EZ-USB FX2
Technical Reference Manual

Cypress Semiconductor
3901 North First Street
San Jose, CA 95134
Tel.: (800) 858-1810 (toll-free in the U.S.)
(408) 943-2600
www.cypress.com

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.

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Cypress Semiconductor Corporation.

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 product.

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

EZ-USB FX2 Technical Reference Manual, Version 2.1.

List of Trademarks

Cypress, the Cypress Logo, EZ-USB, Making USB Universal, Xcelerator, and ReNumeration are trademarks or registered trademarks of Cypress Semiconductor Corporation. Macintosh is a registered trademark of Apple Computer, Inc. Windows is a registered trademark of Microsoft Corporation. I²C is a registered trademark of Philips Electronics. All other product or company names used in this manual may be trademarks, registered trademarks, or servicemarks of their respective owners.

Table of Contents

Chapter 1. Introducing EZ-USB *FX2*

Chapter 2. Endpoint Zero

Chapter 3. Enumeration and ReNumeration™

Chapter 4. Interrupts

Chapter 5. Memory

Chapter 6. Power Management

Chapter 7. Resets

Chapter 8. Access to Endpoint Buffers

Chapter 9. Slave FIFOs

Chapter 10. General Programmable Interface (GPIF)

Chapter 11. CPU Introduction

Chapter 12. Instruction Set

Chapter 13. Input/Output

Chapter 14. Timers/Counters and Serial Interface

Chapter 15. Registers

List of Figures

(List of Figures)

(List of Figures)

(List of Figures)

List of Tables

Chapter 1 Introducing EZ-USB FX2

1.1 Introduction

The Universal Serial Bus (USB) has gained wide acceptance as the connection method of choice for low and medium speed PC peripherals. Equally successful in the Windows and Macintosh worlds, USB has delivered on its promises of easy attachment, an end to configuration hassles, and true plug-and-play operation.

The second generation of the USB specification, “USB 2.0”, extends the original specification to include:

- 480 Mbits/sec signaling rate, a 40× improvement over the USB 1.1 rate of 12 Mbits/sec.
- Full backward and forward compatibility with USB 1.1 devices and cables.
- A new hub architecture that can provide multiple 12 Mbits/sec downstream ports for USB 1.1 devices.

The Cypress Semiconductor EZ-USB FX2 (often abbreviated as “FX2” in this manual) is a single-chip USB 2.0 peripheral whose architecture is similar to that of the Cypress Semiconductor EZ-USB FX family. Although much of the FX architecture is preserved, certain elements have been redesigned to accommodate the higher data rates offered by USB 2.0.

This introductory chapter begins with a brief USB tutorial to put USB and FX2 terminology into context. The remainder of the chapter briefly outlines the FX2 architecture.

1.2 An Introduction to USB

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.

- A USB device can be plugged in anytime, even while 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 auto-

matically 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 simultaneously available to dozens of devices.
- USB is fast enough for printers, hard disk drives, CD-quality audio, and scanners.
- With the introduction of the USB 2.0 Specification, USB supports three speeds:
  - *Low Speed* (1.5 Mbits/sec), suitable for mice, keyboards and joysticks.
  - *Full Speed* (12 Mbits/sec), for devices like modems, speakers and scanners.
  - *High Speed* (480 Mbits/sec), for devices like hard disk drives, CD-ROMs, video cameras, and high-resolution scanners.

The Cypress Semiconductor EZ-USB FX2 augments the EZ-USB family by supporting the high bandwidth offered by the USB 2.0 High Speed mode. The FX2 provides a highly-integrated solution for a USB peripheral device. Like all EZ-USB devices, the FX2 offers the following features:

- An integrated, high-performance CPU based on the industry-standard 8051 processor.
- A *soft* (RAM-based) architecture that allows unlimited configuration and upgrades.
- Full USB throughput. USB devices that use EZ-USB chips are not limited by number of endpoints, buffer sizes, or transfer speeds.
- Automatic handling of most of the USB protocol, which simplifies code and accelerates the USB learning curve.

1.3 The USB Specification

The *Universal Serial Bus Specification Version 2.0* is available on the Internet from the USB Implementers Forum, Inc., at **http://www.usb.org**. Published in April, 2000, the USB Specification is the work of a founding committee of seven industry heavyweights: Compaq, Hewlett-Packard, Lucent, Philips, Intel, Microsoft, and NEC. This impressive list of developers secures USB's position as the low- to high-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 older serial or parallel ports. 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 information about such details as connector types and wire colors.

