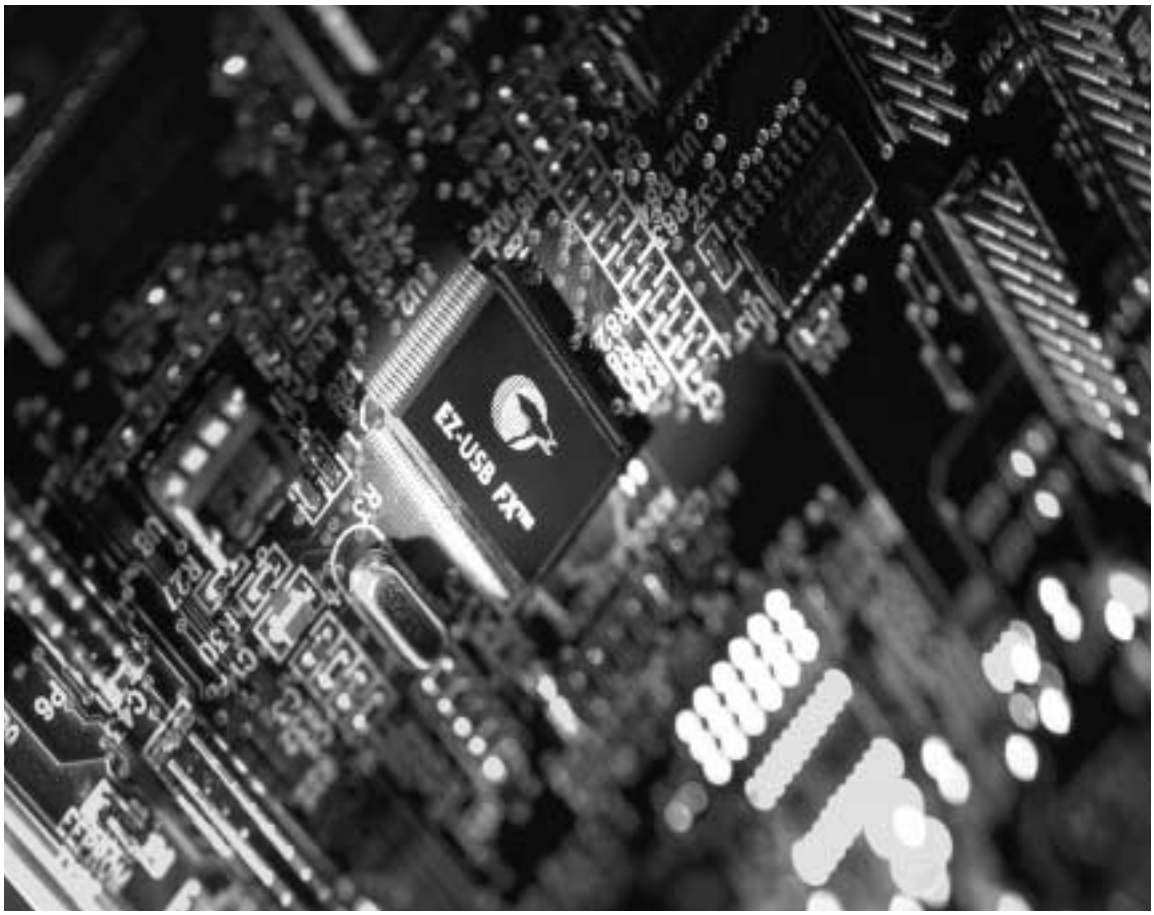
EZ-USB FX
Technical Reference Manual



• Cypress Semiconductor • Interface Products Division •
• 15050 Avenue of Science • Suite 200 • San Diego, CA 92128 •

Cypress Disclaimer Agreement

The information in this document is subject to change without notice and should not be construed as a commitment by Cypress Semiconductor Corporation Incorporated. While reasonable precautions have been taken, Cypress Semiconductor Corporation assumes no responsibility for any errors that may appear in this document.

Cypress Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Cypress Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Cypress Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Cypress Semiconductor and its officers, employees, subsidiaries, affiliates and distributors harmless against all claims, costs, damages, expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Cypress Semiconductor was negligent regarding the design or manufacture of the part.

The acceptance of this document will be construed as an acceptance of the foregoing conditions.

The EZ-USB FX Technical Reference Manual, Version 1.2.

Table of Contents

List of Figures

(List of Figures)

List of Tables

(List of Tables)

Chapter 1. Introducing EZ-USB FX

1.1 Introduction

Like a well designed automobile or appliance, a USB peripheral's outward simplicity hides internal complexity. There's a lot going on "under the hood" of a USB device, which gives the user a new level of convenience. For example:

- A USB device can be plugged in anytime, even when the PC is turned on.
- When the PC detects that a USB device has been plugged in, it automatically interrogates the device to learn its capabilities and requirements. From this information, the PC automatically loads the device's driver into the operating system. When the device is unplugged, the operating system automatically logs it off and unloads its driver.
- USB devices do not use DIP switches, jumpers, or configuration programs. There is never an IRQ, DMA, MEMORY, or I/O conflict with a USB device.
- USB expansion hubs make the bus available to dozens of devices.
- USB is fast enough for printers, CD-quality audio, and scanners.

USB is defined in the *Universal Serial Bus Specification Version 1.1*, a 268-page document describing in elaborate detail all aspects of a USB device. The USB Specification is available at **http://usb.org**. The *EZ-USB FX Technical Reference Manual* describes the EZ-USB FX chip along with USB topics that provide help in understanding the USB Specification.

The Cypress Semiconductor EZ-USB FX is a compact, integrated circuit that provides a highly integrated solution for a USB peripheral device. Three key EZ-USB FX features are:

- The EZ-USB FX family provides a *soft* (RAM-based) solution that allows unlimited configuration and upgrades.
- The EZ-USB FX family delivers full USB throughput. Designs that use EZ-USB FX are not limited by number of endpoints, buffer sizes, or transfer speeds.
- The EZ-USB FX family does much of the USB housekeeping in the USB core, simplifying code and accelerating the USB learning curve.

This chapter introduces key USB concepts and terms to make reading this Technical Reference Manual easier.

1.2 EZ-USB FX Block Diagrams

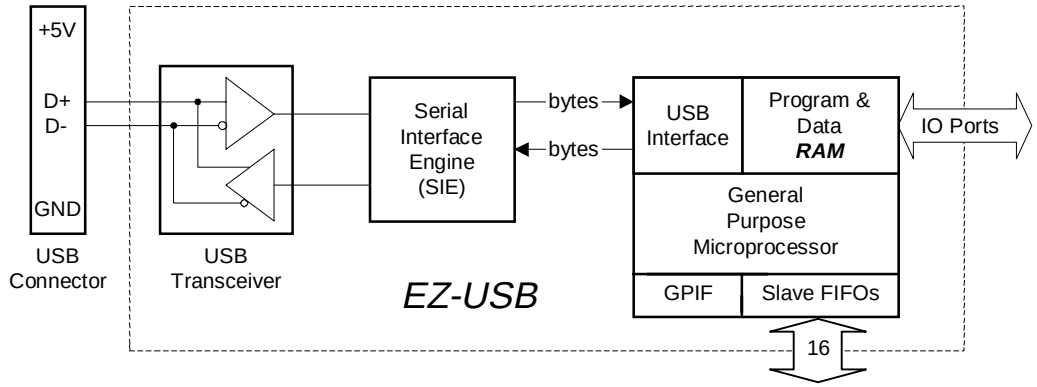


Figure 1-1. CY7C646x3-80NC (80 pin) Simplified Block Diagram

The Cypress Semiconductor EZ-USB FX chip packs the intelligence required by a USB peripheral interface into a compact, integrated circuit. As *Figure 1-1* illustrates, an integrated USB transceiver connects to the USB bus pins D+ and D-. A Serial Interface Engine (SIE) decodes and encodes the serial data and performs error correction, bit stuffing, and other signaling-level details required by USB. Ultimately, the SIE transfers data bytes to and from the USB interface.

The internal microprocessor is an enhanced 8051 with fast execution time and added features. It uses internal RAM for program and data storage, making the EZ-USB FX family a *soft* solution. The USB host downloads 8051 program code and device personality into RAM over the USB bus, and then EZ-USB FX re-connects as the custom device, as defined by the loaded code.

The EZ-USB FX family uses an enhanced SIE/USB interface (USB Core), which has the intelligence to function as a full USB device, even before the 8051 runs. The enhanced core simplifies 8051 code by implementing much of the USB protocol, itself.

EZ-USB FX chips operate at 3.3V. This simplifies the design of bus-powered USB devices, since the 5V power available in the USB connector (USB Specification allows power to be as low as 4.4V) can drive a 3.3V regulator to deliver clean, isolated power to the EZ-USB FX chip.

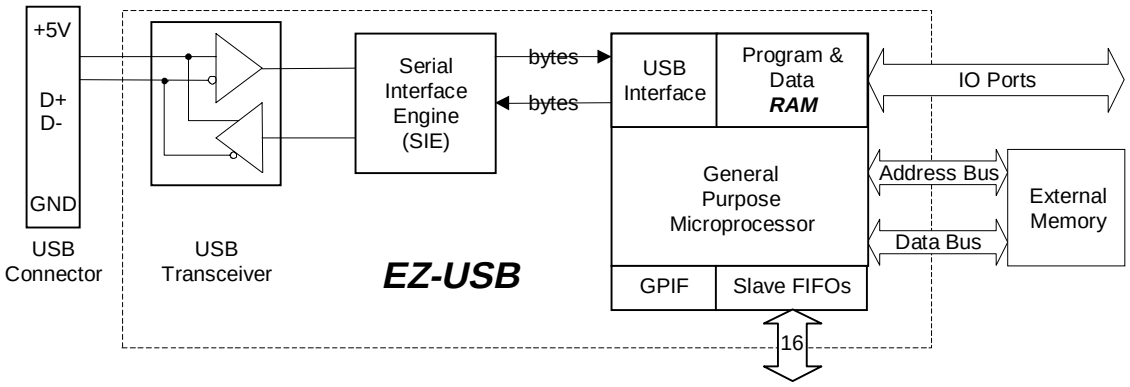


Figure 1-2. CY7C646x3-128NC (128 pin) Simplified Block Diagram

Figure 1-2 illustrates the CY7C646x3-128NC, a 128-pin version of the EZ-USB FX family. In addition to the 40 I/O pins, it contains a 16-bit address bus and an 8-bit data bus for external memory expansion. Slave interface FIFOs and a General Programmable Interface (GPIF) controller provide a flexible, high-bandwidth interface to external logic.

Also included, the DMAEXTFIFO register provides legacy support for invoking the fast transfer mode available on the EZ-USB Series 2100. This allows data to move directly between external logic and internal USB FIFOs. This, along with abundant endpoint resources, allows the EZ-USB FX family to support transfer bandwidths to external logic that exceed the USB delivery/consumption rate.

1.3 The USB Specification

The *Universal Serial Bus Specification Version 1.1* is available on the Internet at **http://usb.org**. Published in January 1998, the USB Specification is the work of a founding committee of seven industry heavyweights: Compaq, DEC, IBM, Intel, Microsoft, NEC, and Northem Telecom. This impressive list of developers secures USB as the low-to-medium speed PC connection method of the future.

A glance at the USB Specification makes it immediately apparent that USB is not nearly as simple as the customary serial or parallel port. The USB Specification uses new terms like “endpoint,” isochronous,” and “enumeration,” and finds new uses for old terms like “configuration,” “interface,” and “interrupt.” Woven into the USB fabric is a software abstraction model that deals with things such as “pipes.” The USB Specification also contains detail about the connector types and wire colors.

1.4 Tokens and PIDs

In this manual, statements like the following appear: “When the host sends an IN token...,” or “The device responds with an ACK.” What do these terms mean? A USB transaction consists of data packets identified by special codes called Packet IDs or PIDs. A PID signifies what kind of packet is being transmitted. There are four PID types, shown in Table 1-1.

Table 1-1. USB PIDs

PID Type	PID Name
Token	IN, OUT, SOF, SETUP
Data	DATA0, DATA1
Handshake	ACK, NAK, STALL
Special	PRE

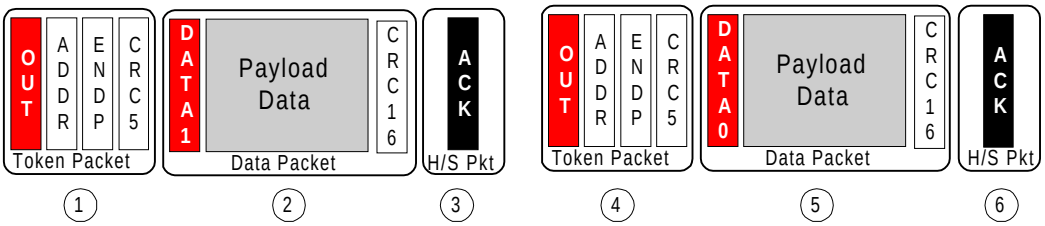


Figure 1-3. USB Packets

Figure 1-3 illustrates a USB transfer. Packet **1** is an OUT token, indicated by the OUT PID. The OUT token signifies that data from the host is about to be transmitted over the bus. Packet **2** contains data, as indicated by the DATA1 PID. Packet **3** is a handshake packet, sent by the device using the ACK (acknowledge) PID to signify to the host that the device received the data error-free.

Continuing with *Figure 1-3*, a second transaction begins with another OUT token **4**, followed by more data **5**, this time using the DATA0 PID. Finally, the device again indicates success by transmitting the ACK PID in a handshake packet **6**.

Why two DATA PIDs, DATA0 and DATA1? It’s because the USB architects took error correction very seriously. As mentioned previously, the ACK handshake is an indication to the host that the peripheral received data without error (the CRC portion of the packet is used to detect errors). But what if the handshake packet itself is garbled in transmission? To detect this, each side (host and device) maintains a *data toggle* bit, which is toggled between data packet transfers. The state of this internal toggle bit is compared with the PID that arrives with the data, either DATA0 or DATA1. When sending data, the host or device sends alternating DATA0-DATA1 PIDs. By comparing the Data PID with the state of the internal toggle bit, the host or device can detect a corrupted handshake packet.

SETUP tokens are unique to CONTROL transfers. They preface eight bytes of data from which the peripheral decodes host Device Requests.

SOF tokens occur once per millisecond, denoting a USB *frame*.

There are three handshake PIDs: ACK, NAK, and STALL.

- ACK means *success*; the data was received error-free.
- NAK means "busy, try again." It's tempting to assume that NAK means "error," but it doesn't. A USB device indicates an error by *not responding*.
- STALL means that something unforeseen went wrong (probably as a result of miscommunication or lack of cooperation between the software and firmware writers). A device sends the STALL handshake to indicate that it doesn't understand a device request, that something went wrong on the peripheral end, or that the host tried to access a resource that wasn't there. It's like HALT, but better, because USB provides a way to recover from a stall.

A PRE (Preamble) PID precedes a low-speed (1.5 Mbps) USB transmission. The EZ-USB FX family supports high-speed (12 Mbps) USB transfers only. It ignores PRE packets and the resultant low-speed transfer.

1.5 Host is Master

This is a fundamental USB concept. There is exactly one master in a USB system: the host computer. **USB devices respond to host requests**. USB devices cannot send information between themselves, as they could if USB were a peer-to-peer topology.

However, there is one case where a USB device can initiate signaling without prompting from the host. After being put into a low-power suspend mode by the host, a device can signal a remote wakeup. A Remote Wakeup is the only method in which the USB becomes the initiator. Everything else happens because the host makes device requests, and the device responds to them.

There's an excellent reason for this host-centric model. The USB architects were keenly mindful of cost, and the best way to make low-cost peripherals is to put most of the smarts into the host side, the PC. If USB had been defined as peer-to-peer, every USB device would have required more intelligence, raising cost.

Here are two important consequences of the "host is master" concept:

1.5.1 Receiving Data from the Host

To send data to a USB peripheral, the host issues an OUT token, followed by the data. If the peripheral has space for the data, and accepts it without error, it returns an ACK to the host. If it is

busy, it sends a NAK. If it finds an error, it sends back nothing. For the latter two cases, the host re-sends the data at a later time.

1.5.2 Sending Data to the Host

A USB device never spontaneously sends data to the host. Nevertheless, in the EZ-USB FX chip, there's nothing to stop the 8051 from loading data for the host into an endpoint buffer (see *"EZ-USB FX Endpoints"*, this chapter) and *arming* it for transfer. However, the data remains in the buffer *until the host sends an IN token to that particular endpoint*. If the host never sends the IN token, the data remains there indefinitely.

1.6 USB Direction

Once you accept that the host is the bus master, it's easy to remember USB direction: OUT means from the host to the device, and IN means from the device to the host. EZ-USB FX nomenclature uses this naming convention. For example, an endpoint that sends data to the host is an IN endpoint. This can be confusing at first, because the 8051 *sends* data by loading an IN endpoint buffer. Keep in mind that an 8051*out* is an IN to the host.

1.7 Frame

The USB host provides a time base to all USB devices by transmitting a SOF (Start Of Frame) packet every millisecond. The SOF packet includes an incrementing, 11-bit frame count. The 8051 can read this frame count from two EZ-USB FX registers. SOF-time has significance for isochronous endpoints; it's the time that the *ping-ponging* buffers switch places. The USB core provides the 8051 with an SOF interrupt request for servicing isochronous endpoint data.

1.8 EZ-USB FX Transfer Types

USB defines four transfer types. These match the requirements of different data types delivered over the bus. (*"EZ-USB FX Endpoints"* explains how the EZ-USB FX family supports the four transfer types.)

1.8.1 Bulk Transfers

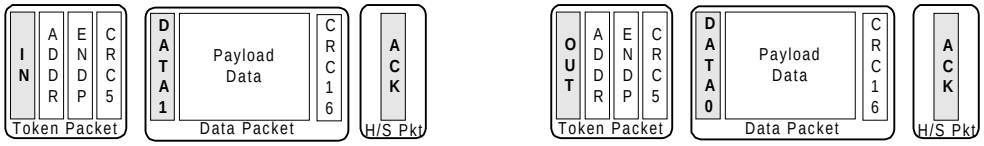


Figure 1-4. Two Bulk Transfers, IN and OUT

Bulk data is *bursty*, traveling in packets of 8, 16, 32, or 64 bytes. Bulk data has guaranteed accuracy, due to an automatic re-try mechanism for erroneous data. The host schedules bulk packets when there is available bus time. Bulk transfers are typically used for printer, scanner, or modem data. Bulk data has built-in flow control provided by handshake packets.

1.8.2 Interrupt Transfers

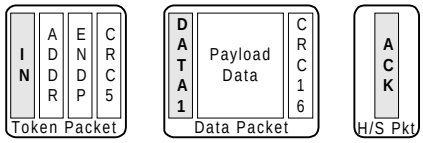


Figure 1-5. An Interrupt Transfer

Interrupt data is like bulk data; it can have packet sizes of 1 through 64 bytes. Interrupt endpoints have an associated polling interval that ensures they will be *pinged* (receive an IN token) by the host on a regular basis.

1.8.3 Isochronous Transfers

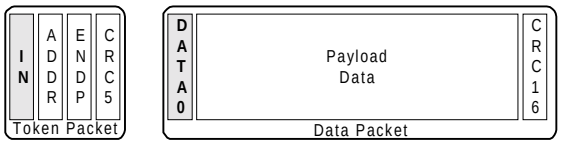


Figure 1-6. An Isochronous Transfer

Isochronous data is time-critical and used to *stream* data like audio and video. Time of delivery is the most important requirement for isochronous data. In every USB frame, a certain amount of USB bandwidth is allocated to isochronous transfers. To lighten the overhead, isochronous transfers have no handshake (ACK/NAK/STALL), and no retries. Error detection is limited to a 16-bit CRC. Isochronous transfers do not use the data toggle mechanism. Isochronous data uses only the DATA0 PID.

1.8.4 Control Transfers

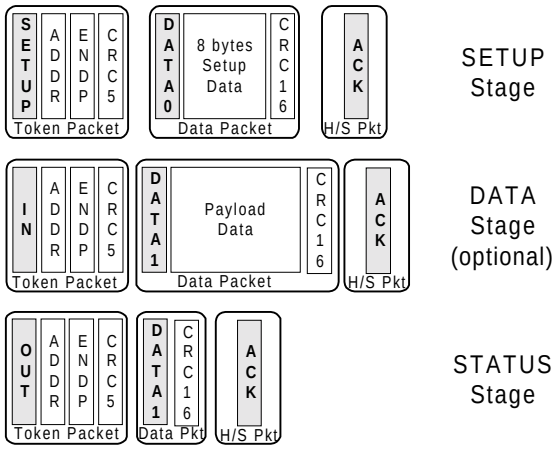


Figure 1-7. A Control Transfer

Control transfers configure and send commands to a device. Being *mission critical*, they employ the most extensive USB error checking USB. Control transfers are delivered on a *best effort* basis by the host (*best effort* is a six-step process defined by the *Universal Serial Bus Specification Version 1.1, "Section 5.5.4"*). The host reserves a part of each USB frame time for Control transfers.

Control transfers consist of two or three stages. The SETUP stage contains eight bytes of USB CONTROL data. An optional DATA stage contains more data, if required. The STATUS (or *handshake*) stage allows the device to indicate successful completion of a control operation.

1.9 Enumeration

Your computer is ON. You plug in a USB device, and the Windows® cursor switches to an hourglass, and then back to a cursor. Magically, your device is connected, and its Windows® driver is loaded! Anyone who has installed a sound card into a PC and had to configure countless jumpers, drivers, and IO/Interrupt/DMA settings knows that a USB connection is miraculous. We've all *heard* about Plug and Play, but USB delivers the real thing.

How does all this happen automatically? Inside every USB device is a table of *descriptors*. This table is the sum total of the device's requirements and capabilities. When you plug into USB, the host goes through a *sign-on* sequence:

1. The host sends a "Get_Descriptor/Device" request to address zero (devices must respond to address zero when first attached).
2. The device responds to the request by sending ID data back to the host to define itself.
3. The host sends the device a *Set_Address* request, which gives it a unique address to distinguish it from the other devices connected to the bus.
4. The host sends more *Get_Descriptor* requests, asking more device information. From this, it learns everything else about the device, like how many endpoints the device has, its power requirements, what bus bandwidth it requires, and what driver to load.

This sign-on process is called *Enumeration*.

1.10 The USB Core

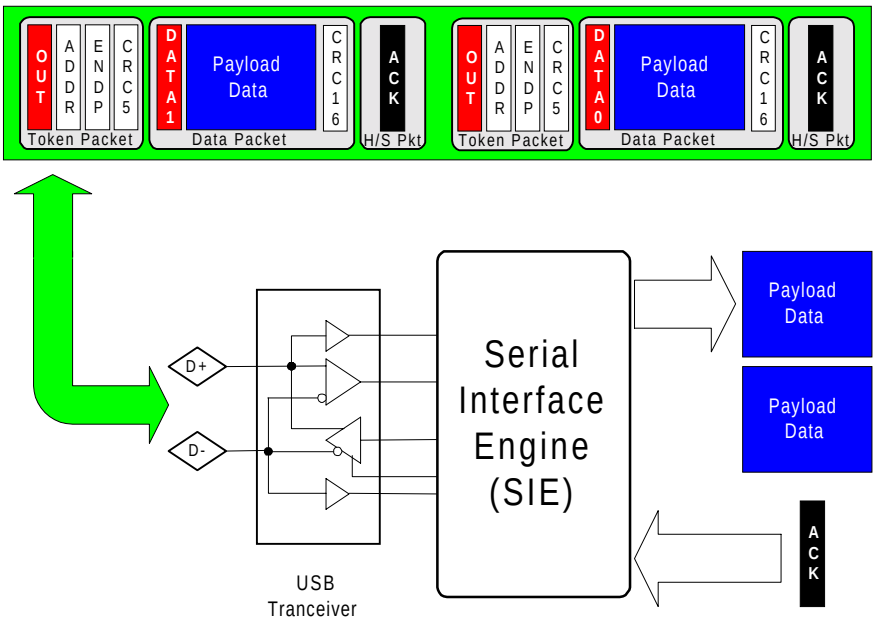


Figure 1-8. What the SIE Does

Every USB device has a Serial Interface Engine (SIE). The SIE connects to the USB data lines D+ and D-, and delivers bytes to and from the USB device. *Figure 1-8* illustrates a USB bulk transfer, with time moving from left to right. The SIE decodes the packet PIDs, performs error checking on the data using the transmitted CRC bits, and delivers payload data to the USB device. If the SIE

encounters an error in the data, it automatically indicates *no response* instead of supplying a handshake PID. This instructs the host to re-transmit the data at a later time.

Bulk transfers, such as the one illustrated in *Figure 1-8*, are *asynchronous*, meaning that they include a flow control mechanism using ACK and NAK handshake PIDs. The SIE indicates *busy* to the host by sending a NAK handshake packet. When the peripheral device has successfully transferred the data, it commands the SIE to send an ACK handshake packet, indicating success.

To send data to the host, the SIE accepts bytes and control signals from the USB device, formats it for USB transfer, and sends it over the two-wire USB. Because the USB uses a self-clocking data format (NRZI), the SIE also inserts bits at appropriate places in the bit stream to guarantee a certain number of transitions in the serial data. This is called “bit stuffing,” and is handled transparently by the SIE.

One of the most important features of the EZ-USB FX family is that it is *soft*. Instead of requiring ROM or other fixed memory, it contains internal program/data RAM downloaded over the USB to give the device its unique personality. This makes modifications, specification revisions, and updates a snap.

The EZ-USB FX family can connect as a USB device and download code into internal RAM. All while, its internal 8051 is held in Reset. This is done by an enhanced SIE, which performs all the work shown in *Figure 1-8*, and more. It contains additional logic to perform a full enumeration, using an internal table of descriptors. It also responds to a vendor specific “Firmware Download” device request to load its internal RAM. Additionally, the added SIE functionality is made available to the 8051. This saves 8051 code and processing time.

Throughout this manual, the SIE and its enhancements are referred to as the “USB Core.”

1.11 EZ-USB FX Microprocessor

The EZ-USB FX microprocessor is an enhanced 8051 core. Use of an 8051-compatible processor makes available immediately extensive software support tools to the EZ-USB FX designer. This enhanced 8051 core (described in *Chapter 2. "EZ-USB FX CPU"*, *Chapter 16. "8051 Introduction"*, *Chapter 17. "8051 Architectural Overview"*, and *Chapter 18. "8051 Hardware Description"*) has the following features:

- 4 clocks/cycle, compared to the 12 clocks/cycle of a standard 8051:a 10X speed improvement.
- 48-MHz clock.
- DMA for 48 MB/second memory-to-memory transfers. Dual data pointers for improved XDATA access.
- Two UARTs.

- Three counter-timers.
- Expanded interrupt system.
- 256 bytes of internal register RAM.
- Standard 8051 instruction set—if you know the 8051, you know EZ-USB FX.

The enhanced 8051 core uses on-chip RAM as program and data memory, giving EZ-USB FX its *soft* feature. *Chapter 3. "EZ-USB FX Memory"* describes the various memory options.

The 8051 communicates with the SIE using a set of registers, occupying the top of the on-chip RAM address space. These registers are grouped and described by function in individual chapters of this reference manual and summarized in register order in *Chapter 15. "EZ-USB FX Registers"*.

The EZ-USB 8051 has two duties. First, it participates in the protocol defined in the *Universal Serial Bus Specification Version 1.1, "Chapter 9, USB Device Framework."* Thanks to EZ-USB FX enhancements to the SIE and USB interface, the 8051 firmware associated with USB overhead is simplified, leaving code space and bandwidth available for the 8051's primary duty, to help implement your device. On the device side, abundant input/output resources are available, including I/O ports, UARTs, and an I^2C-compatible bus master controller. These resources are described in *Chapter 4. "EZ-USB FX Input/Output"*

1.12 ReNumeration™

Because the EZ-USB FX chip is *soft*, it can take on the identities of multiple distinct USB devices. The first device downloads your 8051 firmware and USB descriptor tables over the USB cable when the peripheral device is plugged in. Once downloaded, another device comes on as a totally different USB peripheral as defined by the downloaded information. This patented two-step process, called ReNumeration™, happens instantly when the device is plugged in, with no hint that the initial load step has occurred.

Chapter 5. "EZ-USB FX Enumeration & ReNumeration™" describes this feature in detail, along with other EZ-USB FX boot (startup) modes.

1.13 EZ-USB FX Endpoints

The USB Specification defines an endpoint as a source or sink of data. Since USB is a serial bus, a device endpoint is actually a FIFO, which sequentially empties/fills with USB bytes. The host selects a device endpoint by sending a 4-bit address and one direction bit. Therefore, USB can uniquely address 32 endpoints, IN0 through IN15 and OUT0 through OUT15.

From the EZ-USB FX point of view, an endpoint is a buffer full of bytes received or held for transmitted over the bus. The 8051 reads endpoint data from an OUT buffer, and writes endpoint data for transmission over USB to an IN buffer.

There are four USB endpoint types: Bulk, Control, Interrupt, and Isochronous. These endpoint types are described in the following paragraphs:

1.13.1 EZ-USB FX Bulk Endpoints

Bulk endpoints are unidirectional—one endpoint address per direction. Therefore, endpoint 2-IN is addressed differently than endpoint 2-OUT. Bulk endpoints use maximum packet sizes (buffer sizes) of 8, 16, 32, or 64 bytes. EZ-USB FX provides fourteen bulk endpoints, divided into seven IN endpoints (endpoint 1-IN through 7-IN), and seven OUT endpoints (endpoint 1-OUT through 7-OUT). Each of the fourteen endpoints has a 64-byte buffer.

Bulk data is available to the 8051 in RAM or as FIFO data, using a special EZ-USB FX *Autopointer* (*Chapter 6. "EZ-USB FX Bulk Transfers"*).

1.13.2 EZ-USB FX Control Endpoint Zero

Control endpoints transfer mission-critical control information to and from the USB device. The USB Specification requires every USB device to have a default CONTROL endpoint, endpoint zero. Device enumeration, the process that the host initiates when the device is first plugged in, is conducted over endpoint zero. The host sends all USB requests over endpoint zero.

Control endpoints are bi-directional. If you have an endpoint 0 IN CONTROL endpoint, you automatically have an endpoint 0 OUT endpoint. Only Control endpoints accept SETUP PIDs.

A CONTROL transfer consists of a two or three stage sequence:

- SETUP
- DATA (If needed)
- HANDSHAKE

Eight bytes of data in the SETUP portion of the CONTROL transfer have special USB significance, as defined in the *Universal Serial Bus Specification Version 1.1, “Chapter 9.”* A USB device must respond properly to the requests described in this chapter to pass USB compliance testing (referred to as the USB “Chapter Nine Test”).

Endpoint zero is the only CONTROL endpoint in the EZ-USB FX chip. The 8051 responds to device requests issued by the host over endpoint zero. The USB core is significantly enhanced to simplify the 8051 code required to service these requests. *Chapter 9. "EZ-USB FX Endpoint Zero"* provides a detailed roadmap for writing compliant USB Chapter 9 8051 code.

1.13.3 EZ-USB FX Interrupt Endpoints

Interrupt endpoints are almost identical to bulk endpoints. Fourteen EZ-USB FX endpoints (EP1-EP7, IN, and OUT) may be used as interrupt endpoints. Interrupt endpoints have a maximum packet size 64. They contain a “polling interval” byte in their descriptor to tell the host how often to service them. The 8051 transfers data over interrupt endpoints in exactly the same way as for bulk endpoints. Interrupt endpoints are described in *Chapter 6. "EZ-USB FX Bulk Transfers."*

1.13.4 EZ-USB FX Isochronous Endpoints

Isochronous endpoints deliver high bandwidth, time critical data over USB. Isochronous endpoints are used to stream data to devices such as audio DACs, and from devices such as video cameras. Time of delivery is the most critical requirement, and isochronous endpoints are tailored to this requirement. Once a device has been granted an isochronous bandwidth slot by the host, it is guaranteed the ability to send or receive its data every frame.

EZ-USB FX contains 16 isochronous endpoints, numbered 8-15 (8IN-15IN, and 8OUT-15OUT). 1,024 bytes of FIFO memory are available to the 16 endpoints, and may be divided among them. EZ-USB FX actually contains 2,048 bytes of isochronous FIFO memory to provide double-buffering. Using double buffering, the 8051 reads OUT data from isochronous endpoint FIFOs containing data from the previous frame, while the host writes current frame data into the other buffer. Similarly, the 8051 loads IN data into isochronous endpoint FIFOs that will be transmitted over USB during the next frame, while the host reads current frame data from the other buffer. At every SOF the USB FIFOs and 8051 FIFOs switch, or *ping-pong*.

Isochronous transfers are described in *Chapter 10. "EZ-USB FX Isochronous Transfers."*

1.14 Interrupts

EZ-USB FX adds seven interrupt sources to the standard 8051 interrupt system. Three of the added interrupts are available on device pins: INT4, INT5#, and INT6. The other four are used internally: INT2 is used for all USB interrupts, INT3 is used by the I^2C-compatible interface, INT4 is used by the FIFOs and GPIF, and the remaining interrupt is used for remote wakeup indication.

The USB core automatically supplies jump vectors (Autovectors) for its USB and FIFO interrupts to save the 8051 from having to test bits to determine the source of the interrupt. Each INT2 and INT4 interrupt source has its own vector. When an interrupt requires service, the proper ISR (interrupt service routine) is automatically invoked. *Chapter 12. "EZ-USB FX Interrupts"* describes the EZ-USB FX interrupt system.

1.15 Reset and Power Management

The EZ-USB FX chip contains four resets:

- Power-On-Reset (POR)
- USB bus reset
- 8051 reset
- USB Disconnect/Re-connect

The functions of the various EZ-USB FX resets are described in *Chapter 13. "EZ-USB FX Resets"*

A USB peripheral may be put into a low power state when the host signals a *suspend* operation. The USB Specification states that a bus-powered device cannot draw more than 500 ΩA of current from the VBUS wire while in suspend. The EZ-USB FX chip contains logic to turn off its internal oscillator and enter a *sleep* state. A special interrupt, triggered by a wakeup pin or wakeup signaling on the USB bus, starts the oscillator and interrupts the 8051 to resume operation.

Low power operation is described in *Chapter 14. "EZ-USB FX Power Management"*.

1.16 Slave FIFOs

The EZ-USB FX contains four 64-byte FIFOs to provide a flexible, high-speed interface to a variety of peripherals. These FIFOs can be *slave FIFOs* that accept RD/WR strobes from an external source, or the *GPIF* can be their bus master. See *"GPIF (General Programmable Interface)"* below for more information. Two FIFOs are provided in the IN direction, and two FIFOs transfer data in the OUT direction.

The FIFO module allows the EZ-USB FX to perform the following functions:

- Act as a FIFO or a small RAM on a microprocessor bus
- Create a 16-bit data path
- Transfer data at speeds up to 96 MB per second (burst).

1.17 GPIF (General Programmable Interface)

The GPIF is a programmable state machine that runs at 48 MHz. It can be used to generate custom bus waveforems and control the FIFOs. It has four programs of seven steps each that execute

at 48 MHz. The GPIF program can modify the CTL0-5 lines, branch on the RDY0-5 inputs, and control FIFO data movement.

The GPIF is used to implement many standard interfaces available for the FX, including:

- IDE (ATAPI)
- PC parallel port (EPP)
- Utopia

The GPIF is fully described in *Chapter 8. "General Programmable Interface (GPIF)."*

1.18 EZ-USB FX Product Family

The EZ-USB FX family is available in various pinouts to serve different system requirements and costs. Table 1-2 shows the feature set for each member of the EZ-USB FX family.

Table 1-2. EZ-USB FX Family

Part Number	Package	Ram	ISO Support	I/O	FIFO Width	Addr/Data Bus
CY7C64601-52NC	52-pin PQFP	4 K	No	16	8 Bits	No
CY7C64603-52NC	52-pin PQFP	8 K	No	18	8 Bits	No
CY7C64613-52NC	52-pin PQFP	8 K	Yes	18	8 Bits	No
CY7C64603-80NC	80-pin PQFP	8 K	No	32	16 Bits	No
CY7C64613-80NC	80-pin PQFP	8 K	Yes	32	16 Bits	No
CY7C64603-128NC	128-pin PQFP	8 K	No	40	16 Bits	Yes
CY7C64613-128NC	128-pin PQFP	8 K	Yes	40	16 Bits	Yes

Chapter 2. EZ-USB FX CPU

2.1 Introduction

The EZ-USB FX built-in microprocessor, an enhanced 8051 core, is fully described in *Chapter 16. "8051 Introduction"* , *Chapter 17. "8051 Architectural Overview"* and *Chapter 18. "8051 Hardware Description."* This chapter introduces the processor, its interface to the USB core, and describes architectural differences from a standard 8051.

2.2 8051 Enhancements

The enhanced 8051 core uses the standard 8051 instruction set. Instructions execute faster than with the standard 8051 due to two features:

- Wasted bus cycles are eliminated. A bus cycle uses four clocks, as compared to 12 clocks with the standard 8051.
- The 8051 runs at 24 MHz or 48 MHz.

In addition to speed improvement, the enhanced 8051 core also includes architectural enhancements:

- A second data pointer
- A second UART
- A third, 16-bit timer (TIMER2)
- A high-speed memory interface with a non-multiplexed 16-bit address bus
- Eight additional interrupts (INT2-INT6, WAKEUP, T2, and UART1)
- Variable MOVX timing to accommodate fast/slow RAM peripherals
- 3.3V operation.

