Application Note

AN2294/D
Rev. 1.1, 07/2003

MC68SZ328 USB Configuration Setup

By Ed Kroitor

Contents

1 Introduction

Due to the advancement of computers and computer peripherals, older systems of peripheral communications have been replaced with new and faster technologies. Older technologies such as the RS-232 have changed little since their introduction. Even though this communication interface served as a cornerstone of serial communication, modern applications demanded higher speed data transfers coupled with the ability to be hot swappable. The creation of the USB (Universal System Bus) specification provides a high speed method of data communications with true plug-and-play connectivity. The DragonBall family of microprocessors has taken advantage of this advanced technology by introducing a USB module within the Super VZ (MC68SZ328) DragonBall microprocessor. This application note provides the information to configure the essential registers used to establish communication via USB between a PC host and the DragonBall MC68SZ328 device.

1.1 USB Overview

The Universal System Bus module that is currently in the MC68SZ328 processor can only be used for full speed mode (12 Mbps).

1.1.1 Endpoints and Types of Transfer

The DragonBall MC68SZ328 provides five physical FIFO registers (endpoints) that can be used for communication with a host. These endpoints are used for the different types of transfer that can be accomplished with the MC68SZ328 USB module. The data transfers supported are control, interrupt, and bulk. The MC68SZ328 USB does not support isochronous transfers.

Control endpoints are specifically used to inform the host or device about the data transfer and the required setup information between host and client. Because endpoint zero represents the control transfer buffer, it must be used for all data transfers. Endpoints one through four are used for bulk or interrupt transfers and they can be configured by setting the desired bits in the USB module registers.

1.1.2 Pipes

Pipes are abstract links between the host and the device. These links are not physical, instead, they are software relationships. It is important to understand pipes relative to the types of data exchange and how these exchanges take place between systems (host and device). There are two types of pipes, stream (used in all transfers except control transfers) and message pipe (used in all control transfers).

1.1.3 Descriptors

Descriptors are structures of data sent by the device to the host for information purposes. With this data, the host obtains all the required information necessary to connect to the peripheral device. This data is used every time an enumeration occurs.

2 Hardware Interface

The MC68SZ328 processor has 3 main hardware interfaces: bus, transceiver, and USB crystal. This section describes these interfaces.

2.1 Bus and Power Management

The bus is responsible for transferring the data between the host and the USB module. The USB bus is comprised of four lines D+, D-, Vbus, and ground. The USB module can operate only as a self-powered device with the power supplied by the DragonBall processor's power pins, as the internal USB module does not have separate power pins.

2.2 Transceiver

The following MC68SZ328 output pins are responsible for the interface with the USB transceiver:

- USBD_VPO
- USBD_SUSPND
- USBD_VMO
- USBD_ROEB
- USBD_VM
- USBD_VP
- USBD_RCV
- USBD_AFE

Figure 1 on page 3, shows the connections between the DragonBall processor and transceiver ISP1105W using the USB pin outputs. Note that the SPEED signal is driven high on this diagram. This is necessary for running at full speed mode.

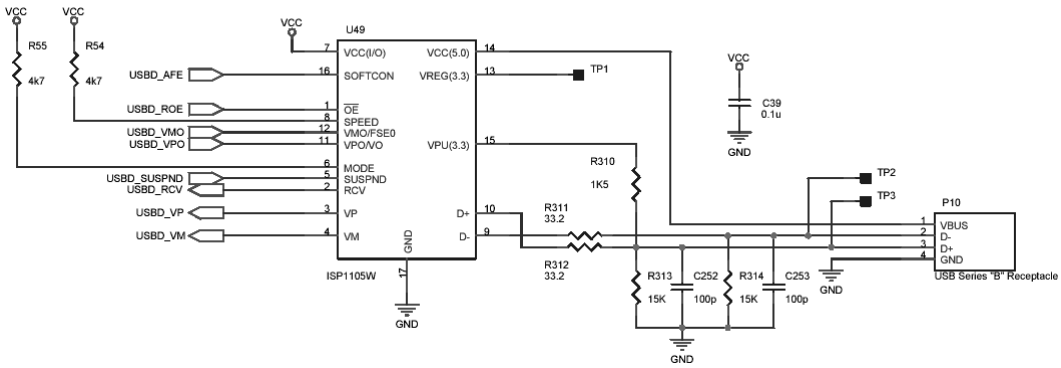

Figure 1. Connection From MC68SZ328 to the Transceiver Using USB Outputs

Components R313, C252, R314 and C253 are not installed normally, they serve the purpose of fine tuning the circuit.