1.4 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 among 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”. This is the only case in which the USB device is the initiator; in all other cases, 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.

1.5 USB Direction

Because the host is always 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. FX2 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 FX2 *sends* data to the host by loading an IN endpoint buffer. Likewise, the FX2 *receives* host data from an OUT endpoint buffer.

1.6 Tokens and PIDs

In this manual, you’ll read statements such as: “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, **DATA2**, **MDATA**
Handshake	ACK, NAK, STALL, **NYET**
Special	PRE, **ERR**, **SPLIT**, **PING**

Bold type indicates PIDs introduced with USB 2.0

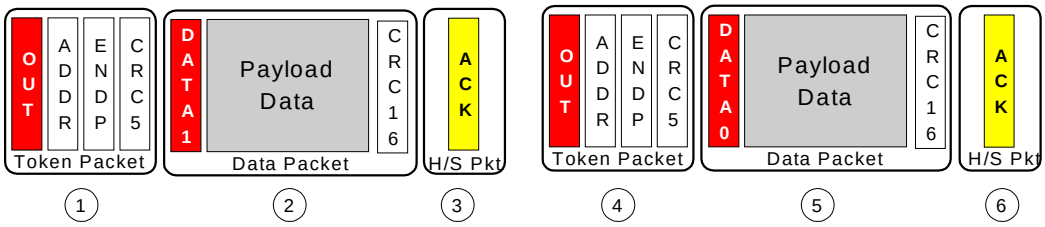


Figure 1-1. USB Packets

Figure 1-1 illustrates a USB OUT transfer. Host traffic is shown in solid shading, while device traffic is shown crosshatched. 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 1-1, 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**.

When operating at full speed, every OUT transfer sends the OUT data, even when the device is busy and can't accept the data. When operating at high speed, this slightly wasteful use of USB bandwidth is remedied by using the new "Ping" PID. The host first sends a short PING token to an OUT endpoint, asking if there is room for OUT data in the peripheral device. Only when the PING is answered by an ACK does the host send the OUT token and data.

There are *two* DATA PIDs (DATA0 and DATA1) in Figure1-1 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 received Data PID with the state of its own internal toggle bit, the receiver 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.

At full speed, SOF (Start of Frame) tokens occur once per millisecond. At high speed, each frame contains eight SOF tokens, each denoting a 125-microsecond *microframe.*

Four handshake PIDs indicate the status of a USB transfer:

- ACK ("Acknowledge") means *success*; the data was received error-free.
- NAK ("Negative Acknowledge") 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 host and device software). 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.
- NYET ("Not Yet") has the same meaning as ACK — the data was received error-free — but also indicates that the endpoint is not yet ready to receive *another* OUT transfer. NYET PIDs occur only in high speed mode.

A PRE (Preamble) PID precedes a low-speed (1.5 Mbits/sec) USB transmission. The FX2 supports full-speed (12 Mbits/sec) and high-speed (480 Mbits/sec) USB transfers only.

1.6.1 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.

1.6.2 Sending Data to the Host

A USB device never spontaneously sends data to the host. Either FX2 firmware or external logic can load data into an FX2 endpoint buffer and 'arm' it for transfer at any time. However, the data is not transmitted to the host until the host issues an IN request to the FX2 endpoint. If the host never sends the IN token, the data remains in the FX2 endpoint buffer indefinitely.

1.7 USB Frames

The USB host provides a time base to all USB devices by transmitting an SOF ("Start of Frame") packet every millisecond. SOF packets include an 11-bit number which increments once per frame; the current frame number [0-2047] may be read from internal FX2 registers at any time.

At high speed (480 Mbits/sec), each one-millisecond frame is divided into eight 125-microsecond *microframes,* each of which is preceded by an SOF packet. The frame number still increments only once per millisecond, so each of those SOF packets contains the same frame number. To keep track of the current microframe number [0-7], the FX2 provides a readable microframe counter.

The FX2 can generate an interrupt request whenever it receives an SOF (once every millisecond at full speed, or once every 125 microseconds at high speed). This SOF interrupt can be used, for example, to service isochronous endpoint data.

1.8 USB Transfer Types

USB defines four transfer types. These match the requirements of different data types delivered over the bus.