2.3 EZ-USB FX Enhancements

EZ-USB FX provides additional enhancements outside the 8051. These include:

- DMA Module
- Fast external transfers (Autopointer, DMAEXTFIFO)
- Vectored USB interrupts (Autovector)
- Separate buffers for SETUP and DATA portions of a CONTROL transfer
- Breakpoint Facility.

2.4 EZ-USB FX Register Interface

The 8051 communicates with the USB core through a set of memory mapped registers. These registers are grouped as follows:

- Endpoint buffers and FIFOs
- Slave FIFOs
- 8051 control
- I/O ports
- DMAEXTFIFO
- I^2C-Compatible Controller
- Interrupts
- USB Functions
- GPIF

These registers and their functions are described throughout this manual. A full description of every register and bit appears in *Chapter 15. "EZ-USB FX Registers."*

2.5 EZ-USB FX Internal RAM

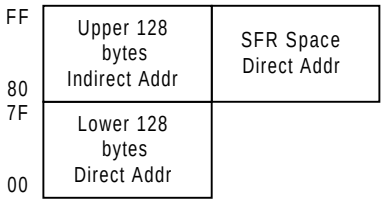


Figure 2-1. 8051 Registers

Like the standard 8051, the EZ-USB 8051 core contains 128 bytes of register RAM at address 00-7F, and a partially populated SFR register space at address 80-FF. An additional 128 indirectly addressed registers (sometimes called “IDATA”) are also available at address 80-FF.

All internal EZ-USB FX RAM, which includes program/data memory, bulk endpoint buffer memory, and the EZ-USB FX register set, is addressed as *add-on* 8051 memory. The 8051 reads or writes these bytes as data using the MOVX (move external) instruction. Even though the MOVX instruction implies external memory, the EZ-USB FX RAM and register set is actually inside the EZ-USB FX chip. External memory attached to the CY7C646x3-128NC address and data busses can also be accessed by the MOVX instruction. The USB core encodes its memory strobe and select signals (RD#, WR#, CS#, and OE#) to eliminate the need for external logic to separate the internal and external memory spaces.

2.6 I/O Ports

A standard 8051 communicates with its I/O ports 0-3 through four Special Function Registers (SFRs). **The USB core implements I/O ports differently than a standard 8051**, as described in *Chapter 4. "EZ-USB FX Input/Output."* The USB core implements a flexible I/O system that is controlled *via* SFRs or *via* the EZ-USB FX register set. Although 8051 SFR bits may be used to control the I/O pins, their addresses and functions are different than in a standard 8051. Each EZ-USB FX I/O pin functions identically, having the following resources:

- An output latch (OUTn). Used when the pin is an output port.
- A register (PINSn) that indicates the state of the I/O pins, regardless of its configuration (input or output).
- An output enable register (OEn) that causes the I/O pin to be driven from the output latch.

Several registers control whether the pin is a port pin or a special function pin. These registers include PORTnCFG, IFCONFIG, PORTACF2, and PORTCGPIF. See *"Port Configuration Tables"* in *Chapter 4. "EZ-USB FX Input/Output"*.

2.7 Interrupts

All standard 8051 interrupts are supported in the enhanced 8051 core. Table 2-1 shows the existing and added 8051 interrupts, and indicates how the added ones are used.

Table 2-1. EZ-USB FX Interrupts

Standard 8051 Interrupts	Enhanced 8051 Interrupts	Used As
INT0		Device Pin INT0#
INT1		Device Pin INT1#
Timer 0		Internal, Timer 0
Timer 1		Internal, Timer 1
Tx0 & Rx0		Internal, UART0
	INT2	Internal, USB
	INT3	Internal, I^2C-compatible Controller
	INT4	Internal, FIFO Interrupt
	INT5	Device Pin, PB5/INT5#
	INT6	Device Pin, PB6/INT6
	WAKEUP	Device Pin, USB WAKEUP#
	Tx1 & Rx1	Internal, UART1
	Timer 2	Internal, Timer 2

The EZ-USB FX chip uses 8051 INT2 for 22 different USB interrupts: 17 bulk endpoints plus SOF, Suspend, SETUP Data, SETUP Token, and USB Bus Reset. To help the 8051 determine which interrupt is active, the USB core provides a feature called Autovectoring. The core inserts an address byte into the low byte of the 3-byte jump instruction found at the 8051 INT2 vector address. This second level of vectoring automatically transfers control to the appropriate USB ISR. The Autovector mechanism, as well as the EZ-USB FX interrupt system is the subject of *Chapter 12. "EZ-USB FX Interrupts."*

2.8 Power Control

The USB core implements a power-down mode that allows it to be used in USB bus-powered devices that must draw no more than 500 ΩA when suspended. Power control is accomplished using a combination of 8051 and USB core resources. The mechanism by which EZ-USB FX powers down for suspend, and then re-powers to resume operation, is described in detail in *Chapter 14. "EZ-USB FX Power Management."*

EZ-USB FX responds to USB suspend using three 8051 resources: the *idle* mode and two interrupts. A USB suspend operation is indicated by a lack of bus activity for 3 ms. The USB core detects this, and asserts an interrupt request via the USB interrupt (8051 INT2). The ISR (Interrupt Service Routine) turns off external sub-systems that draw power. When ready to suspend operation, the 8051 sets an SFR bit, PCON.0. This bit causes the 8051 to suspend, waiting for an interrupt.

When the 8051 sets PCON.0, a control signal from the 8051 to the USB core causes the core to shut down the 12-MHz oscillator and internal PLL. This stops all internal clocks to allow the USB core and 8051 to enter a very low power mode.

The suspended EZ-USB FX chip can be awakened two ways: USB bus activity may resume, or an EZ-USB FX pin (WAKEUP#) can be asserted to activate a USB *Remote Wakeup*. Either event triggers the following chain of events:

- The USB core re-starts the 12-MHz oscillator and PLL, and waits for the clocks to stabilize.
- The USB core asserts a high-priority 8051 interrupt to signal a resume interrupt.
- The 8051 vectors to the resume ISR and, upon completion, resumes executing code at the instruction following the instruction that set the PCON.0 bit to 1.

2.9 SFRs

The EZ-USB FX family was designed to keep 8051 coding as standard as possible, to allow easy integration of existing 8051 software development tools. The added 8051 SFR registers and bits are summarized in Table 2-2.

Table 2-2. Added Registers and Bits

8051 Enhancements	SFR	Addr	Function
Dual Data Pointers	DPL0	0x82	Data Pointer 0 Low Addr
	DPH0	0x83	Data Pointer 0 High Addr
	DPL1	0x84	Data Pointer 1 Low Addr
	DPH1	0x85	Data Pointer 1 High Addr
	DPS	0x86	Data Pointer Select (LSB)
	MPAGE	0x92	Replaces standard 8051 Port 2 for indirect external data memory addressing using R0 or R1
Timer 2	T2CON.6-7	0xC8	Timer 2 Control
	RCAP2L	0xCA	T2 Capture/Reload Value L
	RCAP2H	0xCB	T2 Capture/Reload Value H
	T2L	0xCC	T2 Count L
	T2H	0xCD	T2 Count H
	IE.5	0xA8	ET2-Enable T2 Interrupt Bit
	IP.5	0xB8	PT2-T2 Interrupt Priority Control
UART1	SCON1.0-1	0xC0	Serial Port 1 Control
	SBUF1	0xC1	Serial Port 1 Data
	IE.6	0xA8	ES1-SIO1 Interrupt Enable Bit
	IP.6	0xB8	PS1-SIO1 Interrupt Priority Control
	EICON.7	0xD8	SMOD1-SIO1 Baud Rate Doubler
Interrupts			
INT2-INT5	EXIF	0x91	INT2-INT5 Interrupt Flags
	EIE	0xE8	INT2-INT5 Interrupt Enables
	EIP.0-3	0xF8	INT2-INT5 Interrupt Priority Control
INT6	EICON.3	0xD8	INT6 Interrupt Flag
	EIE.4	0xE8	INT6 Interrupt Enable
	EIP.4	0xF8	INT6 Interrupt Priority Control
WAKEUP#	EICON.4	0xD8	WAKEUP# Interrupt Flag
	EICON.5	0xD8	WAKEUP# Interrupt Enable
Expanded SFRs			
I/O Registers	IOA	0x80	Input/Output A
	IOB	0x90	Input/Output B
	IOC	0xA0	Input/Output C
	IOD	0xB0	Input/Output D
	IOE	0xB1	Input/Output E
Interrupt Clears	INT2CLR	0xA1	Interrupt 2 Clear
	INT4CLR	0xA2	Interrupt 4 Clear
Enables	SOEA	0xB2	Output Enable A
	SOEB	0xB3	Output Enable B
	SOEC	0xB4	Output Enable C

8051 Enhancements	SFR	Addr	Function
	SOED	0xB5	Output Enable D
	SOEE	0xB6	Output Enable E
Idle Mode	PCON.0	0x87	EZ-USB FX Power Down (Suspend)

Members of the EZ-USB FX family that supply pins to expand 8051 memory provide separate non-multiplexed 16-bit address and 8-bit data busses. This differs from the standard 8051, which multiplexes eight device pins between three sources: I/O port 0, the external data bus, and the low byte of the address bus. A standard 8051 system with external memory requires a de-multiplexing address latch, strobed by the 8051 ALE (Address Latch Enable) pin. The external latch is not required by the non-multiplexed EZ-USB FX chip, and no ALE signal is provided. In addition to eliminating the customary external latch, the non-multiplexed bus saves one cycle per memory fetch cycle, further improving 8051 performance.

2.10 Internal Bus

The typical 8051 user must choose between using Port 0 as a memory expansion port or as an I/O port. The CY7C646x3-128NC provides a separate I/O system with its own control registers (in external memory space), and provides the I/O port signals on dedicated (not shared) pins. This allows the external data bus to expand memory without sacrificing I/O pins.

The 8051 is the sole master of the memory expansion bus. It provides read and write signals to external memory. The address bus is output-only.

The DMAEXTFIFO register provides legacy support for invoking the fast transfer mode available on the EZ-USB Series 2100. Refer to *"Dummy Register"* in *Chapter 11. "EZ-USB FX DMA System"* for more information about this register.

2.11 Reset

The internal 8051 RESET signal is not directly controlled by the EZ-USB FX RESET pin. Instead, it is controlled by an EZ-USB FX register bit accessible to the USB host. When the EZ-USB FX chip is powered, the 8051 is held in reset. Using the default USB device (enumerated by the USB core), the host downloads code into RAM. Finally, the host clears an EZ-USB FX register bit that takes the 8051 out of reset.

The EZ-USB FX family also operates with external non-volatile memory, in which case the 8051 exits the reset state automatically at power-on. The various EZ-USB FX resets and their effects are described in *Chapter 13. "EZ-USB FX Resets."*

Chapter 3. EZ-USB FX Memory

3.1 Introduction

EZ-USB FX devices divide RAM into two regions: one for code and data, and the other for USB buffers and control registers.

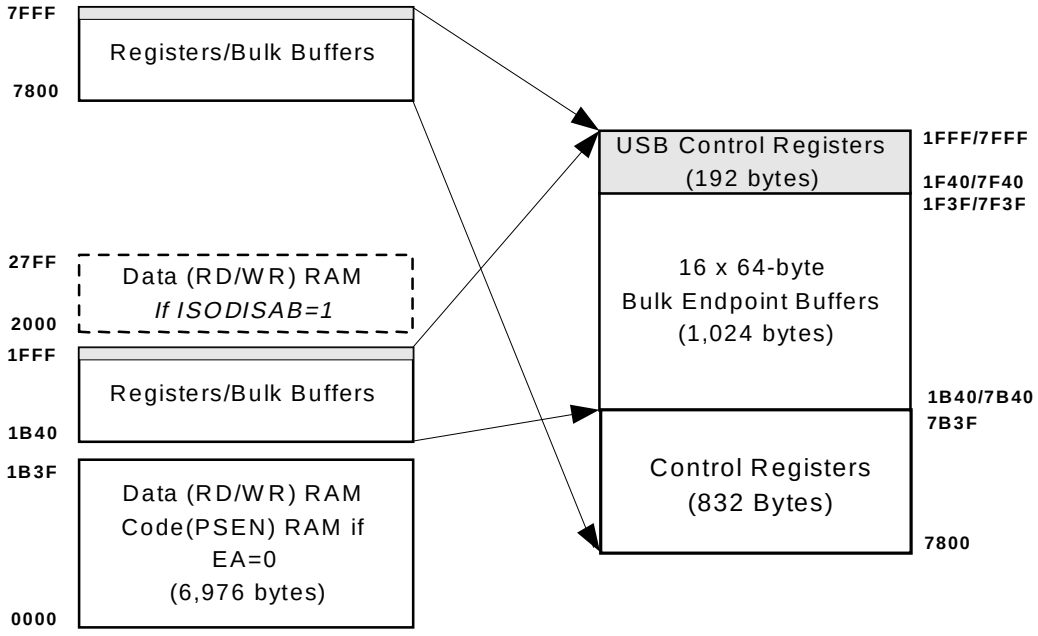


Figure 3-1. EZ-USB FX 8-KB Memory Map - Addresses are in Hexadecimal

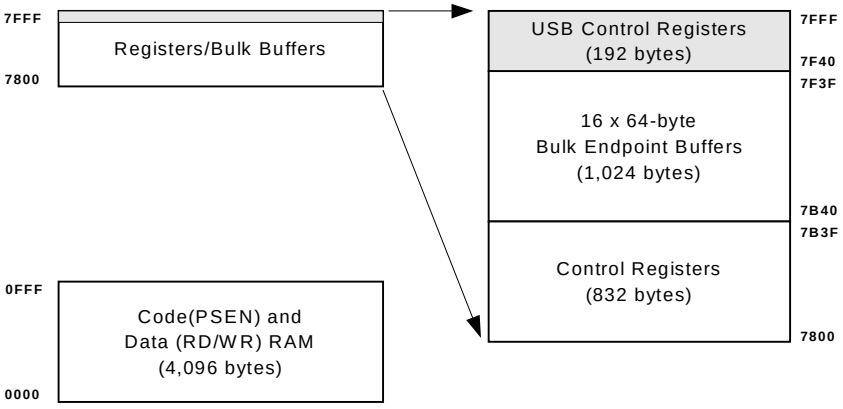


Figure 3-2. EZ-USB FX 4-KB Memory Map - Addresses are in Hexadecimal

3.2 8051 Memory

Figure 3-1 illustrates the two internal EZ-USB FX RAM regions. 6,976 bytes of general-purpose RAM occupy addresses 0x0000-0x1B3F. This RAM is loadable by the USB core or I^2C-compatible bus EEPROM, and contains 8051 code and data.

The EZ-USB FX EA (External Access) pin controls the placement of the bottom segment of code (PSEN) memory — inside (EA=0) or outside (EA=1) the EZ-USB FX chip. If the EA pin is tied low, the USB core internally ORs the two 8051 read signals PSEN and RD for this region, so that code and data share the 0x0000-0x1B3F memory space. If EA=1, all code (PSEN) memory is external.

3.2.1 About 8051 Memory Spaces

The 8051 partitions its memory spaces into code memory and data memory. The 8051 reads code memory using the signal PSEN# (Program Store Enable), reads data memory using the signal RD# (Data Read), and writes data memory using the signal WR# (Data Write). The 8051 MOVX (move external) instruction generates RD# or WR# strobes.

On EZ-USB FX, PSEN# is a dedicated pin, while the RD# and WR# signals share pins with two IO port signals: PC7/RD and PC6/WR. Therefore, if expanded memory is used, the port pins PC7 and PC6 are not available to the system.

1,024 bytes of RAM at 0x7B40-0x7F3F implement the sixteen bulk endpoint buffers. 192 additional bytes at 0x7F40-0x7FFF contain the USB control registers. The 8051 reads and writes this memory using the MOVX instruction. In the 8-KB RAM version of EZ-USB FX, the 1,024 bulk endpoint buffer bytes at 0x7B40-0x7F3F also appear at 0x1B40-0x1F3F. This aliasing allows unused

bulk endpoint buffer memory to be added contiguously to the data memory, as illustrated Figure 3-3. The memory space at 0x1B40-0x1FFF should not be used.

Even though the 8051 can access EZ-USB FX endpoint buffers at either 0x1B40 or 0x7B40, write the firmware to access this memory only at 0x7B40-0x7FFF to maintain compatibility with future versions of EZ-USB FX that contain more than 8 KB of RAM. Future versions will have the bulk buffer space at 0x7B40-0x7F3F, only.

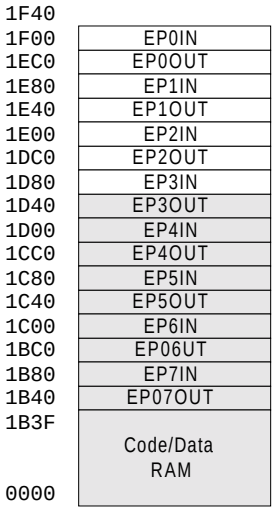


Figure 3-3. Unused Bulk Endpoint Buffers (Shaded) Used as Data Memory

In the example shown in Figure 3-3, only endpoints 0-IN through 3-IN are used for the USB function, so the data RAM (shaded) can be extended to 0x1D7F.

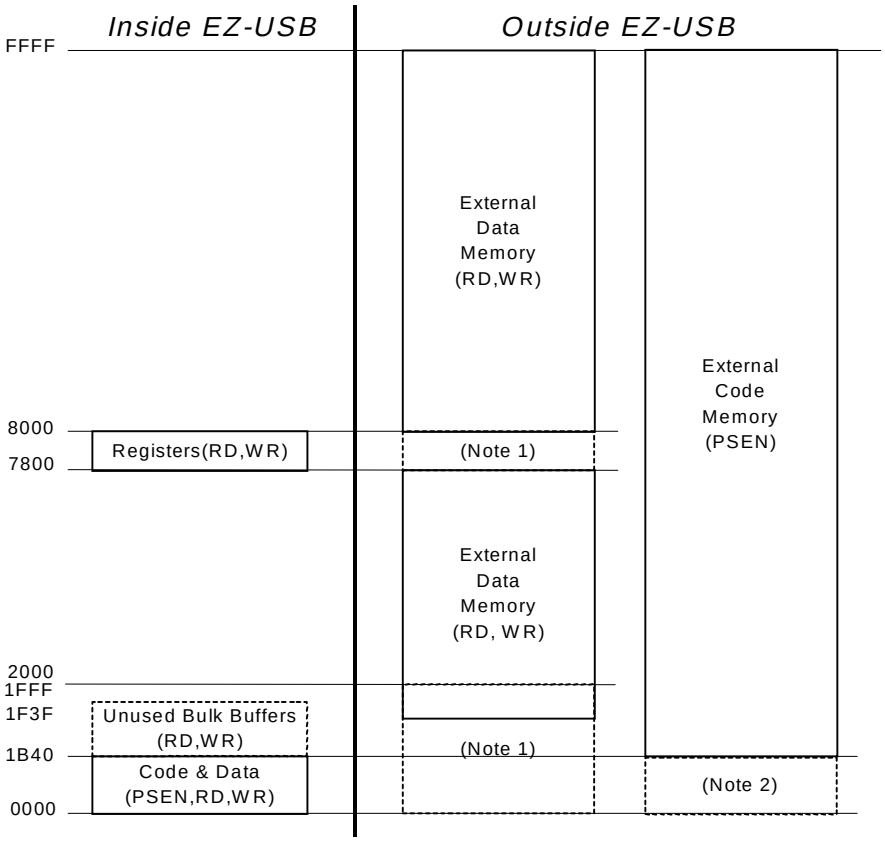
If an application uses *none* of the 16 EZ-USB FX isochronous endpoints, the 8051 can set the ISODISAB bit in the ISOCTL register to disable all 16 isochronous endpoints and make the 2-KB of isochronous FIFO RAM available as 8051 data RAM at 0x2000-0x27FF. **8051 code cannot run in this memory region**.

Setting ISODISAB=1 is an *all or nothing* choice, as all 16 isochronous endpoints are disabled. An application that sets this bit must never attempt to transfer data over an isochronous endpoint.

The memory map figures in the remainder of this chapter assume that ISODISAB=0, the default (and normal) case.

3.3 Expanding EZ-USB FX Memory

The 128-pin EZ-USB FX package provides a 16-bit address bus, an 8-bit bus, and memory control signals PSEN#, RD#, and WR#. These signals are used to expand EZ-USB FX memory.



Note 1: OK to populate data memory here--RD#, WR#, CS# and OE# pins are inactive.

Note 2: OK to populate code memory here--no PSEN# strobe is generated.

Figure 3-4. EZ-USB FX Memory Map with EA=0

Figure 3-4 shows that when EA=0, the code/data memory is internal at 0x0000-0x1B40. External code memory can be added from 0x0000-0xFFFF, but it appears in the memory map only at 0x1B40-0xFFFF. Addressing external code memory at 0x0000-0x1B3F when EA=0 causes the USB core to inhibit the #PSEN strobe. This allows program memory to be added from 0x0000-0xFFFF without requiring decoding to disable it between 0x0000 and 0x1B3F.

The internal block at 0x7B40-0x7FFF (labeled “Registers”) contains the bulk buffer memory and EZ-USB FX control registers. As previously mentioned, they are aliased at 0x1B40-0x1FFF to allow adding unused bulk buffer RAM to general-purpose memory. 8051 code should access this memory only at the 0x7B40-0x7BFF addresses. External RAM may be added from 0x0000 to 0xFFFF, but the regions shown by Note 1 in Figure 3-4 are ignored; no external strobes or select signals are generated when the 8051 executes a MOVX instruction that addresses these regions.

3.4 CS# and OE# Signals

The USB core gates the standard 8051 RD# and WR# signals to exclude selection of external memory that exists internal to the EZ-USB FX part. The PSEN# signal is also available on a pin for connection to external code memory.

Some 8051 systems implement external memory that is used as both data and program memory. These systems must logically OR the PSEN# and RD# signals to qualify the chip enable and output enable signals of the external memory. To save this logic, the USB core provides two additional control signals, CS# and OE#. The equations for these signals are as follows:

- CS# goes low when RD#, WR#, or PSEN# goes low.
- OE# goes low when RD# or PSEN# goes low.

Because the RD#, WR#, and PSEN# signals are already qualified by the addresses allocated to external memory, these strobes are active only when external memory is accessed.

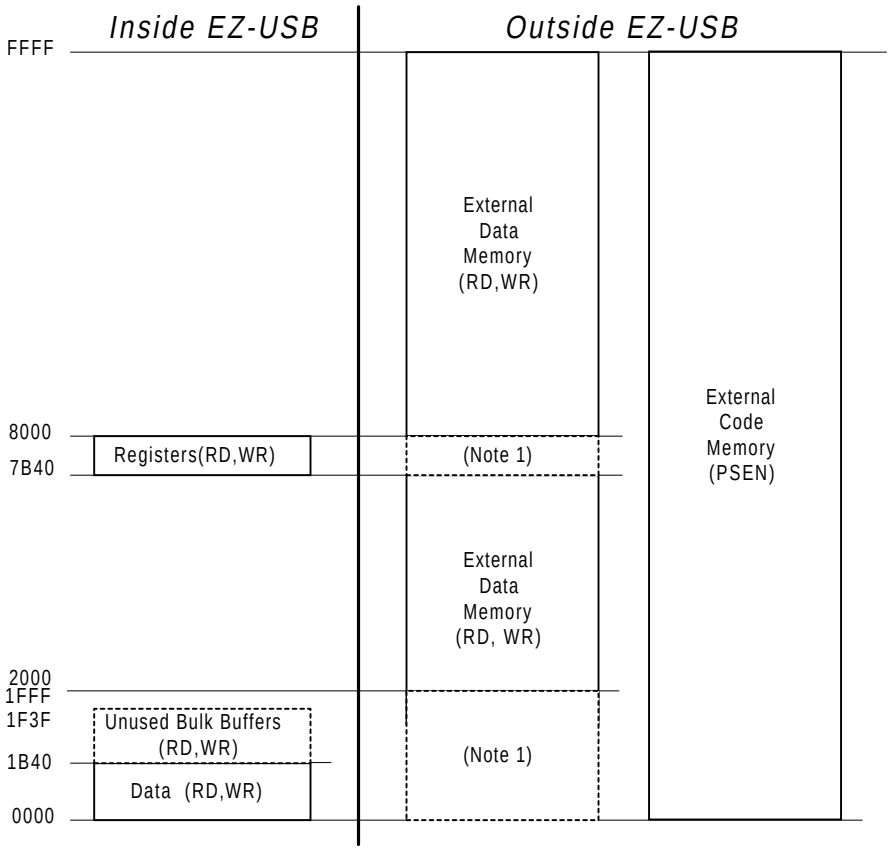


Figure 3-5. EZ-USB FX Memory Map with EA=1

When EA=1 (Figure 3-5), all code (PSEN) memory is external. All internal EZ-USB FX RAM is data memory. This gives the user over 6-KB of general-purpose RAM, accessible by the MOVX instruction.

Figure 3-4 and Figure 3-5 assume that the EZ-USB FX chip uses isochronous endpoints, and therefore that the ISODISAB bit (ISOCTL.0) is LO. If ISODISAB=1, additional data RAM appears internally at 0x2000-0x27FF, and the RD#, WR#, CS#, and OE# signals are modified to exclude this memory space from external data memory.

Chapter 4. EZ-USB FX Input/Output

4.1 Introduction

The EZ-USB FX chip provides two input-output systems:

- A set of programmable I/O pins
- A programmable I^2C-compatible Controller

This chapter describes the programmable I/O pins, and shows how they are shared by a variety of 8051 and EZ-USB FX alternate functions, such as UART and timer and interrupt signals. This chapter provides both the programming information for the 8051 I^2C-compatible interface, and the operating details of the I^2C-compatible boot loader. The role of the boot loader is described in *Chapter 5. "EZ-USB FX Enumeration & ReNumeration™"*.

The I^2C-compatible controller uses the SCL and SDA pins, and performs two functions:

- General-purpose 8051 use
- Boot loading from an EEPROM.

Pullup resistors are required on the SDA and SCL lines, even if nothing is connected to the I^2C-compatible bus. Each line should be pulled-up to Vcc through a 2.2K ohm resistor.

4.2 I/O Ports

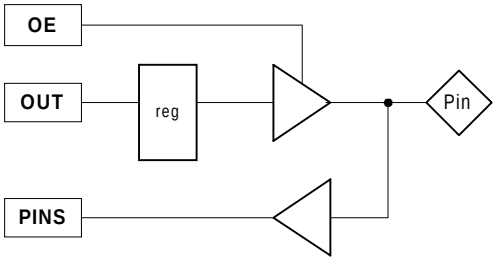


Figure 4-1. EZ-USB FX Input/Output Pin

The EZ-USB FX implements its general purpose I/O ports differently than a standard 8051. Most of the port I/O bits (PINSn and OUTn) are available in bit addressable SFR space or in XDATA space. The OEn bits are also available via SFR registers or XDATA space. See *Figure 4-6* for more information.

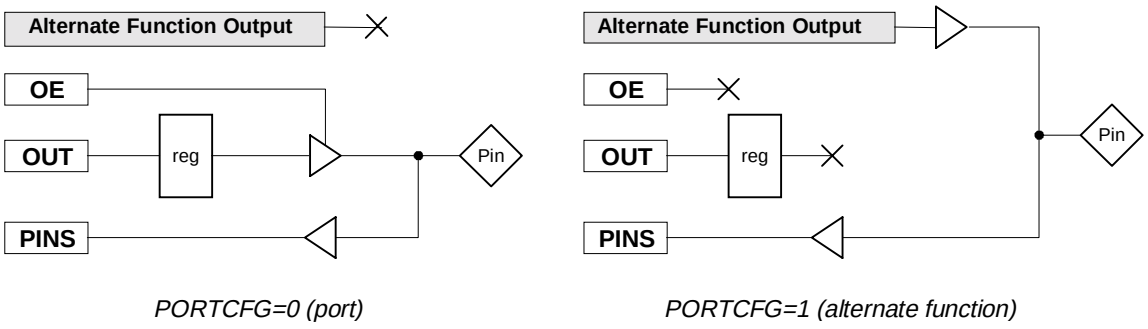
Figure 4-1 shows the basic structure of an EZ-USB FX I/O pin. Forty I/O pins are grouped into five 8-bit ports: PORTA, PORTB, PORTC, PORTD, and PORTE. The CY7C646x3-128NC brings out all five port pins. The CY7C646x3-80NC brings out all port pins for PORTA, PORTB, PORTC, and PORTD. The CY7C646x3-52NC brings out two PORTA pins and all pins of PORTB and PORTC. The 8051 accesses I/O pins using the three control bits shown in *Figure 4-1*: OE, OUT, and PINS. The OUT bit writes output data to a register. The OE bit turns on the output buffer. The PINS bit indicates the state of the pin. Section 4.12, *"SFR Addressing"* explains how this basic structure is enhanced to add SFR access to the I/O pins.

If you are using a small package version of EZ-USB FX, it is important to recognize that I/O ports exist inside the part that are not pinned out. Because I/O ports power-up as inputs, the 8051 code should initialize all of the unused ports as outputs to prevent floating internal nodes. Also, users of the 52-pin package should set IFCONFIG.7 (register 784A.7) to 1 to drive other internal nodes to their lowest power states.

To configure a pin as an input, the 8051 sets OE=0 to turn off the output buffer. To configure a pin as an output, the 8051 sets OE=1 to turn on the output buffer, and writes data to the OUT register. The PINS bit reflects the actual pin value, regardless of the value of OE.

A fourth control bit (in PORTACFG, PORTBCFG, PORTCCFG registers) determines whether a port pin is general-purpose Input/Output (GPIO), as shown in *Figure 4-1*, or connected to an alternate 8051 or EZ-USB FX function. Each bit of PORTA, PORTB, and PORTC has a corresponding control bit in PORTACFG, PORTBCFG, and PORTCCFG, respectively. *Figure 4-1* shows the registers and bits associated with the I/O ports shown in Table 4-1 through Table 4-4.

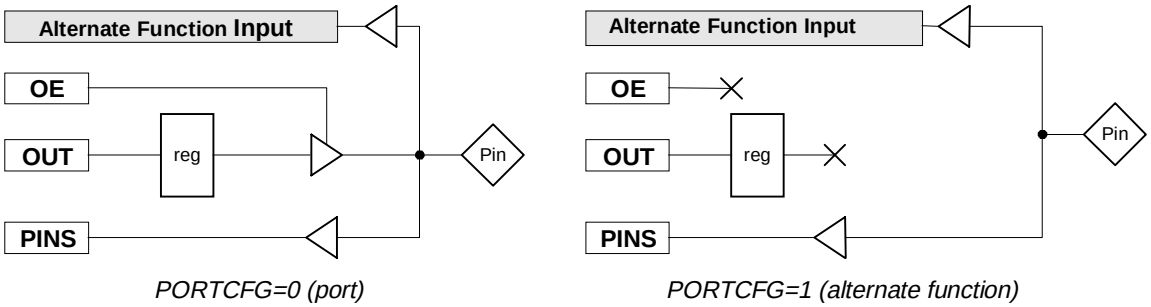
Depending on whether the alternate function is an input or output, the I/O logic is slightly different, as shown in *Figure 4-2* (output) and *Figure 4-3* (input).



PORTCFG=0 (port) *PORTCFG=1 (alternate function)*

Figure 4-2. Alternate Function is an OUTPUT

In *Figure 4-2*, when PORTCFG=0, the I/O port is selected. In this case the alternate function (shaded) is disconnected and the pin functions exactly as shown in *Figure 4-1*. When PORTCFG=1, the alternate function is connected to the I/O pin and the output register and buffer are disconnected. Note that the 8051 can still read the state of the pin, and thus the alternate function value.



PORTCFG=0 (port) *PORTCFG=1 (alternate function)*

Figure 4-3. Alternate Function is an INPUT

In *Figure 4-3*, when PORTCFG=0, the I/O port is selected. This is the general I/O port shown in *Figure 4-1*, with one important difference—the alternate function is always *listening*. Whether the port pin is set for output or input, the pin signal also drives the alternate function. 8051 firmware should ensure that if the alternate function is not used (if the pin is GPIO only), the alternate input function is disabled in the 8051 Special Function Register (SFR) space.

For example, suppose the PB6/INT6 pin is configured for PB6. The pin signal is also routed to INT6. If INT6 is not used by the application, it should not be enabled. Alternatively, enabling INT6 could be useful, allowing I/O bit PB6 to trigger an interrupt.

When PORTxCFG=1, the alternate function is selected. The output register and buffer are disconnected. The PINS bit can still read the pin, and thus the input to the alternate function.

4.3 Input/Output Port Registers

The port control bits (OUT, OE, and PINS) are contained in the six registers shown in Figures *Figure 4-4* through *Figure 4-6*. Section 4.12, *"SFR Addressing"* explains how this basic structure is enhanced to add SFR access to the I/O pins.

The OUTn registers provide the data that drives the port pin when OE=1 **and** the pin is configured for port output. If the port pin is selected as an input (OE=0), the value stored in the corresponding OUTn bit is stored in an output latch but not used.

The OE registers control the output enables on the tri-state drivers connected to the port pins, unless the corresponding PORTnCFG bit is set to a “1.” When a PORTnCFG bit is set to a “1”, the value of the corresponding OE bit has no effect upon the port pin or the alternate function input.

When the corresponding PORTnCFG bit is “0” and OE=”1”, the corresponding value of OUTn is output to the pin.

When the corresponding PORTnCFG bit is “0” and OE=”0”, the corresponding value of OUTn is not output to the pin; it is tri-stated.

OUTA **Port A Outputs** **7F96**

b7	b6	b5	b4	b3	b2	b1	b0
OUTA7	**OUTA6**	**OUTA5**	**OUTA4**	**OUTA3**	**OUTA2**	**OUTA1**	**OUTA0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
x	x	x	x	x	x	x	x

OUTB **Port B Outputs** **7F97**

b7	b6	b5	b4	b3	b2	b1	b0
OUTB7	**OUTB6**	**OUTB5**	**OUTB4**	**OUTB3**	**OUTB2**	**OUTB1**	**OUTB0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
x	x	x	x	x	x	x	x

OUTC **Port C Outputs** **7F98**

b7	b6	b5	b4	b3	b2	b1	b0
OUTC7	**OUTC6**	**OUTC5**	**OUTC4**	**OUTC3**	**OUTC2**	**OUTC1**	**OUTC0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
x	x	x	x	x	x	x	x

OUTD **Port D Outputs** **7841**

b7	b6	b5	b4	b3	b2	b1	b0
OUTD7	**OUTD6**	**OUTD5**	**OUTD4**	**OUTD3**	**OUTD2**	**OUTD1**	**OUTD0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
x	x	x	x	x	x	x	x

OUTE **Port E Outputs** **7845**

b7	b6	b5	b4	b3	b2	b1	b0
OUTE7	**OUTE6**	**OUTE5**	**OUTE4**	**OUTE3**	**OUTE2**	**OUTE1**	**OUTE0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
x	x	x	x	x	x	x	x

Figure 4-4. Output Port Configuration Registers

PINSA **Port A Pins** **7F99**

b7	b6	b5	b4	b3	b2	b1	b0
PINA7	**PINA6**	**PINA5**	**PINA4**	**PINA3**	**PINA2**	**PINA1**	**PINA0**
R	R	R	R	R	R	R	R
x	x	x	x	x	x	x	x

PINSB **Port B Pins** **7F9A**

b7	b6	b5	b4	b3	b2	b1	b0
PINB7	**PINB6**	**PINB5**	**PINB4**	**PINB3**	**PINB2**	**PINB1**	**PINB0**
R	R	R	R	R	R	R	R
x	x	x	x	x	x	x	x

PINSC **Port C Pins** **7F9B**

b7	b6	b5	b4	b3	b2	b1	b0
PINC7	**PINC6**	**PINC5**	**PINC4**	**PINC3**	**PINC2**	**PINC1**	**PINC0**
R	R	R	R	R	R	R	R
x	x	x	x	x	x	x	x

PINSD **Port D Pins** **7842**

b7	b6	b5	b4	b3	b2	b1	b0
PIND7	**PIND6**	**PIND5**	**PIND4**	**PIND3**	**PIND2**	**PIND1**	**PIND0**
R	R	R	R	R	R	R	R
x	x	x	x	x	x	x	x

PINSE **Port E Pins** **7846**

b7	b6	b5	b4	b3	b2	b1	b0
PINE7	**PINE6**	**PINE5**	**PINE4**	**PINE3**	**PINE2**	**PINE1**	**PINE0**
R	R	R	R	R	R	R	R
x	x	x	x	x	x	x	x

Figure 4-5. PINSn Registers

The PINSn registers contain the current value of the port pins, whether they are selected as I/O ports or as alternate functions.

OEA **Port A Output Enable** **7F9C**

b7	b6	b5	b4	b3	b2	b1	b0
OEA7	**OEA6**	**OEA5**	**OEA4**	**OEA3**	**OEA2**	**OEA1**	**OEA0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

OEB **Port B Output Enable** **7F9D**

b7	b6	b5	b4	b3	b2	b1	b0
OEB7	**OEB6**	**OEB5**	**OEB4**	**OEB3**	**OEB2**	**OEB1**	**OEB0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

OEC **Port C Output Enable** **7F9E**

b7	b6	b5	b4	b3	b2	b1	b0
OEC7	**OEC6**	**OEC5**	**OEC4**	**OEC3**	**OEC2**	**OEC1**	**OEC0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

OED **Port D Output Enable** **7843**

b7	b6	b5	b4	b3	b2	b1	b0
OED7	**OED6**	**OED5**	**OED4**	**OED3**	**OED2**	**OED1**	**OED0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

OEE **Port E Output Enable** **7847**

b7	b6	b5	b4	b3	b2	b1	b0
OEE7	**OEE6**	**OEE5**	**OEE4**	**OEE3**	**OEE2**	**OEE1**	**OEE0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Figure 4-6. Output Enable Registers

EZ-USB FX ports A, B, and C have individually selectable, alternate functions for each port pin. Alternate functions, such as UART TxD and RxD, are selected on a pin-by-pin basis for these ports using control bits in registers PORTACFG, PORTBCFG, and PORTCCFG.