2.3 USB Crystal Configuration

The USB uses its own stand-alone crystal that is isolated from the MCU PLL. Its function is to provide the necessary clock signal to run the USB (48 MHz). Both the MCU PLL and the USB PLL clock frequencies can be programmed using Equation 1 on page 5. The values for UMFI, UMFN, UMFD, and UPDF are programmable using the UPFSR0 and UPFSR1 registers. Detailed information about these registers can be found in the MC68SZ328 Reference Manual (Chapter 5). Figure 2 illustrates the crystal configuration hardware for a 32.768 kHz crystal, and Figure 3 on page 4 illustrates the crystal configuration hardware for the 16 MHz crystal.

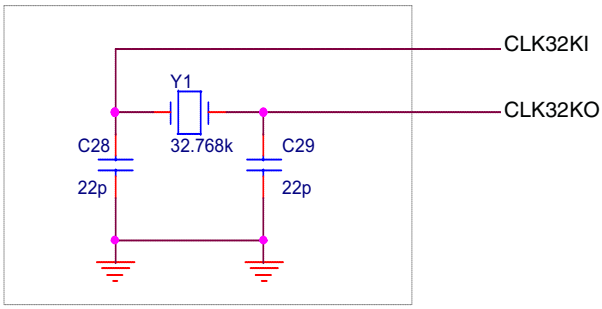

Figure 2. 32.768 kHz Crystal Configuration Hardware

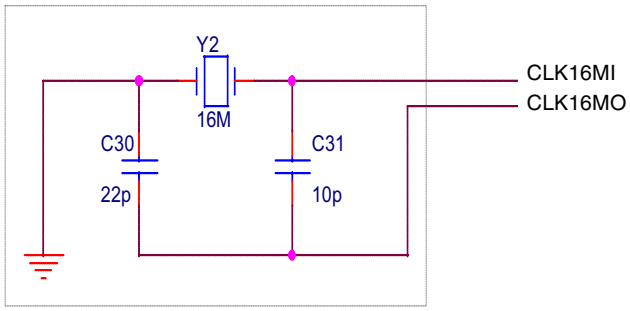

Figure 3. 16 MHz Crystal Configuration Hardware

3 Software Interface

This section provides the information to initialize the USB module with the host, program the USB Device Core (UDC) registers, and receive and transmit data. Example code and instructions to program the registers are provided.

3.1 Initialization

To start the communication process with the host, the PLL registers within the MC68SZ328 are used to program the 48 MHZ clock signal required by the USB module.

3.1.1 USB PLL Initialization

The USB PLL output frequency is controlled by the two USB frequency select registers shown in Figure 4 and Figure 5.

UPFSR1 USBPLL Frequency Select Register 1 **Addr 0x(FF)FFF20A**

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		UPDF[3:0]					UMFD									
TYPE		rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	0x0001															

Figure 4. USBPLL Frequency Select Register 1

UPFSR0 USBPLL Frequency Select Register 0 **Addr 0x(FF)FFF208**

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		UMFI				UBRMO	UMFN									
TYPE		rw	rw	rw	rw	1	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
RESET	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1
	0x0001															

Figure 5. USBPLL Frequency Select Register 0

As these registers show, the values programmed in them control the output frequency for the USB module. By using the Equation 1, the desired output frequency can be derived.

$$F_{usbpll} = 2 \times F_{usbpllin} \times \frac{UMFI + UMFN \div (UMFD)}{UPDF} \qquad \textit{Eqn. 1}$$

Where:

F_{usbpll} = Output frequency of USBPLL

$F_{usbpllin}$ = Output frequency of OSC16M or OSC32K + Premultiplier (USB crystal oscillator)

UMFI = Integer of the USBPLL multiplication factor

UMFN = Numerator of the USBPLL multiplication factor

UMFD = Denominator of the USBPLL multiplication factor

UMFN = Pre-division factor of the USBPLL

The corresponding input values of the UMFD to the UPFSR1 register are shown in Table 1 and Table 2 on page 5.

Table 1. Corresponding UMFD to the UPFSR1 Register Input Values

UMFD [9:0]	UMFD
0	1
1	2
2	3
...	...
0x3FD	0x3FE
0x3FE	0x3FF

Table 2. Corresponding UPDF to the UPFSR1 Register Input Values

UPDF[3:0]	UPDF
0	1

Table 2. Corresponding UPDF to the UPFSR1 Register Input Values (Continued)

UPDF[3:0]	UPDF
1	2
2	3
...	...
0xE	15
0xF	16

3.1.2 Clock Configuration

As shown in Figure 2 and Figure 3, there are two different clocks that can be used to drive the USB module; the 32.768 kHz and the 16 MHz crystals. Clock source selection is controlled by bit 15 in the CSCR register (see Figure 6). If the 16 MHz clock is selected, no premultiplier is required. If the 32.768 kHz crystal is selected, the signal comes from the output of the premultiplier.