1.8.1 Bulk Transfers

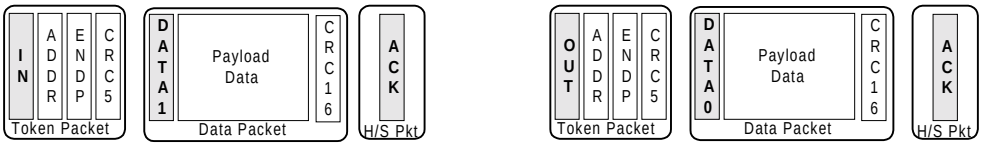


Figure 1-2. Two Bulk Transfers, IN and OUT

Bulk data is *bursty*, traveling in packets of 8, 16, 32 or 64 bytes at full speed or 512 bytes at high speed. Bulk data has guaranteed accuracy, due to an automatic retry 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

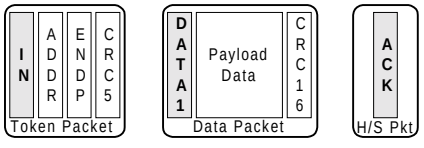


Figure 1-3. An Interrupt Transfer

Interrupt data is like bulk data; it can have packet sizes of 1 through 64 bytes at full speed or up to 1024 bytes at high speed. Interrupt endpoints have an associated polling interval that ensures they will be polled (receive an IN token) by the host on a regular basis.

1.8.3 Isochronous Transfers

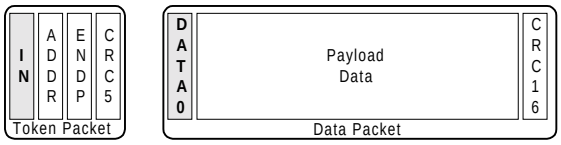


Figure 1-4. An Isochronous Transfer

Isochronous data is time-critical and used to *stream* data like audio and video. An isochronous packet may contain up to 1023 bytes at full speed, or up to 1024 bytes at high speed.

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/NYET), and no retries; error detection is limited to a 16-bit CRC.

Isochronous transfers do not use the data-toggle mechanism. Full-speed isochronous data uses only the DATA0 PID; high-speed isochronous data uses DATA0, DATA1, DATA2 and MDATA.

In full-speed mode, only one isochronous packet can be transferred per endpoint, per frame. In high-speed mode, up to three isochronous packets can be transferred per endpoint, per microframe.

1.8.4 Control Transfers

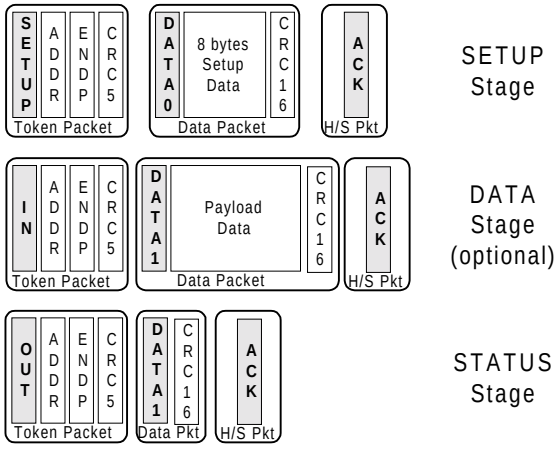


Figure 1-5. A Control Transfer

Control transfers configure and send commands to a device. Because they're so important, they employ the most extensive USB error checking. The host reserves a portion of each USB frame 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 has 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 (all USB 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 identify itself.
3. The host sends a *Set Address* request, which assigns a unique address to the just-attached device so it may be distinguished from the other devices connected to the bus.
4. The host sends more *Get Descriptor* requests, asking for additional device information. From this, it learns everything else about the device: number of endpoints, power requirements, required bus bandwidth, what driver to load, etc.

This sign-on process is called *Enumeration*.

1.9.1 Full-Speed / High-Speed Detection

The USB 2.0 Specification requires that high-speed (480 Mbit/sec) devices must also be capable of enumerating at full-speed (12 Mbit/s). In fact, all high-speed devices begin the enumeration process in full-speed mode; devices switch to high-speed operation only after the host and device have *agreed* to operate at high speed. The high-speed negotiation process occurs during USB reset, via the "Chirp" protocol described in Chapter 7 of the USB 2.0 Specification.

When connected to a full-speed host, the FX2 will enumerate as a full-speed device. When connected to a high-speed host, the FX2 automatically switches to high-speed mode.