Although ports D and E can be used for purposes other than I/O pins, they do not have corresponding, alternate function configuration registers like ports A-C (see Section 15.23, *"PORTA and PORTC Alternate Configurations"*). Instead, their alternate functions are selected in 8-bit groups using a single-interface configuration register called IFCONFIG (see Section 15.22, *"Interface Configuration"*). Two bits, IF[1..0], select four configurations for ports D and E.

4.4 Port Configuration Tables

Table 4-1. Port A Configuration

PORTA Bit 0		
IFCONFIG.3=0		IFCONFIG.3=1
PORT-ACFG.0=0	PORT-ACFG.0=1	
Port pin PA0	**T0out**	**GSTATE[0]**
I/O	O	O

PORTA Bit 1		
IFCONFIG.3=0		IFCONFIG.3=1
PORT-ACFG.1=0	PORT-ACFG.1=1	
Port pin PA1	**T1out**	**GSTATE[1]**
I/O	O	O

PORTA Bit 2		
IFCONFIG.3=0		IFCONFIG.3=1
PORT-ACFG.2=0	PORT-ACFG.2=1	
Port pin PA2	**OE#**	**GSTATE[2]**
I/O	O	O

PORTA Bit 3	
PORT-ACFG.3=0	PORTACFG.3=1
Port pin PA3	**CS#**
I/O	O

Table 4-1. Port A Configuration

PORTA Bit 4			
PORT-ACFG.4=0	PORTACFG.4=1		
	PORTACF2.4=0	PORTACF2.4=1	
		IFCON-FIG[1..0]=10	IFCON-FIG[1..0]=11
Port pin PA4	**FWR#**	**RDY4**	**SLWR**
I/O	O	I	I

PORTA Bit 5			
PORT-ACFG.5=0	PORTACFG.5=1		
	PORTACF2.5=0	PORTACF2.5=1	
		IFCON-FIG[1..0]=10	IFCON-FIG[1..0]=11
Port pin PA5	**FRD#**	**RDY5**	**SLRD**
I/O	O	I	I

PORTA Bit 6	
PORT-ACFG.6=0	PORTACFG.6=1
Port pin PA6	**RxD0out**
I/O	O

PORTA Bit 7	
PORT-ACFG.7=0	PORTACFG.7=1
Port pin PA7	**RxD1out**
I/O	O

Table 4-2. Port B Configuration

PORTB Bit 0				
IFCONFIG[1..0]=00		IFCON-FIG[1..0]=01	IFCON-FIG[1..0]=10	IFCON-FIG[1..0]=11
PORTB-CFG.0=0	PORTB-CFG.0=1			
Port pin PB0	**T2**	**D[0]**	**GDA[0]**	**AFI[0]**
I/O	I	I/O	I/O	I/O

PORTB Bit 1				
IFCONFIG[1..0]=00		IFCON-FIG[1..0]=01	IFCON-FIG[1..0]=10	IFCON-FIG[1..0]=11
PORTB-CFG.1=0	PORTB-CFG.1=1			
Port pin PB1	**T2EX**	**D[1]**	**GDA[1]**	**AFI[1]**
I/O	I	I/O	I/O	I/O

Table 4-2. Port B Configuration

PORTB Bit 2				
IFCONFIG[1..0]=00		IFCON-FIG[1..0]=01	IFCON-FIG[1..0]=10	IFCON-FIG[1..0]=11
PORTB-CFG.2=0	PORTB-CFG.2=1			
Port pin PB2	**RxD1**	**D[2]**	**GDA[2]**	**AFI[2]**
I/O	I	I/O	I/O	I/O

PORTB Bit 3				
IFCONFIG[1..0]=00		IFCON-FIG[1..0]=01	IFCON-FIG[1..0]=10	IFCON-FIG[1..0]=11
PORTB-CFG.3=0	PORTB-CFG.3=1			
Port pin PB3	**TxD1**	**D[3]**	**GDA[3]**	**AFI[3]**
I/O	O	I/O	I/O	I/O

PORTB Bit 4				
IFCONFIG[1..0]=00		IFCON-FIG[1..0]=01	IFCON-FIG[1..0]=10	IFCON-FIG[1..0]=11
PORTB-CFG.4=0	PORTB-CFG.4=1			
Port pin PB4	**INT4**	**D[4]**	**GDA[4]**	**AFI[4]**
I/O	I	I/O	I/O	I/O

PORTB Bit 5				
IFCONFIG[1..0]=00		IFCON-FIG[1..0]=01	IFCON-FIG[1..0]=10	IFCON-FIG[1..0]=11
PORTB-CFG.5=0	PORTB-CFG.5=1			
Port pin PB5	**INT5#**	**D[5]**	**GDA[5]**	**AFI[5]**
I/O	I	I/O	I/O	I/O

PORTB Bit 6				
IFCONFIG[1..0]=00		IFCON-FIG[1..0]=01	IFCON-FIG[1..0]=10	IFCON-FIG[1..0]=11
PORTB-CFG.6=0	PORTB-CFG.6=1			
Port pin PB6	**INT6**	**D[6]**	**GDA[6]**	**AFI[6]**
I/O	I	I/O	I/O	I/O

PORTB Bit 7				
IFCONFIG[1..0]=00		IFCON-FIG[1..0]=01	IFCON-FIG[1..0]=10	IFCON-FIG[1..0]=11
PORTB-CFG.7=0	PORTB-CFG.7=1			
Port pin PB7	**T2OUT**	**D[7]**	**GDA[7]**	**AFI[7]**
I/O	O	I/O	I/O	I/O

Table 4-3. Port C Configuration

PORTC Bit 0			
PORTC-CFG.0=0	PORTCCFG.0=1		
	PORTCCF2.0=0	PORTCCF2.0=1	
		IFCON-FIG[1..0]=10	00, 01, 11 not valid
Port pin PC0	**RxD0**	**RDY0**	**X**
I/O	I	I	

PORTC Bit 1			
PORTC-CFG.1=0	PORTCCFG.1=1		
	PORTCCF2.1=0	PORTCCF2.1=1	
		IFCON-FIG[1..0]=10	00, 01, 11 not valid
Port pin PC1	**TxD0**	**RDY1**	**X**
I/O	O	I	

PORTC Bit 2	
PORTC-CFG.2=0	PORTCCFG.2=1
Port pin PC3	**INT0#**
I/O	I

PORTC Bit 3			
PORTC-CFG.3=0	PORTCCFG.3=1		
	PORTCCF2.3=0	PORTCCF2.3=1	
		IFCON-FIG[1..0]=10	00, 01, 11 not valid
Port pin PC3	**INT1#**	**RDY3**	**X**
I/O	I	I	

PORTC Bit 4			
PORTC-CFG.4=0	PORTCCFG.4=1		
	PORTCCF2.4=0	PORTCCF2.4=1	
		IFCON-FIG[1..0]=10	00, 01, 11 not valid
Port pin PC4	**T0**	**CTL1**	**X**
I/O	I	O	

Table 4-3. Port C Configuration

PORTC Bit 5			
PORTC-CFG.5=0	PORTCCFG.5=1		
	PORTCCF2.5=0	PORTCCF2.5=1	
		IFCON-FIG[1..0]=10	00, 01, 11 not valid
Port pin PC5	**T1**	**CTL3**	**X**
I/O	I	O	

PORTC Bit 6			
PORTC-CFG.6=0	PORTCCFG.6=1		
	PORTCCF2.6=0	PORTCCF2.6=1	
		IFCON-FIG[1..0]=10	00, 01, 11 not valid
Port pin PC6	**WR#**	**CTL4**	**X**
I/O	O	O	

PORTC Bit 7			
PORTC-CFG.7=0	PORTCCFG.7=1		
	PORTCCF2.7=0	PORTCCF2.7=1	
		IFCON-FIG[1..0]=10	00, 01, 11 not valid
Port pin PC7	**RD#**	**CTL5**	**X**
I/O	O	O	

Table 4-4. Port D Bits

PORTD Bits [7..0]					
IFCON-FIG[1..0]=00	IFCON-FIG[1..0]=01	IFCONFIG[1..0]=10		IFCONFIG[1..0]=11	
		IFCONFIG.2=0	IFCONFIG.2=1	IFCONFIG.2=0	IFCONFIG.2=1
Port pins PD[7..0]	**Port pins PD[7..0]**	**Port pins PD[7..0]**	**GDB[7..0]**	**Port pins PD[7..0]**	**BFI[7..0]**
I/O	I/O	I/O	I/O	I/O	I/O

Table 4-5. Port E Bits

Port E Bit 0		
IFCONFIG[1..0]=00, 01	IFCONFIG[1..0]=10	IFCONFIG[1..0]=11
Port pin PE[0]	**adr0**	**BOUTFLAG**
I/O	O	O

Port E Bit 1		
IFCONFIG[1..0]=00, 01	IFCONFIG[1..0]=10	IFCONFIG[1..0]=11
Port pin PE[1]	**adr1**	**AINFULL**
I/O	O	O

Port E Bit 2		
IFCONFIG[1..0]=00, 01	IFCONFIG[1..0]=10	IFCONFIG[1..0]=11
Port pin PE[2]	**adr2**	**BINFULL**
I/O	O	O

Port E Bit 3		
IFCONFIG[1..0]=00, 01	IFCONFIG[1..0]=10	IFCONFIG[1..0]=11
Port pin PE[3]	**adr3**	**AOUTEMTY**
I/O	O	O

Port E Bit 4		
IFCONFIG[1..0]=00, 01	IFCONFIG[1..0]=10	IFCONFIG[1..0]=11
Port pin PE[4]	**adr4**	**BOUTEMTY**
I/O	O	O

Port E Bit 5		
IFCONFIG[1..0]=00, 01	IFCONFIG[1..0]=10	IFCONFIG[1..0]=11
Port pin PE[5]	**CTL3**	**Port pin PE[5]**
I/O	O	I/O

Port E Bit 6		
IFCONFIG[1..0]=00, 01	IFCONFIG[1..0]=10	IFCONFIG[1..0]=11
Port pin PE[6]	**CTL4**	**Port pin PE[6]**
I/O	O	I/O

Port E Bit 7		
IFCONFIG[1..0]=00, 01	IFCONFIG[1..0]=10	IFCONFIG[1..0]=11
Port pin PE[7]	**CTL5**	**Port pin PE[7]**
I/O	O	I/O

4.5 I²C-Compatible Controller

The USB core contains an I²C-compatible controller for boot loading and general-purpose I²C-compatible bus interface. This controller uses the SCL (Serial Clock) and SDA (Serial Data) pins. I2C-compatible controller describes how the boot load operates at power-on to read the contents of an external serial EEPROM to determine the initial EZ-USB FX configuration. The boot loader operates automatically, while the 8051 is held in reset. The last section of this chapter describes the operating details of the boot loader.

After the boot sequence completes and the 8051 is brought out of reset, the general-purpose I²C-compatible controller is available to the 8051 for interface to external I²C-compatible devices, such as other EEPROMS, I/O chips, audio/video control chips, etc.

For I²C-compatible peripherals that support it, the EZ-USB FX I²C-compatible bus can run at 400 KHz. For compatibility, the EZ-USB FX powers-up at the 100-KHz frequency.

4.6 8051 I²C-Compatible Controller

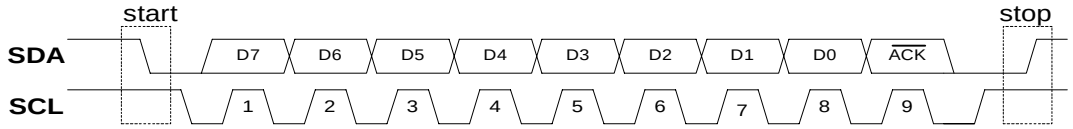


Figure 4-7. General I²C-Compatible Transfer

Figure 4-7 illustrates the waveforms for an I²C-compatible transfer. SCL and SDA are open-drain EZ-USB FX pins, which must be pulled up to Vcc with external resistors. The EZ-USB FX chip is an I²C-compatible bus master only, meaning that it synchronizes data transfers by generating clock pulses on SCL by driving low. Once the master drives SCL low, external slave devices can also drive SCL low to extend clock cycle times.

To synchronize I²C-compatible data, serial data (SDA) is permitted to change state only while SCL is low, and must be valid while SCL is high. Two exceptions to this rule are used to generate START and STOP conditions. A START condition is defined as SDA going low, while SCL is high, and a STOP condition is defined as SDA going high, while SCL is high. Data is sent MSB first. During the last bit time (clock #9 in *Figure 4-7*), the master (EZ-USB FX) floats the SDA line to allow the slave to acknowledge the transfer by pulling SDA low.

Multiple I²C-Compatible Bus Masters *— The EZ-USB FX chip acts only as an I²C-compatible bus master, never a slave. However, the 8051 can detect a second master by checking for BERR=1 (Section 4.8, "Status Bits").*

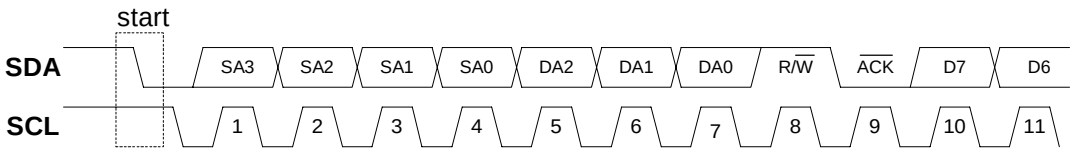


Figure 4-8. Addressing an I²C-compatible Peripheral

The first byte of an I²C-compatible bus transaction contains the address of the desired peripheral. *Figure 4-8* shows the format for this first byte, which is sometimes called a *control* byte.

A master sends the bit sequence shown in *Figure 4-8* after sending a START condition. The master uses this 9-bit sequence to select an I²C-compatible peripheral at a particular address, to establish the transfer direction (using R/W#), and to determine if the peripheral is present by testing for ACK#.

The four most significant bits SA3-SA0 are the peripheral chip's slave address. I²C-compatible devices are pre-assigned slave addresses by device type. For example, slave address 1010 is assigned to EEPROMS. The three bits DA2-DA0 usually reflect the states of I²C-compatible device address pins. Devices with three address pins can be strapped to allow eight distinct addresses for the same device type. The eighth bit (R/W#) sets the direction for the ensuing data transfer, 1 for master read, and 0 for master write. Most address transfers are followed by one or more data transfers, with the STOP condition generated after the last data byte is transferred.

In *Figure 4-8*, a READ transfer follows the address byte (at clock 8, the master sets the R/W# bit high, indicating READ). At clock 9, the peripheral device responds to its address by asserting ACK. At clock 10, the master floats SDA and issues SCL pulses to clock in SDA data supplied by this slave.

Assuming the 12-MHz crystal used by the EZ-USB FX family, the SCL frequency is 90.9 KHz, giving an I²C-compatible transfer rate of 11 microseconds per bit. Operation at four times this rate is available by setting a bit in the boot EEPROM. See Section 5.9, *"Configuration Byte 0"* for details.

I2CS **I²C-Compatible Control and Status** **7FA5**

b7	b6	b5	b4	b3	b2	b1	b0
START	**STOP**	**LASTRD**	**ID1**	**ID0**	**BERR**	**ACK**	**DONE**
R/W	R/W	R/W	R	R	R	R	R
0	0	0	x	x	0	0	0

I2DAT **I²C-Compatible Data** **7FA6**

b7	b6	b5	b4	b3	b2	b1	b0
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
x	x	x	x	x	x	x	x

Figure 4-9. I²C-compatible Registers

The 8051 uses the two registers shown in *Figure 4-9* to conduct I²C-compatible transfers. The 8051 transfers data to and from the I²C-compatible bus by writing and reading the I2DAT register. The I2CS register controls I²C-compatible transfers and reports various status conditions. The three control bits are START, STOP, and LASTRD. The remaining bits are status bits. Writing to a status bit has no effect.

4.7 Control Bits

4.7.1 START

The 8051 sets the START bit to 1 to prepare an I²C-compatible bus transfer. If START=1, the next 8051 load to I2DAT generates the start condition followed by the serialized byte of data in I2DAT. The 8051 loads data in the format shown in *Figure 4-7* after setting the START bit. The I²C-compatible controller clears the START bit during the ACK interval (clock 9 in *Figure 4-7*).

4.7.2 STOP

The 8051 sets STOP=1 to terminate an I²C-compatible bus transfer. The I²C-compatible controller clears the STOP bit after completing the STOP condition. If the 8051 sets the STOP bit during a byte transfer, the STOP condition generates immediately following the ACK phase of the byte transfer. If no byte transfer is occurring when the STOP bit is set, the STOP condition is carried out immediately on the bus. Data should not be written to I2CS or I2DAT until the STOP bit returns low. In most versions of CY7C646x3-128NC, an interrupt request is available to signal that STOP bit transmission is complete.

4.7.3 LASTRD

To read data over the I^{2}C-compatible bus, an I^{2}C-compatible master floats the SDA line and issues clock pulses on the SCL line. After every eight bits, the master drives SDA low for one clock to indicate ACK. To signal the last byte of the read transfer, the master floats SDA at ACK time to instruct the slave to stop sending. This is controlled by the 8051 by setting LASTRD=1 before reading the last byte of a read transfer. The I^{2}C-compatible controller clears the LASTRD bit at the end of the transfer (at ACK time).

Setting LASTRD does not automatically generate a STOP condition. The 8051 should also set the STOP bit at the end of a read transfer.

4.8 Status Bits

After a byte transfer, the I^{2}C-compatible controller updates the three status bits BERR, ACK, and DONE. If no STOP condition was transmitted, they are updated at ACK time. If a STOP condition was transmitted they are updated after the STOP condition is transmitted.

4.8.1 DONE

The I^{2}C-compatible controller sets this bit whenever it completes a byte transfer, right after the ACK stage. The controller also generates an I^{2}C-compatible interrupt request (8051 INT3) when it sets the DONE bit. The I^{2}C-compatible controller clears the DONE bit when the 8051 reads or writes the I2DAT register, and it clears the I^{2}C-compatible interrupt request bit whenever the 8051 reads or writes the I2CS or I2DAT register.

4.8.2 ACK

Every ninth SCL of a write transfer, the slave indicates reception of the byte by asserting ACK. The I^{2}C-compatible controller floats SDA during this time, samples the SDA line, and updates the ACK bit with the complement of the detected value. ACK=1 indicates acknowledge, and ACK=0 indicates not-acknowledge. The I^{2}C-compatible controller updates the ACK bit at the same time it sets DONE=1. The ACK bit should be ignored for read transfers on the bus.

4.8.3 BERR

This bit indicates an I^{2}C-compatible bus error. BERR=1 indicates that there was bus contention, which results when an outside device drives the bus LO when it shouldn't, or when another bus

master wins arbitration, taking control of the bus. BERR is cleared when the 8051 reads or writes the I2DAT register.

4.8.4 ID1, ID0

These bits are set by the boot loader (Section 4.11, *"I2C-Compatible Boot Loader"*) to indicate whether an 8-bit address or 16-bit address EEPROM at slave address 000 or 001 was detected at power-on. They are normally used only for debug purposes. Table 4-7 shows the encoding for these bits.

4.9 Sending I²C-Compatible Data

To send a multiple byte data record over the I²C-compatible bus, follow these steps:

1. Set the START bit.
2. Write the peripheral address and direction=0 (for write) to I2DAT.
3. Wait for DONE=1*. If BERR=1 or ACK=0, go to step 7.
4. Load I2DAT with a data byte.
5. Wait for DONE=1*. If BERR=1 or ACK=0 go to step 7.
6. Repeat steps 4 and 5 for each byte until all bytes have been transferred.
7. Set STOP=1.

* If the I²C-compatible interrupt (8051 INT3) is enabled, each “Wait for DONE=1” step can be interrupt driven, and handled by an interrupt service routine. See *Chapter 12. "EZ-USB FX Interrupts"* for more details regarding the I²C-compatible interrupt.

4.10 Receiving I²C-Compatible Data

To read a multiple-byte data record, follow these steps:

1. Set the START bit.
2. Write the peripheral address and direction=1 (for read) to I2DAT.
3. Wait for DONE=1*. If BERR=1 or ACK=0, terminate by setting STOP=1.
4. Read I2DAT and discard the data. This initiates the first burst of nine SCL pulses to clock in the first byte from the slave.
5. Wait for DONE=1*. If BERR=1, terminate by setting STOP=1.
6. Read the data from I2DAT. This initiates another read transfer.
7. Repeat steps 5 and 6 for each byte until ready to read the second-to-last byte.
8. Before reading the second-to-last I2DAT byte, set LASTRD=1.

9. Read the data from I2DAT. With LASTRD=1, this initiates the final byte read on the I²C-compatible bus.
10. Wait for DONE=1*. If BERR=1, terminate by setting STOP=1.
11. Set STOP=1.
12. Read the last byte from I2DAT immediately (the next instruction) after setting the STOP bit. This retrieves the last data byte without initiating an extra read transaction (nine more SCL pulses) on the I²C-compatible bus.

* If the I²C-compatible interrupt (8051 INT3) is enabled, each "Wait for DONE=1" step can be interrupt-driven, and handled by an interrupt service routing. See *Chapter 12. "EZ-USB FX Interrupts"* for more details regarding the I²C-compatible interrupt.

4.11 I²C-Compatible Boot Loader

When the EZ-USB FX chip comes out of reset, the EZ-USB FX boot loader checks for the presence of an EEPROM on its I²C-compatible bus. If an EEPROM is detected, the loader reads the first EEPROM byte to determine how to enumerate (specifically, whether to supply ID information from the USB core or from the EEPROM). The various enumeration modes are described in *Chapter 5. "EZ-USB FX Enumeration & ReNumeration™"*.

Prior to reading the first EEPROM byte, the boot loader must set to zero an address counter inside the EEPROM. It does this by sending a control byte (write) to select the EEPROM, followed by a zero address to set the internal EEPROM address pointer to zero. Then, it issues a control byte (read), and reads the first EEPROM byte.

The EZ-USB FX boot loader supports two I²C-compatible EEPROM types:

- EEPROMs with address A[7..4]=1010 that use an 8-bit address, (example: 24LC00, 24LC01/B, 24LC02/B).
- EEPROMs with address A[7..4]=1010 that use a 16-bit address, (example: 24AA64, 24LC128, 24AA256).

EEPROMs with densities up to 256 bytes require loading a single address byte. Larger EEPROMs require loading two address bytes.

The EZ-USB FX I²C-compatible controller needs to determine which EEPROM type is connected—one or two address bytes—so that it can properly reset the EEPROM address pointer to zero before reading the EEPROM. For the single-byte address part, it must send a single zero byte of address, and for the two-byte address part it must send two zero bytes of address.

Because there is no direct way to detect which EEPROM type—single or double address—is connected, the I²C-compatible controller uses the EEPROM address pins A2, A1, and A0 to determine whether to send out one or two bytes of address. This algorithm requires that the EEPROM

address lines are strapped as shown in Table 4-6. Single-byte-address EEPROMs are strapped to address 000 and double-byte-address EEPROMs are strapped to address 001.

Table 4-6. Strap Boot EEPROM Address Lines to These Values

Bytes	Example EEPROM	A2	A1	A0
16	24LC00*	N/A	N/A	N/A
128	24LC01	0	0	0
256	24LC02	0	0	0
4K	24LC32	0	0	1
8K	24LC64	0	0	1

* This EEPROM does not have address pins

The I^2C-compatible controller performs a three-step test at power-on to determine whether a one-byte-address or a two-byte-address EEPROM is attached. This test proceeds as follows:

1. The I^2C-compatible controller sends out a “read current address” command to I^2C-compatible sub-address 000 (10100001). If no ACK is returned, the controller proceeds to step 2. If ACK is returned, the one-byte-address device is indicated. The controller discards the data and proceeds to step 3.
2. The I^2C-compatible controller sends out a “read current address” command to I^2C-compatible sub-address 001 (10100011). If ACK is returned, the two-byte-address device is indicated. The controller discards the data and proceeds to step 3. If no ACK is returned, the controller assumes that a valid EEPROM is not connected, assumes the “No Serial EEPROM” mode, and terminates the boot load.
3. The I^2C-compatible controller resets the EEPROM address pointer to zero (using the appropriate number of address bytes), then reads the first EEPROM byte. If it does not read 0xB4 or 0xB6, the controller assumes the “No Serial EEPROM” mode. If it reads either 0xB4 or 0xB6, the controller copies the next six bytes into internal storage. If it reads 0xB6, it proceeds to load the EEPROM contents into internal RAM.

The results of this power-on test are reported in the ID1 and ID0 bits, as shown in Table 4-7.

Table 4-7. Results of Power-On I²C-Compatible Test

ID1	ID0	Meaning
0	0	No EEPROM detected
0	1	One-byte-address load EEPROM detected
1	0	Two-byte-address load EEPROM detected
1	1	Not used

Other EEPROM devices (with device address of 1010) can be attached to the I²C-compatible bus for general purpose 8051 use, as long as they are strapped for address other than 000 or 001. If a 24LC00 EEPROM is used, no other EEPROMS with device address 1010 may be used because the 24LC00 responds to all eight sub-addresses.

4.12 SFR Addressing

The 8051 architecture includes a directly-addressable bank of registers from 0x80-0xFF, called Special Function Registers or SFRs. These registers control various 8051 peripheral functions such as the timers, interrupts, and UARTs. Because they are directly addressable, they allow quick transfer of bytes in and out of the 8051 accumulator.

A portion of the 8051 SFR space is bit-addressable. The 8051 architecture assigns 256 bit addresses to individual bits in certain registers, including SFR registers with addresses ending in 0 or 8. The advantage of bit addressing is that special bit manipulation instructions can set, test, or toggle individual bits without dealing with bytes—reading a byte, modifying one bit, or writing back the byte. This bit manipulation is especially useful for I/O, when a single I/O pin needs attention.

The EZ-USB FX preserves the I/O architecture used in EZ-USB Series 2100, where I/O is controlled using memory mapped registers in external RAM space. To allow quick access to the I/O control registers, EZ-USB FX also maps the I/O control registers into SFR registers. In addition, four of the I/O control registers are bit-addressable.

Table 4-8. EZ-USB FX Special Function Registers*

	80	90	A0	B0	C0	D0	E0	F0
0	**IOA**	**IOB**	**IOC**	**IOD**	**SCON1**	PSW	ACC	B
1	SP	**EXIF**	**INT2CLR**	**IOE**	**SBUF1**			
2	DPL0	**MPAGE**	**INT4CLR**	**SOEA**				
3	DPH0			**SOEB**				
4	DPL1			**SOEC**				
5	DPH1			**SOED**				
6	DPS			**SOEE**				
7	PCON							
8	TCON	SCON0	IE	IP	T2CON	**EICON**	**EIE**	**EIP**
9	TMOD	SBUF0						
A	TL0				RCAP2L			
B	TL1				RCAP2H			
C	TH0				TL2			
D	TH1				TH2			
E	**CKCON**							
F								

* 8051 enhancements appear in bold. EZ-USB FX SFRs are shaded. Bit-addressable registers (rows 0 and 8) are highlighted.

In the standard 8051, ports 0-3 are addressed using SFRs 80, 90, A0, and B0. Because these ports are not implemented in EZ-USB FX, the SFRs are available. The EZ-USB FX chip maps the input-output data for four of its I/O ports, A-D, into these registers. Also, the I/O register for PORT E and the port output enable registers are mapped into non-bit-addressable SFRs as shown in Table 4-8.

INT2CLR and **INT4CLR** are dummy registers (no data) that provide a fast method for clearing IRQ2 and IRQ4 flags. The 8051 writes any value to these registers to clear the IRQ2 or IRQ4 interrupt request flags.

INT2 is used for all USB interrupts. INT4 is used for all slave FIFO and GPIF interrupts.

Two enable bits turn on the SFR interrupt clearing:

- INT2 SFR clearing is enabled by setting USBBAV.4=1.
- INT4 SFR clearing is enabled by setting INT4SETUP.2=1.

The two code examples (*Figure 4-10* and *Figure 4-11*) illustrate the speed advantage gained by using the INT2CLR SFR to clear a pending USB interrupt request for endpoint 6 OUT. The first example uses the EZ-USB FX method, and the second example uses the new SFR method to clear an interrupt request for bulk endpoint EP6OUT.

```
EP6OUT_ISR_A:
               push   dps
               push   dpl
               push   dph
               push   dpl1
               push   dph1
               push   acc
;
               mov    a,EXIF            ; clear INT2 (USB) IRQ flag
               clr    acc.4
               mov    EXIF,a
;
               mov    dptr,#OUT07IRQ
               mov    a,#01000000b      ; clear OUT6 IRQ bit by writing 1
               movx   @dptr,a

; Do interrupt processing here —set flags, whatever...
;
               pop    acc
               pop    dph1
               pop    dpl1
               pop    dph
               pop    dpl
               pop    dps
               reti
```

Figure 4-10. EZ-USB FX Method, sample code

Because the OUT6 interrupt request bit is in the memory-mapped register OUT07IRQ, the 8051 clears it using the data pointer and a MOVX instruction. Because this is an interrupt service routine, all registers used by the ISR must be saved and restored. It is not known at the time of the interrupt which data pointer is in use, so both of them along with the data pointer select register “dps” are pushed and later restored (popped).

Next, the INT2 request bit is cleared in EXIF.4. It is important to clear INT2 before clearing the individual source of the interrupt—in this example EP6OUT. (This is explained in *Chapter 12. "EZ-USB FX Interrupts"*). Finally, the data pointer is set to OUT07IRQ, and the bit corresponding to OUT6 is set, and written to OUT0IRQ. Writing a “1” clears the OUT6 interrupt request.

```
init:           movx    dptr,#USBBAV
                movx    a,@dptr
                setb    acc.4               ; enable the SFR-clearing feature
                movx    @dptr,a             ; for INT2
;
EP6OUT_ISR_B:
                push    acc
                mov     a,EXIF              ; clear INT2 (USB) IRQ flag
                clr     acc.4
                mov     EXIF,a
;
                mov     INT2CLR,a           ; use whatever value is in acc
;
; Do interrupt processing here
;
                pop     acc
                reti
```

Figure 4-11. SFR Method, sample code

The “init” routine should be included in general initialization code, and is executed only once. Setting bit 4 of USBBAV enables the SFR clearing feature for INT2 (but not INT4).

The ISR clears the INT2 request bit in EXIF.4, as before. But now, only one instruction is required to clear the endpoint 6-OUT IRQ, due to the fact that the SFR is directly addressable.

There are two important points about this operation:

The data in acc is *don’t care*, because the act of writing INT2CLR, and not the data written, actually clears the IRQ. Second, the particular USB interrupt cleared by this instruction is the one currently pending (the interrupt source is displayed in the INT2IVEC register).

4.13 SFR Control of PORTs A-E

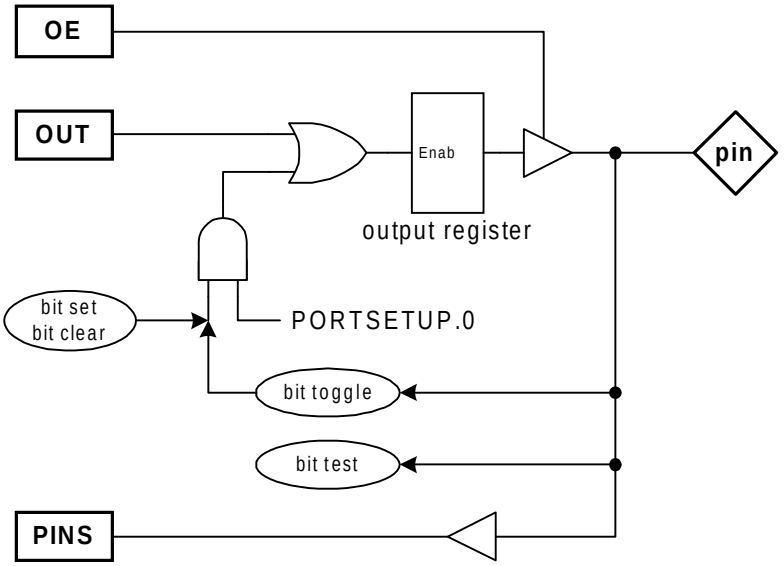


Figure 4-12. EZ-USB FX I/O Structure

Figure 4-12 shows a block diagram of the EZ-USB FX I/O structure. The signals in rectangles, OE, OUT, and PINS, represent the memory mapped register bits that access I/O bits using 8051 MOVX instructions. The ovals represent access via the SFRs. The 8051 sets a single bit, PORTSETUP.0, to enable SFR access to all of the I/O pins.

When PORTSETUP.0=1, both I/O access methods operate simultaneously. Both the MOVX method and SFR addressing method can be used to set the state of an output pin. To elaborate, the following code example sets PA0 using a MOVX instruction, clears it using a bit clear instruction, and then toggles it using a bit toggle instruction.

```
                mov     dptr,#OUTA          ; set PA0 the old way
                movx    a,@dptr             ; get value of OUTA register
                setb    acc.0               ; set bit 0
                movx    @dptr,a             ; write it back
;
                clrb    IOA.0               ; clear PA0 bit the new way
;
                cpl     IOA.0               ; complement PA0 bit the new way
```

Figure 4-13. Use MOVX to Set PA0, sample code

This simple example illustrates two important points. First, both the old (MOVX) and new (SFR) methods can be used on the same I/O bits. Second, the SFR method is much more efficient, because setting the bit using the MOVX takes nine cycles and seven bytes, while a bit set, clear, or toggle instruction takes two cycles and two bytes. In practice, there is no reason to use the first method in EZ-USB FX except for backward compatibility; the example is meant to illustrate that each method can be used independently.

The data registers for I/O ports A, B, C, and D are mapped into SFRs that are bit-addressable (0x80, 0x90, 0xA0, and 0xB0, respectively). Because the 8051 uses the rest of the SFRs, the remaining EZ-USB FX I/O registers (PORTE data and the output enables) are mapped into SFRs that are not bit-addressable. This still gives faster access to these I/O bits because direct addressing takes less time and fewer bytes than MOVX addressing, using the data pointer.

Although not shown in *Figure 4-13*, the output enables are also registered in exactly the same manner as the data register, and the SFR access is enabled using the PORTSETUP.0 bit.

The 8051 can read the state of a pin at any time by:

- Reading a PINS register using a MOVX instruction, or
- Reading the corresponding SFR register or bit.

For the bit-addressable registers IOA, IOB, IOC, or IOD, the bit test instructions (jb, jnb) may be used on individual input pins. Bit test instructions may not be used with IOE (at 0xB1) because it is not bit-addressable. However, SFR access is still faster for the IOE register than MOVX access.

The 8051 can read an I/O pin using SFRs, regardless of the state of the PORTSETUP.0 bit.

The following example code tests the state of PORTC bit 2, and jumps to two different routines depending on the result.

```
checkbit:     jb      IOC.2, process_the_one  ; jump if bit set
              jmp     process_the_zero        ; it's low
```

Figure 4-14. Test the State of PORTC, sample code

Chapter 5. EZ-USB FX Enumeration & ReNumeration™

5.1 Introduction

The EZ-USB FX chip is *soft*. 8051 code and data is stored in internal RAM, which is loaded from the host using the USB interface. Peripheral devices that use the EZ-USB FX chip can operate without ROM, EPROM, or FLASH memory, shortening production lead times and making firmware updates a breeze.

To support the soft feature, the EZ-USB FX chip enumerates automatically as a USB device *without firmware*, so the USB interface itself can download 8051 code and descriptor tables. The USB core performs this initial (power-on) enumeration and code download while the 8051 is held in RESET. This initial USB device, which supports code download, is called the “Default USB Device.”

After the code descriptor tables have been downloaded from the host to EZ-USB FX RAM, the 8051 is brought out of reset and begins executing the device code. The EZ-USB FX device enumerates again, this time as the loaded device. This patented enumeration process is called “ReNumeration™.” The EZ-USB FX chip accomplishes ReNumeration™ by electrically simulating a physical disconnection and re-connection to the USB.

An EZ-USB FX control bit called “RENUM” (ReNumerated) determines which entity, the core or the 8051, handles device requests over endpoint zero. At power-on, the RENUM bit (USBCS.1) is zero, indicating that the USB core automatically handles device requests. Once the 8051 is running, it can set RENUM to 1 to indicate that user 8051 code handles subsequent device requests using its downloaded firmware. *Chapter 9. "EZ-USB FX Endpoint Zero"* describes how the 8051 handles device requests while RENUM=1.

It is also possible for the 8051 to run with RENUM=0 and have the USB core handle certain endpoint zero requests. (See Info Box below).

This chapter deals with the various EZ-USB FX startup modes, and describes the default USB device that is created at initial enumeration.

