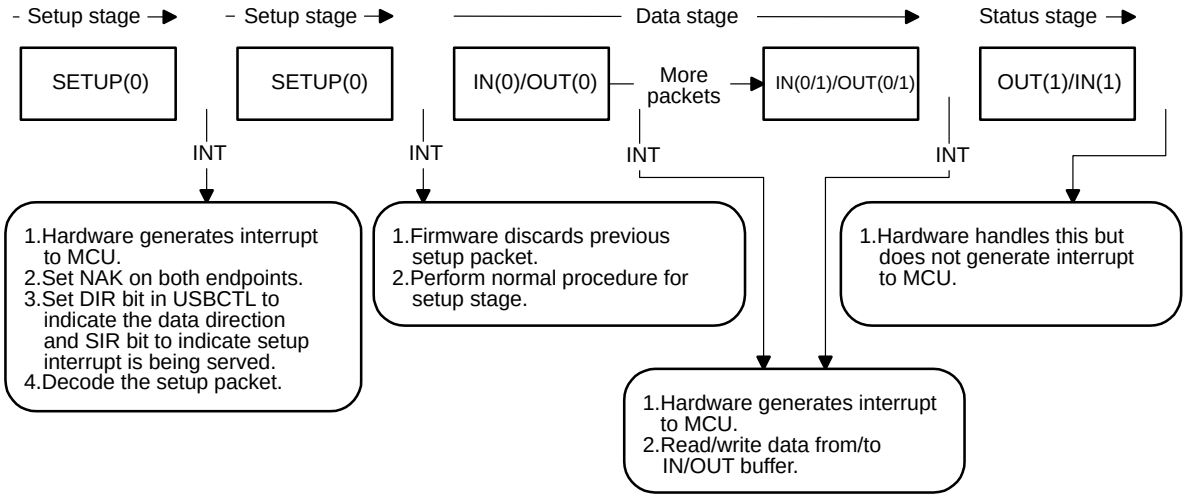


Figure 2−9. Incomplete Transfer

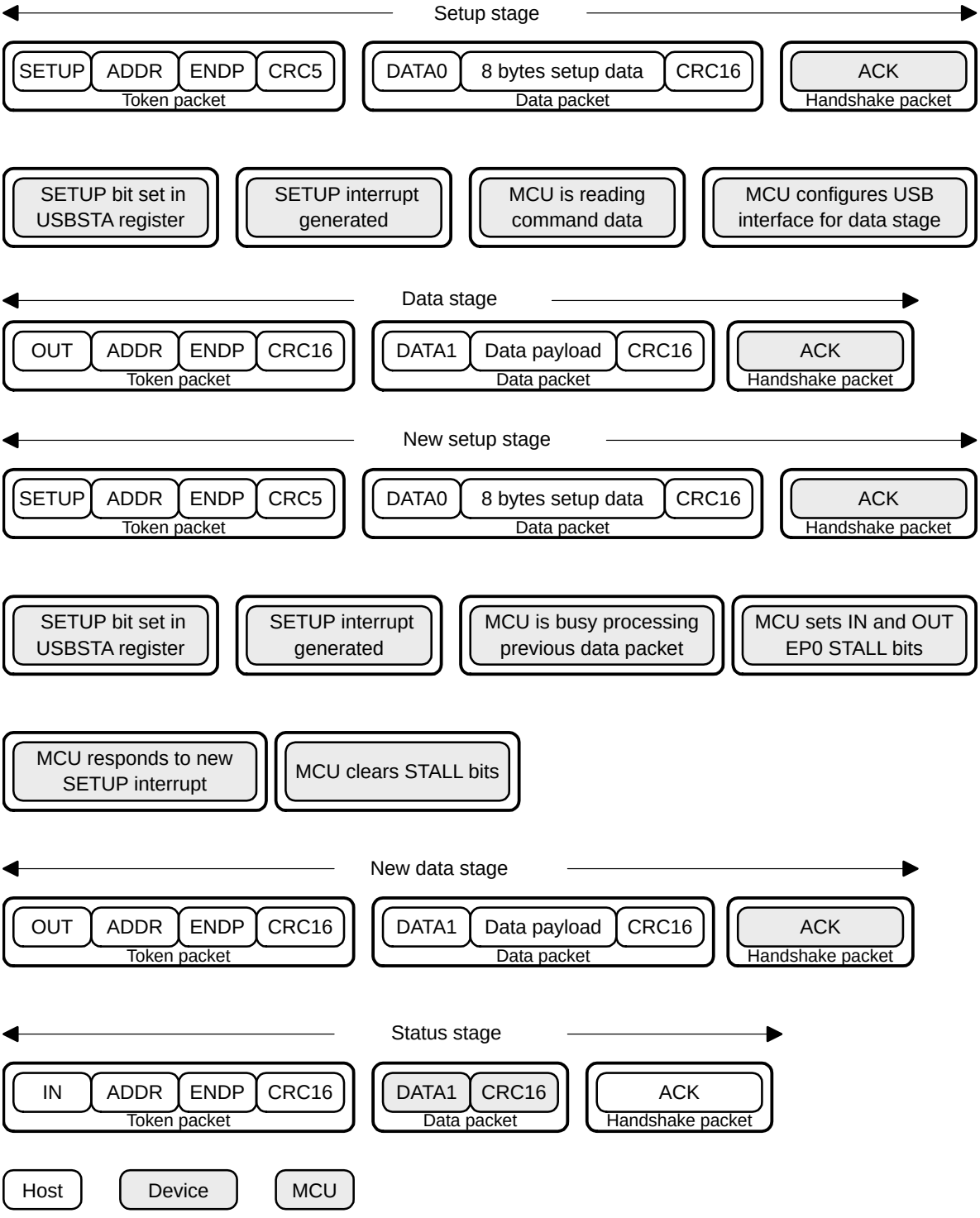
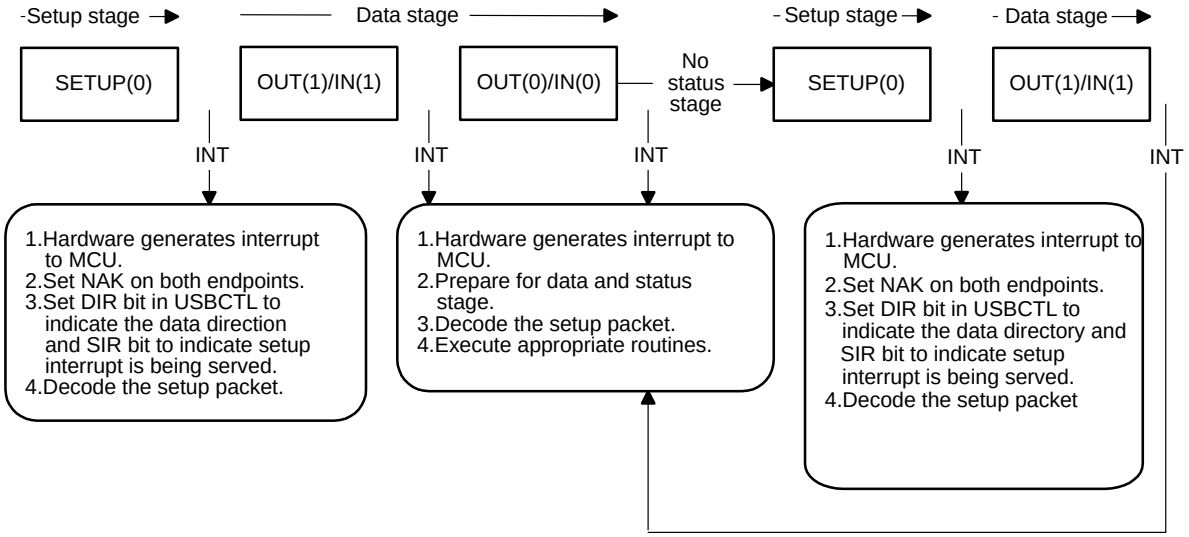


Figure 2−10. Interrupt Incomplete Transfer



2.6 Bulk IN Transfer

As shown in Figure 2−11 and Figure 2−12, after the hardware receives an IN token on IEP1, it either transmits a NAK or data to the host depending on whether the hardware has data or not. If the hardware transmits data to the host, it generates an interrupt to the MCU at the end of data transmission. Therefore, the application firmware senses data has been sent and it updates the IN buffer with new data.

Figure 2−13 shows dual buffer bulk in transfer. The toggle bit in the IEPCNF_1 register indicates which buffer the application firmware accesses.