1.10 The Serial Interface Engine (SIE)

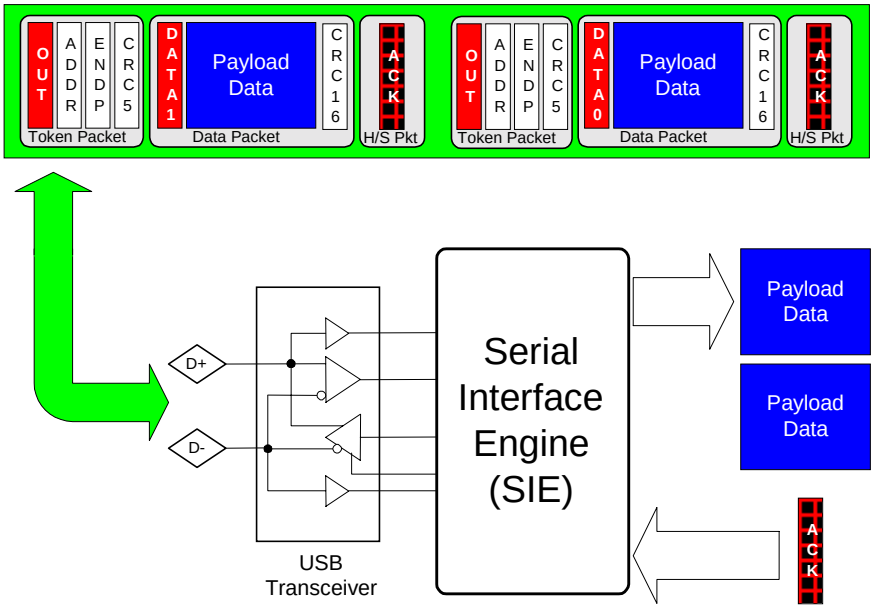


Figure 1-6. What the SIE Does

Every USB device has a Serial Interface Engine (SIE) which connects to the USB data lines (D+ and D-) and delivers data to and from the USB device. Figure 1-6 illustrates the SIE's role: it decodes the packet PIDs, performs error checking on the data using the transmitted CRC bits, and delivers payload data to the USB device.

Bulk transfers 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 USB device has successfully transferred the data, it commands the SIE to send an ACK handshake packet, indicating success. 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 retransmit the data at a later time.

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 D+ and D-. Because 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 automatically by the FX2's SIE.

One of the most important features of the FX2 (and the other EZ-USB chips) family is that its configuration is *soft*. Instead of requiring ROM or other fixed memory, it contains internal program/data

RAM which can be loaded over the USB. This makes modifications, specification revisions, and updates a snap.

The FX2's "smart" SIE performs much more than the basic functions shown in Figur e1-6; it can perform a full enumeration by itself, which allows the FX2 to connect as a USB device and download code into its RAM while its CPU is held in reset. This added SIE functionality is also made available to the FX2 programmer, to make development easier and save code and processing time.

1.11 ReNumeration™

Because the FX2's configuration is *soft*, one chip can take on the identities of multiple distinct USB devices.

When first plugged into USB, the FX2 enumerates automatically and downloads firmware and USB descriptor tables over the USB cable. Next, the FX2 enumerates again, this time as a device 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 download step has occurred.

Alternately, FX2 can also load its firmware from an external EEPROM.

Chapter 3, "Enumeration and ReNumeration™" describes these processes in detail.