The clock sources control register (CSCR) is used to select the clock source and control the divide ratios for the USB module. This register is shown in Figure 6.

CSCR Clock Sources Control Register **Addr 0x(FF)FFF20C**

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	USBSEL	PLLBYPB	USBCDIV			DMACDIV							OSC16EN	CLKOSEL		
TYPE		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
RESET	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1
	0x0001															

Figure 6. Clock Sources Control Register

The PLL registers work in two different modes according to the hardware settings programmed. The following examples show the configuration for a module to run the PLL with the 32.768 kHz crystal or using the 16 MHz crystal.

Using Equation 1 on page 5:

To produce an output of 48 MHz, use the values shown in Table 3 and use a USB Divider = 4.

Table 3. Programming Values

Variable	32.768 kHz	16 MHz
UMFI	5	6
UMFN	361	0
UMFD[9:0]	0	0
UPDF[3:0]	499	0

The results are 192 MHz. However, because USBCDIV / 4 (CSCR bit 13-11) the USB_CLK output frequency will be 48 MHz. Refer to the Code Example 1 on page 8 for more information on programming the USBPLL registers.

System Configuration for the 32.768 kHz Crystal

1. Set the value of CSCR register to 0x4C04 using the following:
   - — Consider the premultiplier output (512 *32.768 KHz)
   - — Enable USBPLL
   - — Set USBCLK / 4
   - — Set MCUPLL_CLK /1
   - — Disable the internal clock
   - — Enable the USB clock
2. Set the values of UMFN, UMFI, UPDF, and UMFD in the two frequency select registers:

   UPFSR0—0x2969

   UPFSR1—0x01F3

   PLLCR—0x6400
3. Set the value of CSCR after the previous registers are initialized to equal 0x4C03.

   The system is now setup to use the system clock.

System Configuration for the 16 MHz Crystal

1. Set the value of the CSCR register to 0xCC0C by enabling the following:
   - — 16 MHz external clock
   - — USBPLL
   - — Internal clock
   - — USB clock
2. Set the values of UMFN, UMFI, UPDF, and UMFD in the two frequency select registers:

   UPFSR0—0x3400

   UPFSR1—0x0000

   PLLCR—0x6400
3. CSCR remains the same because the system clock has already been enabled.

Code Example 1. Programming the USBPLL Registers

```
void init_USBPLL(void)
{
/* using 32.768 KHz crystal. */
              reg_CSCR=     0x4C04;
              reg_UPFSR0=   0x2969;
              reg_UPFSR1=   0x01f3;
              reg_PLLCR=    0x6400;
              reg_CSCR=     0x4c03;       }
void init_USBPLL(void)
{
/* using 16 MHz crystal. */
              int i;
              reg_CSCR =    0xCC0C;
              reg_PLLCR =   0x2400;
              reg_UPFSR0 = 0x3400;
              reg_UPFSR1 = 0x0000;
              reg_PLLCR =   0x6400;       }
```

3.2 Programming the USB Device Core (UDC)

The USB Device Core implements most of the USB protocol in hardware. As shown in Figure 7, the UDC is the front-end device for communication with the USB transceiver and the peripheral bus.