Figure 2−11. Interrupt/NAK in Bulk IN Transfer

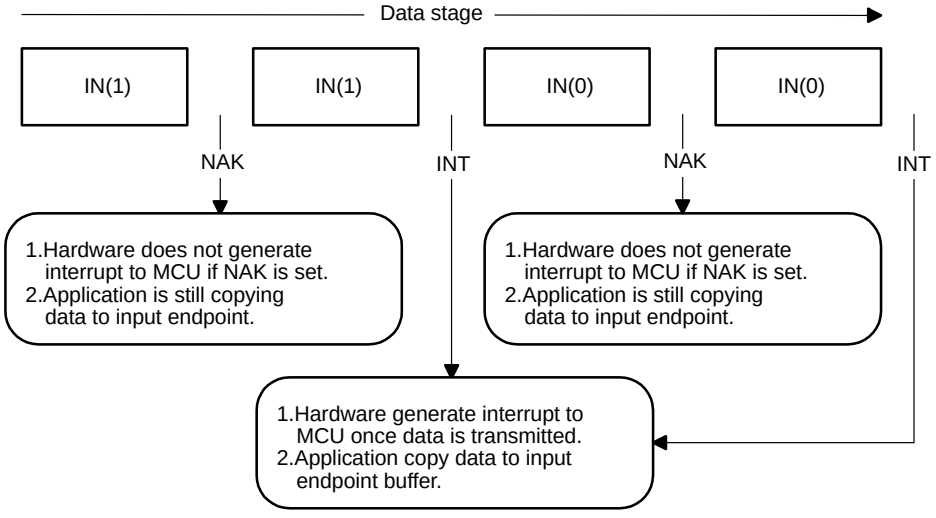


Figure 2−12. Single