Another Use for the Default USB Device

The Default USB Device is established at power-on to set up a USB device capable of down-loading firmware into EZ-USB FX RAM. Another useful feature of the EZ-USB FX default device is that 8051 code can be written to support the already-configured Generic USB device. Before bringing the 8051 out of reset, the USB core enables certain endpoints and reports them to the host via descriptors. By utilizing the USB default machine (by keeping RENUM=0), the 8051 can, with very little code, perform meaningful USB transfers that use these default endpoints. This accelerates the USB learning curve. To see an example of how little code is actually necessary, take a look at Section 6.11. "Polled Bulk Transfer Example."

5.2 The Default USB Device

The Default USB Device consists of a single USB configuration containing one interface (interface 0) with three alternate settings, 0, 1, and 2. The endpoints reported for this device are shown in Table 5-1. Note that alternate setting zero consumes no interrupt or isochronous bandwidth, as recommended by the USB Specification.

Table 5-1. EZ-USB FX Default Endpoints

Endpoint	Type	Alternate Setting		
		0	1	2
		Maximum Packet Size (Bytes)		
0	CTL	64	64	64
1-IN	INT	0	16	64
2-IN	BULK	0	64	64
2-OUT	BULK	0	64	64
4-IN	BULK	0	64	64
4-OUT	BULK	0	64	64
6-IN	BULK	0	64	64
6-OUT	BULK	0	64	64
8-IN	ISO	0	16	256
8-OUT	ISO	0	16	256
9-IN	ISO	0	16	16
9-OUT	ISO	0	16	16
10-IN	ISO	0	16	16
10 OUT	ISO	0	16	16

For the purpose of downloading 8051 code, the Default USB Device requires only CONTROL endpoint zero. Nevertheless, the USB default machine is enhanced to support other endpoints as shown in *Figure 5-2* (note the alternate settings 1 and 2). This enhancement is provided to allow the developer to get a head start generating USB traffic and learning the USB system. All the descriptors are handled automatically by the USB core, so the developer can immediately start writing code to transfer data over USB using these pre-configured endpoints.

When the USB core establishes the Default USB Device, it also sets the proper endpoint configuration bits to match the descriptor data supplied by the USB core. For example, bulk endpoints 2, 4, and 6 are implemented in the Default USB Device, so the USB core sets the corresponding EPVAL bits. *Chapter 6. "EZ-USB FX Bulk Transfers"* contains a detailed explanation of the EPVAL bits.

Tables 5-9 through 5-13 show the various descriptors returned to the host by the USB core when RENUM=0. These tables describe the USB endpoints defined in Table 5-1, along with other USB details. These tables should help you understand the structure of USB descriptors.

5.3 USB Core Response to EP0 Device Requests

Table 5-2 shows how the USB core responds to endpoint zero requests when RENUM=0.

Table 5-2. How the USB Core Handles EP0 Requests When RENUM=0

bRequest	Name	Action: RENUM=0
0x00	Get Status/Device	Returns two zero bytes
0x00	Get Status/Endpoint	Supplies EP Stall bit for indicated EP
0x00	Get Status/Interface	Returns two zero bytes
0x01	Clear Feature/Device	None
0x01	Clear Feature/Endpoint	Clears Stall bit for indicated EP
0x02	(reserved)	None
0x03	Set Feature/Device	None
0x03	Set Feature/Endpoint	Sets Stall bit for indicated EP
0x04	(reserved)	None
0x05	Set Address	Updates FNADD register
0x06	Get Descriptor	Supplies internal table
0x07	Set Descriptor	None
0x08	Get Configuration	Returns internal value
0x09	Set Configuration	Sets internal value
0x0A	Get Interface	Returns internal value (0-3)
0x0B	Set Interface	Sets internal value (0-3)

Table 5-2. How the USB Core Handles EP0 Requests When RENUM=0

bRequest	Name	Action: RENUM=0
0x0C	Sync Frame	None
Vendor Requests		
0xA0	Firmware Load	Upload/Download RAM
0xA1-0xAF	Reserved	Reserved by Cypress Semiconductor
all other		None

The USB host enumerates by issuing:

- Set_Address
- Get_Descriptor
- Set_Configuration (to 1)

As shown in Table 5-2, after enumeration, the USB core responds to the following host requests:

- Set or clear an endpoint stall (Set/Clear Feature_Endpoint).
- Read the stall status for an endpoint (Get_Status_Endpoint).
- Set/Read an 8-bit configuration number (Set/Get_Configuration).
- Set/Read a 2-bit interface alternate setting (Set/Get_Interface).
- Download or upload 8051 RAM.

5.3.1 Port Configuration Bits

To ensure proper operation of the default Keil Monitor, which uses SIO-1 (RXD1 and TXD1), never change the following Port Config bits from “1”:

- PORTBCFG bits 2 (RXD1) and 3 (TXD1).

To ensure the 8051 processor can access the external SRAM (including the Keil Monitor), do not change the following bits from “1”:

- PORTCCFG bits 6 (WR#) and 7 (RD#).

To ensure that no bits are unintentionally changed, all writes to the PORTxCFG registers should use a read-modify-write series of instructions.

5.4 Firmware Load

The USB Specification provides for *vendor-specific requests* to be sent over CONTROL endpoint zero. The EZ-USB FX chip uses this feature to transfer data between the host and EZ-USB FX RAM. The USB core responds to two “Firmware Load” requests, as shown in Tables 5-3 and 5-4.

Table 5-3. Firmware Download

Byte	Field	Value	Meaning	8051 Response
0	bmRequest	**0x40**	Vendor Request, OUT	*None required*
1	bRequest	**0xA0**	“Firmware Load”	
2	wValueL	**AddrL**	Starting Address	
3	wValueH	**AddrH**		
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLenghtL	**LenL**	Number of Bytes	
7	wLengthH	**LenH**		

Table 5-4. Firmware Upload

Byte	Field	Value	Meaning	8051 Response
0	bmRequest	**0xC0**	Vendor Request, IN	*None required*
1	bRequest	**0xA0**	“Firmware Load”	
2	wValueL	**AddrL**	Starting Address	
3	wValueH	**AddrH**		
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	**LenL**	Number of Bytes	
7	wLengthH	**LenH**		

These requests are always handled by the USB core (RENUM=0 or 1). The bRequest value 0xA0 is reserved by the EZ-USB FX chip. It should never be used for a vendor request. Cypress Semiconductor also reserves bRequest values 0xA1 through 0xAF. Your system should not use these bRequest values.

A host loader program typically writes 0x01 to the CPUCS register to put the 8051 into RESET, loads all or part of the EZ-USB FX RAM with 8051 code, and finally reloads the CPUCS register with 0 to take the 8051 out of RESET. The CPUCS register is the only USB register that can be written using the Firmware Download command.

Firmware loads are restricted to internal EZ-USB FX memory.

When RENUM=1 at Power-On

At power-on, the RENUM bit is normally set to zero so that the EZ-USB FX to handle device requests over CONTROL endpoint zero. This allows the core to download 8051 firmware and then reconnect as the target device.

At power-on, the USB core checks the I^2C-compatible bus for the presence of an EEPROM. If it finds one, and the first byte of the EEPROM is 0xB6, the core copies the contents of the EEPROM into internal RAM, sets the RENUM bit to 1, and un-RESETS the 8051. The 8051 wakes up ready to run the firmware in RAM. The required data format for this load module is described in Section 5.8. "Serial EEPROM Present, First Byte is 0xB6".

5.5 Enumeration Modes

When the EZ-USB FX chip comes out of RESET, the USB core decides how to enumerate based on the contents of an external EEPROM on its I^2C-compatible bus. Table 5-5 shows the choices. In Table 5-5, PID means Product ID, VID means Version ID, and DID means Device ID.

Table 5-5. USB Core Action at Power-Up

First EEPROM byte	USB Core Action
Not 0xB4 or 0xB6	Supplies descriptors, PID/VID/DID from USB Core. Sets RENUM=0.
0xB4	Supplies descriptors from USB core, PID/VID/DID from EEPROM. Sets RENUM=0.
0xB6	Loads EEPROM into EZ-USB FX RAM. Sets RENUM=1; therefore 8051 supplies descriptors, PID/VID/DID.

If no EEPROM is present, or if one is present but the first byte is neither 0xB4 nor 0xB6, the USB core enumerates using internally stored descriptor data, which contains the Cypress Semiconductor VID, PID, and DID. These ID bytes cause the host operating system to load a Cypress Semi-

conductor device driver. The USB core also establishes the *Default USB device*. This mode is only used for code development and debug.

If a serial EEPROM is attached to the I^{2}C-compatible bus and its first byte is 0xB4, the USB core enumerates with the same internally stored descriptor data as for the no-EEPROM case, but with one difference. It supplies the PID/VID/DID data from six bytes in the external EEPROM rather than from the USB core. The custom VID/PID/DID in the EEPROM causes the host operating system to load a device driver that is matched to the EEPROM VID/PID/DID. This EZ-USB FX operating mode provides a *soft* USB device using ReNumeration™

If a serial EEPROM is attached to the I^{2}C-compatible bus and its first byte is 0xB6, the USB core transfers the contents of the EEPROM into internal RAM. The USB core also sets the RENUM bit to 1 to indicate that the 8051 (and not the USB core) responds to device requests over CONTROL endpoint zero (see the Info Box on page 5-6). Therefore, all descriptor data, including VID/DID/PID values, are supplied by the 8051 firmware. The last byte loaded from the EEPROM (to the CPUCS register) releases the 8051 reset signal, allowing the EZ-USB FX chip to come up as a fully, custom device with firmware in RAM.

The following sections discuss these enumeration methods in detail.

The Other Half of the I^{2}C-Compatible Story

The EZ-USB FX I^{2}C-compatible controller serves two purposes. First, as described in this chapter, it manages the serial EEPROM interface that operates automatically at power-on to determine the enumeration method. Second, once the 8051 is up and running, the 8051 can access the I^{2}C-compatible controller for general-purpose use. This makes a wide range of standard I^{2}C-compatible peripherals available to an EZ-USB FX system.

Other I^{2}C-compatible devices can be attached to the SCL and SDA lines of the I^{2}C-compatible bus as long as there is no address conflict with the serial EEPROM described in this chapter. Chapter 4. "EZ-USB FX Input/Output" describes the general-purpose nature of the I^{2}C-compatible interface.

5.6 No Serial EEPROM

In the simplest scenario, no serial EEPROM is present on the I^{2}C-compatible bus or an EEPROM is present, but its first byte is not 0xB4 or 0xB6. In this case, descriptor data is supplied by a table internal to the USB core. The EZ-USB FX chip comes on as the *USB Default Device*, with the ID bytes shown in Table 5-6.

Pullup resistors are required on SCL/SDA, even if no device is connected. The resistors are required to allow EZ-USB FX to detect the “no-EEPROM” condition.

Table 5-6. EZ-USB FX Device Characteristics, No Serial EEPROM

Vendor ID	0x0547 (Cypress Semiconductor/ Anchor Chips)
Product ID	0x2235 (EZ-USB FX)
Device Release	0xXXYY (depends on revision)

The USB host queries the device during enumeration, reads the device descriptor, and uses the bytes described in Table 5-6 to determine which software driver to load into the operating system. This is a major USB feature — drivers are dynamically matched with devices and automatically loaded when a device is plugged in.

The “no EEPROM” scenario is the simplest configuration, and also the most limiting. This mode is used only for code development, utilizing Cypress software tools matched to the ID values in Table 5-6.

Reminder

The USB core uses the data in Table 5-6 for enumeration only if the RENUM bit is zero. If RENUM=1, enumeration data is supplied by 8051 code.

5.7 Serial EEPROM Present, First Byte is 0xB4

Table 5-7. EEPROM Data Format for “B4” Load

EEPROM Address	Contents
0	**0xB4**
1	Vendor ID (VID) L
2	Vendor ID (VID) H
3	Product ID (PID) L
4	Product ID (PID) H
5	Device ID (DID) L
6	Device ID (DID) H
7	Config 0
8	Reserved (set to 0x00)

If at power-on, the USB core detects an EEPROM connected to its I^{2}C-compatible port with the value **0xB4** at address 0, the USB core copies the Vendor ID (VID), Product ID (PID), and Device ID (DID) from the EEPROM (Table 5-7) into internal storage. The USB core then supplies these bytes to the host as part of the Get_Descriptor-Device request. (These six bytes replace only the VID/PID/DID bytes in the default USB device descriptor.) This causes a driver matched to the VID/PID/DID values in the EEPROM, instead of those in the USB core, to be loaded into the OS.

After initial enumeration, the driver downloads 8051 code and USB descriptor data into EZ-USB FX RAM and starts the 8051. The code then ReNumerates™ and comes on as the fully, custom device.

A recommended EEPROM for this application is the Microchip 24LC00, a small (5-pin SOT package) inexpensive 16-byte serial EEPROM. A 24LC01 (128 bytes) or 24LC02 (256 bytes) may be substituted for the 24LC00, but as with the 24LC00, only the first nine bytes are used.

5.8 Serial EEPROM Present, First Byte is 0xB6

If at power-on, the USB core detects an EEPROM connected to its I^{2}C-compatible port with the value **0xB6** at address 0, the USB core loads the EEPROM data into EZ-USB FX RAM. It also sets the RENUM bit to 1, causing device requests to be fielded by the 8051 instead of the USB core. The EEPROM data format is shown in Table 5-8.

Table 5-8. EEPROM Data Format for "B6" Load

EEPROM Address	Contents
0	**0xB6**
1*	Vendor ID (VID) L
2*	Vendor ID (VID) H
3*	Product ID (PID) L
4*	Product ID (PID) H
5*	Device ID (DID) L
6*	Device ID (DID) H
7	Config 0
8	Reserved (set to 0x00)
9	Length H
10	Length L
11	StartAddr H
12	StartAddr L
---	Data block

Table 5-8. EEPROM Data Format for "B6" Load

EEPROM Address	Contents
---	Length H
---	Length L
---	StartAddr H
---	StartAddr L
---	Data block

---	0x80
---	0x01
---	0x7F
---	0x92
Last	00000000

* Ignored — see Info Box below.

The first byte tells the USB core to copy EEPROM data into RAM. The next six bytes are ignored (See the Info Box below).

One or more data records follow, starting at EEPROM address 9. The maximum value of Length H is 0x03, allowing a maximum of 1,023 bytes per record. Each data record consists of a length, a starting address, and a block of data bytes. The last data record must have the MSB of its Length H byte set to 1. The last data record consists of a single-byte load to the CPUCS register at 0x7F92. Only the LSB of this byte is significant—8051RES (CPUCS.0) is set to zero to bring the 8051 out of reset.

Serial EEPROM data can be loaded into two EZ-USB FX RAM spaces only.

- 8051 program/data RAM at 0x0000-0x1B3F.
- The CPUCS register at 0x7F92 (only bit 0, 8051 RESET, is host-loadable).

VID/PID/DID in a "B6" EEPROM

Bytes 1-6 of a B6 EEPROM can be loaded with VID/PID/DID bytes if it is desired at some point to run the 8051 program with RENUM=0 (USB core handles device requests), using the EEPROM VID/PID/DID rather than the Cypress Semiconductor values built into the USB core.

5.9 Configuration Byte 0

The first configuration byte, **Config 0,** is valid for both EEPROM load formats; B4 and B6.

Config 0							
b7	b6	b5	b4	b3	b2	b1	b0
0	**0**	**0**	**0**	**0**	**48MHZ**	**CLKINV**	**400KHZ**

Figure 5-1. Configuration 0

Bit 2: **48MHZ** *24- or 48-MHz clock*

If 48MHZ=1, the 8051 operates at a clock rate of 48 MHz, and the CLKOUT pin is a 48-MHz square wave. If 48MHZ=0 the 8051 operates at a clock rate of 24 MHz, and the CLKOUT pin is a 24-MHz square wave. This bit is copied to the CPUCS Register (Bit 3, "24/48"), which is read-only to the 8051. Thus the 8051 clock rate is fixed at 24 or 48 MHz at boot time according to the EEPROM contents, and cannot be changed subsequently by the 8051.

If no EEPROM is present the default value is zero, selecting 24-MHz operation.

Bit 1: **CLKINV** *Invert CLKOUT signal*

If CLKINV=0, the CLKOUT signal is not inverted (as shown in all timing diagrams in this manual). If CLKINV=1, the CLKOUT signal is inverted. This bit is copied to the CPUCS Register Bit 2, which is read-only to the 8051. Thus, the CLKOUT polarity is set to invert or non-invert at boot time according to the EEPROM contents, and cannot be changed subsequently by the 8051.

If no EEPROM is present the default value is zero, selecting non-inverting operation.

Bit 0: **400KHZ** *High-speed I^2C-compatible Bus*

If 400KHZ=0, the I^2C-compatible bus operates at approximately 100 KHz. If 400KHZ=1, the I^2C-compatible bus operates at approximately 400 KHz. This bit is copied to the I2CCTL register bit 0, which is read-write to the 8051. Thus the I^2C-compatible bus speed is initially set by the EEPROM bit, and may be changed subsequently by the 8051.

When the EZ-USB FX comes out of RESET, the I^2C-compatible bus operates at 100 KHz mode, ensuring that a 100 KHz device can be used as the boot EEPROM.

5.10 ReNumeration™

Three EZ-USB FX control bits in the USBCS (USB Control and Status) Register control the ReNumeration™ process: DISCON, DISCOE, and RENUM.

USBCS	USB Control and Status	7FD6

b7	b6	b5	b4	b3	b2	b1	b0
WAKESRC	-	-	-	DISCON	DISCOE	RENUM	SIGRSUME
R/W	R	R	R	R/W	R/W	R/W	R/W
0	0	0	0	0	1	0	0

Figure 5-2. USB Control and Status Register

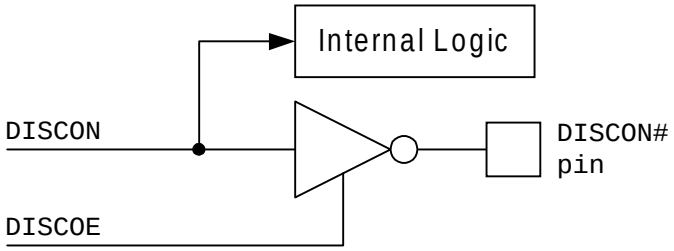


Figure 5-3. Disconnect Pin Logic

The logic for the DISCON and DISCOE bits is shown in *Figure 5-3*. To simulate a USB disconnect, the 8051 writes the value 00001010 to USBCS. This floats the DISCON# pin, and provides an internal DISCON=1 signal to the USB core that causes it to perform disconnect housekeeping.

To re-connect to USB, the 8051 writes the value 00000110 to USBCS. This presents a logic HI to the DISCON# pin, enables the output buffer, and sets the RENUM bit HI to indicate that the 8051 (and not the USB core) is now in control for USB transfers. This arrangement allows connecting the 1,500-ohm resistor directly between the DISCON# pin and the USB D+ line (*Figure 5-4*).

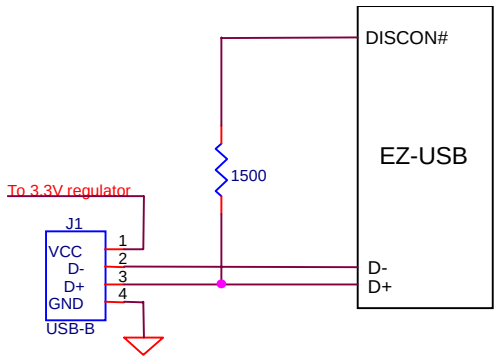


Figure 5-4. Typical Disconnect Circuit

5.11 Multiple ReNumeration™

The 8051 can ReNumerate™ anytime. One use for this capability might be to *fine tune* an isochronous endpoint's bandwidth requests by trying various descriptor values and ReNumerating.

5.12 Default Descriptor

Tables 5-9 through 5-19 show the descriptor data built into the USB core. The tables are presented in the order that the bytes are stored.

Table 5-9. USB Default Device Descriptor

Offset	Field	Description	Value
0	bLength	Length of this Descriptor = 18 bytes	12H
1	bDescriptorType	Descriptor Type = **Device**	01H
2	bcdUSB (L)	USB Specification Version 1.10 (L)	10H
3	bcdUSB (H)	USB Specification Version 1.10 (H)	01H
4	bDeviceClass	Device Class (FF is Vendor-Specific)	FFH
5	bDeviceSubClass	Device Sub-Class (FF is Vendor-Specific)	FFH
6	bDeviceProtocol	Device Protocol (FF is Vendor-Specific)	FFH
7	bMaxPacketSize0	Maximum Packet Size for EP0 = **64 bytes**	40H
8	idVendor (L)	Vendor ID (L) Cypress Semiconductor = 0547H	47H
9	idVendor (H)	Vendor ID (H)	05H
10	idProduct (L)	Product ID (L) EZ-USB FX = 2235H	35H
11	idProduct (H)	Product ID (H)	22H
12	bcdDevice (L)	Device Release Number (BCD,L) (see individual data sheet)	xxH
13	bcdDevice (H)	Device Release Number (BCD,H) (see individual data sheet)	YYH
14	iManufacturer	Manufacturer Index String = None	00H
15	iProduct	Product Index String = None	00H
16	iSerialNumber	Serial Number Index String = None	00H
17	bNumConfigurations	Number of Configurations in this Interface = 1	01H

The Device Descriptor specifies a MaxPacketSize of 64 bytes for endpoint 0, contains Cypress Semiconductor Vendor, Product and Release Number IDs, and uses no string indices. Release Number IDs (*XX* and *YY*) are found in individual Cypress Semiconductor data sheets. The USB core returns this information response to a "Get_Descriptor/Device" host request.

Table 5-10. USB Default Configuration Descriptor

Offset	Field	Description	Value
0	bLength	Length of this Descriptor = 9 bytes	09H
1	bDescriptorType	Descriptor Type = **Configuration**	02H
2	wTotalLength (L)	Total Length (L) Including Interface and Endpoint Descriptors	DAH
3	wTotalLength (H)	Total Length (H)	00H
4	bNumInterfaces	Number of Interfaces in this Configuration	01H
5	bConfigurationValue	Configuration Value Used by Set_Configuration Request to Select this Configuration	01H
6	iConfiguration	Index of String Describing this Configuration = None	00H
7	bmAttributes	Attributes - Bus-Powered, No Wakeup	80H
8	MaxPower	Maximum Power - 100 mA	32H

The configuration descriptor includes a total length field (offset 2-3) that encompasses all interface and endpoint descriptors that follow the configuration descriptor. This configuration describes a single interface (offset 4). The host selects this configuration by issuing a Set_Configuration requests specifying configuration #1 (offset 5).

Table 5-11. USB Default Interface 0, Alternate Setting 0 Descriptor

Offset	Field	Description	Value
0	bLength	Length of the Interface Descriptor	09H
1	bDescriptorType	Descriptor Type = **Interface**	04H
2	bInterfaceNumber	Zero-based Index of this Interface = **0**	00H
3	bAlternateSetting	Alternate Setting Value = **0**	00H
4	bNumEndpoints	Number of Endpoints in this Interface (Not Counting EPO) = **0**	00H
5	bInterfaceClass	Interface Class = Vendor Specific	FFH
6	bInterfaceSubClass	Interface Sub-class = Vendor Specific	FFH
7	bInterfaceProtocol	Interface Protocol = Vendor Specific	FFH
8	iInterface	Index to String Descriptor for this Interface = None	00H

Interface 0, Alternate Setting 0 describes endpoint 0 only. This setting consumes *zero bandwidth*. The interface has no string index.

Table 5-12. USB Default Interface 0, Alternate Setting 1 Descriptor

Offset	Field	Description	Value
0	bLength	Length of the Interface Descriptor	09H
1	bDescriptorType	Descriptor Type = **Interface**	04H
2	bInterfaceNumber	Zero-based Index of this Interface = **0**	00H
3	bAlternateSetting	Alternate Setting Value = **1**	01H
4	bNumEndpoints	Number of Endpoints in this Interface (Not Counting EPO) = **13**	0DH
5	bInterfaceClass	Interface Class = Vendor Specific	FFH
6	bInterfaceSubClass	Interface Sub-class = Vendor Specific	FFH
7	bInterfaceProtocol	Interface Protocol = Vendor Specific	FFH
8	iInterface	Index to String Descriptor for this Interface = None	00H

Interface 0, Alternate Setting 1 has thirteen endpoints, whose individual descriptors follow the interface descriptor. The alternate settings have no string indices.

Table 5-13. Default Interface 0, Alternate Setting 1, INT Endpoint Descriptor

Offset	Field	Description	Value
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **IN1**	81H
3	bmAttributes	XFR Type = **INT**	03H
4	wMaxPacketSize (L)	Maximum Packet Size = **16 Bytes**	10H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds = **10 ms**	0AH

Interface 0, Alternate Setting 1 has one interrupt endpoint, IN1, which has a maximum packet size of 16 and a polling interval of 10 ms.

Table 5-14. Default Interface 0, Alternate Setting 1, Bulk Endpoint Descriptors

Offset	Field	Description	Value
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **IN2**	82H
3	bmAttributes	XFR Type = **BULK**	02H
4	wMaxPacketSize (L)	Maximum Packet Size = **64 Bytes**	40H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds	00H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **OUT2**	02H
3	bmAttributes	XFR Type = **BULK**	02H
4	wMaxPacketSize (L)	Maximum Packet Size = **64 Bytes**	40H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds	00H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **IN4**	84H
3	bmAttributes	XFR Type = **BULK**	02H
4	wMaxPacketSize (L)	Maximum Packet Size = **64 Bytes**	40H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds	00H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **OUT4**	04H
3	bmAttributes	XFR Type = **BULK**	02H
4	wMaxPacketSize (L)	Maximum Packet Size = **64 Bytes**	40H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds	00H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **IN6**	86H
3	bmAttributes	XFR Type = **BULK**	02H
4	wMaxPacketSize (L)	Maximum Packet Size = **64 Bytes**	40H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds	00H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **OUT6**	06H
3	bmAttributes	XFR Type = **BULK**	02H
4	wMaxPacketSize (L)	Maximum Packet Size = **64 Bytes**	40H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds	00H

Interface 0, Alternate Setting 1 has six bulk endpoints with max packet sizes of 64 bytes. Even numbered endpoints were chosen to allow endpoint pairing. For more on endpoint pairing, see *Chapter 6. "EZ-USB FX Bulk Transfers"*.

Table 5-15. Default Interface 0, Alternate Setting 1, ISO Endpoint Descriptors

Offset	Field	Description	Value
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **IN8**	88H
3	bmAttributes	XFR Type = **ISO**	01H
4	wMaxPacketSize (L)	Maximum Packet Size = **16 Bytes**	10H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds (1 for iso)	01H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **OUT8**	08H
3	bmAttributes	XFR Type = **ISO**	01H
4	wMaxPacketSize (L)	Maximum Packet Size = **16 Bytes**	10H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds (1 for iso)	01H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **IN9**	89H
3	bmAttributes	XFR Type = **ISO**	01H
4	wMaxPacketSize (L)	Maximum Packet Size = **16 Bytes**	10H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds (1 for iso)	01H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **OUT9**	09H
3	bmAttributes	XFR Type = **ISO**	01H
4	wMaxPacketSize (L)	Maximum Packet Size = **16 Bytes**	10H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds (1 for iso)	01H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **IN10**	8AH
3	bmAttributes	XFR Type = **ISO**	01H
4	wMaxPacketSize (L)	Maximum Packet Size = **16 Bytes**	10H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds (1 for iso)	01H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **OUT10**	0AH
3	bmAttributes	XFR Type = **ISO**	01H
4	wMaxPacketSize (L)	Maximum Packet Size = **16 Bytes**	10H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds (1 for iso)	01H

Interface 0, Alternate Setting 1 has six isochronous endpoints with maximum packet sizes of 16 bytes. This is a *low bandwidth* setting.

Table 5-16. USB Default Interface 0, Alternate Setting 2 Descriptor

Offset	Field	Description	Value
0	bLength	Length of the Interface Descriptor	09H
1	bDescriptor Type	Descriptor Type = **Interface**	04H
2	bInterfaceNum-ber	Zero-based Index of this Interface = **0**	00H
3	bAlternateSetting	Alternate Setting Value = **2**	02H
4	bNumEndpoints	Number of Endpoints in this Interface (Not Counting EPO) = **13**	0DH
5	bInterfaceClass	Interface Class = Vendor Specific	FFH
6	bInterfaceSub-Class	Interface Sub-class = Vendor Specific	FFH
7	bInterfaceProto-col	Interface Protocol = Vendor Specific	FFH
8	iInterface	Index to String Descriptor for this Interface = None	00H

Interface 0, Alternate Setting 2 has thirteen endpoints, whose individual descriptors follow the interface descriptor. Alternate Setting 2 differs from Alternate Setting 1 in the maximum packet sizes of its interrupt endpoint and two of its isochronous endpoints (EP8IN and EP8OUT).

Table 5-17. Default Interface 0, Alternate Setting 1, INT Endpoint Descriptor

Offset	Field	Description	Value
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **IN1**	81H
3	bmAttributes	XFR Type = **INT**	03H
4	wMaxPacketSize (L)	Maximum Packet Size = **64 Bytes**	40H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds = **10 ms**	0AH

Alternate Setting 2 for the Interrupt 1-IN increases the maximum packet size for the interrupt endpoint to 64.

Table 5-18. Default Interface 0, Alternate Setting 2, Bulk Endpoint Descriptors

Offset	Field	Description	Value
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptor Type	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **IN2**	82H
3	bmAttributes	XFR Type = **BULK**	02H
4	wMaxPacketSize (L)	Maximum Packet Size = **64 Bytes**	40H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds	00H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **OUT2**	02H
3	bmAttributes	XFR Type = **BULK**	02H
4	wMaxPacketSize (L)	Maximum Packet Size = **64 Bytes**	40H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds	00H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **IN4**	84H
3	bmAttributes	XFR Type = **BULK**	02H
4	wMaxPacketSize (L)	Maximum Packet Size = **64 Bytes**	40H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds	00H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **OUT4**	04H
3	bmAttributes	XFR Type = **ISO**	02H
4	wMaxPacketSize (L)	Maximum Packet Size = **64 Bytes**	40H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds	00H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **IN6**	86H
3	bmAttributes	XFR Type = **BULK**	02H
4	wMaxPacketSize (L)	Maximum Packet Size = **64 Bytes**	40H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds	00H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **OUT6**	06H
3	bmAttributes	XFR Type = **BULK**	02H
4	wMaxPacketSize (L)	Maximum Packet Size = **64 Bytes**	40H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds	00H

The bulk endpoints for Alternate Setting 2 are identical to Alternate Setting 1.

Table 5-19. Default Interface 0, Alternate Setting 2, ISO Endpoint Descriptors

Offset	Field	Description	Value
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **IN8**	88H
3	bmAttributes	XFR Type = **ISO**	01H
4	wMaxPacketSize (L)	Maximum Packet Size = **256 Bytes**	00H
5	wMaxPacketSize (H)	Maximum Packet Size - High	01H
6	bInterval	Polling Interval in Milliseconds (1 for iso)	01H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **OUT8**	08H
3	bmAttributes	XFR Type = **ISO**	01H
4	wMaxPacketSize (L)	Maximum Packet Size = **256 Bytes**	00H
5	wMaxPacketSize (H)	Maximum Packet Size - High	01H
6	bInterval	Polling Interval in Milliseconds (1 for iso)	01H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **IN9**	89H
3	bmAttributes	XFR Type = **ISO**	01H
4	wMaxPacketSize (L)	Maximum Packet Size = **16 Bytes**	10H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds (1 for iso)	01H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **OUT9**	09H
3	bmAttributes	XFR Type = **ISO**	01H
4	wMaxPacketSize (L)	Maximum Packet Size = **16 Bytes**	10H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds (1 for iso)	01H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **IN10**	8AH
3	bmAttributes	XFR Type = **ISO**	01H
4	wMaxPacketSize (L)	Maximum Packet Size = **16 Bytes**	10H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds (1 for iso)	01H
0	bLength	Length of this Endpoint Descriptor	07H
1	bDescriptorType	Descriptor Type = **Endpoint**	05H
2	bEndpointAddress	Endpoint Direction (1 is in) and Address = **OUT10**	0AH
3	bmAttributes	XFR Type = **ISO**	01H
4	wMaxPacketSize (L)	Maximum Packet Size = **16 Bytes**	10H
5	wMaxPacketSize (H)	Maximum Packet Size - High	00H
6	bInterval	Polling Interval in Milliseconds (1 for iso)	01H

The only differences between Alternate Settings 1 and 2 are the maximum packet sizes for EP8IN and EP8OUT. This is a *high-bandwidth* setting.

Chapter 6. EZ-USB FX Bulk Transfers

6.1 Introduction

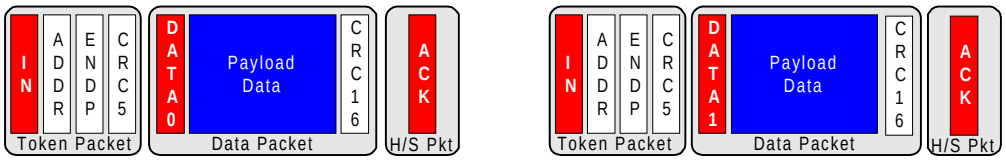


Figure 6-1. Two BULK Transfers, IN and OUT

EZ-USB FX provides sixteen endpoints for BULK, CONTROL, and INTERRUPT transfers, numbered 0-7 as shown in Table 6-1 This chapter describes BULK and INTERRUPT transfers. INTERRUPT transfers are a special case of BULK transfers. EZ-USB FX CONTROL endpoint zero is described in *Chapter 9. "EZ-USB FX Endpoint Zero".*

Table 6-1. EZ-USB FX Bulk, Control, and Interrupt Endpoints

Endpoint	Direction	Type	Size
0	Bidir	Control	64/64
1	IN	Bulk/Int	64
1	OUT	Bulk/Int	64
2	IN	Bulk/Int	64
2	OUT	Bulk/Int	64
3	IN	Bulk/Int	64
3	OUT	Bulk/Int	64
4	IN	Bulk/Int	64
4	OUT	Bulk/Int	64
5	IN	Bulk/Int	64
5	OUT	Bulk/Int	64
6	IN	Bulk/Int	64
6	OUT	Bulk/Int	64
7	IN	Bulk/Int	64
7	OUT	Bulk/Int	64

The USB Specification allows maximum packet sizes of 8, 16, 32, or 64 bytes for bulk data, and 1 - 64 bytes for interrupt data. EZ-USB FX provides the maximum 64 bytes of buffer space for each of

its sixteen endpoints: 0-7 IN and 0-7 OUT. Six of the bulk endpoints, 2-IN, 4-IN, 6-IN, 2-OUT, 4-OUT, and 6-OUT may be paired with the next consecutively numbered endpoint to provide double-buffering. This allows one data packet to be serviced by the 8051, while another is in transit over USB. Six *endpoint pairing bits* (USBPAIR Register) control double-buffering.

The 8051 sets fourteen *endpoint valid bits* (IN07VAL, OUT07VAL Registers) at initialization time to tell the USB core which endpoints are active. The default CONTROL endpoint zero is always valid.

Bulk data appears in RAM. Each bulk endpoint has a reserved 64-byte RAM space, a 7-bit count register, and a 2-bit control and status (CS) register. The 8051 can read one bit of the CS Register to determine *endpoint busy*, and write the other to force an endpoint STALL condition.

The 8051 should never read or write an endpoint buffer or byte count register while the endpoint's busy bit is set.

When an endpoint becomes ready for 8051 service, the USB core sets an interrupt request bit. The EZ-USB FX vectored interrupt system separates the interrupt requests by endpoint to automatically transfer control to the ISR (Interrupt Service Routine) for the endpoint requiring service. *Chapter 12. "EZ-USB FX Interrupts"* fully describes this mechanism.

Figure 6-2 illustrates the registers and bits associated with bulk transfers.