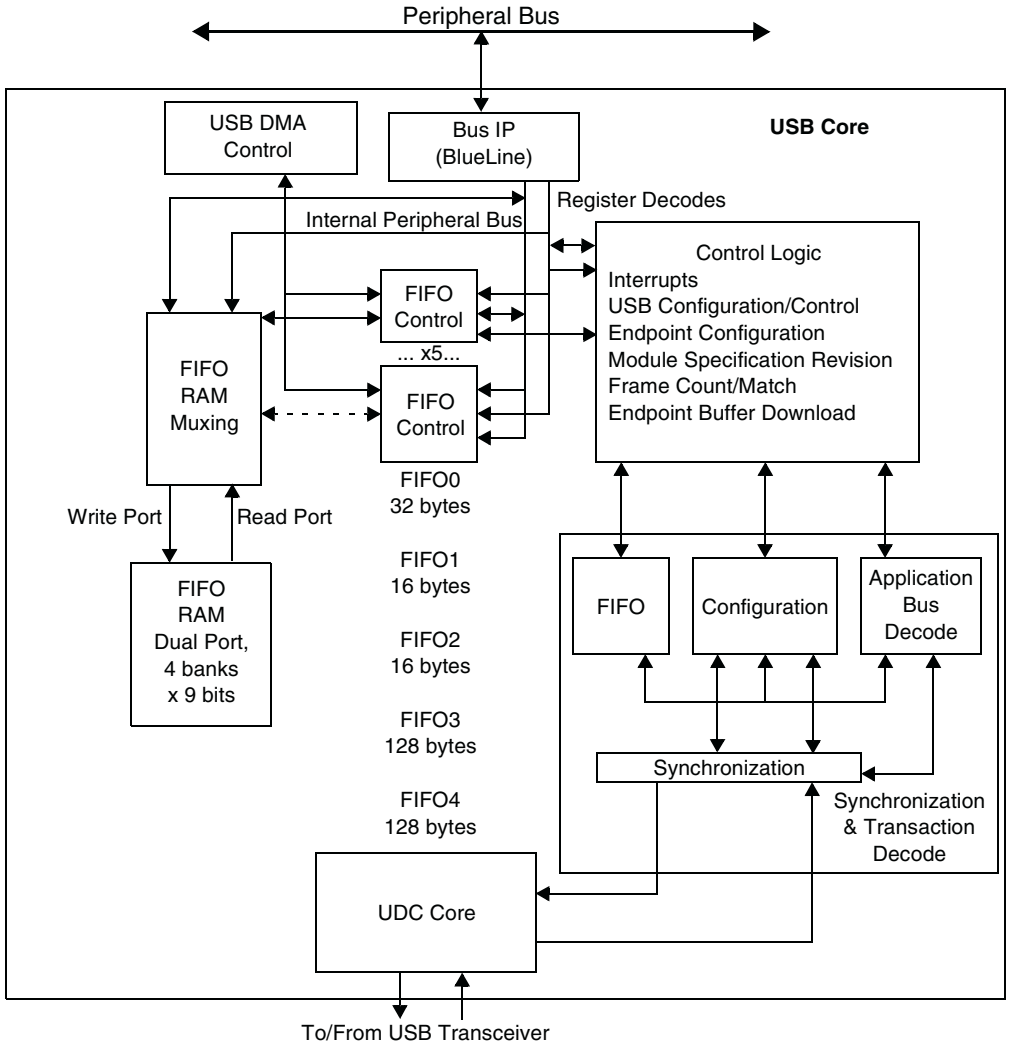

Figure 7. USB Device Core (UDC) Block Diagram

The following steps are found in the MC68SZ328 Reference's Manual on programming the UDC registers but are explained in greater detail in this application note.

UDC Register Programming Instructions

1. Perform a hard reset or a software rest (RST bit in USB_ENAB register).

   Software option—USB_ENAB = 0x8000 0000

   This sets the RST bit and the ENAB bits. Wait for the bit to clear before continuing to program registers. Check that the CFG bit in the CFGSTAT register is set.

2. Download configuration data (EndptBufs) to the device via the USB_DDAT register shown in Figure 8.

   EndptBufs—A personality file that contains the bytes of the five endpoint buffers (40 bits) that loads directly into the UDC module via DDAT[7:0]. The EndPtBufs register contains the necessary setup data for each endpoint while configuring the USB module in the MC68SZ328.

USB_DDAT USB Endpoint Buffer Register **Addr 0x(FF)FE0414**

BIT	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
TYPE																
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
									0x0000							

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
									DDAT[7:0]							
TYPE									rw	rw	rw	rw	rw	rw	rw	rw
RESET	0	0	0	0	0	0	0	0	X	X	X	X	X	X	X	X
									0x0001							

Figure 8. USB Endpoint Buffer Register

To program all required 40 bits (5 bytes) per endpoint into the EndPtBufs buffer, the information must be programmed into the DDAT field byte per byte. The USB Device Core in the MC68SZ328 processor has five EndPutBuf registers that must be programmed. Each register represents one endpoint (Ep0–EP4). Once loaded, the values in these buffers cannot be changed while the device is powered on.

The first EndPtBuf is reserved for information about control endpoint (or endpoint 0). The value for this register is always 0x0000080000. Figure 9 on page 11 explains the EndPtBuf register.

When entering the endpoint buffer (EndPtBufs) information to the USB_DDAT register, the first byte written is EPn[39:32] and the last byte written is EPn[7:0]. After writing each byte, and before performing any other operation on the USB module, check that the BSY bit and the CFG bit have cleared in the USB_CFGSTAT register. This check tells the program when the byte is written to the UDC, therefore allowing the device to program this DDAT field with the byte, USB_CFGSTAT = 0x0000 0000.

Code Example 2 shows how to program the EndptBufs via the USB_DDAT field.