Buffer Bulk IN Transfer

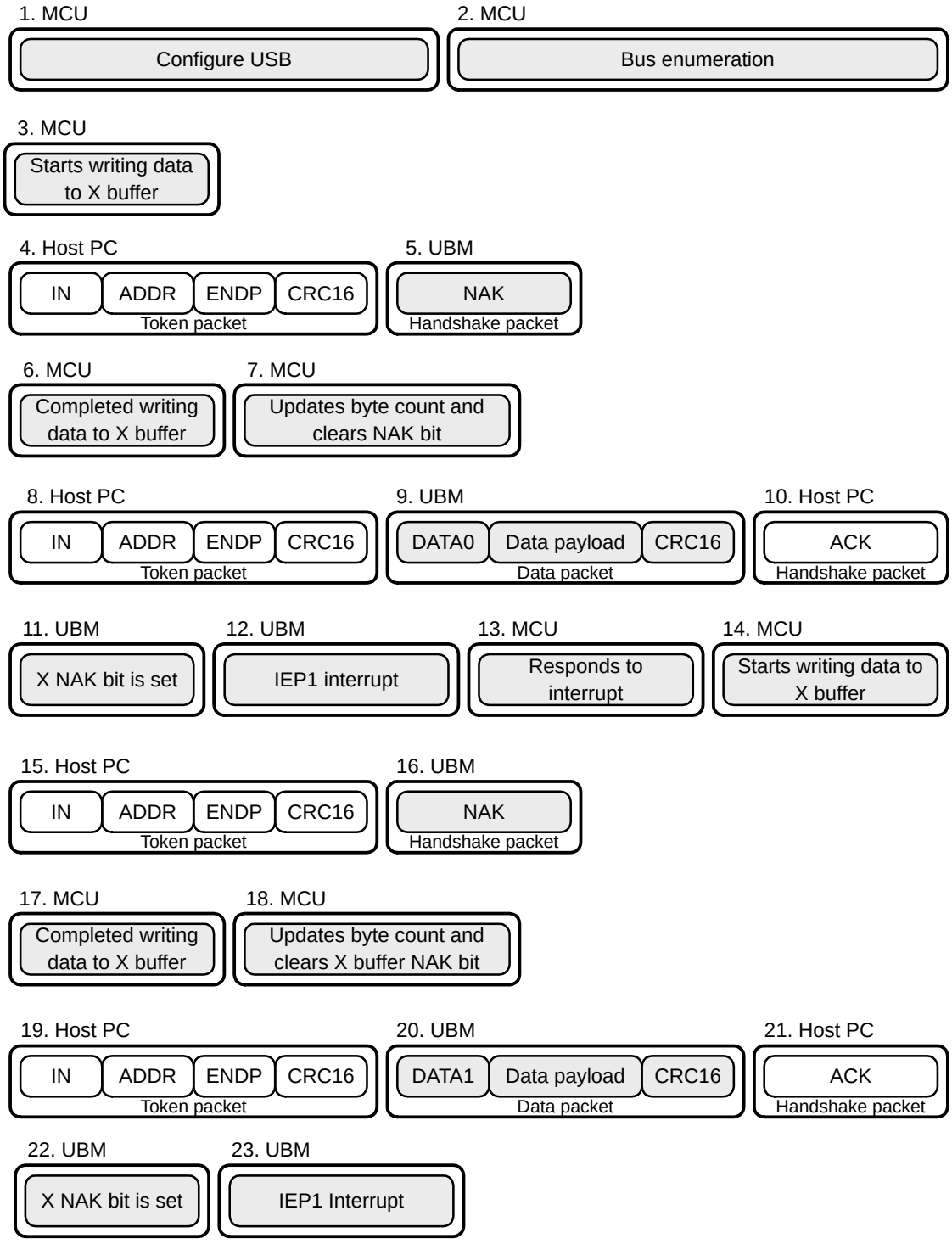
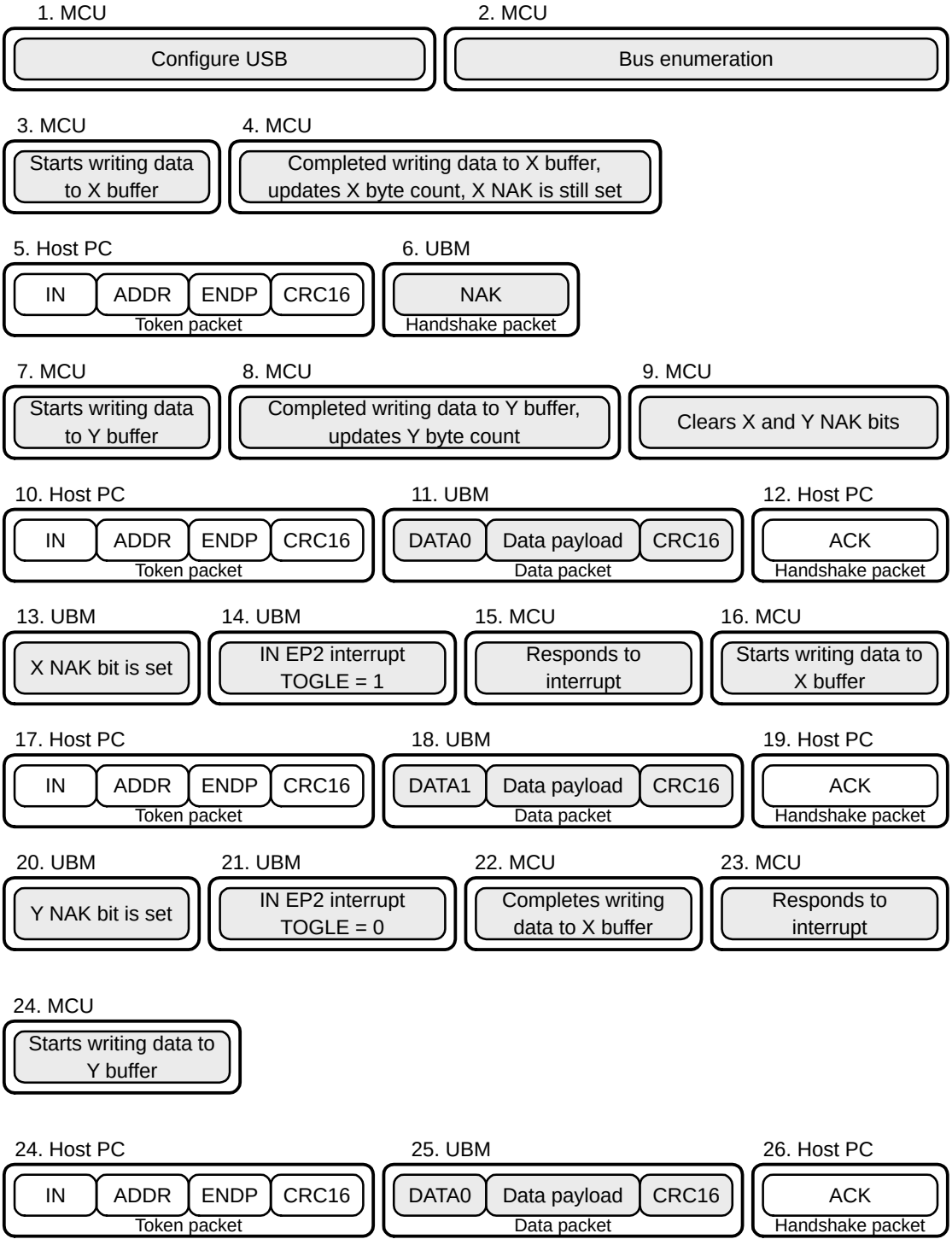