1.12 EZ-USB FX2 Architecture

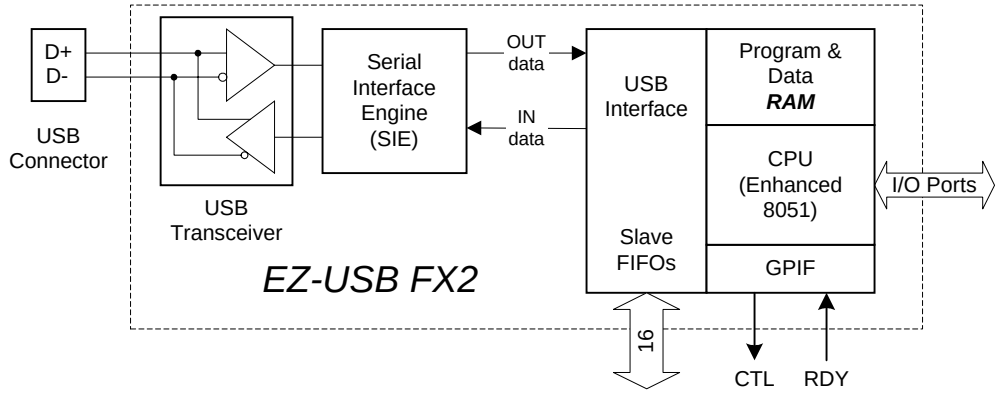


Figure 1-7. FX2 56-pin Package Simplified Block Diagram

The FX2 packs all the intelligence required by a USB peripheral interface into a compact integrated circuit. As Figure 1-7 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 the other signaling-level tasks required by USB. Ultimately, the SIE transfers parallel data to and from the USB interface.

The FX2 SIE operates at Full Speed (12 Mbits/sec) and High Speed (480 Mbits/sec) rates. To accommodate the increased bandwidth of USB 2.0, the FX2 endpoint FIFOs and slave FIFOs (which interface to external logic or processors) are unified to eliminate internal data transfer times.

The CPU is an enhanced 8051 with fast execution time and added features. It uses internal RAM for program and data storage.

The role of the CPU in a typical FX2-based USB peripheral is twofold:

- It implements the high-level USB protocol by servicing host requests over the control endpoint (endpoint zero)
- It is available for general-purpose system use

The high-level USB protocol is not bandwidth-critical, so the FX2's CPU is well-suited for handling host requests over the control endpoint. However, the data rates offered by USB 2.0 are too high for the CPU to process the USB data directly. For this reason, the CPU is not usually in the high-bandwidth data path between endpoint FIFOs and the external interface. Instead, *the CPU simply configures the interface, then "gets out of the way" while the unified FX2 FIFOs move the data directly between the USB and the external interface.*

The FIFOs can be controlled by an external master, which either supplies a clock and clock-enable signals to operates synchronously, or strobe signals to operate asynchronously.

Alternately, the FIFOs can be controlled by an internal FX2 timing generator called the General Programmable Interface (GPIF). The GPIF serves as an *internal* master, interfacing directly to the FIFOs and generating user-programmed control signals for the interface to external logic. Additionally, the GPIF can be made to wait for external events by sampling external signals on its RDY pins. The GPIF runs much faster than the FIFO data rate to give good programmable resolution for the timing signals. It can be clocked from either the internal FX2 clock or an externally supplied clock.

The FX2's CPU is rich in features. Up to five I/O ports are available, as well as two USARTs, three counter/timers, and an extensive interrupt system. It runs at a clock rate of up to 48 MHz and uses four clocks per instruction cycle instead of the twelve required by a standard 8051.

The FX2 chip family uses an enhanced SIE/USB interface which simplifies FX2 code by implementing much of the USB protocol. In fact, the FX2 can function as a full USB device even without firmware.

Like all EZ-USB family chips, FX2 operates at 3.3V. This simplifies the design of bus-powered USB devices, since the 5V power available at the USB connector (which the USB Specification allows to be as low as 4.4V) can drive a 3.3V regulator to deliver clean, isolated power to the FX2 chip.

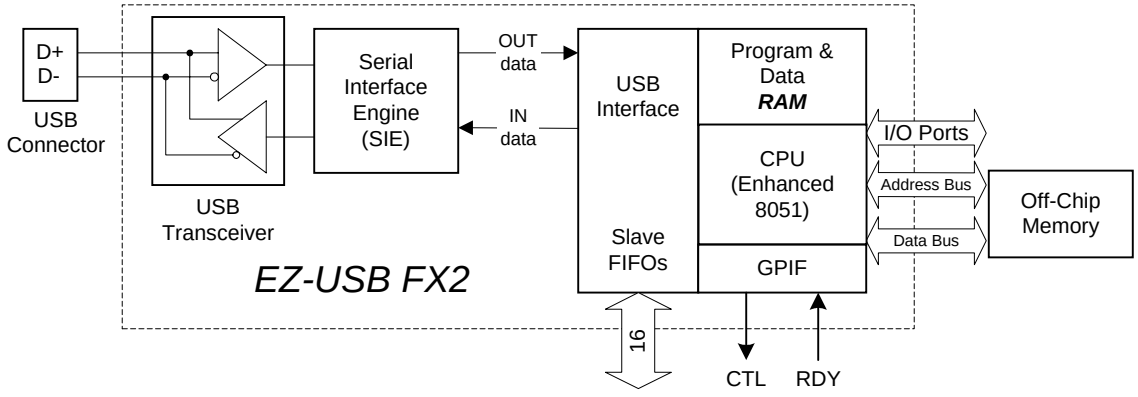


Figure 1-8. FX2 128-pin Package Simplified Block Diagram

FX2 is available in a 128-pin package which brings out the 8051 address bus, data bus, and control signals to allow connection of external memory and/or memory-mapped I/O. Figure 1-8 is a block diagram for this package; *Chapter 5, "Memory",* gives full details of the external-memory interface.