Code Example 2. Programming EndptBufs via the USB_DDAT

```
//Setup information that will go into the EndPtBufs register
static unsigned char epcfg[NUM_ENDPOINT][5] = { { 0x00,0x00,0x08,0x00,0x00 },
                                                { 0x14,0x10,0x10,0xC0,0x01 },
                                                { 0x24,0x14,0x10,0xC0,0x02 },
                                                { 0x34,0x10,0x40,0xC0,0x03 },
                                                { 0x44,0x14,0x40,0xC0,0x04 } };
{
            int i,ep;

            for (ep = 0; ep < NUM_ENDPOINT; ep++)
            {
                  for (i = 0; i < 5; i++)
                  {
// setup information into DDAT register byte for byte

                        USB_DDAT = epcfg[ep][i];
                        while( USB_CFGSTAT & 0x40000000);
                  }

            }
            while (USB_CFGSTAT & 0x80000000);

            return( TRUE )
}
```

Figure 9 represents the EndPBufs register with the appropriate register numbers as they should be programmed into the UDC. Table 4 provides the register descriptions.

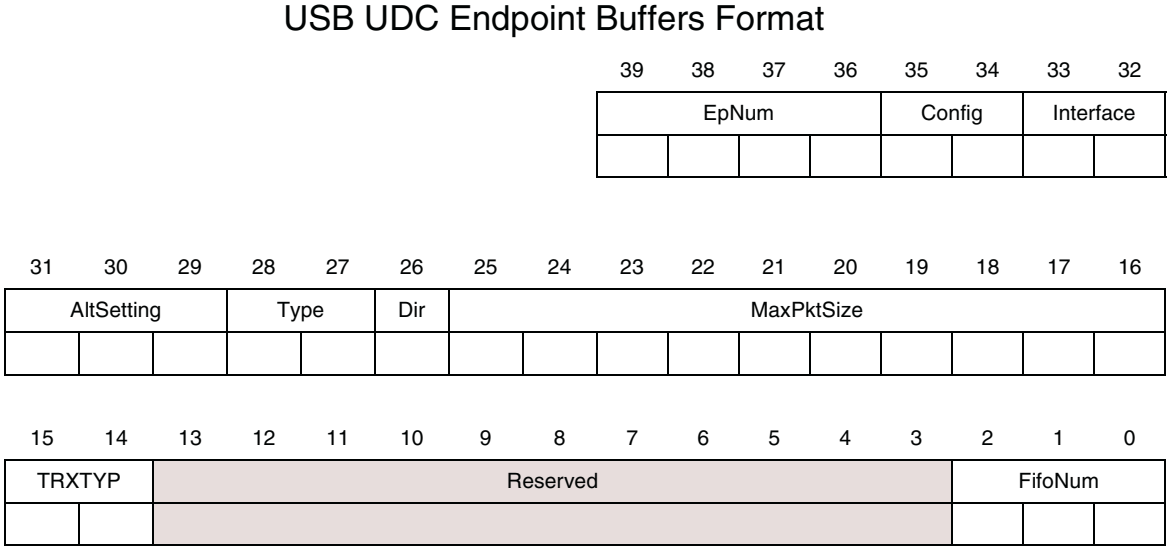

Figure 9. EndPBufs Register

Table 4. EndPBufs Register Description Table

Bit Field	Type	Description
[39:36]	EpNum	Logical endpoint number
[35:34]	Config	Configuration number—Maximum up to 1 configurations.
[33:32]	Interface	Interface number—Maximum up to 1 interface.

Table 4. EndPBufs Register Description Table (Continued)

Bit Field	Type	Description
[31:29]	AltSetting	Alternate setting number—Maximum up to 4 alternate settings.
[28:27]	Type	Type of endpoint: 00 = Control 01 = Reserved 10 = Bulk 11 = Interrupt
[26]	Dir	Direction of the Endpoint: 0 = OUT endpoint 1 = IN endpoint
[25:16]	MaxPktSize	Maximum packet size for endpoint: 0x08 = 8 bytes 0x10 = 16 bytes 0x20 = 32 bytes 0x40 = 64 bytes
[15:14]	TRXTYP	These bits must be set to 2'b00 for endpoint 0 and 2'b11 for all other endpoints.
[13:3]	Reserved	
[2:0]	FifoNum	FIFO Number—This field maps the endpoint to one of the USB Device module's hardware FIFOs. Multiple UDC endpoints may map to a single hardware FIFO. It is up to the software to monitor and control any data hazards related to operation in this way. The hardware FIFOs that are available are: FIFO 0 (32 bytes) FIFO 1 (16 bytes) FIFO 2 (16 bytes) FIFO 3 (128 bytes) FIFO 4 (128 bytes)

Table 5 shows the register configuration for endpoints 0 through 4 in Code Example 2 on page 11.

Table 5.