Figure 2−13. Double Buffer Bulk IN Transfer



2.7 Bulk OUT Transfer

As shown in Figure 2−14 and Figure 2−15, after the hardware receives an OUT token on IEP2, it either transmits a NAK, if the NAK bit is not cleared, or generates an interrupt to the application firmware.

Figure 2−16 shows dual buffer bulk out transfer. The toggle bit in the IEPCNF_2 register indicates which buffer the application firmware accesses.

Figure 2−14. Interrupt/NAK in Single Buffer Bulk OUT Transfer

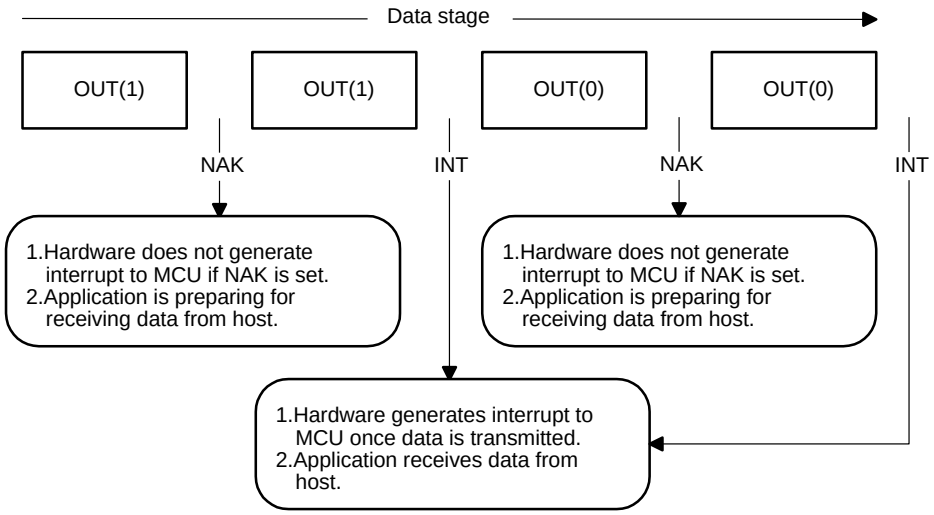


Figure 2−15. Single Buffer Bulk OUT Transfer

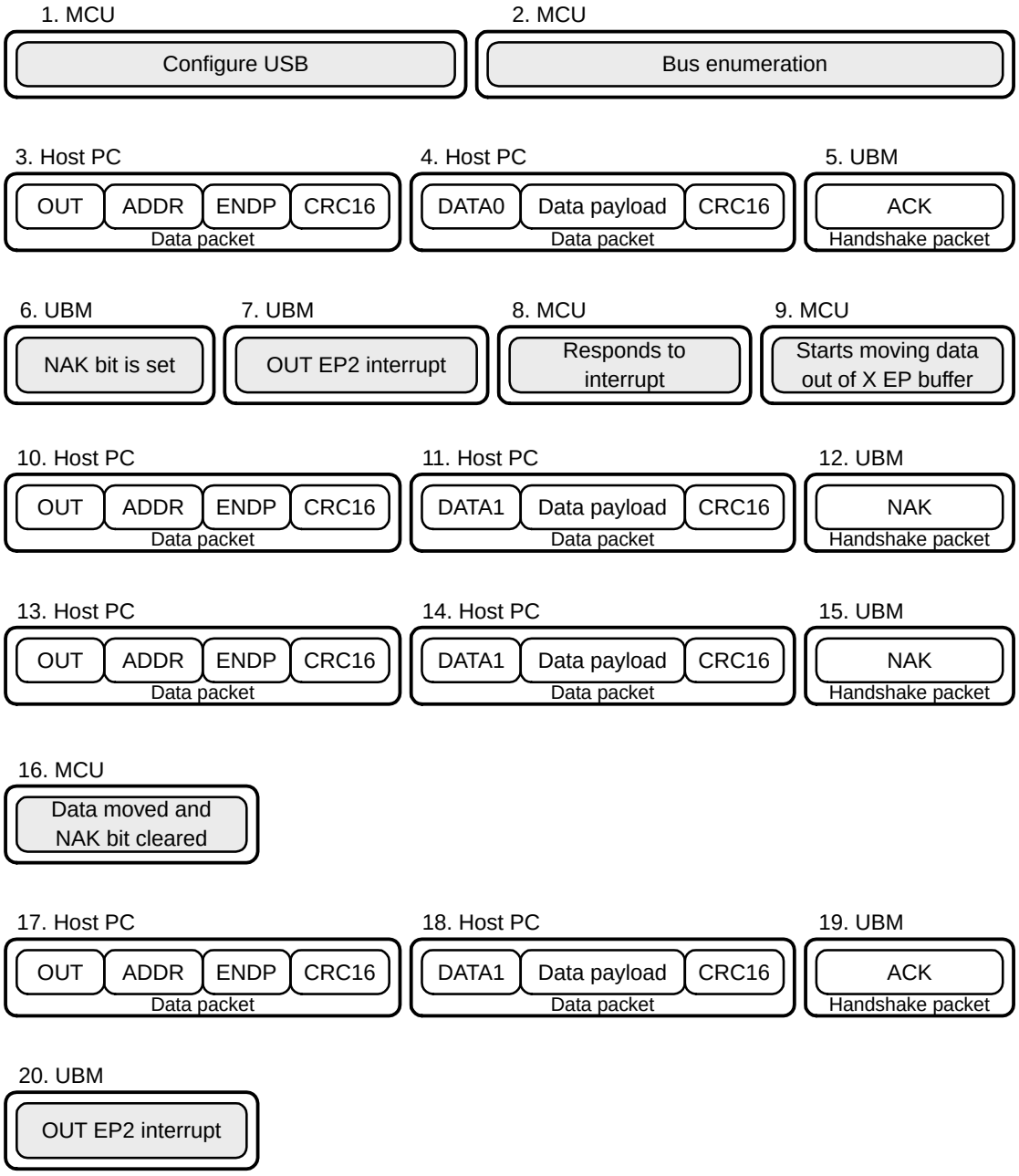
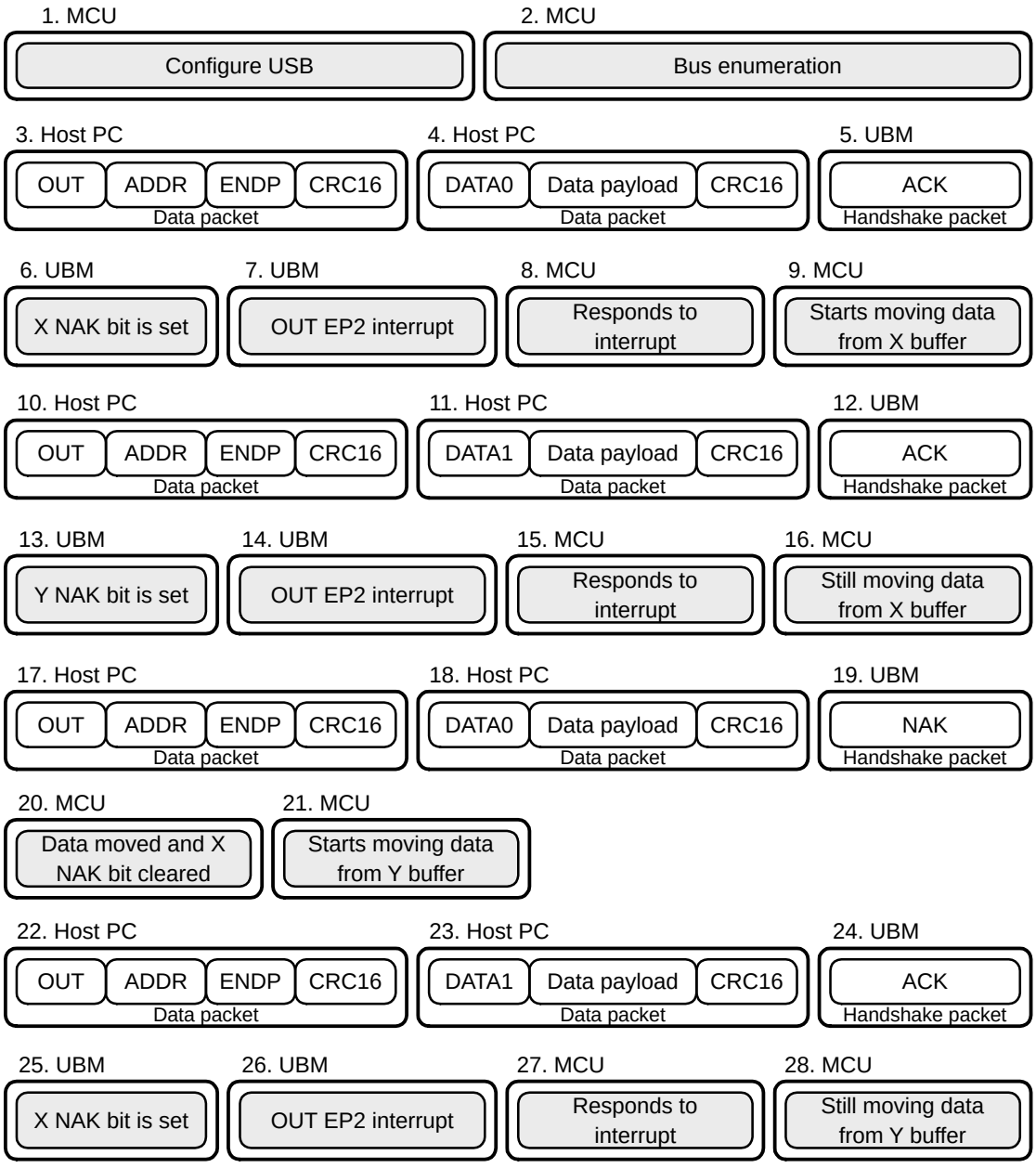


Figure 2−16. Double Buffer Bulk OUT Transfer



2.8 Interrupt IN Transfer

Interrupt IN transfer is similar to bulk IN transfer except the interrupt transfer ensures reception of an IN token from the host within a period of time, specified in the endpoint descriptor. The interrupt handling routine is the same as the bulk interrupt handling routine. Application firmware uses this type of interrupt transfer to provide status of the device or application related information. The wait time between two consecutive interrupts is also specified in the endpoint descriptor in 1 ms increments.

Chapter 3

Application Firmware Programming Examples

The step-by-step examples show how the application sets up the USB environment and responds to various USB transfers.