1.13 FX2 Feature Summary

FX2 includes the following features:

- On-chip 480 Mbits/sec transceiver, PLL and SIE—the entire USB 2.0 physical layer (PHY).
- Double-, triple- and quad-buffered endpoint FIFOs accommodate the 480 MBits/sec USB 2.0 data rate.
- Built-in, enhanced 8051 running at up to 48 MHz.
  - Fully featured: 256 bytes of register RAM, two USARTs, three timers, two data pointers.
  - Fast: four clocks (83.3 nanoseconds at 48 MHz) per instruction cycle.
  - SFR access to control registers (including I/O ports) that require high speed.
  - USB-vectored interrupts for low ISR latency.
  - Used for USB housekeeping and control, not to move high speed data.
- “Soft” operation—USB firmware can be downloaded over USB, eliminating the need for hard-coded memory.
- Four interface FIFOs that can be internally or externally clocked. The endpoint and interface FIFOs are unified to eliminate data transfer time between USB and external logic.
- General Programmable Interface (GPIF), a microcoded state machine which serves as a timing master for ‘glueless’ interface to the FX2 FIFOs.

FX2 is a single-chip USB 2.0 peripheral solution. Unlike designs that use an external PHY, the FX2 integrates everything on one chip, eliminating costly high pin-count packages and the need to route high-speed signals between chips.

1.14 FX2 Integrated Microprocessor

The FX2’s CPU uses on-chip RAM as program and data memory. *Chapter 5, "Memory",* describes the various internal/external memory options.

The CPU communicates with the SIE using a set of registers occupying on-chip RAM addresses 0xE600-0xE6FF. These registers are grouped and described by function in individual chapters of this reference manual and summarized in register order in *Chapter 15, "Registers"*.

The CPU has two duties. First, it participates in the protocol defined in the *Universal Serial Bus Specification Version 2.0, “Chapter 9, USB Device Framework.”* Thanks to the FX2’s “smart” SIE,

the firmware associated with the USB protocol is simplified, leaving code space and bandwidth available for the CPU's primary duty—to help implement your device. On the device side, abundant input/output resources are available, including I/O ports, USARTs, and an I²C-compatible bus master controller. These resources are described in *Chapter 13, "Input/Output",* and *Chapter 14, "Timers/Counters and Serial Interface"*.

It's important to recognize that the FX2 architecture is such that the CPU sets up and controls data transfers, but it normally does not ***participate*** in high bandwidth transfers. It is not in the data path; instead, the large data FIFOs that handle endpoint data connect directly to outside interfaces. To make the interface versatile, a programmable timing generator (GPIF, General Programmable Interface) can create user-programmed waveforms for high bandwidth transfers between the internal FIFOs and external logic.

FX2 adds eight interrupt sources to the standard 8051 interrupt system:

- INT2: USB Interrupt
- INT3: I²C-Compatible Bus Interrupt
- INT4: FIFO/GPIF Interrupt
- INT4: External Interrupt 4
- INT5: External Interrupt 5
- INT6: External Interrupt 6
- USART1: USART1 Interrupt
- WAKEUP: USB Resume Interrupt

The FX2 provides 27 individual USB-interrupt sources which share the INT2 interrupt, and 14 individual FIFO/GPIF-interrupt sources which share the INT4 interrupt. To save the code and processing time which normally would be required to identify an individual interrupt source, the FX2 provides a second level of interrupt vectoring called *Autovectoring*. Each INT2 and INT4 interrupt source has its own autovector, so when an interrupt requires service, the proper ISR (interrupt service routine) is automatically invoked. *Chapter 4, "Interrupts"* describes the FX2 interrupt system.