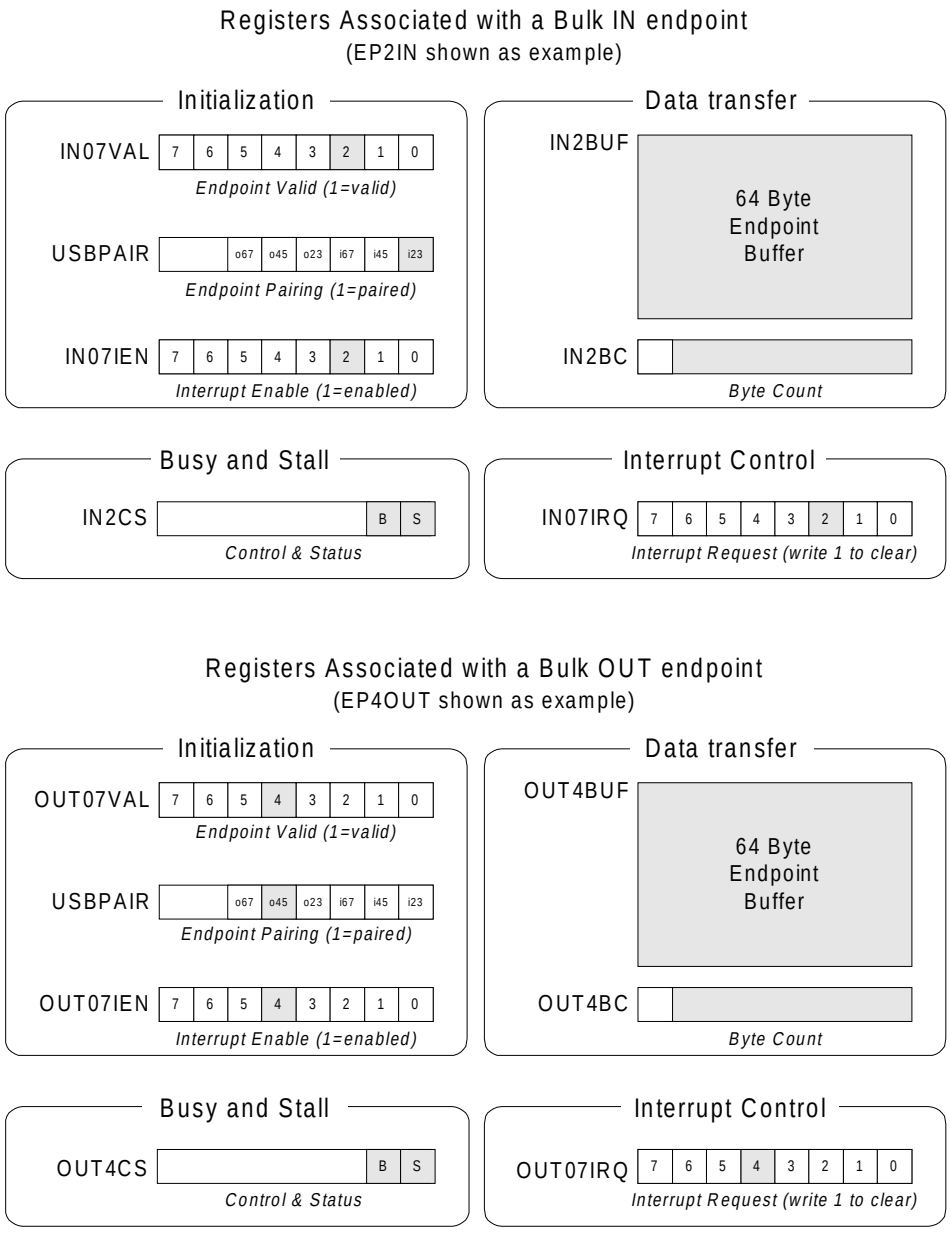


Figure 6-2. Registers Associated with Bulk Endpoints

6.2 Bulk IN Transfers

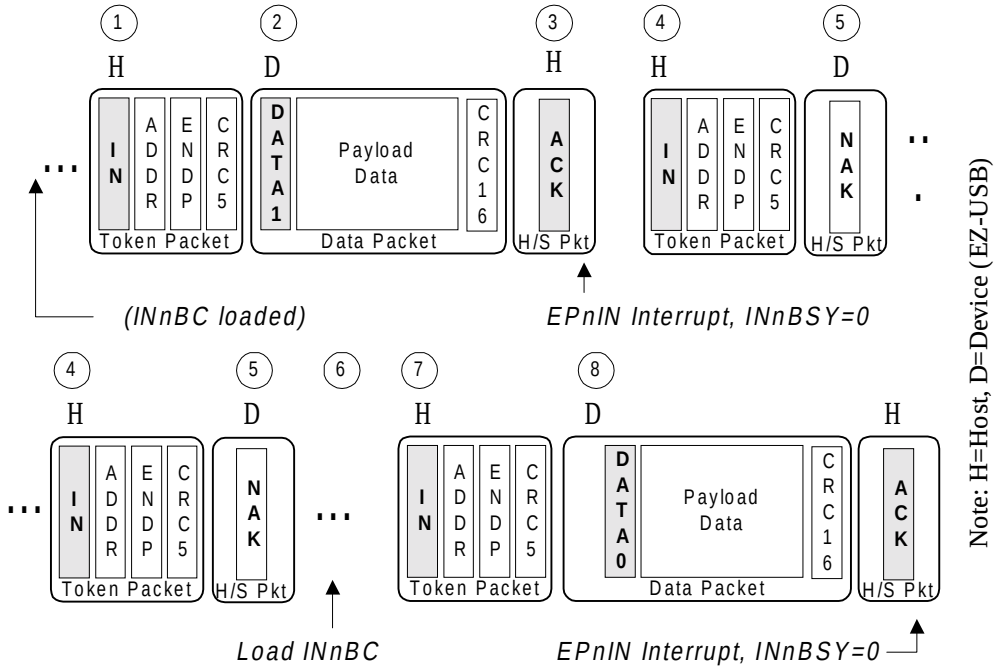


Figure 6-3. Anatomy of a Bulk IN Transfer

USB bulk *IN* data travels from device to host. The host requests an IN transfer by issuing an IN token to the USB core, which responds with data when it is ready. The 8051 indicates *ready* by loading the endpoint's byte count register. If the USB core receives an IN token for an endpoint that is not ready, it responds to the IN token with a *NAK* handshake.

In the bulk IN transfer illustrated in *Figure 6-3*, the 8051 has previously loaded an endpoint buffer with a data packet, and then loaded the endpoint's byte count register with the number of bytes in the packet to arm the next IN transfer. This sets the endpoint's BUSY Bit. The host issues an IN token (1), to which the USB core responds by transmitting the data in the IN endpoint buffer (2). When the host issues an ACK (3), indicating that the data has been received error-free, the USB core clears the endpoint's BUSY Bit and sets its interrupt request bit. This notifies the 8051 that the endpoint buffer is empty. If this is a multi-packet transfer, the host then issues another IN token to get the next packet.

If the second IN token (4) arrives before the 8051 has had time to fill the endpoint buffer, the EZ USB core issues a NAK handshake, indicating *busy* (5). The host continues to send IN tokens (4) and (7) until the data is ready. Eventually, the 8051 fills the endpoint buffer with data, and then

loads the endpoint's byte count register (INnBC) with the number of bytes in the packet (6). Loading the byte count re-arms the given endpoint. When the next IN token arrives (7) the USB core transfers the next data packet (8).

6.3 Interrupt Transfers

Interrupt transfers are handled just like bulk transfers.

The only difference between a bulk endpoint and an interrupt endpoint exists in the endpoint descriptor, where the endpoint is identified as type *interrupt*, and a *polling interval* is specified. The polling interval determines how often the USB host issues IN/OUT tokens to the interrupt endpoint.

6.4 EZ-USB FX Bulk IN Example

Suppose 220 bytes are to be transferred to the host using endpoint 6-IN. Further assume that MaxPacketSize of 64 bytes for endpoint 6-IN has been reported to the host during enumeration. Because the total transfer size exceeds the maximum packet size, the 8051 divides the 220-byte transfer into four transfers of 64, 64, 64, and 28 bytes.

After loading the first 64 bytes into IN6BUF (at 0x7C00), the 8051 loads the byte count register IN6BC with the value 64. Writing the byte count register instructs the EZ-USB core to respond to the next host IN token by transmitting the 64 bytes in the buffer. Until the byte count register is loaded to *arm* the IN transfer, any IN tokens issued by the host are answered by EZ-USB FX with NAK (Not-Acknowledge) tokens, telling the USB host that the endpoint is not yet ready with data. The host continues to issue IN tokens to endpoint 6-IN until data is ready for transfer—whereupon the USB core replaces NAKs with valid data.

When the 8051 initiates an IN transfer by loading the endpoint's byte count register, the EZ-USB core sets a busy bit to instruct the 8051 to hold off loading IN6BUF until the USB transfer is finished. When the IN transfer is complete and successfully acknowledged, the EZ-USB core resets the endpoint 6-IN busy bit and generates an endpoint 6-IN interrupt request. If the endpoint 6-IN interrupt is enabled, program control automatically vectors to the data transfer routine for further action (Autovectoring is enabled by setting AVEN=1. Refer to *Chapter 12. "EZ-USB FX Interrupts"*).

The 8051 now loads the next 64 bytes into IN6BUF and then loads the EPINBC Register with 64 for the next two transfers. For the last portion of the transfer, the 8051 loads the final 28 bytes into IN6BUF, and loads IN6BC with 28. This completes the transfer.

Initialization

When the EZ-USB FX chip comes out of RESET, or when the USB host issues a bus reset, the EZ-USB core unarms IN endpoint 1-7 by setting their busy bits to 0. Any IN transfer requests are NAKd until the 8051 loads the appropriate INxBC Register(s). The endpoint valid bits are not affected by an 8051 reset or a USB reset. Chapter 13. "EZ-USB FX Resets" describes the various reset conditions in detail.

The EZ-USB core takes care of USB housekeeping chores, such as handshake verification. When an endpoint 6-IN interrupt occurs, the user is assured that the data loaded by the 8051 into the endpoint buffer was received error-free by the host. The EZ-USB core automatically checks the handshake information from the host and re-transmits the data, if the host indicates an error by not ACKing.

6.5 Bulk OUT Transfers

USB bulk *OUT* data travels from host to device. The host requests an OUT transfer by issuing an OUT token to EZ-USB FX, followed by a packet of data. The USB core then responds with an ACK, if it correctly received the data. If the endpoint buffer is not ready to accept data, the USB core discards the host's OUT data and returns a NAK token, indicating "not ready." In response, the host continues to send OUT tokens *and data* to the endpoint until the USB core responds with an ACK.

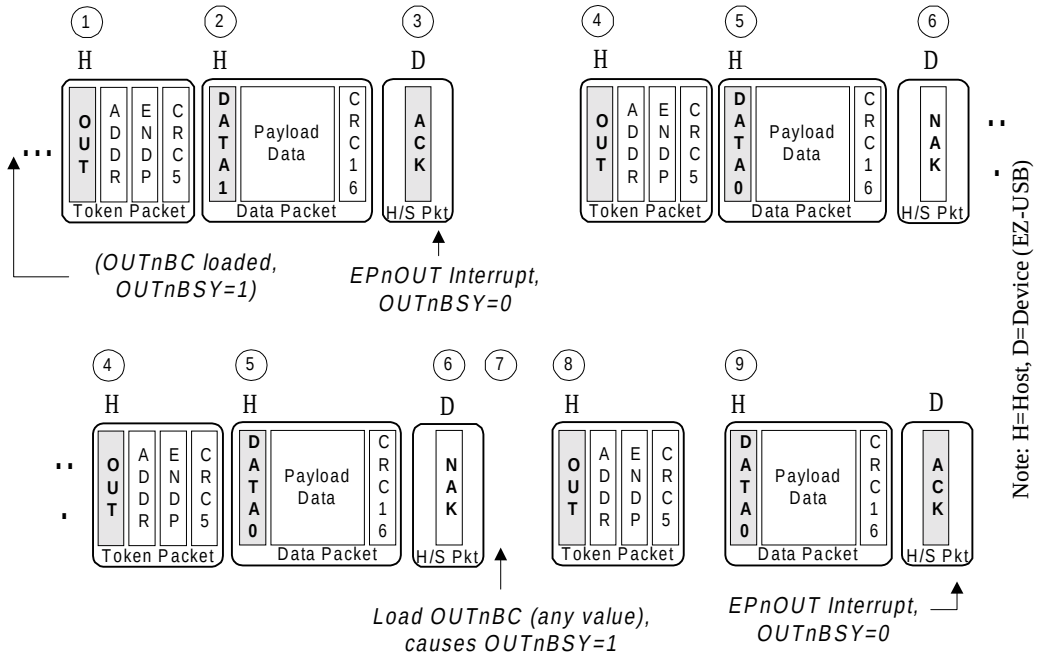


Figure 6-4. Anatomy of a Bulk OUT Transfer

Each EZ-USB FX bulk OUT endpoint has a byte count register, which serves two purposes. The 8051 *reads* the byte count register to determine how many bytes were received during the last OUT transfer from the host. The 8051 *writes* the byte count register (with any value) to tell the USB core that is has finished reading bytes from the buffer, making the buffer available to accept the next OUT transfer. The OUT endpoints come up (after reset) *armed*, so the byte count register writes are required only for OUT transfers after the first one.

In the bulk OUT transfer illustrated in *Figure 6-4*, the 8051 has previously loaded the endpoint's byte count register with any value to arm receipt of the next OUT transfer. Loading the byte count register causes the EZ-USB core to set the OUT endpoint's busy bit to 1, indicating that the 8051 should not use the endpoint's buffer.

The host issues an OUT token (1), followed by a packet of data (2), which the USB core acknowledges, clears the endpoint's busy bit and generates an interrupt request (3). This notifies the 8051 that the endpoint buffer contains valid USB data. The 8051 reads the endpoint's byte count register to find out how many bytes were sent in the packet, and transfers that many bytes out of the endpoint buffer.

In a multi-packet transfer, the host then issues another OUT token (4) along with the next data packet (5). If the 8051 has not finished emptying the endpoint buffer, the EZ-USB FX host issues a NAK, indicating *busy* (6). The data at (5) is shaded to indicate that the USB core discards it, and does not over-write the data in the endpoint's OUT buffer.

The host continues to send OUT tokens (4, 5, and 6) that are greeted by NAKs until the buffer is ready. Eventually, the 8051 empties the endpoint buffer data, and then loads the endpoint's byte count register (7) with any value to re-arm the USB core. Once armed and when the next OUT token arrives (8) the USB core accepts the next data packet (9).

Initializing OUT Endpoints

When the EZ-USB FX chip comes out of reset, or when the USB host issues a bus reset, the USB core arms OUT endpoints 1-7 by setting their busy bits to 1. Therefore, they are initially ready to accept one OUT transfer from the host. Subsequent OUT transfers are NAKd until the appropriate OUTnBC Register is loaded to re-arm the endpoint.

The EZ-USB core takes care of USB housekeeping chores such as CRC checks and data toggle PIDs. When an endpoint 6-OUT interrupt occurs and the busy bit is cleared, the user is assured that the data in the endpoint buffer was received error-free from the host. The USB core automatically checks for errors, and requests the host to re-transmit data if it detects any errors using the built-in USB error checking mechanisms (CRC checks and data toggles).

6.6 Endpoint Pairing

Table 6-2. Endpoint Pairing Bits (in the USB PAIR Register)

Bit	5	4	3	2	1	0
Name	**PR6OUT**	**PR4OUT**	**PR2OUT**	**PR6IN**	**PR4IN**	**PR2IN**
Paired	6 OUT	4 OUT	2 OUT	6 IN	4 IN	2 IN
Endpoints	7 OUT	5 OUT	3 OUT	7 IN	5 IN	3 IN

The 8051 sets endpoint pairing bits to 1 to enable double-buffering of the bulk endpoint buffers. With double-buffering enabled, the 8051 can operate on one data packet while another is being transferred over USB. The endpoint busy and interrupt request bits function identically, so the 8051 code requires little code modification to support double-buffering.

When an endpoint is paired, the 8051 uses only the even-numbered endpoint of the pair. The 8051 should not use the paired odd endpoint. For example, suppose it is desired to use endpoint 2-IN as a double-buffered endpoint. This pairs the IN2BUF and IN3BUF buffers, although the 8051 accesses the IN2BUF buffer only. The 8051 sets PR2IN=1 (in the USBPAIR Register) to enable pairing; sets IN2VAL=1 (in the IN07VAL Register) to make the endpoint valid; and then uses the IN2BUF buffer for all data transfers. The 8051 should not write the IN3VAL Bit, enable IN3 interrupts, access the EP3IN buffer, or load the IN3BC byte count register.

6.7 Paired IN Endpoint Status

INnBSY=1 indicates that *both* endpoint buffers are in use, and the 8051 should not load new IN data into the endpoint buffer. When INnBSY=0, either one or both of the buffers is available for loading by the 8051. The 8051 can keep an internal count that increments on EPnIN interrupts and decrements on byte count loads to determine whether one or two buffers are free. Or, the 8051 can simply check for INnBSY=0 after loading a buffer (and loading its byte count register to re-arm the endpoint) to determine if the other buffer is free.

If an IN endpoint is paired and it is desired to clear the busy bit for that endpoint, do the following: (a) write any value to the even endpoint's byte count register twice, and (b) clear the busy bit for both endpoints in the pair. This is the only code difference between paired and unpaired use of an IN endpoint.

A bulk IN endpoint interrupt request is generated whenever a packet is successfully transmitted over USB. The interrupt request is independent of the busy bit. If both buffers are filled and one is sent, the busy bit transitions from 1 to 0. If one buffer is filled and then sent, the busy bit starts and remains at 0. In either case, an interrupt request is generated to tell the 8051 that a buffer is free.

6.8 Paired OUT Endpoint Status

OUTnBSY=1 indicates that both endpoint buffers are empty, and no data is available to the 8051. When OUTnBSY=0, either one or both of the buffers holds USB OUT data. The 8051 can keep an internal count that increments on EPnOUT interrupts and decrements on byte count loads to determine whether one or two buffers contain data. Or, the 8051 can simply check for OUTnBSY=0 after unloading a buffer (and loading its byte count register to re-arm the endpoint) to determine if the *other* buffer contains data.

6.9 Reusing Bulk Buffer Memory

Table 6-3. EZ-USB FX Endpoint 0-7 Buffer Addresses

Endpoint Buffer	Address	Mirrored
IN0BUF	7F00-7F3F	1F00-1F3F
OUT0BUF	7EC0-7EFF	1EC0-1EFF
IN1BUF	7E80-7EBF	1E80-1EBF
OUT1BUF	7E40-7E7F	1E40-1E7F
IN2BUF	7E00-7E3F	1E00-1E3F
OUT2BUF	7DC0-7DFF	1DC0-1DFF
IN3BUF	7D80-7DBF	1D80-1DBF
OUT3BUF	7D40-7D7F	1D40-1D7F
IN4BUF	7D00-7D3F	1D00-1D3F
OUT4BUF	7CC0-7CFF	1CC0-1CFF
IN5BUF	7C80-7CBF	1C80-1CBF
OUT5BUF	7C40-7C7F	1C40-1C7F
IN6BUF	7C00-7C3F	1C00-1C3F
OUT6BUF	7BC0-7BFF	1BC0-1BFF
IN7BUF	7B80-7BBF	1B80-1BBF
OUT7BUF	7B40-7B7F	1B40-1B7F

Table 6-3 shows the RAM locations for the sixteen 64-byte buffers for endpoints 0-7 IN and OUT. These buffers are positioned at the bottom of the EZ-USB FX register space so that any buffers not used for endpoints can be reclaimed as general purpose data RAM. The top of memory for the 8-KB EZ-USB FX part is at 0x1B3F. However, if the endpoints are allocated in ascending order, starting with the lowest numbered endpoints, the higher numbered unused endpoints can effectively move the top of memory to utilize the unused endpoint buffer RAM as data memory. For example, an application that uses endpoint 1-IN, 2-IN/OUT (paired), 4-IN and 4-OUT can use 0x1B40-0x1CBF as data memory. *Chapter 3. "EZ-USB FX Memory"* provides full details of the EZ-USB FX memory map.

Uploads or Downloads to unused bulk memory can be done only at the Mirrored (low) addresses shown in Table 6-3.

6.10 Data Toggle Control

The EZ-USB core automatically maintains the data toggle bits during bulk, control and interrupt transfers. As explained in *Chapter 1. "Introducing EZ-USB FX"*, the toggle bits are used to detect certain transmission errors so that erroneous data can be re-sent.

In certain circumstances, the host resets its data toggle to “DATA0”:

- After sending a Clear_Feature: Endpoint Stall request to an endpoint.
- After setting a new interface.
- After selecting a new alternate setting.

In these cases, the 8051 can directly clear the data toggle for each of the bulk/interrupt/control endpoints, using the TOGCTL Register (*Figure 6-5*).

TOGCTL **Data Toggle Control** **7FD7**

b7	b6	b5	b4	b3	b2	b1	b0
Q	**S**	**R**	**IO**	**0**	**EP2**	**EP1**	**EP0**
R	R/W	R/W	R/W	R/W	R/W	R/W	R/W
x	x	x	x	x	x	x	x

Figure 6-5. Bulk Endpoint Toggle Control

The I/O bit selects the endpoint direction (1=IN, 0=OUT), and the EP2-EP1-EP0 Bits select the endpoint number. The *Q* Bit, which is read-only, indicates the state of the data toggle for the selected endpoint. Writing R=1 sets the data toggle to DATA0, and writing S=1 sets the data toggle to DATA1.

Currently, there appears to be no reason to set a data toggle to DATA1. The S Bit is provided for generality.

To clear an endpoint's data toggle, the 8051 performs the following sequence:

1. Selects the endpoint by writing the value 000D0EEE binary to the TOGCTL Register, where D is the direction and EEE is the endpoint number.
2. Clears the toggle bit by writing the value 001D0EEE binary to the TOGCTL Register.

After Step 1, the 8051 may read the state of the data toggle by reading the TOGCTL Register checking Bit 7.

6.11 Polled Bulk Transfer Example

The following code sample illustrates the EZ-USB FX registers used for a simple bulk transfer. In this example, 8051 Register R1 keeps track of the number of endpoint 2-IN transfers and Register R2 keeps track of the number of endpoint 2-OUT transfers (mod-256). Every endpoint 2-IN transfer consists of 64 bytes of a decrementing count, with the first byte replaced by the number of IN transfers and the second byte replaced by the number of OUT transfers.

```
1  start:         mov       SP,#STACK-1         ; set stack
2                 mov       dptr,#IN2BUF        ; fill EP2IN buffer with
3                 mov       r7,#64              ; decrementing counter
4  fill:          mov       a,r7
5                 movx      @dptr,a
6                 inc       dptr
7                 djnz      r7,fill
8  ;
9                 mov       r1,#0               ; r1 is IN token counter
10                mov       r2,#0               ; r2 is OUT token counter
11                mov       dptr,#IN2BC         ; Point to EP2 Byte Count Register
12                mov       a,#40h              ; 64-byte transfer
13                movx      @dptr,a             ; arm the IN2 transfer
14 ;
15 loop:          mov       dptr,#IN2CS         ; poll the EP2-IN Status
16 movx           a,@dptr
17                jnb       acc.1,serviceIN2    ; not busy--service endpoint
18                mov       dptr,#OUT2CS
19                movx      a,@dptr
20                jb        acc.1,loop          ; EP2OUT is busy--keep looping
21 ;
22 serviceOUT2:
23                inc       r2                  ; OUT packet counter
24                mov       dptr,#OUT2BC        ; load byte count Register to re-arm
25                movx      @dptr,a             ; (any value)
26                sjmp      loop
27 ;
28 serviceIN2:
29                inc       r1                  ; IN packet counter
30                mov       dptr,#IN2BUF        ; update the first data byte
31                mov       a,r1                ; in EP2IN buffer
32                movx      @dptr,a
33                inc       dptr                ; second byte in buffer
```

```
34                mov       a,r2                ; get number of OUT packets
35                movx      @dptr,a
36                mov       dptr,#IN2BC         ; point to EP2IN Byte Count Register
37                mov       a,#40h
38                movx      @dptr,a             ; load bc=64 to re-arm IN2
39                sjmp      loop
40  ;
41                END
```

Figure 6-6. Example Code for a Simple (Polled) BULK Transfer

The code at lines 2-7 fills the endpoint 2-IN buffer with 64 bytes of a decrementing count. Two 8-bit counts are initialized to zero at lines 9 and 10. An endpoint 2-IN transfer is *armed* at lines 11-13, which load the endpoint 2-IN byte count register IN2BC with 64. Then, the program enters a polling loop at lines 15-20, where it checks two flags for endpoint 2 servicing. Lines 15-17 check the endpoint 2-IN busy bit in IN2CS Bit 1. Lines 18-20 check the endpoint 2-OUT busy bit in OUT2CS Bit 1. When busy=1, the EZ-USB core is currently using the endpoint buffers, and the 8051 should not access them. When busy=0, new data is ready for service by the 8051.

For both IN and OUT endpoints, the busy bit is set when the EZ-USB core is using the buffers, and cleared by loading the endpoint's byte count register. The byte count value is meaningful for IN transfers because it tells the USB core how many bytes to transfer in response to the next IN token. The 8051 can load any byte count OUT transfers, because only the act of loading the register is significant—loading OUTnBC arms the OUT transfer and sets the endpoint's busy bit.

When an OUT packet arrives in OUT2BUF, the service routine at lines 22-26 increments R2, loads the byte count (any value) into OUT2BC to re-arm the endpoint (lines 24-25), and jumps back to the polling routine. This program does not use OUT2BUF data. It simply counts the number of endpoint 2-OUT transfers.

When endpoint 2-IN is ready for the 8051 to load another packet into IN2BUF, the polling loop jumps to the endpoint 2-IN service routine at lines 28-39. First, R1 is incremented (line 29). The data pointer is set to IN2BUF at line 30, and Register R1 is loaded into the first byte of the buffer (lines 31-32). The data pointer is advanced to the second byte of IN2BUF at line 33, and Register R2 is loaded into the buffer (lines 34-35). Finally, the byte count 40H (64 decimal bytes) is loaded into the byte count Register IN2BC to arm the next IN transfer at lines 36-38, and the routine returns the polling loop.

6.12 Enumeration Note

The code in the example listed above is complete, and it runs on the EZ-USB FX chip. You may be wondering about the *missing step*, which reports the endpoint characteristics to the host during the enumeration process. The reason this code runs without any enumeration code is that the EZ-USB FX chip comes on as a fully-functional USB device with certain endpoints already configured and reported to the host. Endpoint 2 is included in this default configuration. The full default configuration is described in *Chapter 5. "EZ-USB FX Enumeration & ReNumeration™"*.

6.13 Bulk Endpoint Interrupts

All USB interrupts activate the 8051 *INT 2* interrupt. If enabled, INT2 interrupts cause the 8051 to push the current program counter onto the stack, and then execute a jump to location 0x43, where the programmer has inserted a jump instruction to the interrupt service routine (ISR). If the AVEN (Autovector Enable) bit is set, the USB core inserts a special byte at location 0x45, which directs the jump instruction to a table of jump instructions that transfer control the endpoint-specific ISR.

Table 6-4. 8051 INT2 Interrupt Vector

Location	Op-Code	Instruction
0x43	02	LJMP
0x44	AddrH	
0x45	AddrL*	

*Replaced by EZ-USB Core if AVEN=1.

The byte inserted by the EZ-USB core at address 0x45 depends on which bulk endpoint requires service. Table 6-5 shows all INT2 vectors, with the bulk endpoint vectors shaded.

Table 6-5. Byte Inserted by USB Core at Location 0x45 if AVEN=1

Interrupt	Inserted Byte at 0x45
SUDAV	**0x00**
SOF	**0x04**
SUTOK	**0x08**
SUSPEND	**0x0C**
USBRES	**0x10**
Reserved	**0x14**
EP0-IN	**0x18**
EP0-OUT	**0x1C**
EP1-IN	**0x20**
EP1OUT	**0x24**
EP2IN	**0x28**
EP2OUT	**0x2C**
EP3-IN	**0x30**
EP3-OUT	**0x34**
EP4-IN	**0x38**
EP4-OUT	**0x3C**
EP5-IN	**0x40**
EP5-OUT	**0x44**
EP6-IN	**0x48**
EP6-OUT	**0x4C**
EP7-IN	**0x50**
EP7-OUT	**0x54**

The vector values are four bytes apart. This allows the programmer to build a jump table to each of the interrupt service routines. Note that the jump table must begin on a page (256 byte) boundary

because the first vector starts at 00. If Autovectoring is not used (AVEN=0), the IVEC Register may be directly inspected to determine the USB interrupt source. (See *Section 12.11. "Autovector Coding"*).

Each bulk endpoint interrupt has an associated interrupt enable bit (in IN07IEN and OUT07IEN), and an interrupt request bit (in IN07IRQ and OUT07IRQ). These IRQ bits can be cleared by writing to the INT2CLR SFR Register.

Any USB ISR should clear the 8051 INT2 interrupt request bit before clearing any of the EZ-USB FX endpoint IRQ bits, to avoid losing interrupts. Interrupts are discussed in more detail in Chapter 12. "EZ-USB FX Interrupts"

Individual interrupt request bits are cleared by writing "1" to them to simplify code. For example, to clear the endpoint 2-IN IRQ, simply write "0000100" to IN07IRQ. This will not disturb the other interrupt request bits. ***Do not read the contents of IN07IRQ, logical-OR the contents with 01, and write it back****. This clears all other pending interrupts because you are writing "1"s to them.*

6.14 Interrupt Bulk Transfer Example

This following simple (but fully-functional) example illustrates the bulk transfer mechanism using interrupts. In the example program, BULK endpoint 6 is used to loop data back to the host. Data sent by the host over endpoint 6-OUT is sent back over endpoint 6-IN.

1. Set up the jump table.

```
                CSEG    AT 300H         ; any page boundary
USB_Jump_Table:
                ljmp    SUDAV_ISR       ; SETUP Data Available
                db      0               ; make a 4-byte entry
                ljmp    SOF_ISR         ; SOF
                db      0
                ljmp    SUTOK_ISR       ; SETUP Data Loading
                db      0
                ljmp    SUSP_ISR        ; Global Suspend
                db      0
                ljmp    URES_ISR        ; USB Reset
                db      0
                ljmp    IBN_ISR
                db      0
                ljmp    EP0IN_ISR
                db      0
                ljmp    EP0OUT_ISR
                db      0
                ljmp    EP1IN_ISR
                db      0
                ljmp    EP1OUT_ISR
                db      0
                ljmp    EP2IN_ISR
                db      0
                ljmp    EP2OUT_ISR
                db      0
                ljmp    EP3IN_ISR
                db      0
                ljmp    EP3OUT_ISR
                db      0
                ljmp    EP4IN_ISR
                db      0
                ljmp    EP4OUT_ISR
                db      0
                ljmp    EP5IN_ISR
                db      0
                ljmp    EP5OUT_ISR
                db      0
                ljmp    EP6IN_ISR       ; Used by this example
                db      0
                ljmp    EP6OUT_ISR      ; Used by this example
                db      0
                ljmp    EP7IN_ISR
                db      0
                ljmp    EP7OUT_ISR
                db      0
```

Figure 6-7. Interrupt Jump Table

This table contains all of the USB interrupts, even though only the jumps for endpoint 6 are used for the example. It is convenient to include this table in any USB application that uses interrupts. Be sure to locate this table on a page boundary.

2. Write the INT2 interrupt vector.

```
; -----------------
; Interrupt Vectors
; -----------------
        org             43h                     ; int2 is the USB vector
        ljmp            USB_Jump_Table          ; Autovector will replace byte 45
```

Figure 6-8. INT2 Interrupt Vector

3. Write the interrupt service routine.

   Put it anywhere in memory and the jump table in step 1 will automatically jump to it.

```
; ----------------------------
; USB Interrupt Service Routine
; ----------------------------
EP6OUT_ISR
            push   acc
;
            mov    a,EXIF           ; clear INT2 (USB) IRQ flag
            clr    acc.4
            mov    EXIF,a
;
            mov    INT2CLR,a        ; use whatever value is in acc
;
            setb   got_EP6-DATA
; Do Interrupt processing here – set flags, whatever . . .
;                                     spend time here or not
            pop    acc
            reti
```

Figure 6-9. Interrupt Service Routine (ISR) for Endpoint 6-OUT

In this example, the ISR simply sets the 8051 flag “got_EP6_data” to indicate to the background program that the endpoint requires service.

4. Write the endpoint 6 transfer program.

```
1  loop:      jnb    got_EP6_data,loop
2             clr    got_EP6_data          ; clear my flag
3  ;
4  ; The user sent bytes to OUT6 endpoint using the USB Control Panel.
5  ; Find out how many bytes were sent.
6  ;
7             mov    dptr,#OUT6BC          ; point to OUT6 byte count register
8             movx   a,@dptr               ; get the value
9             mov    r7,a                  ; stash the byte count
10            mov    r6,a                  ; save here also
11 ;
12 ; Transfer the bytes received on the OUT6 endpoint to the IN6 endpoint
13 ; buffer.  Number of bytes in r6 and r7.
14 ;
15            mov    dptr,#OUT6BUF         ; first data pointer points to EP2OUT buffer
16            inc    dps                   ; select the second data pointer
17            mov    dptr,#IN6BUF          ; second data pointer points to EP2IN buffer
18            inc    dps                   ; back to first data pointer
19 transfer:  movx   a,@dptr               ; get OUT byte
20            inc    dptr                  ; bump the pointer
21            inc    dps                   ; second data pointer
22            movx   @dptr,a               ; put into IN buffer
23            inc    dptr                  ; bump the pointer
24            inc    dps                   ; first data pointer
25            djnz   r7,transfer
26 ;
27 ; Load the byte count into IN6BC.  This arms in IN transfer
28 ;
29            mov    dptr,#IN6BC
30            mov    a,r6                  ; get other saved copy of byte count
31            movx   @dptr,a               ; this arms the IN transfer
32 ;
33 ; Load any byte count into OUT6BC.  This arms the next OUT transfer.
34 ;
35            mov    dptr,#OUT6BC
36            movx   @dptr,a               ; use whatever is in acc
37            sjmp   loop                  ; start checking for another OUT6 packet
```

Figure 6-10. Background Program Transfers Endpoint 6-OUT Data to Endpoint 6-IN

The main program loop tests the “got_EP6_data” flag, waiting until it is set by the endpoint 6 OUT interrupt service routine in *Figure 6-10*. This indicates that a new data packet has arrived in OUT6BUF. Then the service routine is entered, where the flag is cleared in line 2. The number of bytes received in OUT6BUF is retrieved from the OUT6BC Register (Endpoint 6 Byte Count) and saved in Registers R6 and R7 in lines 7-10.

The dual data pointers are initialized to the source (OUT6BUF) and destination (IN6BUF) buffers for the data transfer in lines 15-18. These labels represent the start of the 64-byte buffers for endpoint 6-OUT and endpoint 6-IN, respectively. Each byte is read from the OUT6BUF buffer and written to the IN6BUF buffer in lines 19-25. The saved value of OUT6BC is used as a loop counter in R7 to transfer the exact number of bytes that were received over endpoint 6-OUT.

When the transfer is complete, the program loads the endpoint 6-IN byte count Register IN6BC with the number of loaded bytes (from R6) to *arm* the next endpoint 6-IN transfer in lines 29-31. Finally, the 8051 loads any value into the endpoint 6 OUT byte count Register OUT6BC to arm the next OUT transfer in lines 35-36. Then the program loops back to check for more endpoint 6-OUT data.

DMA cannot be used for this Loopback since the source and destination would be in the same RAM block.

5. Initialize the endpoints and enable the interrupts.

```
start:    mov     SP,#STACK-1       ; set stack
;
; Enable USB interrupts and Autovector
;
          mov     dptr,#USBBAV      ; enable Autovector
          movx    a,@dptr
          setb    acc.0             ; AVEN bit is bit 0
          movx    @dptr,a
;
          movx    dptr,#USBBAV
          movx    a, @dptr
          setb    acc.4             ; enable the SFR-clearing feature
          movx    @dptr, a          ; for INT2
;
          mov     dptr,#OUT07IEN    ; 'EP0-7 OUT int enables' Register
;         mov     a,#01000000b      ; set bit 6 for EP6OUT interrupt enable
          movx    @dptr,a           ; enable EP6OUT interrupt
;
; Enable INT2 and 8051 global interrupts
;
          setb    ex2               ; enable int2 (USB interrupt)
          setb    EA                ; enable 8051 interrupts
          clr     got_EP6_data      ; clear my flag
```

Figure 6-11. Initialization Routine

The initialization routine sets the stack pointer, and enables the EZ-USB FX Autovector by setting USBBAV.0 to 1. Then it enables the endpoint 6-OUT interrupt, all USB interrupts (INT2), and the 8051 global interrupt (EA) and finally clears the flag indicating that endpoint 6-OUT requires service.

Once this structure is put into place, it is quite easy to service any or all of the bulk endpoints. To add service for endpoint 2-IN, for example, simply write an endpoint 2-IN interrupt service routine with starting address EP2IN_ISR (to match the address in the jump table in step 1), and add its valid and interrupt enable bits to the "init" routine.

6.15 Enumeration Note

The code in the previous example is complete, and runs on the EZ-USB FX chip. You may be wondering about the *missing step*, which reports the endpoint characteristics to the host during the enumeration process. The reason this code runs without any enumeration code is that the EZ-USB FX chip comes on as a fully-functional USB device with certain endpoints already configured and reported to the host. Endpoint 6 is included in this default configuration. The full default configuration is described in *Chapter 5. "EZ-USB FX Enumeration & ReNumeration™"*.

Portions of the above code are not necessary for the default configuration (such as setting the endpoint valid bits), but the code is included to illustrate all of the EZ-USB FX registers used for bulk transfers

6.16 The Autopointer

Bulk endpoint data is available in 64-byte buffers in EZ-USB FX RAM. In some cases it is preferable to access bulk data as a FIFO register rather than as a RAM. The EZ-USB core provides a special data pointer that automatically increments when data is transferred. Using this Autopointer, the 8051 can access any contiguous block of internal EZ-USB FX RAM or off-chip memory as a FIFO.

AUTOPTRH **Autopointer Address High** **7FE3**

b7	b6	b5	b4	b3	b2	b1	b0
A15	**A14**	**A13**	**A12**	**A11**	**A10**	**A9**	**A8**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
x	x	x	x	x	x	x	x

AUTOPTRL **Autopointer Address Low** **7FE4**

b7	b6	b5	b4	b3	b2	b1	b0
A7	**A6**	**A5**	**A4**	**A3**	**A2**	**A1**	**A0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
x	x	x	x	x	x	x	x

AUTODATA	Autopointer Data	7FE5

b7	b6	b5	b4	b3	b2	b1	b0
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
x	x	x	x	x	x	x	x

Figure 6-12. Autopointer Registers

The 8051 first loads AUTOPTRH and AUTOPTRL with a RAM address (for example the address of a bulk endpoint buffer). Then, as the 8051 reads or writes data to the data Register AUTODATA, the address is supplied by AUTOPTRH/L, which automatically increments after every read or write to the AUTODATA Register. The AUTOPTRH/L Registers may be written or read at anytime. These registers maintain the current pointer address, so the 8051 can read them to determine where the next byte will be read or written.