Topic **Page**

3.1 System Initialization Routine

Application firmware does not rely fully on the default register value after power on or warm boot since the application can not determine the previous status. Registers like bUSBCTL, bFUNADR are especially critical. They must be set to the proper value prior to USB enumeration. If the application firmware prefers a hardware reset (MCU reset is tied to USB reset) instead of the firmware control, the FRSTE bit in bUSBCTL register is not cleared. The following example shows that the device is disconnected from the USB and application firmware handles the USB reset as well as I^2C and MCU speed.

```
VOID SystemInitialization(VOID)
{
  EA  = DISABLE;
  // disconnect from USB, firmware handles USB reset
  bUSBCTL = 0x00;
  // set i2c speed
  i2cSetBusSpeed(I2C_400KHZ);
  // set to 48Mhz
  bMCNFG = MCNFG_48MHZ;
  // Enable the USB−specific Interrupts; SETUP, RESET and STPOW
  bUSBMSK = USBMSK_STPOW | USBMSK_SETUP | USBMSK_RSTR | USBMSK_RESR | USBMSK_FSPR;
  // Enable global and USB interrupt
  EA  = ENABLE;
  EX0 = ENABLE;

  // Enable Embedded Function
  // Enable port 3 for TUSB2136/TUSB3210
  // Enable port 6 for TUSB5052
  // connect to USB
  bUSBCTL = USB_CONT;
}
```

3.2 USB Reset Routine

Because application firmware handles USB reset, all endpoints and internal USB-related variables are initialized in the usbReset() routine. If a hardware reset is preferred, the routine can be changed to a usbInitialization() routine.

```
VOID usbReset(VOID)
{
  bFUNADR           = 0x00;      // no device address
                                 // this register needs to be
                                 // cleared before enumeration
  wBytesRemainingOnIEP0 = NO_MORE_DATA;
  wBytesRemainingOnOEP0 = NO_MORE_DATA;
  bStatusAction         = STATUS_ACTION_NOTHING;
  pbIEP0Buffer          = (PBYTE)0x0000;
  pbOEP0Buffer          = (PBYTE)0x0000;
  bConfigurationNumber  = 0x00;  // device unconfigured
  bInterfaceNumber        = 0x00;
                                 // enable endpoint 0 interrupt
  tEndPoint0DescriptorBlock.bIEPCNFG = EPCNF_USBIE | EPCNF_UBME;
  tEndPoint0DescriptorBlock.bOEPCNFG = EPCNF_USBIE | EPCNF_UBME;
                                 // enable all necessary endpoints here...
  tOutputEndPointDescriptorBlock[0].bEPCNF = EPCNF_USBIE | EPCNF_UBME |
     EPCNF_DBUF;
  tOutputEndPointDescriptorBlock[0].bEPBBAX = (BYTE)(OEP1_X_BUFFER_ADDRESS >> 3 &
    0x00ff);
  tOutputEndPointDescriptorBlock[0].bEPBCTX = 0x0000;
  tOutputEndPointDescriptorBlock[0].bEPBBAY = (BYTE)(OEP1_Y_BUFFER_ADDRESS >> 3 &
    0x00ff);
  tOutputEndPointDescriptorBlock[0].bEPBCTY  = 0x0000;
  tOutputEndPointDescriptorBlock[0].bEPSIZXY = EP_MAX_PACKET_SIZE;
}
```

3.3 USB ISR Routine

With the device connected to the USB, the first event is USB RESET. This instructs the MCU to execute a usbISR() routine. Generally, any USB event triggers an interrupt.

It is possible to disable the interrupt globally during the interrupt service routine. The following example clears the global interrupt first. The interrupt source and bVECINT register must be cleared prior to or after executing appropriate subroutines. This allows the hardware to detect any new USB events. Be aware that some events have the same interrupt source as the bVECINT register. If this is the case, the application firmware only needs to clear the bVECINT register. This is especially important when the application firmware needs more time to process the USB request.

```
interrupt [0x03] VOID usbISR(VOID)
{
    EA = DISABLE;        // Disable any further interrupts
    switch (bVECINT){   // Identify Interrupt ID
        case VECINT_OUTPUT_ENDPOINT0:
                bVECINT  = 0x00;
                OEP0InterruptHandler();
                break;
        case VECINT_INPUT_ENDPOINT0:
                bVECINT  = 0x00;
                IEP0InterruptHandler();
                break;
        case VECINT_OUTPUT_ENDPOINT1:
                bVECINT  = 0x00;
                OEP1InterruptHandler();
                break;
        case VECINT_STPOW_PACKET_RECEIVED:
         // hardware clears STALL in both data end
            points once valid setup packet is
         // received
         // NAK data endpoints
         tEndPoint0DescriptorBlock.bIEPBCNT = EPBCNT_NAK;
         tEndPoint0DescriptorBlock.bOEPBCNT = EPBCNT_NAK;
         // clear setup packet flag
         bUSBSTA = USBSTA_STPOW;
         bVECINT  = 0x00;
         SetupPacketInterruptHandler();
         break;
        case VECINT_SETUP_PACKET_RECEIVED:
         // hardware clears STALL in both data endpoints
         once valid setup packet is
         // received
         // NAK data endpoints
         tEndPoint0DescriptorBlock.bIEPBCNT  = EPBCNT_NAK;
         tEndPoint0DescriptorBlock.bOEPBCNT  = EPBCNT_NAK;
         SetupPacketInterruptHandler();
         break;
        case VECINT_RSTR_INTERRUPT:
         // clear reset flag
         bUSBSTA = USBSTA_RSTR;
         bVECINT  = 0x00;
         UsbReset();
         break;
        default:break;       // unknown interrupt ID
    }
    EA      = ENABLE;       // Enable the interrupts again
}
```