Endpoint (EP#)	Configuration	Interface	Alternate Setting	Type	Direction	Maximum Packet Size	FIFO Number
Endpt0	00	00	000	Control	In/Out	8 bytes	0
Endpt1	01	00	000	Bulk	Out	16 bytes	1
Endpt2	01	00	000	Bulk	In	16 bytes	2
Endpt3	01	00	000	Bulk	Out	64 bytes	3
Endpt4	01	00	000	Bulk	In	64 bytes	4

The selection of the current EndPoint information from all the EndPtBuf registers is based on the current configuration and current alternate setting of the interface with which the EndPtBuf is associated. EndPtBuf0 (EndPtBuf information for End Point 0) is always fixed for EndPoint 0.

There will be only one EndPoint0 for the entire device and this EndPoint is associated to all configurations and interfaces in the device. The UDC ignores the values programmed in the Ep_Config, Ep_Interface, and Ep_AltSetting fields of the EndPtBuf0.

3.2.1 USB Module General Registers

The general registers in the USB module cover the memory address locations, 0xFFFE0400 to 0XFFFE0428. The following list identifies specific characteristics of those registers.

- USB_FRAME—Used for debugging purposes.
- USB_SPEC—This register identifies the version of the USB specification to which the underlying USB core is compliant. (MC68SZ328 USB module is version 1.1 and only covers full speed).
- USBB_STAT—Bits CFG, INTF, and ALTSET indicate the current USB configuration. These bits are programmed during configuration download (using the EndPtBf register).
- USB_CTRL—Bit CMD_OVER (see note) is responsible for monitoring transactions between host and device.
- USB_SPD—Bit is set, full speed (low and high speeds are not supported in the MC68SZ328).
- USBD_AFE—Bit high (full speed communication activation), enables USB module front-end logic (ready for communication) with host.
- USB_CFGSTAT—Bits CFG and BSY are monitored during device configuration.
- USB_GEN_ISR—Sets interrupts necessary for chip to work properly.
- USB_MASK—Interrupt disable.
- USB_ENAB—No reset in progress, enables USB for communication with the host.
- USB_ISR—When the device is ready to receive and transmit data, all bits read as zero.

NOTE:

In the USB_CTRL register, the CMD_OVER bit is cleared after a request is completed. The bit remains set until the current request is completed.

Actions to consider when setting up for initialization:

1. Program the USB interrupt mask register (USB_MASK) to enable USB general interrupts.

   This enables all interrupts except the SOF and MSOF interrupts.

2. Clear interrupts by writing 1 to all bits in the general and Endpoint interrupt registers as shown in Code Example 3.

Code Example 3. Clearing Interrupts in the General and Endpoint Interrupt Registers

```
void usbdIsrInit( void )
{
            int i;

            USB_GEN_ISR = 0x800000FF; //Clear interrupts
            USB_MASK = 0x800000C0; // Mask interrupts
}
```

3.2.2 USB Module Endpoint Specific Registers

The USB module Endpoint specific registers cover the memory address locations identified in Table 6.

Table 6. Endpoint Specific Register Memory Address Range

Endpoint	Memory Address Range
Endpoint 0	0xFFFE0430 – 0xFFFE0458
Endpoint 1	0xFFFE0460 – 0xFFFE0488
Endpoint 2	0xFFFE0490 – 0xFFFE04B8
Endpoint 3	0xFFFE04C0 – 0xFFFE04E8
Endpoint 4	0xFFFE04F0 – 0xFFFE0518
Endpoint 5	0xFFFE0520 – 0xFFFE0548

Each of these registers is programmed according to the characteristics of each endpoint used.

During initialization the endpoint configuration register USB_EPn_STATCR must include the same configuration setup for each endpoint in the EndPtBuf register. In the USB_EPn_STATCR register, the direction, maximum packet size, and type of transfers are matched to the EndPtBufs register.

Before downloading the information to the endpoint registers, it is important to reset the interrupt requests and to mask all desired interrupts in the USB_EP(n)_ISR registers (where n = endpoint 0–4). To clear any interrupt in the USB module, including interrupts for the endpoints, registers need to be set to 1, because interrupts do not clear even if the interrupt requests go away. By setting all bits to 1, the particular interrupt clears. Writing 0 to registers that are already set does not clear these interrupts.

All other endpoint registers can be set with the default setting for initialization purposes.

In Code Example 4 all endpoints 0 through 5 are set for initialization.