The 8051 code example in *Figure 6-13* uses the Autopointer to transfer a block of eight data bytes from the endpoint 4 OUT buffer to internal 8051 memory.

```
Init:   mov     dptr,#AUTOPTRH
        mov     a,#HIGH(OUT4BUF)  ; High portion of OUT4BUF buffer
        movx    @dptr,a           ; Load AUTOPTRH
        mov     dptr,#AUTOPTRL
        mov     a,#LOW(OUT4BUF)   ; Low portion of OUT4BUF buffer address
        movx    @dptr,a           ; Load AUTOPTRL
        mov     dptr,#AUTODATA    ; point to the 'fifo' Register
        mov     r0,#80H           ; store data in upper 128 bytes of 8051 RAM
        mov     r2,#8             ; loop counter
;
loop:   movx    a,@dptr           ; get a 'fifo' byte
        mov     @r0,a             ; store it
        inc     r0                ; bump destination pointer
                                  ; (NOTE: no 'inc dptr' required here)
        djnz    r2,loop           ; do it eight times
```

Figure 6-13. Use of the Autopointer

As the comment in the second to last line indicates, the Autopointer saves an "inc dptr" instruction that would be necessary if one of the 8051 data pointers were used to access the OUT4BUF RAM data. This improves the transfer time.

The Autopointer works only with internal program/data RAM. It does not work with memory outside the chip, or with internal RAM that is made available when ISODISAB=1. See Section 10.6.1. "Disable ISO" for a description of the ISODISAB bit.

The EZ-USB FX chip should never be a speed bottleneck in a USB system since it can DMA Data at 24Mhz. It also gives the 8051 ample time for other processing duties between endpoint buffer loads.

The Autopointer can be used to quickly move data anywhere in RAM, not just the bulk endpoint buffers. For example, it can be used to good effect in an application that calls for transferring a block of data into RAM, processing the data, and then transferring the data to a bulk endpoint buffer.

Chapter 7. EZ-USB FX Slave FIFOs

7.1 Introduction

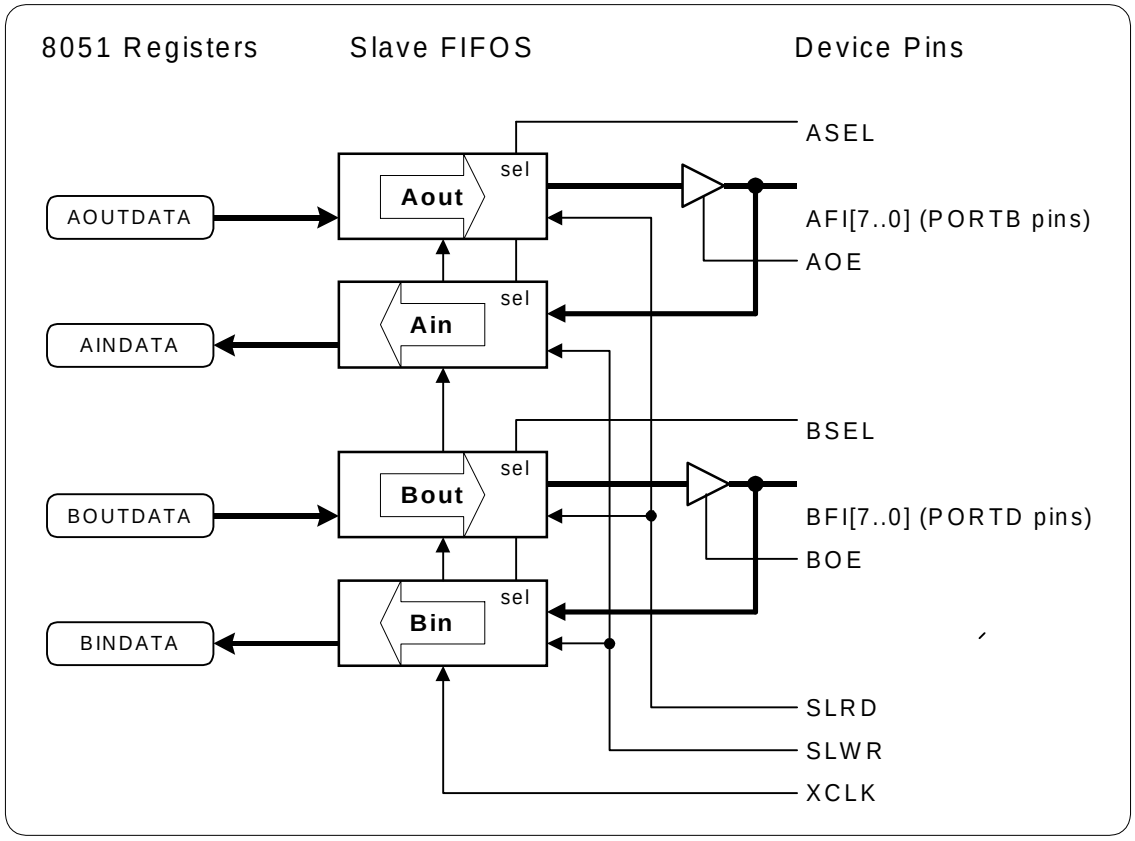


Figure 7-1. The Four 64-Byte Slave FIFOs Configured for 16-Bit Mode

Figure 7-1 illustrates the four slave FIFOs in EZ-USB FX. The slave FIFOs, each 64 bytes in length, serve as general-purpose buffers between external logic and 8051 registers. They are called “slave” FIFOs because the outside logic can supply the timing signals. The FIFOs are

grouped into identical *A* and *B* pairs, each pair having an IN and OUT FIFO. *Figure 7-1* illustrates *16-bit mode*, in which outside logic can read or write data either independently or simultaneously from/to the two 8-bit FIFOs.

7.1.1 8051 FIFO Access

The 8051 accesses the slave FIFOs using four registers in XDATA memory: AOUTDATA, AINDATA, BOUTDATA, and BINDATA. These registers can be read and written by 8051 code (using the MOVX instruction), or they can serve as sources and destinations for the DMA mechanism, built into the EZ-USB FX. Section 7.2. *"Slave FIFO Register Descriptions"* describes these registers in detail.

7.1.2 External Logic FIFO Access

External logic can access the slave FIFOs either asynchronously or synchronously:

- Asynchronous—SLRD and SLWR pins are read and write strobes.
- Synchronous—SLRD and SLWR pins are enables for the XCLK clock pin.

External logic accesses the FIFOs through two 8-bit data buses, which double as general-purpose I/O ports PORTB and PORTD. When used for FIFO access, the data buses are bi-directional, with output drivers controlled by the AOE and BOE pins.

Two FIFO select signals, ASEL and BSEL, are used to select the FIFO in two modes that use both FIFOs: 8-bit mode, and double-byte mode. These modes and the role of the ASEL and BSEL pins are illustrated in *Figure 7-2* and *Figure 7-3*.

7.1.3 ASEL, BSEL in 8-Bit Mode

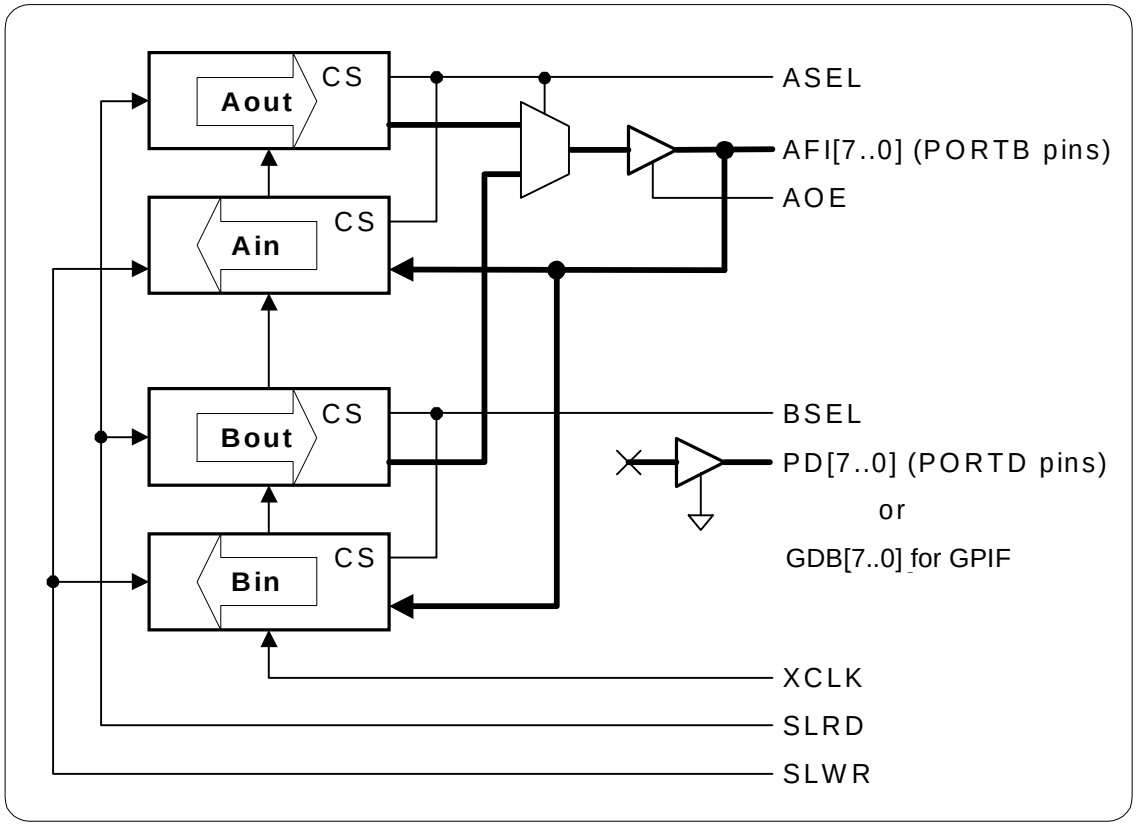


Figure 7-2. Slave FIFOs in 8-Bit Mode

In 8-bit mode, data from the PORTB pins can be read/written from either the A or B FIFOs, as selected by the ASEL and BSEL pins. In 8-bit mode, the input/output port or GPIF data is available on the PORTD pins.

7.1.4 ASEL, BSEL in Double-Byte Mode

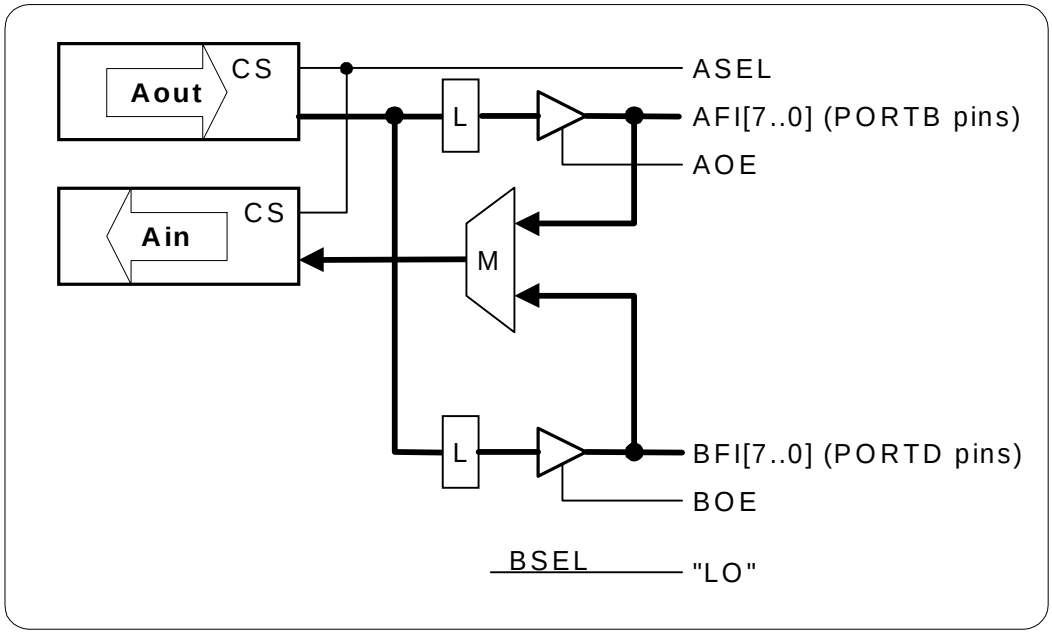


Figure 7-3. Double-Byte Mode with A-FIFO Selected

Figure 7-3 illustrates double-byte mode. For this illustration, signals ASEL, BSEL, AOE, and BOE are programmed to be active high polarity. In this mode, the ASEL and BSEL pins determine which of the FIFO pairs, A or B, accept or transmit interleaved byte data, as follows:

- The IN FIFO receives 16-bit data as double bytes, interleaved from PORTB first and then from PORTD. The data interleaving is automatic, with two bytes written to the FIFO per external write strobe. The interleave order input from the ports is the same whether the destination FIFO is A-IN or B-IN.
- The OUT FIFO transmits two bytes, the first to PORTB and the second to PORTD, per external read strobe. The interleave order output to the ports is the same whether the source FIFO is A-OUT or B-OUT.

7.1.5 FIFO Registers

The 8051 accesses a variety of control and status registers to control the slave FIFO operation. These registers perform the following functions:

- Data registers give the 8051/DMA access to IN FIFO and OUT FIFO data.

- Byte Count registers indicate the number of bytes in each FIFO.
- Flag bits indicate FIFO full, empty, and a programmable level.
- Mode bits control the various FIFO modes.

7.1.6 FIFO Flags and Interrupts

The slave FIFOs have two independent sets of flags, internal and external. The 8051 can directly test the internal flags, or these flags can automatically create 8051 interrupts using INT4. The external flags are available as device pins, to be used by external logic. Two independent sets of programmable flags allow different FIFO *fullness* levels to be set for internal and external use.

The internal FIFO flags are connected to the 8051 interrupt system using INT4. To streamline the 8051 code that deals with these interrupts, the 8051 INT4 vector locations have a special property when a mode bit called “AV4EN” (Autovector 4 Enable) is set. Referring toTable 7-1, when a FIFO flag interrupt occurs with AV4EN=1, internal logic replaces the third byte of the jump instruction at location 0x55 with a different address for each FIFO interrupt source.

Table 7-1. Autovector for INT4*

8051 Addr	Instruction	Notes
0x53	LJMP	Loc 53-55 are the INT4 Interrupt Vector.
0x54	AddrH	
0x55 *	AddrL	**EZ-USB FX logic replaces this byte when AV4EN=1.**

* (Table 7-2 shows bytes inserted at address 55H)

To set up autovectoring, the user places an LJMP instruction at location 0x53. This jumps to a table of instructions that jump to the various FIFO ISRs. Then, every FIFO interrupt automatically vectors to the individual interrupt service routines for the particular FIFO flags. The autovector mechanism saves the 8051 from having to check for the source of each interrupt shared on INT4.

The FIFO interrupts that share INT4 are shown in Table 7-2. The last three are not FIFO-related, and are described in other chapters. The bytes inserted by the EZ-USB FX logic (the low-address byte of the LJMP instruction) are separated by four to allow four bytes per LJMP instruction in the jump table. (An 8051 LJMP instruction requires three bytes).

Note that the bytes inserted for the INT4 autovector start at 0x80, rather than 0x00. This is because another EZ-USB FX autovector, for INT2 (used for all USB interrupts), uses jump table offsets from 0x00 to 0x57. The autovector jump table must start on a page boundary (8051 address XX00). Therefore, separating the two groups of jumps allows a single page of 8051 memory to be used for both INT2 and INT4 jump tables. The INT2 jump table can start at 0x00, and the INT4 jump table can start at 0x80, both in the same page.

Table 7-2. INT4 Autovectors

IVEC4 Value	Byte Inserted at 0x55	Source	Meaning
0x40	0x80	AINPF	A-IN FIFO Programmable Flag
0x44	0x84	BINPF	B-IN FIFO Programmable Flag
0x48	0x88	AOUTPF	A-OUT FIFO Programmable Flag
0x4C	0x8C	BOUTPF	B-OUT FIFO Programmable Flag
0x50	0x90	AINEF	A-IN FIFO Empty Flag
0x54	0x94	BINEF	B-IN FIFO Empty Flag
0x58	0x98	AOUTEF	A-OUT FIFO Empty Flag
0x5C	0x9C	BOUTEF	B-OUT FIFO Empty Flag
0x60	0xA0	AINFF	A-IN FIFO Full Flag
0x64	0xA4	BINFF	B-IN FIFO Full Flag
0x68	0xA8	AOUTFF	A-OUT FIFO Full Flag
0x6C	0xAC	BOUTFF	B-OUT FIFO Full Flag
0x70	0xB0	GPIF-DONE	See *Chapter 8. "General Programmable Interface (GPIF)"*
0x74	0xB4	GPIFWF	See *Chapter 8. "General Programmable Interface (GPIF)"*
0x78	0xB8	DMADONE	See *Chapter 8. "General Programmable Interface (GPIF)"*

The first column shows the value in the IVEC4 Register for each FIFO interrupt source.

If two or more INT4 interrupt requests occur simultaneously, they are serviced in the order shown in Table 7-2, with AINPF having the highest priority and DMADONE the lowest. Interrupt requests remain pending while a higher level interrupt is serviced.

7.2 Slave FIFO Register Descriptions

In the following FIFO diagrams, the 8051-access side is on the left, and the external pins are on the right.

7.2.1 FIFO A Read Data

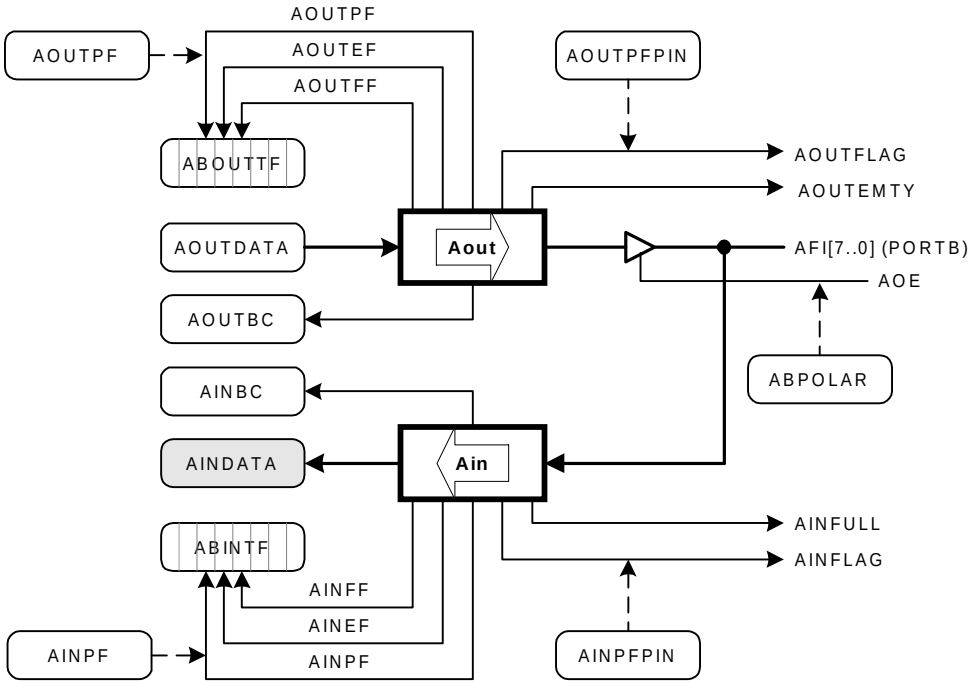


Figure 7-4. AINDATA's Role in the FIFO A Register

AINDATA	FIFO A Read Data	7800

b7	b6	b5	b4	b3	b2	b1	b0
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R	R	R	R	R	R	R	R
x	x	x	x	x	x	x	x

Figure 7-5. FIFO A Read Data

Each time the 8051 reads a byte from this register, the A-IN FIFO advances to the next byte in the FIFO, and the AINBC (byte count) decrements. Reading this register when there is one byte remaining in the A-IN FIFO sets the A-IN FIFO Empty Flag (AINEF, in ABINCS.4). This causes an interrupt request on INT4 (Table 7-2). Reading this register when the A-IN FIFO is empty returns indeterminate data and has no effect on the FIFO flags byte counts.

7.2.2 A-IN FIFO Byte Count

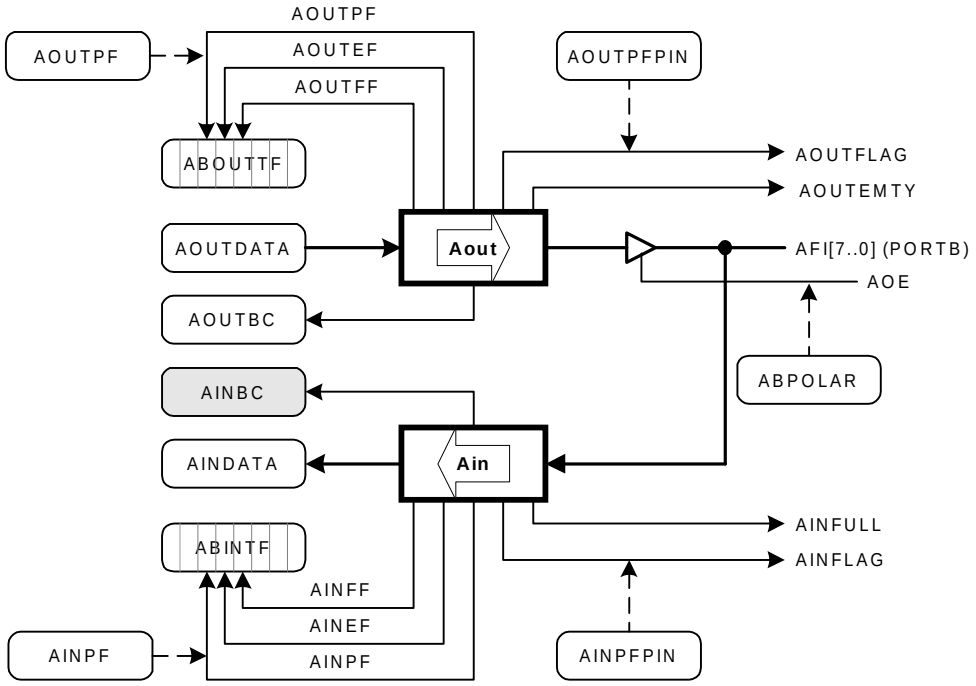


Figure 7-6. AINBC's Role in the FIFO A Register

AINBC **A-IN FIFO Byte Count** **7801**

b7	b6	b5	b4	b3	b2	b1	b0
0	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R	R	R	R	R	R	R	R
0	0	0	0	0	0	0	0

Figure 7-7. A-IN FIFO Byte Count

This count reflects the number of bytes remaining in the A-IN FIFO. Valid byte counts are 0-64. Every byte written by outside logic increments this count, and every 8051 read of AINDATA decrements this count. If AINBC is zero, an 8051 read of AINDATA returns indeterminate data and results in the byte count in AINBC remaining at zero. Data bytes should never be written to the FIFO from outside logic when the AINFULL flag is HI.

7.2.3 A-IN FIFO Programmable Flag

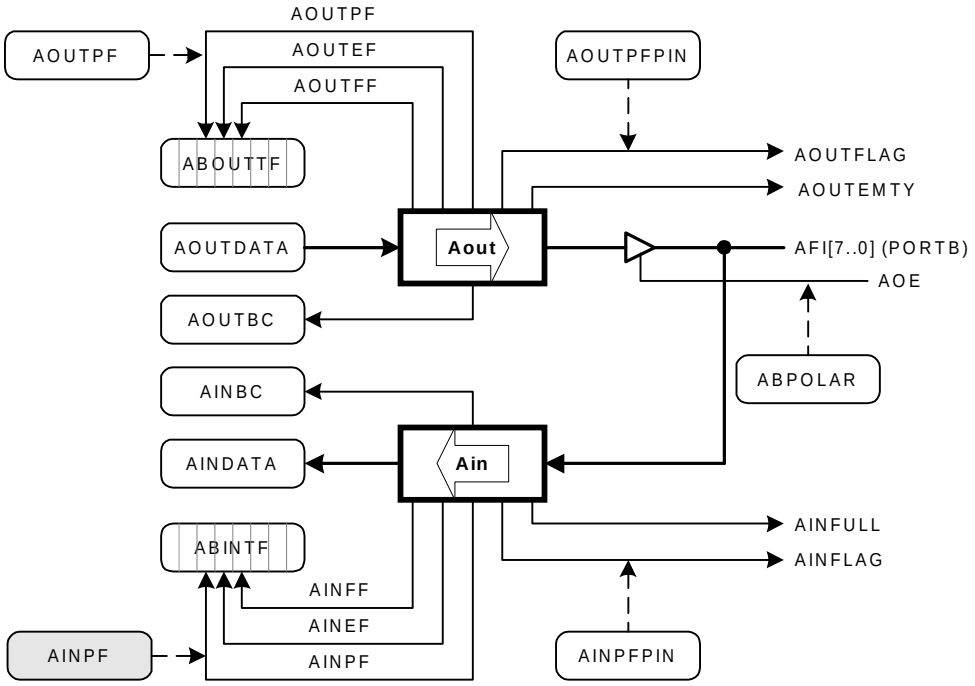


Figure 7-8. AINPF's Role in the FIFO A Register

AINPF	A-IN FIFO Programmable Flag	7802

b7	b6	b5	b4	b3	b2	b1	b0
LTGT	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	1	0	0	0	0	0

Figure 7-9. A-IN FIFO Programmable Flag

This register controls the sense and value for the internal A-IN FIFO programmable flag. This flag is testable by the 8051.

*Another register, AINPF**PIN** (*Section 7.2.3.3. *"A-IN FIFO Pin Programmable Flag") corresponds to an A-IN FIFO programmable flag that drives an output pin, not an internal flag bit.*

The 8051 tests the internal FIFO programmable flag by reading the AINPF Bit in ABINCS.5. This flag can also be enabled to cause an interrupt request on INT4(Table 7-2) when it makes a zero-to-one transition. The default value of the AINPF Register indicates half-empty.

Bit 7: **LTGT** *Less-than, Greater-than flag*

If LTGT=0, the AINPF flag goes true, if the number of bytes in the FIFO is less than or equal to the programmed value in D[6..0].

If LTGT=1, the AINPF flag goes true, if the number of bytes in the FIFO is greater than or equal to the value programmed into D[6..0].

Bit 6-0: **PFVAL** *Programmable Flag Value*

This value, along with the LTGT Bit, determines when the programmable flag for the A-IN FIFO becomes active. The 8051 programs this register to indicate various degrees of A-IN FIFO *fullness* to suit the application. The following two sections show the interaction of the LTGT Bit and the programmed value for two cases, a filling FIFO and an emptying FIFO.

7.2.3.1 Filling FIFO

When a FIFO is filling with data, it is useful to generate an 8051 interrupt when a programmed level is reached. Because the interrupt request is triggered on a zero-to-one transition of the programmable flag AINPF, the LTGT Bit should be set to "1." In Table 7-3, D[6..0] is set to 48 bytes and the LTGT Bit is set to "1." When the FIFO reaches 48 bytes, the AINPF Bit goes high, generating an interrupt request.

Table 7-3. Filling FIFO

LTGT	D[6..0]	Bytes in FIFO	AINPF
1	48	45	0
1	48	46	0
1	48	47	0
1	48	48	1
1	48	49	1
1	48	50	1

7.2.3.2 Emptying FIFO

When a FIFO is being emptied of data, the LTGT Bit should be set to "0," so the zero-to-one transition of the AINPF flag (and therefore the interrupt request) occurs when the byte count descends to below the programmed value. In Table 7-4, D[6..0] is set to 48 bytes, and when the FIFO goes from 49 bytes to 48 bytes, the AINPF Bit goes high, generating an interrupt request.

Table 7-4. Emptying FIFO

LTGT	D[6..0]	Bytes in FIFO	AINPF
0	48	51	0
0	48	50	0
0	48	49	0
0	48	48	1
0	48	47	1
0	48	46	1

7.2.3.3 A-IN FIFO Pin Programmable Flag

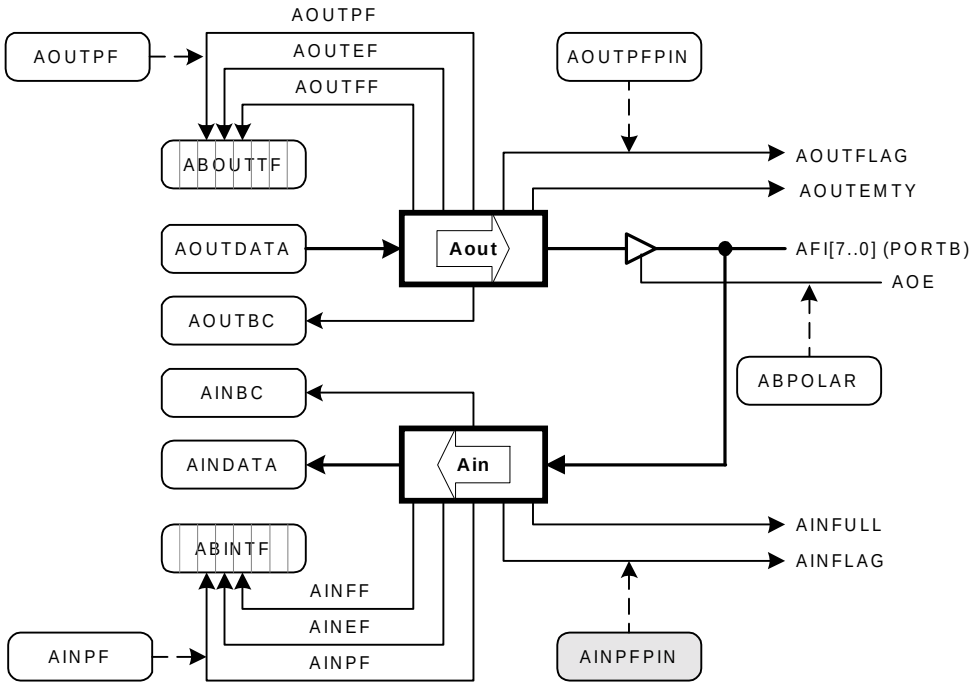


Figure 7-10. AINPFPIN's Role in the FIFO A Register

AINPFPIN **A-IN FIFO Pin Programmable Flag** **7803**

b7	b6	b5	b4	b3	b2	b1	b0
LTGT	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Figure 7-11. A-IN FIFO Pin Programmable Flag

This register controls the sense and value for the A-IN FIFO Programmable Flag that appears on the AINFLAG *pin*. This pin is used by external logic to regulate external writes to the A-IN FIFO. The AINPFPIN Register is programmed with the same data format as the previous register, AINPF. The only operational difference is that the flag drives a hardware pin rather than existing as an internal register bit.

Having separate programmable flags allows the 8051 and external logic to have independent gauges of FIFO fullness. It may be desirable, for example, for one side (8051 or external logic) to have advance notice over the other side about a FIFO becoming full or empty.

The default value of the AINPFPIN Register indicates empty.

7.2.4 B-IN FIFO Read Data

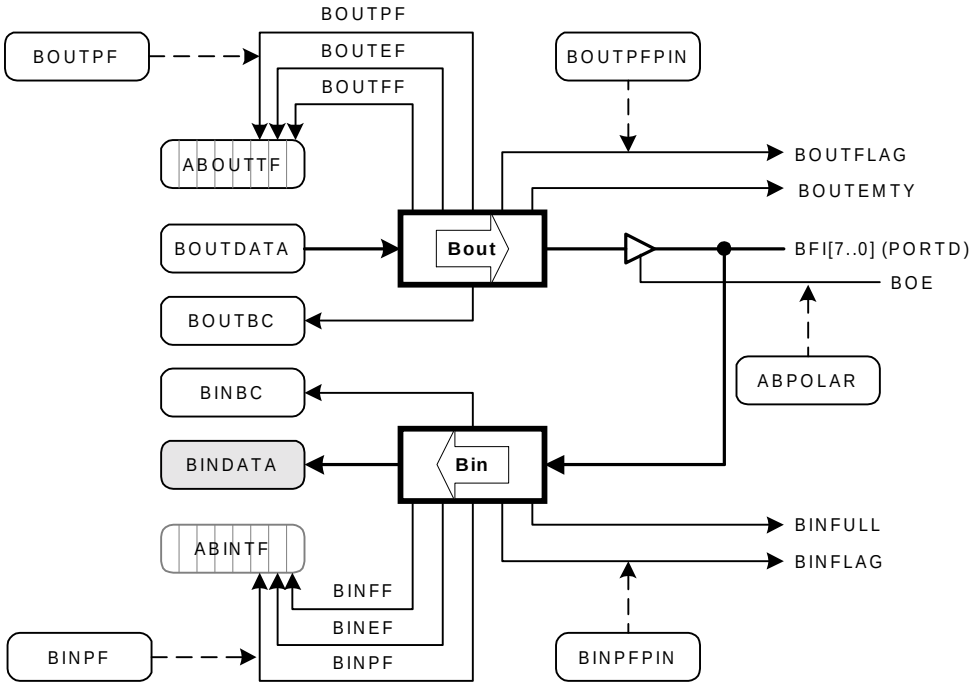


Figure 7-12. BINDATA's Role in the FIFO B Register

BINDATA	B-IN FIFO Read Data	7805

b7	b6	b5	b4	b3	b2	b1	b0
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R	R	R	R	R	R	R	R
x	x	x	x	x	x	x	x

Figure 7-13. B-IN FIFO Read Data

Each time the 8051 reads a byte from this register, the B-IN FIFO advances to the next byte in the FIFO, and the BINBC (byte count) decrements. Reading this register when there is one byte remaining in the FIFO sets the B-IN FIFO Empty Flag (BINEF, in ABINCS.1), which causes an INT4 request. Reading this register when the B-IN FIFO is empty returns indeterminate data and has no effect on the FIFO flags or byte count.

7.2.5 B-IN FIFO Byte Count

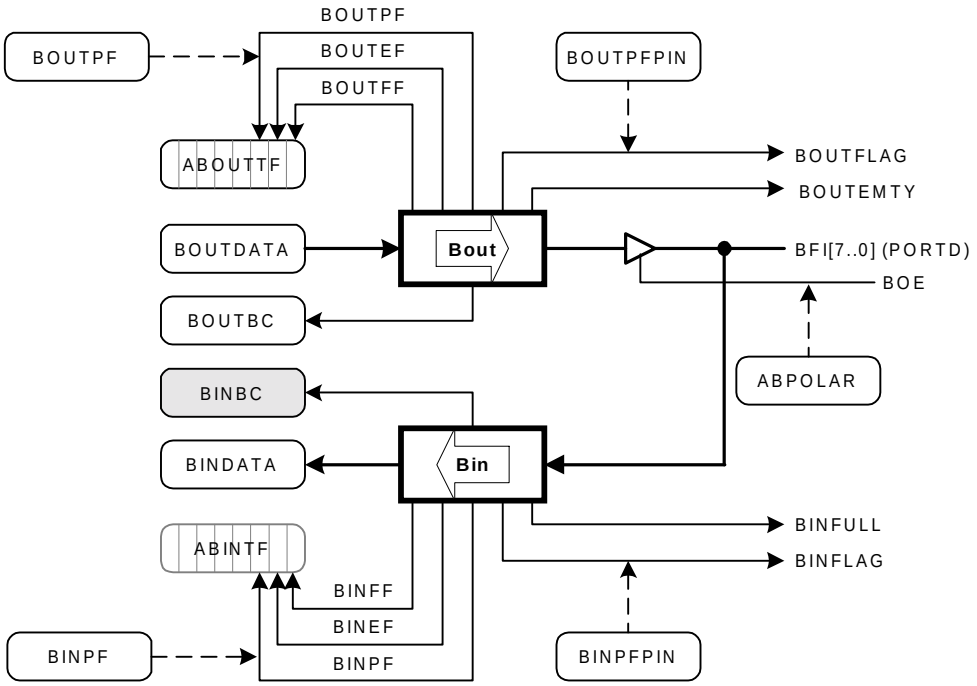


Figure 7-14. BINBC's Role in the FIFO B Register

BINBC **B-IN FIFO Byte Count** **7806**

b7	b6	b5	b4	b3	b2	b1	b0
0	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R	R	R	R	R	R	R	R
0	0	0	0	0	0	0	0

Figure 7-15. B-IN FIFO Byte Count

This count reflects the number of bytes remaining in the B-IN FIFO. Valid byte counts are 0-64. Every byte written by outside logic increments this count, and every 8051 read of BINDATA decrements this count. If BINBC is zero, an 8051 read of BINDATA returns indeterminate data. This results in the byte count in BINBC to remain at zero. Data bytes should never be written to the FIFO from outside logic when the BINFULL flag is HI.