3.4 USB Setup Packet Routine

If the device receives a setup packet from the host, it generates an interrupt and assigns VECINT_SETUP_PACKET_RECEIVED to the bVECINT register. As shown in the previous program, the SetupPacketInterrupt Handler() routine was requested. During this routine the NAK sets both endpoints in case the host attempts to get data from the IN buffer with obsolete data. It also sets the direction of the current transfer and action in the status stage before requesting the usbDecodeAndProcessUsbRequest() routine.

```
//---------------------------------------------------------------------------
VOID SetupPacketInterruptHandler(VOID)
{
    // copy the MSB of bmRequestType to DIR bit of USBCTL
    if((tSetupPacket.bmRequestType & USB_REQ_TYPE_INPUT) == USB_REQ_TYPE_INPUT)
        bUSBCTL |= USBCTL_DIR;
    else bUSBCTL &= ~USBCTL_DIR;
    // SET SIR bit
    bUSBCTL |= USBCTL_SIR;
    bStatusAction         = STATUS_ACTION_NOTHING;
    usbDecodeAndProcessUsbRequest();
}
```

The usbDecodeAndProcessUsbRequest() routine compares the setup packet buffer with a predefined setup packet list. If any item in the list matches the setup packet in the buffer, it executes the routine assigned in the list. If there is not a match in the list and the setup packet is not a vendor or class specific request, the routine delays the setup packet due to an unsupported request.

```
VOID usbDecodeAndProcessUsbRequest(VOID)
{
  BYTE bMask,bResult,bTemp;
  BYTE *pbUsbRequestList;
  pbUsbRequestList = (PBYTE)&tUsbRequestList[0];
  while(1){
    bResult = 0x00;
    bMask   = 0x80;
    // compare all the fields first
    for(bTemp = 0; bTemp < 8; bTemp++){
      if(*(pbEP0_SETUP_ADDRESS+bTemp) == *(pbUsbRequestList+bTemp)) bResult |=
bMask;
      bMask = bMask >> 1;
      }
     // now we have the result
     if((*(pbUsbRequestList+bTemp) & bResult) == *(pbUsbRequestList+bTemp)) break;
     // advance to next one
```

```
    pbUsbRequestList += sizeof(tDEVICE_REQUEST_COMPARE);
  }
  // check if any more setup packet(s) is in
  if(bUSBSTA & (USBSTA_SETUP | USBSTA_STPOW) != 0x00) return;
  // now we found the match and jump to the function accordingly.
  ((ptDEVICE_REQUEST_COMPARE)pbUsbRequestList)->pUsbFunction();
}
```

3.5 USB Data Endpoint 0 Routine

If the device receives a data packet from the host, it generates an interrupt and assigns VECINT_INPUT_ENDPOINT0 to the bVECINT register. This same rule applies to send a data packet to the host. Firmware gets an interrupt after data is sent to the host.

```
VOID IEP0InterruptHandler(VOID)
{
  tEndPoint0DescriptorBlock.bOEPBCNT = 0x00;     // will be set by the hardware
  if(bStatusAction == STATUS_ACTION_DATA_IN) usbSendNextPacketOnIEP0();
  else tEndPoint0DescriptorBlock.bIEPCNFG |= EPCNF_STALL; // no more data
}
//---------------------------------------------------------------------------
VOID OEP0InterruptHandler(VOID)
{
  tEndPoint0DescriptorBlock.bIEPBCNT = 0x00;     // will be set by the hardware
  if(bStatusAction == STATUS_ACTION_DATA_OUT) usbReceiveNextPacketOnOEP0();
  else tEndPoint0DescriptorBlock.bOEPCNFG |= EPCNF_STALL; // no more data
}
```

3.6 USB Data Endpoint 1 Routine

For the OUT endpoint, the application firmware receives data then clears the NAK if more data is required from the host. For the IN endpoint, the application firmware copies data to the IN endpoint buffer, sets buffer size, and clears the NAK bit. The device sends the data back to the host after receiving an IN token from the host. When the device finishes sending the data, it generates an interrupt. The application firmware then either sends more data back to the host or does nothing if data is not available.