Code Example 4. Setting Endpoints 0–5 for Initialization

```
static unsigned int epstat[NUM_ENDPOINT] = { 0x0002, 0x0032, 0x00B2, 0x0072, 0x00F2 };
// Same configuration as the already configured EndPtBufs register
bool usbdFifoInit( void ) {
for (ep = 0; ep < NUM_ENDPOINT; ep++)
            {

                  USB_EP_STATCR(ep) = epstat[ep];// Program dir, max size, type
                  USB_EP_FCTRL(ep) = 0x0900000 // Frame mode disable

            for (i = 0; i < NUM_ENDPOINT; i++)
            {
                        USB_EP_ISR(i) = 0x1FF; //Clear interrupts
                        USB_EP_MASK(i) = 0x1F0; //Set desired masks
            }
}
```

3.3 Receive and Transmit Data

When receiving and transmitting data certain rules of engagement must be taken into account. The first distinction between both actions is the way that they are processed. Both actions transmit from the same differential lines, and the same endpoint registers in the MC68SZ328 using the USB_EPn_FDAT field (where n = endpoint 0–4).

The USB_EPn_FDAT is a 16 bit register that is controlled by events coming from the USB_EPn_STATCR (endpoint status registers). During configuration, the information in the status register has to be the same as the information of the EndPtBfs buffer for each of the endpoints used. The status register is responsible for the direction of transfer, the maximum amounts of stored bytes in FIFO, type of transfer, FIFO flushing, stall acknowledgement, and so on.

At the same time, the USB_EPn_ISR (USB endpoint interrupt request registers) handle their specific endpoint requests and create interrupt requests based on those software calls. Interrupts that are triggered on this registers are: FIFO full, FIFO empty, error, high-level alarm, low-level alarm, multiple device request, end-of-transfer, device request for single requests, and end-of-frame. When any interrupt bit is set (interrupts occur) the USB module moves data accordingly. Follow the necessary steps to clear such interrupts when needed.

3.3.1 Transmitting Data

When transmitting data, each endpoint in the device interfaces with its counterpart in the host through pipes. Each FIFO in the device has a maximum amount of bytes of storage. For endpoint 0 there are 32 bytes of physical space available. Therefore, 32 byte maximum packet size that can be transmitted.

In Code Example 2 on page 11 only endpoint 0 is programmed to a maximum packet size transfer of 8 bytes (setup packet maximum size) for configuration purposes. "Ep0 - (0x0000080000)" (which constitutes an 8 byte transfer maximum) is ignored because EP0 configuration is hard mapped, but the download is still required to properly configure endpoints 1 to 4. However, after the endpoint configuration is complete, the actual maximum packet size transfer for endpoint 0 is determined by the information programmed in the MAX[1:0] of the USB_EP0_STATCR register (which should not be greater than 32 bytes).

Endpoints 1, 2, 3, and 4 can be used for control, interrupt or bulk types of transfer only. As mentioned before, the MC68SZ328 does not support isochronous transfers. Endpoints 1 and 2 are 16 bytes and endpoints 3, 4 are 128 bytes. Even though FIFOs for endpoints 3 and 4 are 128 bytes, the maximum amount of packet size for a bulk transfer is 64 bytes. This doubling of space in the FIFO for endpoint 3 and 4 serves as a step to maintain data transfer timing requirements of the MC68SZ328. The behavior of each endpoint changes according to how each endpoint register is programmed, and the behavior of each transfer follows the characteristics of each endpoint. To send all necessary transfers from the device to the host, the host needs to send requests to the device. The device acknowledges these requests and responds to the host with the required information.

To Send a Packet of Data to the USB Host

Perform the following steps to send a packet of data to the USB host using programmed I/O. Code Example 5 on page 17 provides the programming instructions to accompany these procedures.

For odd byte size packets:

1. For an N (N = odd) byte packet, write the first N-1 bytes to the FIFO data register (USB_EPn_FDAT). Data may be written as words.
2. Set the WFR bit in the USB_EPn_FCTRL register, then write the last data word, (byte N-1 and byte N) to the USB_EPn_FDAT register.

When the WFR bit is set, it tells the device that the next transaction that takes place is the last data word (either 16 or 8 bit) for that particular transaction.

Even byte size packets:

1. For an N (N = even) byte packet, write the first N-2 bytes to the FIFO data register (USB_EPn_FDAT). Data may be written as words.
2. Set the WFR bit in the USB_EPn_FCTRL register and then write the last data word (byte N-1 and byte N) to the USB_EPn_FDAT register.