7.2.6 B-IN FIFO Programmable Flag

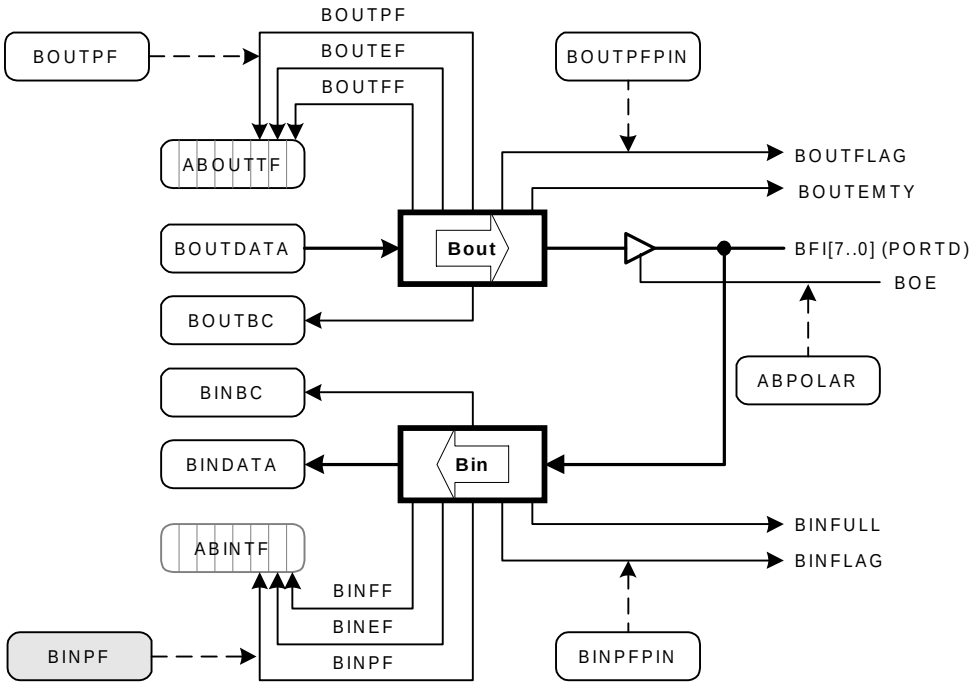


Figure 7-16. BINPF's Role in the FIFO B Register

BINPF	B-IN FIFO Programmable Flag	7807

b7	b6	b5	b4	b3	b2	b1	b0
LTGT	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	1	0	0	0	0	0

Figure 7-17. B-IN FIFO Programmable Flag

This register controls the sense and value for the internal B-IN FIFO programmable flag.

*Another register, BINPF**PIN** (Section 7.2.7. "B-IN FIFO Pin Programmable Flag") corresponds to a B-IN FIFO programmable flag that drives an output pin, not an internal flag bit.*

The 8051 tests the internal FIFO programmable flag by reading the BINPF Bit in ABINCS.2. This flag can also be enabled to cause an interrupt request on INT4(Table 7-2) when it makes a zero-to-one transition. The default value of the BINPF Register indicates half-empty.

Bit 7: **LTGT** *Less-than, Greater-than flag*

If LTGT=0, the BINPF flag goes true if the number of bytes in the FIFO is less than or equal to the programmed value in D[6..0].

If LTGT=1, the BINPF flag goes true if the number of bytes in the FIFO is greater than or equal to the value programmed into D[6..0].

Bit 6-0: **PFVAL** *Programmable Flag Value*

This value, along with the LTGT Bit, determines when the programmable flag for the B-FIFO becomes active. The 8051 programs this register to indicate various degrees of B-FIFO *fullness* to suit the application. The following two sections in this chapter show the interaction of the LTGT Bit and the programmed value for two cases, a filling FIFO and an emptying FIFO.

7.2.6.1 Filling FIFO

When a FIFO is filling with data, it is useful to generate an 8051 interrupt when a programmed level is reached. Because the interrupt request is triggered on a zero-to-one transition of the programmable flag BINPF, the LTGT Bit should be set to "1." In Table 7-5, D[6..0] is set to 48 bytes and the LTGT Bit is set to "1." When the FIFO reaches 48 bytes, the BINPF Bit goes high, generating an interrupt request.

Table 7-5. Filling FIFO

LTGT	D[6..0]	Bytes in FIFO	BINPF
1	48	45	0
1	48	46	0
1	48	47	0
1	48	48	1
1	48	49	1
1	48	50	1

7.2.6.2 Emptying FIFO

When a FIFO is being emptied of data, the LTGT Bit should be set to "0," so the zero-to-one transition of the BINPF flag (therefore, the interrupt request) occurs when the byte count descends to below the programmed value. In Table 7-6, D[6..0] is set to 48 bytes, and when the FIFO goes from 49 bytes to 48 bytes, the BINPF Bit goes high, generating an interrupt request.

Table 7-6. Emptying FIFO

LTGT	D[6..0]	Bytes in FIFO	BINPF
0	48	51	0
0	48	50	0
0	48	49	0
0	48	48	1
0	48	47	1
0	48	46	1

7.2.7 B-IN FIFO Pin Programmable Flag

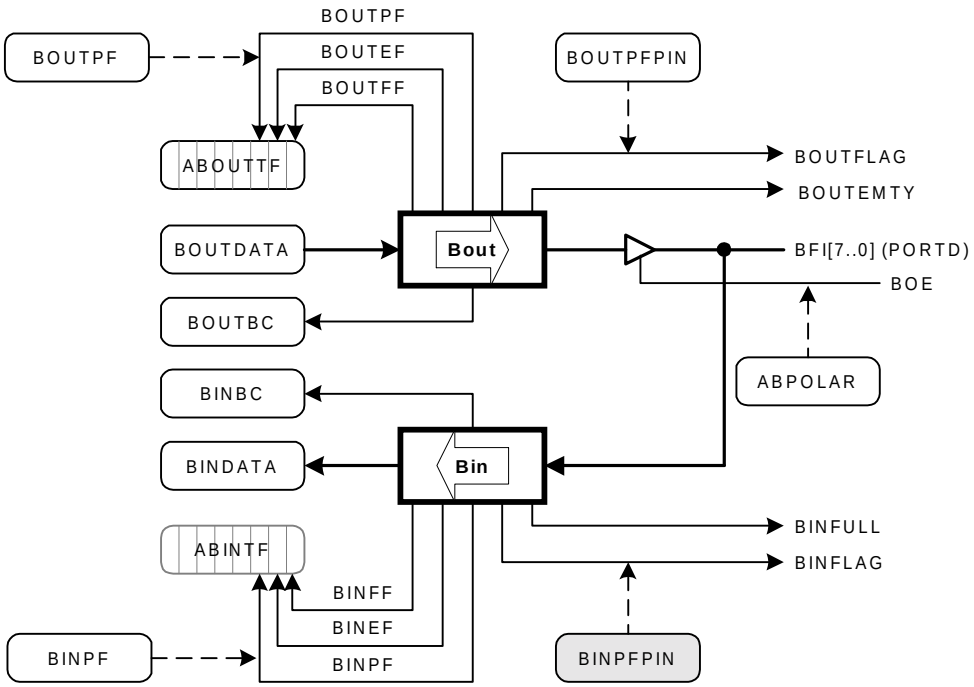


Figure 7-18. BINPFPIN's Role in the FIFO B Register

BINPFPIN **B-IN FIFO Pin Programmable Flag** **7808**

b7	b6	b5	b4	b3	b2	b1	b0
LTGT	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

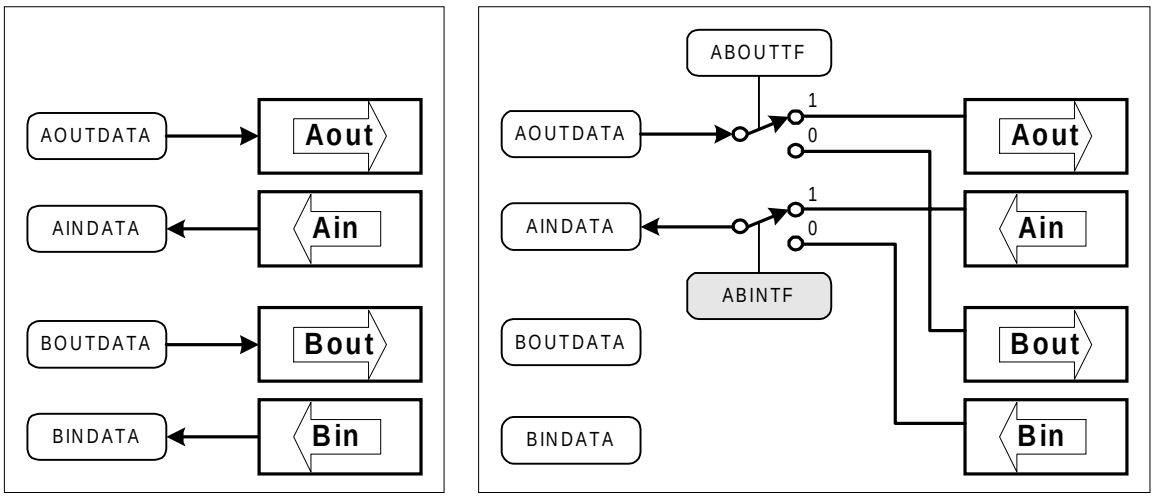
Figure 7-19. B-IN FIFO Pin Programmable Flag

This register controls the sense and value for the B-IN FIFO Programmable Flag that appears on the BINFLAG *pin*. This pin is used by external logic to regulate external writes to the B-IN FIFO. The BINPFPIN Register is programmed with the same data format as the previous register, BINPF. The only operational difference is that the flag drives a hardware pin rather than existing as an internal register bit.

Having separate programmable flags allows the 8051 and external logic to have independent gauges of FIFO fullness. It may be desirable, for example, for one side (8051 or external logic) to have advance notice over the other side about a FIFO becoming full or empty.

The default value of the BINPFPIN Register indicates empty.

7.2.8 Input FIFOs A/B Toggle CTL and Flags



(a) Normal Mode (b) 8051 FIFO Toggle Mode

Figure 7-20. 8051 FIFO Toggle Mode vs. Normal Mode Diagram

ABINCS	Input FIFOs A/B Toggle CTL and Flags	780A

b7	b6	b5	b4	b3	b2	b1	b0
INTOG	**INSEL**	**AINPF**	**AINEF**	**AINFF**	**BINPF**	**BINEF**	**BINFF**
R/W	R/W	R	R	R	R	R	R
0	1	1	1	0	1	1	0

Figure 7-21. Input FIFOs A/B Toggle CTL and Flags

Bit 7: INTOG *Enable Input FIFO Toggle*

A special FIFO toggle mode switches automatically between the A-IN and B-IN FIFOs each time the 8051 reads data from the AINDATA Register. The toggle mechanism works only for programmed 8051 transfers, not DMA transfers.

When INTOG=0, the A-IN and B-IN FIFOs operate in Normal Mode, as illustrated in diagram (a) in *Figure 7-20* on the previous page.

When INTOG=1, the FIFOs operate in Toggle Mode, as illustrated in diagram (b) in *Figure 7-20*. The selected FIFO switches between the A-IN and B-IN FIFOs after every 8051 read of the AINDATA Register. The selected FIFO is indicated by the INSEL Bit (Bit 6).

Bit 6: INSEL *Input Toggle Select*

If INTOG=1 when enabling the Toggle Mode:

- This bit selects IN FIFO A or B when the 8051 reads the AINDATA Register. When INSEL=0, the B-IN FIFO is read. When INSEL=1, the A-IN FIFO is read. When INTOG=1, this bit complements automatically (toggles) after every 8051 read of AINDATA. This has the effect of automatically toggling between the A-IN and B-IN FIFOs for successive reads of AINDATA.
- The 8051 can directly write this bit to select *manually* the A-IN or B-IN FIFO. More commonly, the Toggle Mode will be used since it allows 16-bit transfers using the 8051 without requiring the 8051 to switch between the FIFOs.

If INTOG=0 when enabling the Toggle Mode:

- The INSEL Bit has no effect.

Bit 5: AINPF *A-IN FIFO Programmable Flag*

AINPF=1 when the A-IN FIFO byte count satisfies the conditions programmed into the programmable FIFO flag register AINPF; otherwise, AINPF=0. A zero-to-one transition of this flag sets the interrupt request bit AINPFIR.

Bit 4: **AINEF** *A-IN FIFO Empty Flag*

AINEF=1 when the A-IN FIFO is empty; otherwise, AINEF=0. The flag goes active after the 8051 or DMA system reads the last byte in the A-IN FIFO. A zero-to-one transition of this flag sets the interrupt request bit AINEFIR.

AINFF=1 when the A-IN FIFO is full; otherwise, AINFF=0. The flag goes active after external logic writes the 64th byte into the A-IN FIFO. A zero-to-one transition of this flag sets the interrupt request bit AINFFIR.

Bit 2: **BINPF** *B-IN FIFO Programmable Flag*

BINPF=1 when the number of bytes in the B-IN FIFO satisfies the requirements programmed into the BINPF Register; otherwise, BINPF=0. A zero-to-one transition of this flag sets the interrupt request bit BINPFIR.

Bit 1: **BINEF** *B-IN FIFO Empty Flag*

BINEF=1 when the B-IN FIFO is empty; otherwise, BINEF=0. The flag goes active after the 8051 or DMA system reads the last byte in the B-IN FIFO. A zero-to-one transition of this flag sets the interrupt request bit BINEFIR.

Bit 0: **BINFF** *B-IN FIFO Full Flag*

BINFF=1 when the B-IN FIFO is full. The flag goes valid after external logic writes the 64th byte into the B-IN FIFO. A zero-to-one transition of this flag sets the interrupt request bit BINFFIR.

7.2.9 Input FIFOs A/B Interrupt Enables

ABINIE **Input FIFOs A/B Interrupt Enables** **780B**

b7	b6	b5	b4	b3	b2	b1	b0
0	**0**	**AINPFIE**	**AINEFIE**	**AINFFIE**	**BINPFIE**	**BINEFIE**	**BINFFIE**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Figure 7-22. Input FIFOs A/B Interrupt Enables

Bit 5: **AINPFIE** *A-IN FIFO Programmable Flag Interrupt Enable*

The 8051 sets AINPFIE=1 to enable an INT4 interrupt when the AINPFIR interrupt request bit makes a zero-to-one transition. This transition indicates that the A-IN FIFO byte count has satisfied the *fullness* level programmed into the programmable FIFO flag register AINPF. The

8051 clears AINPFIE to prevent the associated interrupt request bit from causing an INT4 interrupt.

Bit 4: **AINEFIE** *A-IN FIFO Empty Interrupt Enable*

The 8051 sets AINEFIE=1 to enable an INT4 interrupt when the AINEFIR interrupt request bit makes a zero-to-one transition. This indicates an A-IN FIFO byte count of zero. The 8051 clears AINEFIE to prevent the associated interrupt request bit from causing an INT4 interrupt.

Bit 3: **AINFFIE** *A-IN FIFO Bull Interrupt Enable*

The 8051 sets AINFFIE=1 to enable an INT4 interrupt when the AINFFIR interrupt request bit makes a zero-to-one transition. This indicates an A-IN FIFO byte count of 64. The 8051 clears AINFFIE to prevent the associated interrupt request bit from causing an INT4 interrupt.

Bit 2: **BINPFIE** *B-IN FIFO Programmable Flag Interrupt Enable*

The 8051 sets BINPFIE=1 to enable an INT4 interrupt when the BINPFIR interrupt request bit makes a zero-to-one transition. This transition indicates that the B-IN FIFO byte count has satisfied the *fullness* level programmed into the programmable FIFO flag register BINPF. The 8051 clears BINPFIE to prevent the associated interrupt request bit from causing an INT4 interrupt.

Bit 1: **BINEFIE** *B-IN FIFO Empty Interrupt Enable*

The 8051 sets BINEFIE=1 to enable an INT4 interrupt when the BINEFIR interrupt request bit makes a zero-to-one transition. This indicates a B-IN FIFO byte count of zero. The 8051 clears BINEFIE to prevent the associated interrupt request bit from causing an INT4 interrupt.

Bit 0: **BINFFIE** *B-IN FIFO Full Interrupt Enable*

The 8051 sets BINFFIE=1 to enable an INT4 interrupt when the BINFFIR interrupt request bit makes a zero-to-one transition. This indicates a B-IN FIFO byte count of 64. The 8051 clears BINFFIE to prevent the associated interrupt request bit from causing an INT4 interrupt.

7.2.10 Input FIFOs A/B Interrupt Requests

ABINIRQ **Input FIFOs A/B Interrupt Requests** **780C**

b7	b6	b5	b4	b3	b2	b1	b0
0	**0**	**AINPFIR**	**AINEFIR**	**AINFFIR**	**BINPFIR**	**BINEFIR**	**BINFFIR**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
x	x	x	x	x	x	x	x

Figure 7-23. Input FIFOs A/B Interrupt Requests

Bit 5: **AINPFIR** *A-IN FIFO Programmable Flag Interrupt Request*

AINPFIR makes a zero-to-one transition when the A-IN FIFO byte count satisfies the required condition programmed into the programmable FIFO flag register AINPF. If enabled by the AINPFIE Bit, this transition causes an INT4 interrupt request.

The 8051 writes a “1” to this bit to clear the interrupt request. The 8051 should clear the 8051 INT4 Bit (EXIF.6) before clearing the AINPFIR Bit in the interrupt service routine to guarantee that pending INT4 interrupts will be recognized.

Bit 4: **AINEFIR** *A-IN FIFO Empty Interrupt Request*

AINEFIR makes a zero-to-one transition when the A-IN FIFO byte count reaches zero (FIFO empty). If enabled by the AINEFIE Bit, this transition causes an INT4 interrupt request.

The 8051 writes “1” to this bit to clear the interrupt request. The 8051 should clear the 8051 INT4 Bit (EXIF.6) before clearing the AINEFIR Bit in the interrupt service routine to guarantee that pending INT4 interrupts will be recognized.

Bit 3: **AINFFIR** *A-IN FIFO Full Interrupt Request*

AINFFIR makes a zero-to-one transition when the A-IN FIFO byte count reaches 64 (FIFO full). If enabled by the AINFFIE Bit, this transition causes an INT4 interrupt request.

The 8051 writes a “1” to this bit to clear the interrupt request. The 8051 should clear the 8051 INT4 Bit (EXIF.6) before clearing the AINFFIR Bit in the interrupt service routine to guarantee that pending INT4 interrupts will be recognized.

Bit 2: **BINPFIR** *B-IN FIFO Programmable Flag Interrupt Request*

BINPFIR makes a zero-to-one transition when the B-IN FIFO byte count satisfies the required condition programmed into the programmable FIFO flag register BINPF. If enabled by the BINPFIE Bit, this transition causes an INT4 interrupt request.

The 8051 writes a “1” to this bit to clear the interrupt request. The 8051 should clear the 8051 INT4 Bit (EXIF.6) before clearing the BINPFIR Bit in the interrupt service routine to guarantee that pending INT4 interrupts will be recognized.

Bit 1: **BINEFIR** *B-IN FIFO Empty Interrupt Request*

BINEFIR makes a zero-to-one transition when the B-IN FIFO byte count reaches zero (FIFO empty). If enabled by the BINEFIE Bit, this transition causes an INT4 interrupt request.

The 8051 writes a “1” to this bit to clear the interrupt request. The 8051 should clear the 8051 INT4 Bit (EXIF.6) before clearing the BINEFIR Bit in the interrupt service routine to guarantee that pending INT4 interrupts will be recognized.

Bit 0: **BINFFIR** *B-IN FIFO Full Interrupt Request*

BINFFIR makes a zero-to-one transition when the B-IN FIFO byte count reaches 64 (FIFO full). If enabled by the BINFFIE Bit, this transition causes an INT4 interrupt request.

The 8051 writes a “1” to this bit to clear the interrupt request. The 8051 should clear the 8051 INT4 Bit (EXIF.6) before clearing the BINFFIR Bit in the interrupt service routine to guarantee that pending INT4 interrupts will be recognized.

7.2.11 FIFO A Write Data

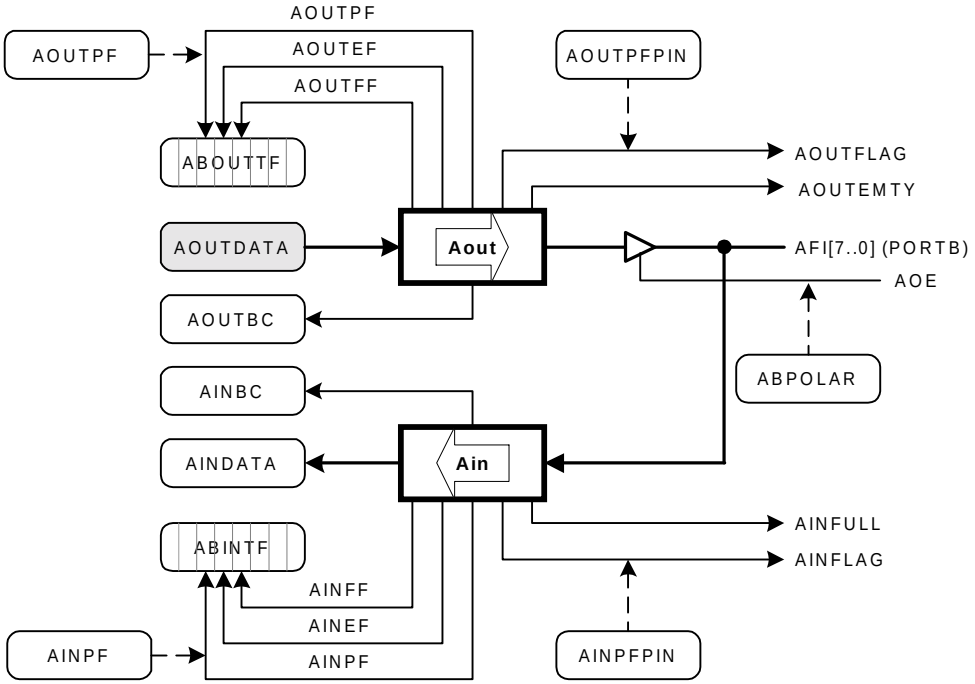


Figure 7-24. AOUTDATA's Role in the FIFO A Register

AOUTDATA	FIFO A Write Data	780E

b7	b6	b5	b4	b3	b2	b1	b0
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
W	W	W	W	W	W	W	W
x	x	x	x	x	x	x	x

Figure 7-25. FIFO A Write Data

Each time the 8051/DMA writes a byte to this register, the A-OUT FIFO advances to the next open position in the FIFO and the AOUTBC (byte count) increments. Writing this register when there are 63 bytes remaining in the A-OUT FIFO sets the A-FIFO Full Flag (AOUTFF, in ABOUTCS.3), which causes an INT4 request. Writing this register when the A-OUT FIFO is full (64 bytes) does not update the FIFO or byte count, and has no effect on the FIFO flags or byte count.

7.2.11.1 A-OUT FIFO Byte Count

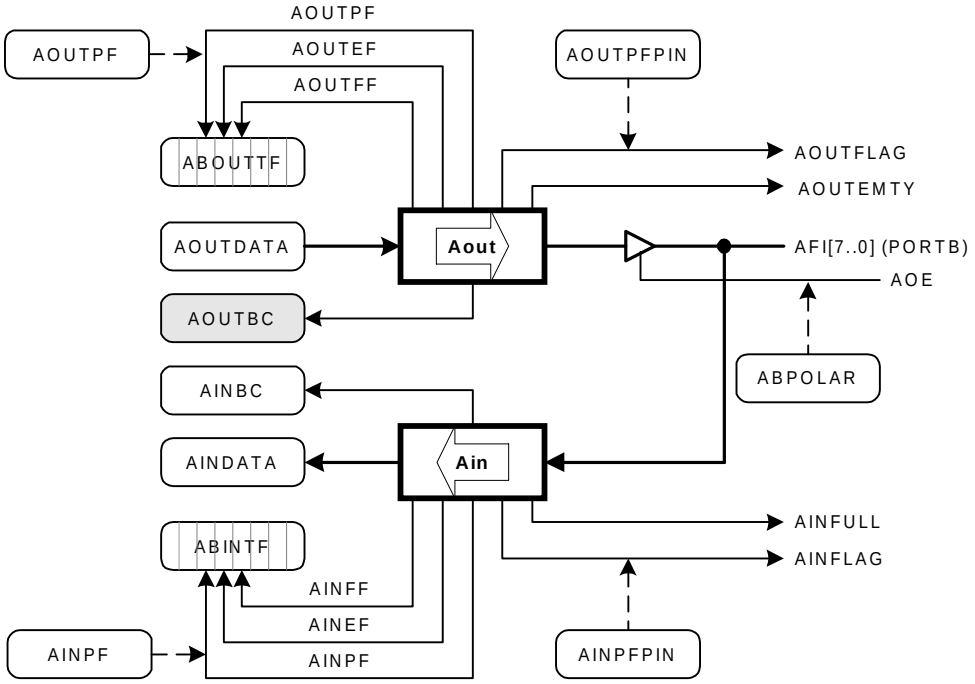


Figure 7-26. AOUTBC's Role in the FIFO A Register

AOUTBC	A-OUT FIFO Byte Count	780F

b7	b6	b5	b4	b3	b2	b1	b0
0	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R	R	R	R	R	R	R	R
0	0	0	0	0	0	0	0

Figure 7-27. Input FIFOs A/B Interrupt Requests

This count reflects the number of bytes remaining in the A-OUT FIFO. Valid byte counts are 0-64. When non-zero, every byte read by outside logic decrements this count, and every 8051 write of AOUTDATA increments this count. If AOUTBC is zero, reading a data byte by outside logic returns indeterminate data and results in the byte count in AOUTBC remaining at zero.

7.2.12 A-OUT FIFO Programmable Flag

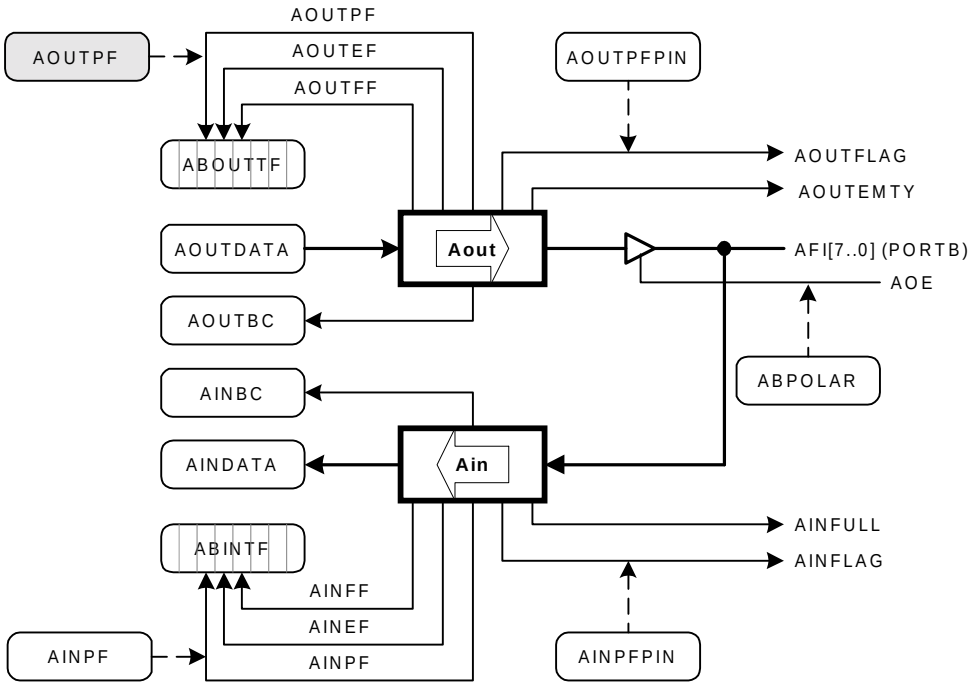


Figure 7-28. AOUTPF's Role in the FIFO A Register

AOUTPF	A-OUT FIFO Programmable Flag	7810

b7	b6	b5	b4	b3	b2	b1	b0
LTGT	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
1	0	1	0	0	0	0	0

Figure 7-29. Input FIFOs A/B Interrupt Requests

This register controls the sense and value for the internal A-OUT FIFO Programmable Flag. The internal flag may be tested by the 8051 and/or enabled to cause an INT4 interrupt request. The default value of the AOUTPF Register indicates a half-full condition.

The 8051 tests the internal FIFO programmable flag by reading the AOUTPF Bit in ABOUTCS.5 (Register at 0x7818).

Bit 7: **LTGT** *Less-than, Greater-than flag*

If LTGT=0, the AOUTPF flag goes true if the number of bytes in the FIFO is less than or equal to the programmed value in D[6..0].

If LTGT=1, the AOUTPF flag goes true if the number of bytes in the FIFO is greater than or equal to the value programmed into D[6..0].

Bit 6-0: **PFVAL** *Programmable Flag Value*

This value, along with the LTGT Bit, determines when the programmable flag for the A-OUT FIFO becomes active. The 8051 programs this register to indicate various degrees of A-OUT FIFO *fullness* to suit the application. The following two sections in this chapter show the interaction of the LTGT Bit and the programmed value for two cases, a filling FIFO and an emptying FIFO.

7.2.12.1 Filling FIFO

When a FIFO is filling with data, it is useful to generate an 8051 interrupt when a programmed level is reached. Because the interrupt request is triggered on a zero-to-one transition of the programmable flag AOUTPF, the LTGT Bit should be set to "1." In Table 7-7, D[6..0] is set for 48 bytes, and the LTGT Bit is set to "1." When the FIFO reaches 48 bytes, the AINPF Bit goes high, generating an interrupt request.

Table 7-7. Filling FIFO

LTGT	D[6..0]	Bytes in FIFO	AOUTPF
1	48	45	0
1	48	46	0
1	48	47	0
1	48	48	1
1	48	49	1
1	48	50	1

7.2.12.2 Emptying FIFO

When a FIFO is being emptied of data, the LTGT Bit should be set to "0," so that the zero-to-one transition of the PF flag (therefore, the interrupt request) occurs when the byte count descends to below the programmed value. In Table 7-8, D[6..0] is set to 48 bytes, and when the FIFO goes from 49 bytes to 48 bytes, the AOUTPF Bit goes high, generating an interrupt request.

Table 7-8. Emptying FIFO

LTGT	D[6..0]	Bytes in FIFO	AOUTPF
0	48	51	0
0	48	50	0
0	48	49	0
0	48	48	1
0	48	47	1
0	48	46	1

7.2.13 A-OUT FIFO Pin Programmable Flag

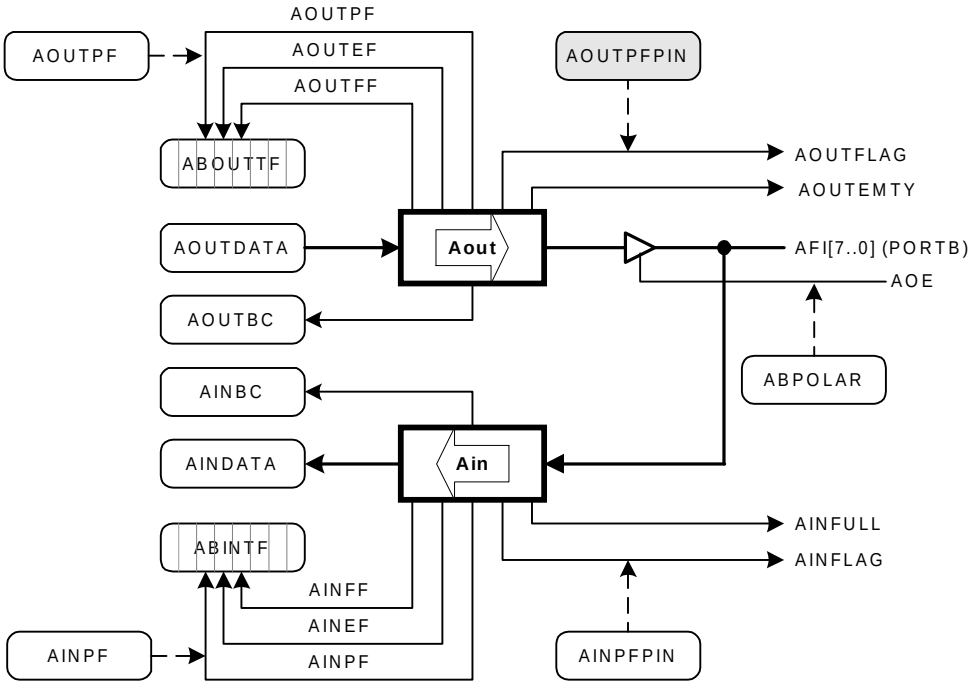


Figure 7-30. AOUTPFPIN's Role in the FIFO A Register

AOUTPFPIN	A-OUT FIFO Pin Programmable Flag	7811

b7	b6	b5	b4	b3	b2	b1	b0
LTGT	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
1	1	0	0	0	0	0	0

Figure 7-31. A-OUT FIFO Pin Programmable Flag

This register controls the sense and value for the A-OUT FIFO Programmable Flag that appears on the AOUTFLAG *pin*. This pin is used by external logic to regulate external reads from the A-OUT FIFO. The AOUTPFPIN Register is programmed with the same data format as the previous register, AOUTPF. The only operational difference is that the flag drives a hardware pin, rather than existing as an internal register bit.

Having separate programmable flags allows the 8051 and external logic to have independent gauges of FIFO fullness. It may be desirable, for example, for one side (8051 or external logic) to have advance notice over the other side about a FIFO becoming full or empty.

The default value of the AOUTPFPIN Register indicates full (bytes in FIFO greater than or equal to 64.

7.2.14 B-OUT FIFO Write Data

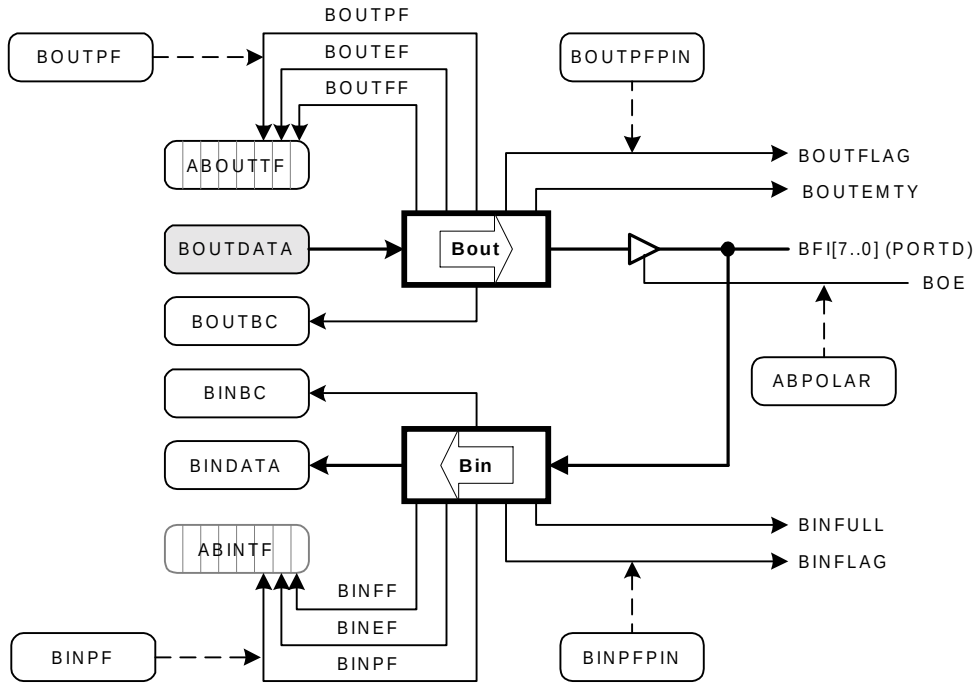


Figure 7-32. BOUTDATA's Role in the FIFO B Register

BOUTDATA	B-OUT FIFO Write Data	7813

b7	b6	b5	b4	b3	b2	b1	b0
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
W	W	W	W	W	W	W	W
x	x	x	x	x	x	x	x

Figure 7-33. B-OUT FIFO Write Data

Each time the 8051/DMA writes a byte to this register, the B-OUT FIFO advances to the next open position in the FIFO and the BOUTBC (Byte count) increments. Writing this register when there are 63 bytes remaining in the B-OUT FIFO sets the B-FIFO Full Flag (BOUTFF, in ABOUTCS.0). This causes an INT4 interrupt request. Writing this register when the B-OUT FIFO is full (64 bytes) does not update the FIFO or byte count, and has no effect on the FIFO flags or byte count.

7.2.15 B-OUT FIFO Byte Count

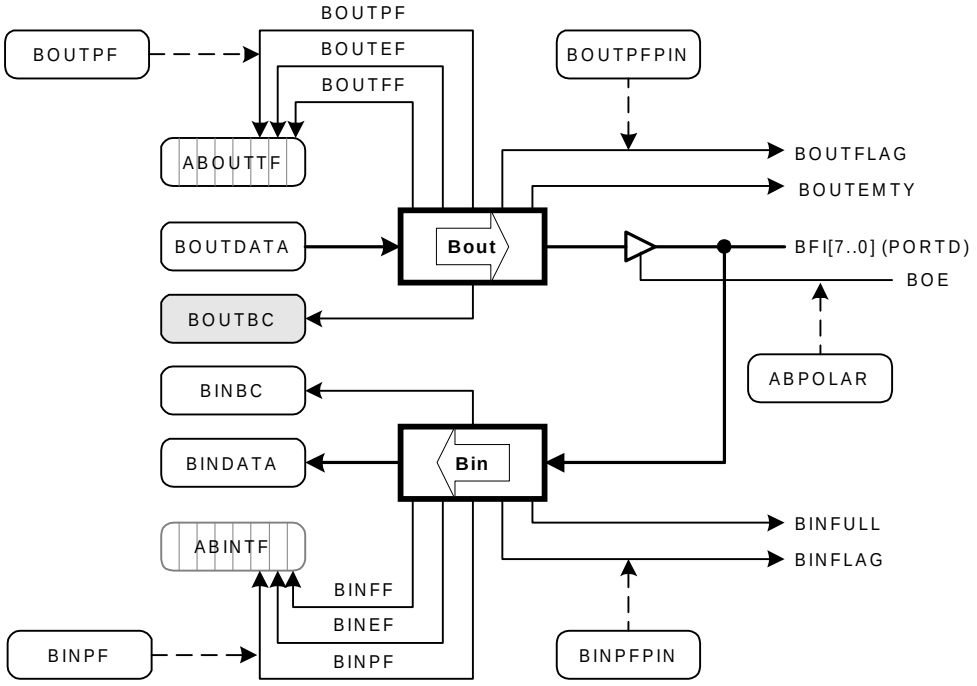


Figure 7-34. BOUTBC's Role in the FIFO B Register

BOUTBC	B-OUT FIFO Byte Count	7814

b7	b6	b5	b4	b3	b2	b1	b0
0	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R	R	R	R	R	R	R	R
0	0	0	0	0	0	0	0

Figure 7-35. B-OUT FIFO Byte Count

This count reflects the number of bytes remaining in the B-OUT FIFO. Valid byte counts are 0-64. When non-zero, every byte read by outside logic decrements this count, and every 8051 write of BOUTDATA increments this count. If BOUTBC is zero, reading a data byte by outside logic returns indeterminate data and results in the byte count in BOUTBC remaining at zero.