Code Example 5. Sending Packets

```
void Send_Odd_Packet(U32 start, U32 end)NOTES
{
            U32 i;
// write first 2 bytes of last packet
//          for (i=start; i<end; i=i++)
//          {

//          _reg_USBD_EP0_FDAT_byte_access = (USB_CONFIG_DESC[i]);

//          count++
}
            if (

            _reg_USBD_EP0_FDAT_byte_access = (USB_CONFIG_DESC[80]);
            _reg_USBD_EP0_FDAT_byte_access = (USB_CONFIG_DESC[81]);

// set WFR bit of USB_EP0_FCTRL register
            _reg_USBD_EP0_FCTRL |= 0x20000000;

// write last byte of packet last
            _reg_USBD_EP0_FDAT_byte_access = (USB_CONFIG_DESC[82]);
}


void Send_Even_Packet(U32 start, U32 end) //Example for 8 byte packet
{
            U32 i;

            if (start - end== 0)
            {
// set WFR bit of USB_EP0_FCTRL register
                  _reg_USBD_EP0_FCTRL |= 0x20000000;

                  _reg_USBD_EP0_FDAT_byte_access = (USB_CONFIG_DESC[start]);
    }
    else
    {
// write first 6 bytes of packet 2
                  for (i=start; i<end+1; i=i+2)
                  {

                  _reg_USBD_EP0_FDAT = (USB_CONFIG_DESC[i] << 8) |
USB_CONFIG_DESC[i+1];

                  }

// set WFR bit of USB_EP0_FCTRL register
                  _reg_USBD_EP0_FCTRL |= 0x20000000;

// write last 2 bytes of packet 2 (8 byte total)
                  _reg_USBD_EP0_FDAT = (USB_CONFIG_DESC[end-1] << 8) |
USB_CONFIG_DESC[end];
            }
}
```

3.3.2 Receiving Data

Customize the program to read USB_EPn_ISR (n=0-4) bit 1. After this bit is set, the end-of-transfer interrupt enables. When set, it tells the device that the last packet of the transfer has been received. Read the USB_EPn_FDAT register for the corresponding endpoint.

In special cases, such as the setup packet monitor shown in Code Example 6, USB_EPn_STATCR (SIP) bit 8 and USB_EPn_ISR DEVREQ bit 1 are set. Clear corresponding EOF and DEVREQ interrupts. It is important to clear the bit that caused the interrupt (by writing a 1 in the specific register) in the USB_EPn_ISR register to clear that particular interrupt, otherwise interrupts will not clear automatically. The UDC acknowledges to the host that the setup data is received, then the process of flushing the FIFOs for next transfer takes place.

Code Example 6. Receiving Packets

```
void Setup_Packet_Detect()
{
// Verify the Registers
             peek_USBD_ISR[0] = _reg_USBD_ISR;
             peek_USBD_CTRL[0] = _reg_USBD_CTRL;
             peek_EP0_STATCR[0] = _reg_USBD_EP0_STATCR;
             peek_EP0_ISR[0] = _reg_USBD_EP0_ISR;
             peek_EP0_FSTAT[0] = _reg_USBD_EP0_FSTAT;

             _reg_USBD_MASK = (~0x000000FF);// clear mask
             _reg_USBD_EP0_MASK = (~0x000001FF);

//   1. HOST Sends a setup packet to the device.

//   2. Wait until setup packet received, SIP set
             while (!(_reg_USBD_EP0_STATCR & BIT8));

//   3. Wait until Device Request Received, DEVREQ set
             while (!(_reg_USBD_EP0_ISR & DEVREQ_BIT));

// Clear DEVREQ and EOF if get it...
             _reg_USBD_EP0_ISR |= 0x00000003;

// Read the 8 byte Packet
             fifo_peek[0] = _reg_USBD_EP0_FDAT;
             fifo_peek[1] = _reg_USBD_EP0_FDAT;
             fifo_peek[2] = _reg_USBD_EP0_FDAT;
             fifo_peek[3] = _reg_USBD_EP0_FDAT;
                    .
                    .
                    .

void ACK_Detect()
{
//   3. Wait until Device Request Received, DEVREQ set
             while (!(_reg_USBD_EP0_ISR & EOF_BIT));

// Clear EOF
             _reg_USBD_EP0_ISR |= 0x00000001;

// Flush the FIFO
             _reg_USBD_EP0_STATCR |= 0x00000002
```

NOTES

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:

Motorola Literature Distribution;
P.O. Box 5405, Denver, Colorado 80217
1-303-675-2140 or 1-800-441-2447

JAPAN:

Motorola Japan Ltd.; SPS, Technical Information Center,
3-20-1, Minami-Azabu Minato-ku, Tokyo 106-8573 Japan
81-3-3440-3569

ASIA/PACIFIC:

Motorola Semiconductors H.K. Ltd.; Silicon Harbour
Centre, 2 Dai King Street, Tai Po Industrial Estate,
Tai Po, N.T., Hong Kong
852-26668334

TECHNICAL INFORMATION CENTER:

1-800-521-6274

HOME PAGE:

http://www.motorola.com/semiconductors

AN2294/D