7.2.16 B-OUT FIFO Programmable Flag

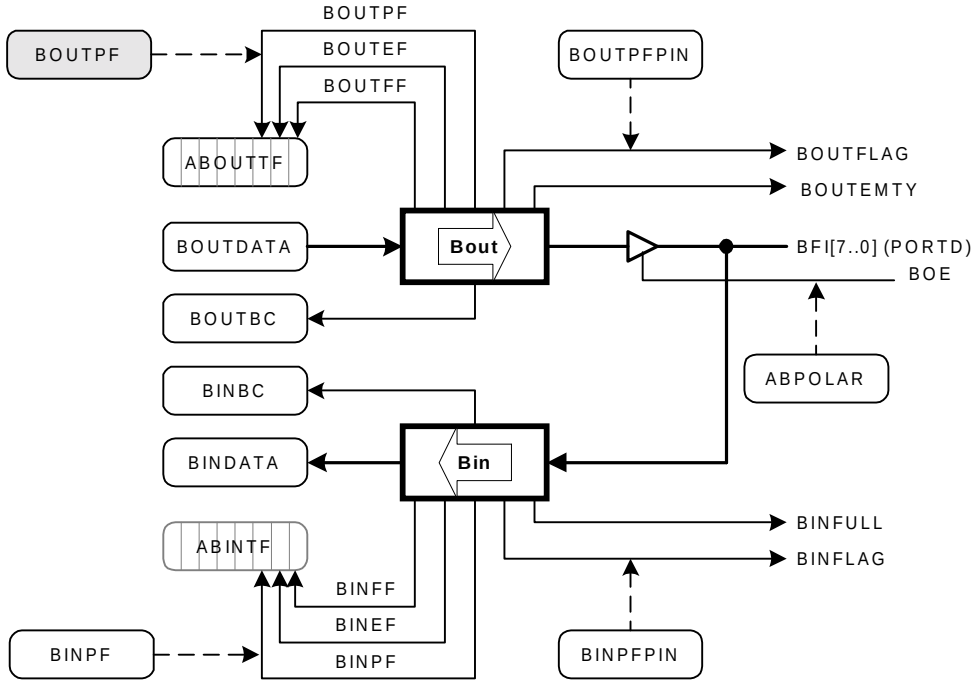


Figure 7-36. BOUTPF's Role in the FIFO B Register

BOUTPF **B-OUT FIFO Programmable Flag** **7815**

b7	b6	b5	b4	b3	b2	b1	b0
LTGT	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
1	0	1	0	0	0	0	0

Figure 7-37. B-OUT FIFO Programmable Flag

This register controls the sense and value for the internal B-OUT FIFO Programmable Flag. The internal flag may be tested by the 8051, and/or enabled to cause an INT4 interrupt request. The default value of the BOUTPF Register indicates a half-full condition.

The 8051 tests the internal FIFO programmable flag by reading the BOUTPF Bit in ABOUTCS.2.

Bit 7: **LTGT** *Less-than, Greater-than flag*

If LTGT=0, the BOUTPF flag goes true if the number of bytes in the FIFO is less than or equal to the programmed value in D[6..0].

If LTGT=1, the BOUTPF flag goes true if the number of bytes in the FIFO is greater than or equal to the value programmed into D[6..0].

Bit 6-0: **PFVAL** *Programmable Flag Value*

This value, along with the LTGT Bit, determines when the programmable flag for the B-FIFO becomes active. The 8051 programs this register to indicate various degrees of B-FIFO *fullness* to suit the application. The following two sections of this chapter show the interaction of the LTGT Bit and the programmed value for two cases, a filling FIFO and an emptying FIFO.

7.2.16.1 Filling FIFO

When a FIFO is filling with data, it is useful to generate an 8051 interrupt when a programmed level is reached. Because the interrupt request is triggered on a zero-to-one transition of the programmable flag BOUTPF, the LTGT Bit should be set to "1." In Table 7-9, D[6..0] is set for 48 bytes and the LTGT Bit is set to "1." When the FIFO reaches 48 bytes, the BOUTPF Bit goes high, generating an interrupt request.

Table 7-9. Filling FIFO

LTGT	D[6..0]	Bytes in FIFO	BOUTPF
1	48	45	0
1	48	46	0
1	48	47	0
1	48	48	1
1	48	49	1
1	48	50	1

7.2.16.2 Emptying FIFO

When a FIFO is being emptied of data, the LTGT Bit should be set to "0," so the zero-to-one transition of the BOUTPF flag (therefore, the interrupt request) occurs when the byte count descends to below the programmed value. In Table 7-10, D[6..0] is set to 48 bytes. When the FIFO goes from 49 bytes to 48 bytes, the BOUTPF Bit goes high, generating an interrupt request.

Table 7-10. Emptying FIFO

LTGT	D[6..0]	Bytes in FIFO	BOUTPF
0	48	51	0
0	48	50	0
0	48	49	0
0	48	48	1
0	48	47	1
0	48	46	1

7.2.17 B-OUT FIFO Pin Programmable Flag

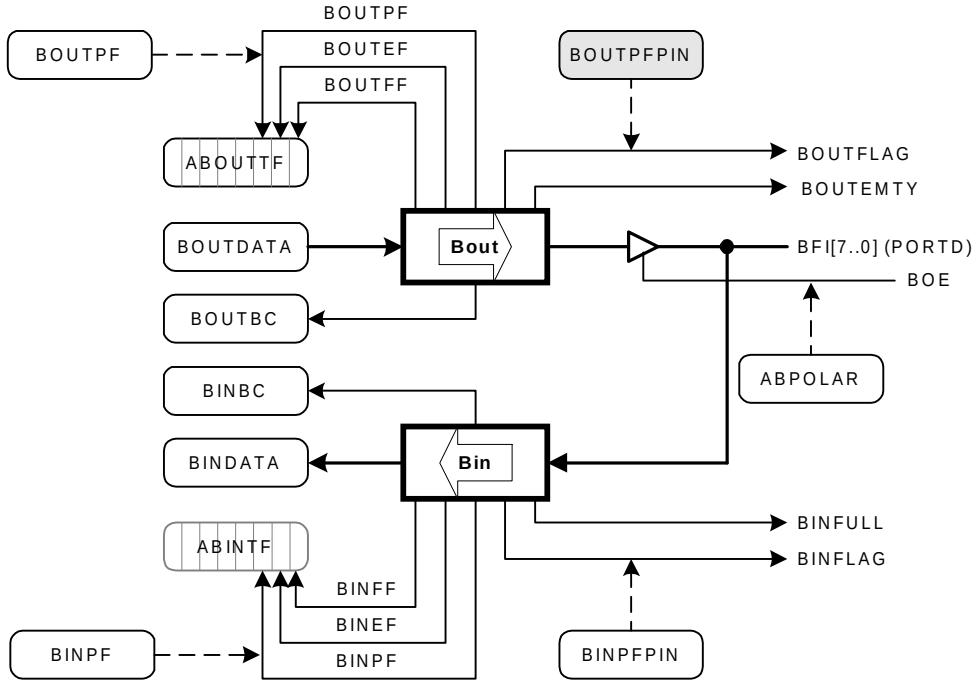


Figure 7-38. BOUTPFPIN's Role in the FIFO B Register

BOUTPFPIN	B-OUT FIFO Pin Programmable Flag	7816

b7	b6	b5	b4	b3	b2	b1	b0
LTGT	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
1	1	0	0	0	0	0	0

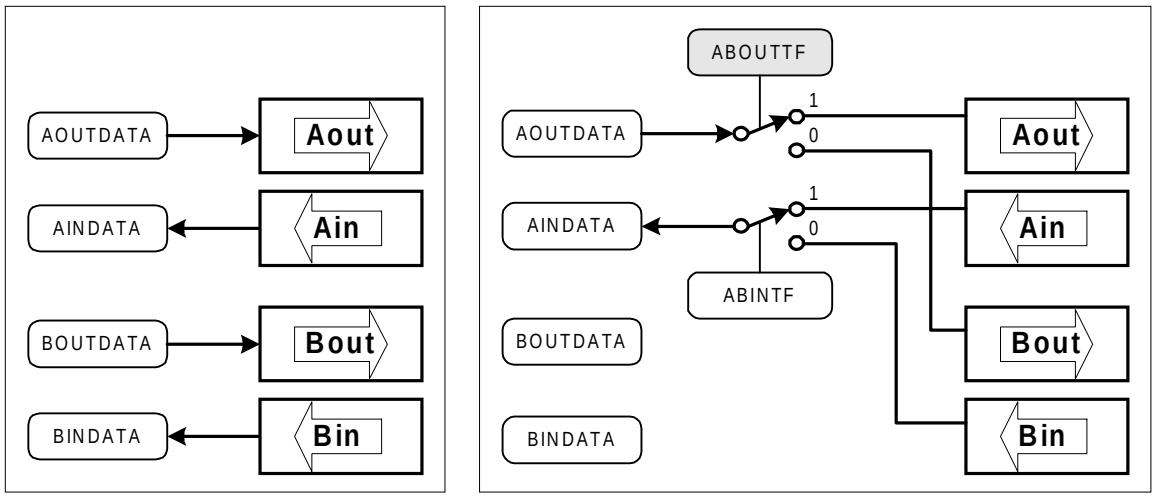
Figure 7-39. B-OUT FIFO Pin Programmable Flag

This register controls the sense and value for the B-OUT FIFO Programmable Flag that appears on the BOUTFLAG *pin*. This pin is used by external logic to regulate external reads from the B-OUT FIFO. The BOUTPFPIN Register is programmed with the same data format as the previous register, BOUTPF. The only operational difference is that the flag drives a hardware pin rather than existing as an internal register bit.

Having separate, programmable flags allows the 8051 and external logic to have independent gauges of FIFO fullness. It may be desirable, for example, for one side (8051 or external logic) to have advance notice over the other side about a FIFO becoming full or empty.

The default value of the BOUTPFPIN Register indicates full.

7.2.18 Output FIFOs A/B Toggle CTL and Flags



(a) Normal Mode (b) 8051 FIFO Toggle Mode

Figure 7-40. 8051 FIFO Toggle Mode vs. Normal Mode Diagram

ABOUTCS **Output FIFOs A/B Toggle CTL and Flags** **7818**

b7	b6	b5	b4	b3	b2	b1	b0
OUTTOG	**OUTSEL**	**AOUTPF**	**AOUTEF**	**AOUTFF**	**BOUTPF**	**BOUTEF**	**BOUTFF**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	1	0	1	0	0	1	0

Figure 7-41. Output FIFOs A/B Toggle CTL and Flags

Bit 7: **OUTTOG** *Enable Output FIFO Toggle*

A special FIFO toggle mode switches automatically between the A-OUT and B-OUT FIFOs each time the 8051 writes data to the AOUTDATA Register. The toggle mechanism works only for programmed 8051 transfers, not DMA transfers.

When OUTTOG=0, the A-OUT and B-OUT FIFOs operate in Normal Mode, as shown by diagram (a) in *Figure 7-40* on the previous page.

When OUTTOG=1, the FIFOs operate in Toggle Mode, as shown by diagram (b) in *Figure 7-40*. The selected FIFO switches between the A-OUT and B-OUT FIFOs after every 8051 write to the AOUTDATA Register. The selected FIFO is indicated by the OUTSEL Bit (Bit 6).

Bit 6: **OUTSEL** *Input Toggle Select*

If OUTTOG=1 when enabling the Toggle Mode:

- This bit selects OUT FIFO A or B when the 8051 writes to the AOUTDATA Register. When OUTSEL=0, the B-OUT FIFO is written. When OUTSEL=1, the A-OUT FIFO is written. When OUTTOG=1, this bit complements automatically (toggles) after every 8051 write to AOUTDATA. This has the effect of automatically toggling between the A-OUT and B-OUT FIFOs for successive 8051 writes to AOUTDATA.
- The 8051 can directly write this bit to select *manually* the A-OUT or B-OUT FIFO. More commonly, the Toggle Mode is used, since it allows 16-bit transfers using the 8051 without requiring the 8051 to switch between the FIFOs.

If OUTTOG=0 when enabling the Toggle Mode:

- The OUTSEL Bit has no effect.

Bit 5: **AOUTPF** *A-OUT FIFO Programmable Flag*

AOUTPF=1 when the number of bytes in the A-OUT FIFO satisfies the requirements programmed into the AOUTPF Register; otherwise, AOUTPF=0. This bit may be tested by the

8051and/or used to generate an interrupt request. A zero-to-one transition of this flag sets the interrupt request bit AOUTPFIR.

Bit 4: **AOUTEF** *A-OUT FIFO Empty Flag*

AOUTEF=1 when the A-OUT FIFO is empty; otherwise, AOUTEF=0. The flag goes valid after external logic reads the last byte in the A-OUT FIFO. This bit may be tested by the 8051, and/or used to generate an interrupt request. A zero-to-one transition of this flag sets the interrupt request bit AOUTEFIR.

Bit 3: **AOUTFF** *A-OUT FIFO Full Flag*

AOUTFF=1 when the A-OUT FIFO is full; otherwise, AOUTFF=0. The flag goes valid after the 8051/DMA writes the 64th byte into the A-OUT FIFO. A zero-to-one transition of this flag sets the interrupt request bit AOUTFFIR.

Bit 2: **BOUTPF** *B-OUT FIFO Programmable Flag*

BOUTPF=1 when the number of bytes in the B-OUT FIFO satisfies the requirements programmed into the BOUTPF Register; otherwise, BOUTPF=0. A zero-to-one transition of this flag sets the interrupt request bit BOUTPFIR.

Bit 1: **BOUTEF** *B-OUT FIFO Empty Flag*

BOUTEF=1 when the B-OUT FIFO is empty; otherwise, BOUTEF=0. The flag goes valid after external logic reads the last byte in the B-OUT FIFO. A zero-to-one transition of this flag sets the interrupt request bit BOUTEFIR.

Bit 0: **BOUTFF** *B-OUT FIFO Full Flag*

BOUTFF=1 when the B-OUT FIFO is full; otherwise, BOUTFF=0. The flag goes valid after the 8051/DMA writes the 64th byte into the B-OUT FIFO. A zero-to-one transition of this flag sets the interrupt request bit BOUTFFIR.

7.2.19 Output FIFOs A/B Interrupt Enables

ABOUTIE **Input FIFOs A/B Interrupt Enables** **7819**

b7	b6	b5	b4	b3	b2	b1	b0
0	**0**	**AOUTPFIE**	**AOUTEFIE**	**AOUTFFIE**	**BOUTPFIE**	**BOUTEFIE**	**BOUTFFIE**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Figure 7-42. Output FIFOs A/B Interrupt Enables

Bit 5: **AOUTPFIE** *A-OUT FIFO Programmable Flag Interrupt Enable*

The 8051 sets AOUTPFIE=1 to enable an INT4 interrupt when the AOUTPFIR interrupt request bit makes a zero-to-one transition. This transition indicates that the A-OUT FIFO byte count has satisfied the *fullness* level programmed into the programmable FIFO flag register AOUTPF.

Bit 4: **AOUTEFIE** *A-OUT FIFO Empty Interrupt Enable*

The 8051 sets AOUTEFIE=1 to enable an INT4 interrupt when the AOUTEFIR interrupt request bit makes a zero-to-one transition. This indicates an A-OUT FIFO byte count of zero (FIFO empty).

Bit 3: **AOUTFFIE** *A-OUT FIFO Bull Interrupt Enable*

The 8051 sets AOUTFFIE=1 to enable an INT4 interrupt when the AOUTFFIR interrupt request bit makes a zero-to-one transition. This indicates an A-OUT FIFO byte count of 64 (FIFO full).

Bit 2: **BOUTPFIE** *B-OUT FIFO Programmable Flag Interrupt Enable*

The 8051 sets BOUTPFIE=1 to enable an INT4 interrupt when the BOUTPFIR interrupt request bit makes a zero-to-one transition. This indicates a B-OUT FIFO byte count has satisfied the *fullness* level programmed into the programmable FIFO flag register BOUTPF.

Bit 1: **BOUTEFIE** *B-OUT FIFO Empty Interrupt Enable*

The 8051 sets BOUTEFIE=1 to enable an INT4 interrupt when the BOUTEFIR interrupt request bit makes a zero-to-one transition. This indicates a B-OUT FIFO byte count of zero (FIFO empty).

Bit 0: **BOUTFFIE** *B-OUT FIFO Full Interrupt Enable*

The 8051 sets BOUTFFIE=1 to enable an INT4 interrupt when the BOUTFFIR interrupt request bit makes a zero-to-one transition. This indicates a B-OUT FIFO byte count of 64 (FIFO full).

7.2.20 Output FIFOs A/B Interrupt Requests

ABOUTIRQ	Output FIFOs A/B Interrupt Requests						781A

b7	b6	b5	b4	b3	b2	b1	b0
0	**0**	**AOUTPFIR**	**AOUTEFIR**	**AOUTFFIR**	**BOUTPFIR**	**BOUTEFIR**	**BOUTFFIR**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
x	x	x	x	x	x	x	x

Figure 7-43. Output FIFOs A/B Interrupt Requests

Bit 5: AOUTPFIR *A-OUT FIFO Programmable Flag Interrupt Request*

AOUTPFIR makes a zero-to-one transition when the A-OUT FIFO byte count satisfies the required condition programmed into the programmable FIFO flag register AOUTPF. If enabled by the AOUTPFIE Bit, this transition causes an INT4 interrupt request.

The 8051 writes a “1” to this bit to clear the interrupt request. The 8051 should clear the 8051 INT4 Bit (EXIF.6) before clearing the AOUTPFIR Bit in the interrupt service routine to guarantee that pending INT4 interrupts will be recognized.

Bit 4: AOUTEFIR *A-OUT FIFO Empty Interrupt Request*

AOUTEFIR makes a zero-to-one transition when the A-OUT FIFO byte count reaches zero (FIFO empty). If enabled by the AOUTEFIE Bit, this transition causes an INT4 interrupt request.

The 8051 writes “1” to this bit to clear the interrupt request. The 8051 should clear the 8051 INT4 Bit (EXIF.6) before clearing the AOUTEFIR Bit in the interrupt service routine to guarantee that pending INT4 interrupts will be recognized.

Bit 3: AOUTFFIR *A-OUT FIFO Full Interrupt Request*

AOUTFFIR makes a zero-to-one transition when the A-OUT FIFO byte count reaches 64 (FIFO full). If enabled by the AOUTFFIE Bit, this transition causes an INT4 interrupt request.

The 8051 writes a “1” to this bit to clear the interrupt request. The 8051 should clear the 8051 INT4 Bit (EXIF.6) before clearing the AOUTFFIR Bit in the interrupt service routine to guarantee that pending INT4 interrupts will be recognized.

Bit 2: **BOUTPFIR** *B-OUT FIFO Programmable Flag Interrupt Request*

BOUTPFIR makes a zero-to-one transition when the B-OUT FIFO byte count satisfies the required condition programmed into the programmable FIFO flag register BOUTPF. If enabled by the BOUTPFIE Bit, this transition causes an INT4 interrupt request.

The 8051 writes a "1" to this bit to clear the interrupt request. The 8051 should clear the 8051 INT4 Bit (EXIF.6) before clearing the BOUTPFIR Bit in the interrupt service routine to guarantee that pending INT4 interrupts will be recognized.

Bit 1: **BOUTEFIR** *B-OUT FIFO Empty Interrupt Request*

BOUTEFIR makes a zero-to-one transition when the B-OUT FIFO byte count reaches zero (FIFO empty). If enabled by the BOUTEFIE Bit, this transition causes an INT4 interrupt request.

The 8051 writes a "1" to this bit to clear the interrupt request. The 8051 should clear the 8051 INT4 Bit (EXIF.6) before clearing the BOUTEFIR Bit in the interrupt service routine to guarantee that pending INT4 interrupts will be recognized.

Bit 0: **BOUTFFIR** *B-OUT FIFO Full Interrupt Request*

BOUTFFIR makes a zero-to-one transition when the B-OUT FIFO byte count reaches 64 (FIFO full). If enabled by the BOUTFFIE Bit, this transition causes an INT4 interrupt request.

The 8051 writes a "1" to this bit to clear the interrupt request. The 8051 should clear the 8051 INT4 Bit (EXIF.6) before clearing the BOUTFFIR Bit in the interrupt service routine to guarantee that pending INT4 interrupts will be recognized.

7.2.21 FIFO A/B Setup

ABSETUP **FIFO A/B Setup** **781C**

b7	b6	b5	b4	b3	b2	b1	b0
0	**0**	**ASYNC**	**DBLIN**	**0**	**OUTDLY**	**0**	**DBLOUT**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Figure 7-44. FIFO A/B Setup

Bit 5: **ASYNC** *Select SYNC/ASYNC Slave FIFO Clocking*

The ASYNC Bit controls how external logic synchronizes accesses to the A and B FIFOs.

When the 8051 sets ASYNC=1, the A and B FIFOs operate asynchronously, whereby the SLRD (Slave FIFO-READ) and SLWR (Slave FIFO-WRITE) pins are used as direct read and write strobes.

When the 8051 sets ASYNC=0, the A and B FIFOs operate synchronously, whereby the SLRD and SLWR pins are used as enable signals for the externally supplied FIFO clock XCLK. The polarity of the enables, active-high or active-low, is controlled by the ABPOLAR Register (Section 7.2.22. *"FIFO A/B Control Signal Polarities"*).

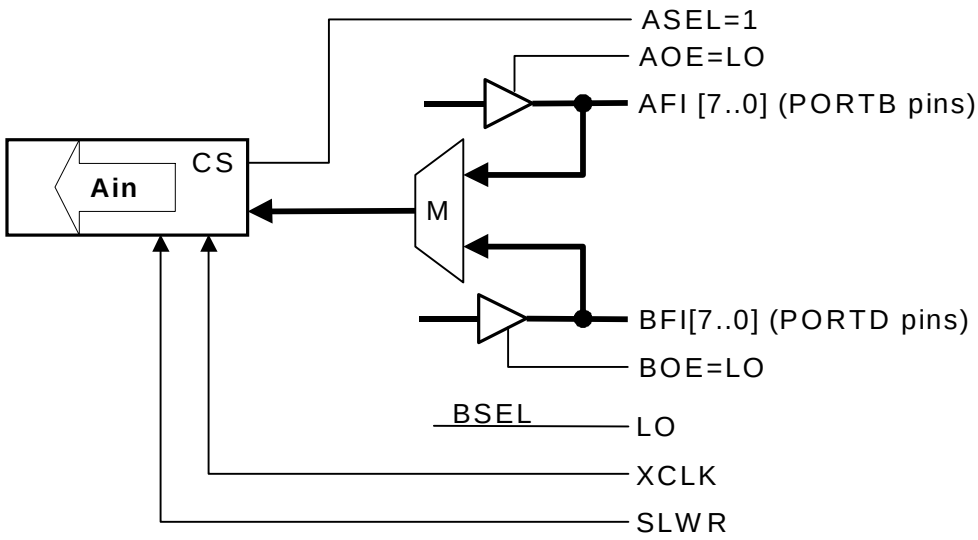


Figure 7-45. A-IN FIFO Double-Byte Mode

Bit 4: DBLIN *Enable IN Double-Byte Mode*

The 8051 sets DBLIN=1 to turn on the IN-FIFO double-byte mode. *Figure 7-45* illustrates the double-byte mode for the A-IN FIFO. The B-IN FIFO may also use this mode, in which case the outside logic sets ASEL=0 and BSEL=1. For this illustration, signals ASEL, BSEL, AOE, and BOE are programmed to be active high polarity.

In double-byte mode, external logic writes 16 bits of data into the A-IN or B-IN FIFO each time it asserts the SLWR signal. The double-byte mode automatically writes two bytes for every SLWR pulse in ASYNC mode or two bytes for every clock pulse in SYNC mode. The bytes are taken from PORTD and PORTB, in that order. This provides a very efficient mechanism for transferring 16-bit data into the 8-bit slave FIFOs.

If synchronous clocking is used in double-byte mode, consecutive writes must be separated by at least one XCLK period to give the internal logic time to write both bytes into the FIFO. This clocking restriction applies only to the double-byte mode. In normal operation, one byte per clock can be loaded into a slave IN-FIFO.

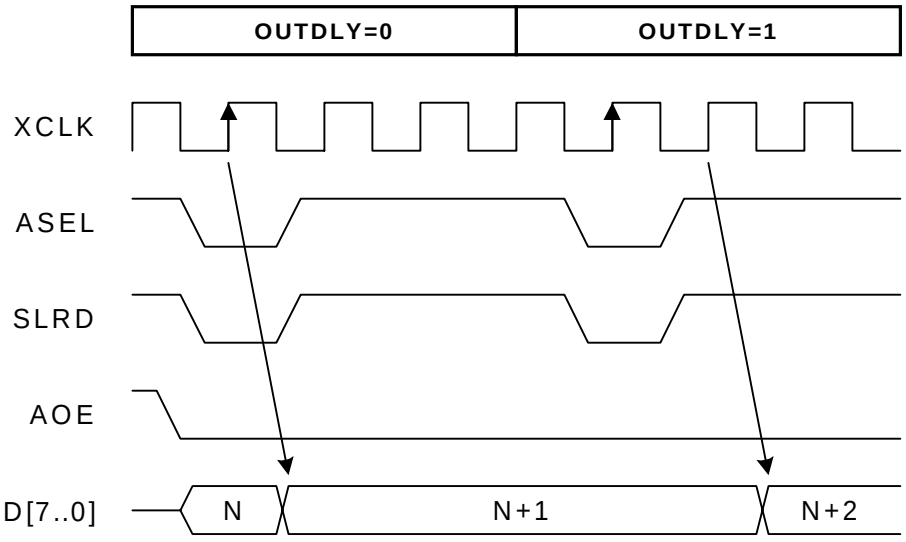


Figure 7-46. A-OUT FIFO Delay Synchronous Reads

Bit 2: **OUTDLY** *Delay Synchronous Reads*

The OUTDLY Bit affects only synchronous reads of a slave FIFO. When OUTDLY=0, output data is valid on the clock edge that corresponds to the SLRD signal being valid. When OUTDLY=1, the output data is valid one clock later.

Figure 7-46 shows two synchronous reads of the A-OUT FIFO, with the OUTDLY Bit first equal to 0, then equal to 1. For this example, the SLRD, AOE, and ASEL signals are programmed to be active low.

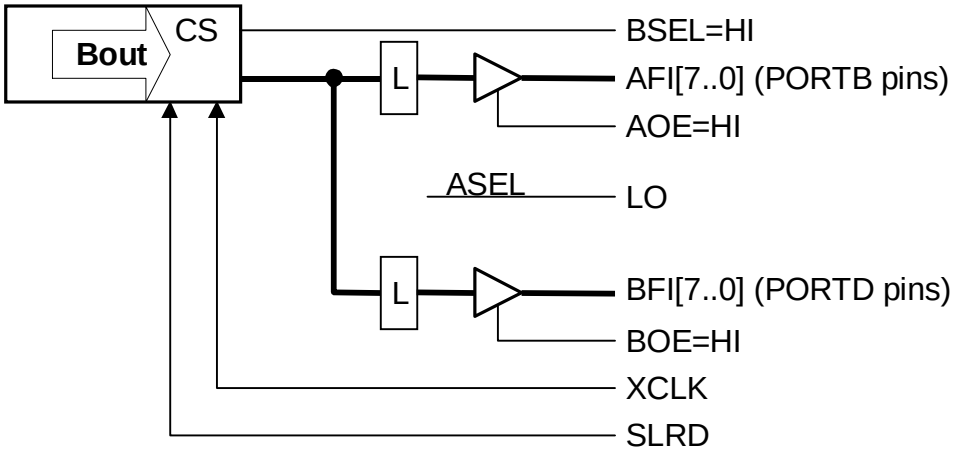


Figure 7-47. B-OUT FIFO Double-Byte Mode

Bit 0: **DBLOUT** *Enable FIFO-OUT Double-Byte Mode*

The 8051 sets DBLOUT=1 to turn on the OUT-FIFO double-byte mode. *Figure 7-47* illustrates the double-byte mode for the B-OUT FIFO. The A-OUT FIFO may also use this mode, in which case the outside logic sets ASEL=1 and BSEL=0. For this illustration, signals ASEL, BSEL, AOE, and BOE are programmed to be active high polarity.

The double-byte mode automatically provides two FIFO bytes on PORTDand PORTB, in that order, for every SLRD pulse in ASYNC mode or two bytes for every clock pulse in SYNC mode. This provides a very efficient mechanism for transferring 16-bit data out of the 8-bit slave FIFOs.

In SYNC mode, consecutive reads must be separated by at least one XCLK period, to give the internal logic time to retrieve both bytes from the FIFO.

7.2.22 FIFO A/B Control Signal Polarities

ABPOLAR **FIFO A/B Control Signal Polarities** **781D**

b7	b6	b5	b4	b3	b2	b1	b0
0	**0**	**BOE**	**AOE**	**SLRD**	**SLWR**	**ASEL**	**BSEL**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0

Figure 7-48. FIFO A/B Control Signal Polarities

These bits define the pin polarities for the indicated signals. The 8051 sets a bit LOW for active low, and HI for active high. The default setting for all FIFO A/B control signals is active low polarity.

7.2.23 FIFO Flag Reset

ABFLUSH	Reset All FIFO Flags	781E

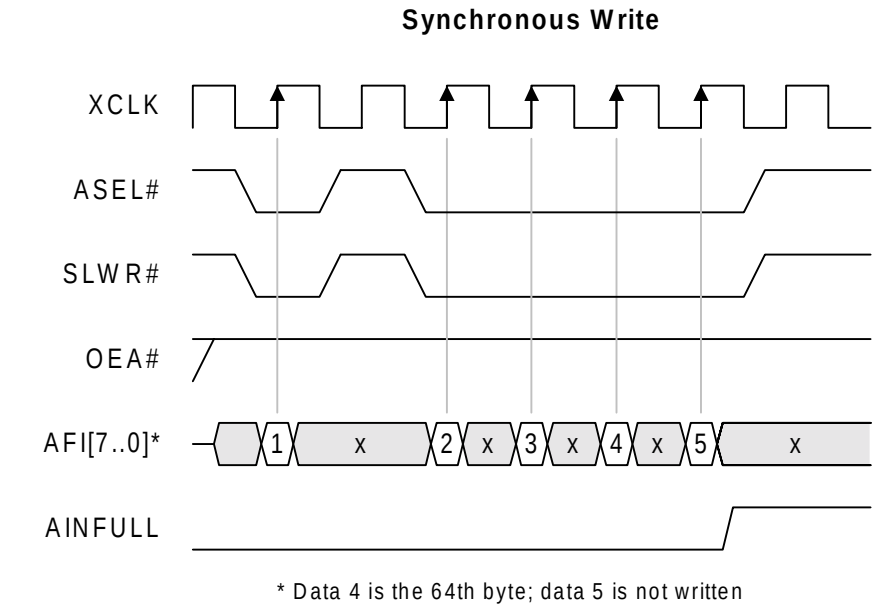
b7	b6	b5	b4	b3	b2	b1	b0
x	**x**	**x**	**x**	**x**	**x**	**x**	**x**
W	W	W	W	W	W	W	W
x	x	x	x	x	x	x	x

Figure 7-49. FIFO Flag Reset

The 8051 writes any value to this register to reset the FIFO byte counts to zero, effectively *flushing* the FIFOs. Consequently, the byte counts are set to zero, the empty flags are set, and the full flags are cleared.

Reading this register returns indeterminate data.

7.3 FIFO Timing



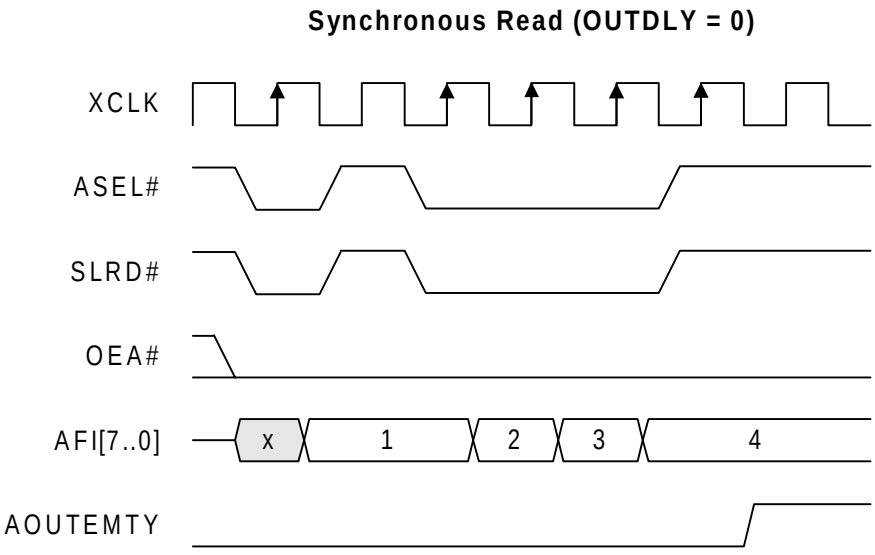


Figure 7-50. Synchronous Write/Read Timing

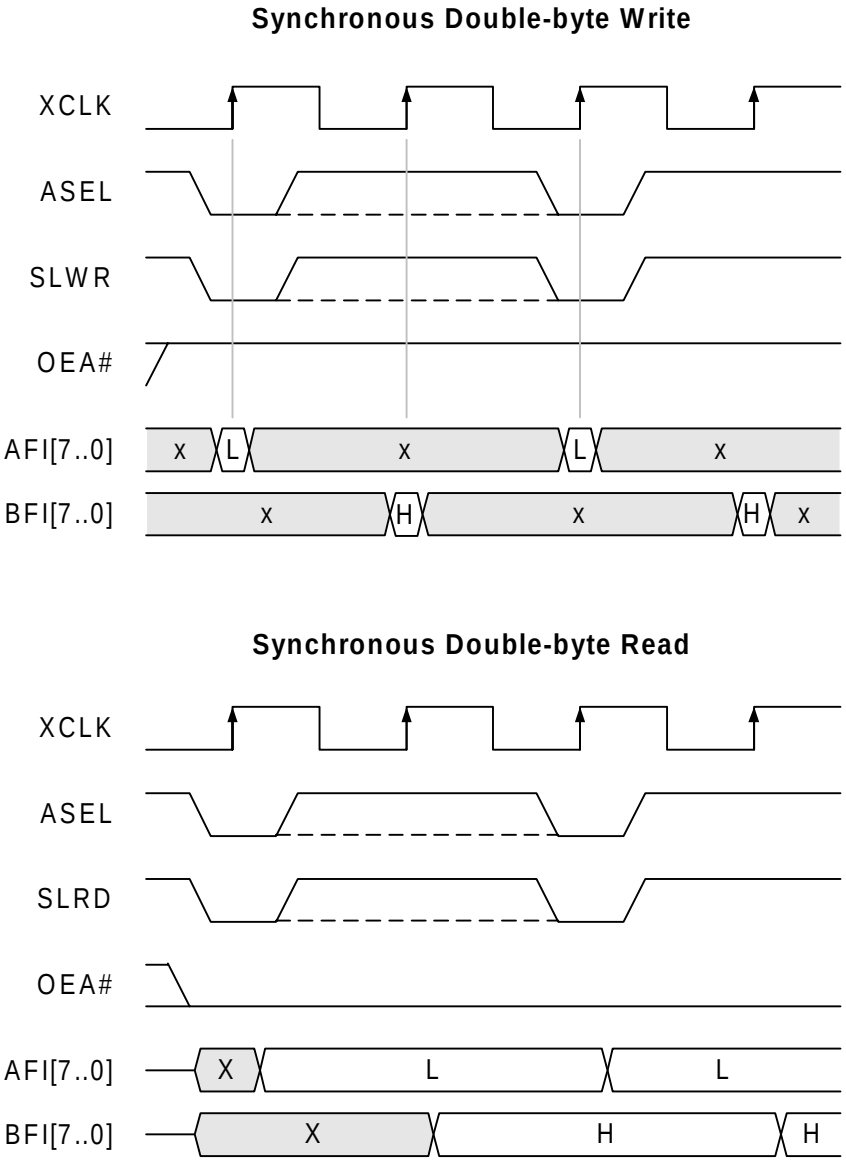


Figure 7-51. Synchronous Double-byte Write/Read