1.15 FX2 Block Diagram

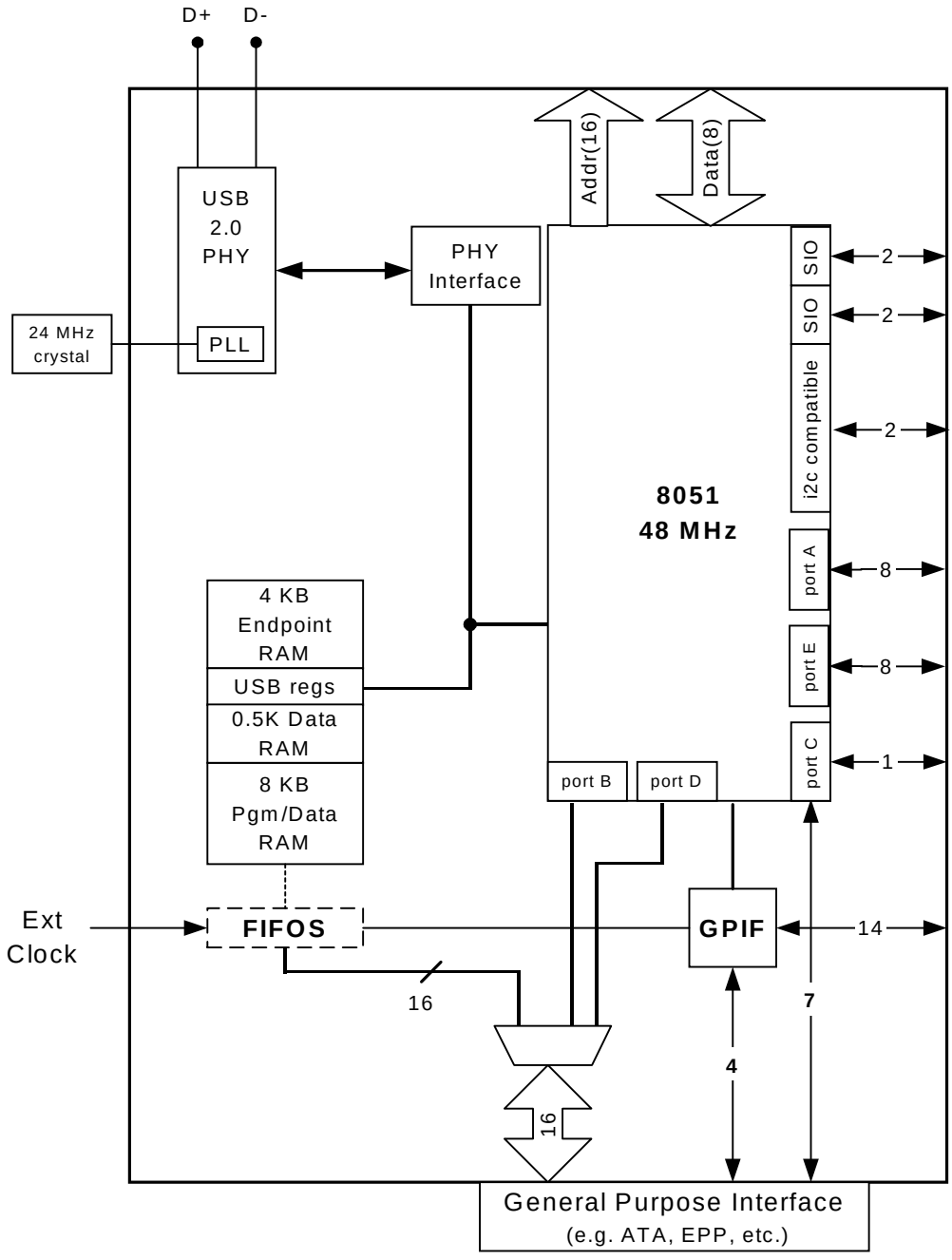


Figure 1-9. FX2 Block Diagram

1.16 Packages

FX2 is available in three packages:

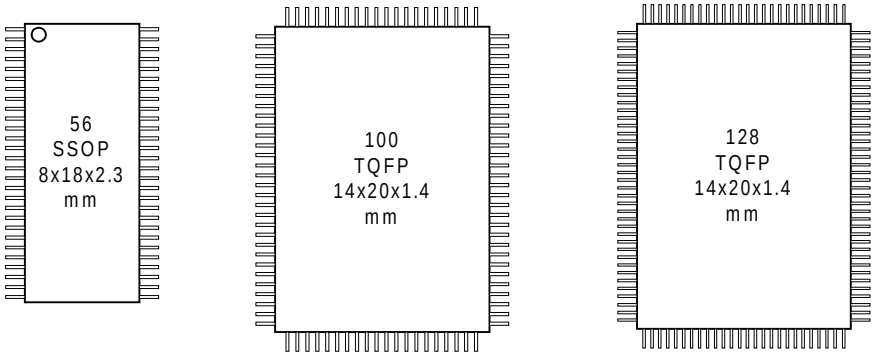


Figure 1-10. 56-pin, 100-pin, and 128-pin FX2 Packages

1.16.1 56-Pin Package

Twenty-four general-purpose I/O pins (ports A, B, and D) are available. Sixteen of these I/O pins can be configured as the 16-bit data interface to the FX2's internal high-speed 16-bit FIFOs, which can be used to implement low cost, high-performance interfaces such as ATAPI, UTOPIA, EPP, etc. The 56-pin package has the following:

- Three 8-bit I/O ports: PORTA, PORTB, and PORTD
- I²C-compatible bus
- An 8- or 16-bit General Programmable Interface (GPIF) multiplexed onto PORTB and PORTD, with five non-multiplexed control signals
- Four 8- or 16-bit Slave FIFOs, with five non-multiplexed control signals and four or five control signals multiplexed with PORTA

1.16.2 100-Pin Package

The 100-pin package adds functionality to the 56-pin package:

- Two additional 8-bit I/O ports: PORTC and PORTE
- Seven additional GPIF Control (CTL) and Ready (RDY) signals
- Nine non-multiplexed peripheral signals (two USARTs, three timer inputs, INT4, and $\overline{INT5}$)
- Eight additional control signals multiplexed onto PORTE
- Nine GPIF address lines, multiplexed onto PORTC (eight) and PORTE (one)
- $\overline{RD}$ and $\overline{WR}$ signals which may be used as read and write strobes for PORTC

1.16.3 128-Pin Package

The 128-pin package adds the 8051 address and data buses and control signals. The $\overline{RD}$, $\overline{PSEN}$, and $\overline{WR}$ strobes are standard 8051 control strobes, serving as read/write strobes for external memory attached to the 8051 address and data buses. The FX2 encodes the $\overline{CS}$ and $\overline{OE}$ signals to automatically exclude external access to memory spaces which exist on-chip, and optionally to combine off-chip data- and code-memory read accesses. The 128-pin package adds the following:

- 16-bit 8051 address bus
- 8-bit 8051 data bus
- Address/data bus control signals

1.16.4 Signals Available in the Three Packages

Three interface modes are available: Ports, GPIF Master, and Slave FIFO.

Figure 1-11 shows a logical diagram of the signals available in the three packages. The signals on the left edge of the diagram are common to all interface modes, while the signals on the right are specific to each mode. The interface mode is software-selectable via an internal mode register.

In “Ports” mode, all the I/O pins are general-purpose I/O ports.

“GPIF master” mode uses the PORTB and PORTD pins as a 16-bit data interface to the four FX2 endpoint FIFOs EP2, EP4, EP6 and EP8. In this “master” mode, the FX2 FIFOs are controlled by the internal GPIF, a programmable waveform generator that responds to FIFO status flags, drives timing signals using its CTL outputs, and waits for external conditions to be true on its RDY inputs. Note that only a subset of the GPIF signals (CTL0-2, RDY0-1) is available in the 56-pin package, while the full set (CTL0-5, RDY0-5) is available in the 100- and 128-pin packages.

In the “Slave FIFO” mode, external logic or an external processor interfaces directly to the FX2 endpoint FIFOs. In this mode, the GPIF is not active, since external logic has direct FIFO control. Therefore, the basic FIFO signals (flags, selectors, strobes) are brought out on FX2 pins. The external master can be asynchronous or synchronous, and it may supply its own independent clock to the FX2 interface.

The 100-pin package includes all the functionality of the 56-pin package, and brings out the two additional I/O ports PORTC and PORTE as well as all the USART, Timer, Interrupt, and GPIF signals. The RD and WR pins function as PORTC strobes in the 100-pin package, and as expansion memory strobes in the 128-pin package.

The 128-pin package adds 28 pins to the 100-pin package to bring out the full 8051 expansion memory bus. This allows for the connection of external memory for applications that run at power-on and before connection to USB. The 128-pin package also provides the foundation for the Cypress FX2 Development Kit boards, in which code is developed using a debug monitor that runs in external RAM.

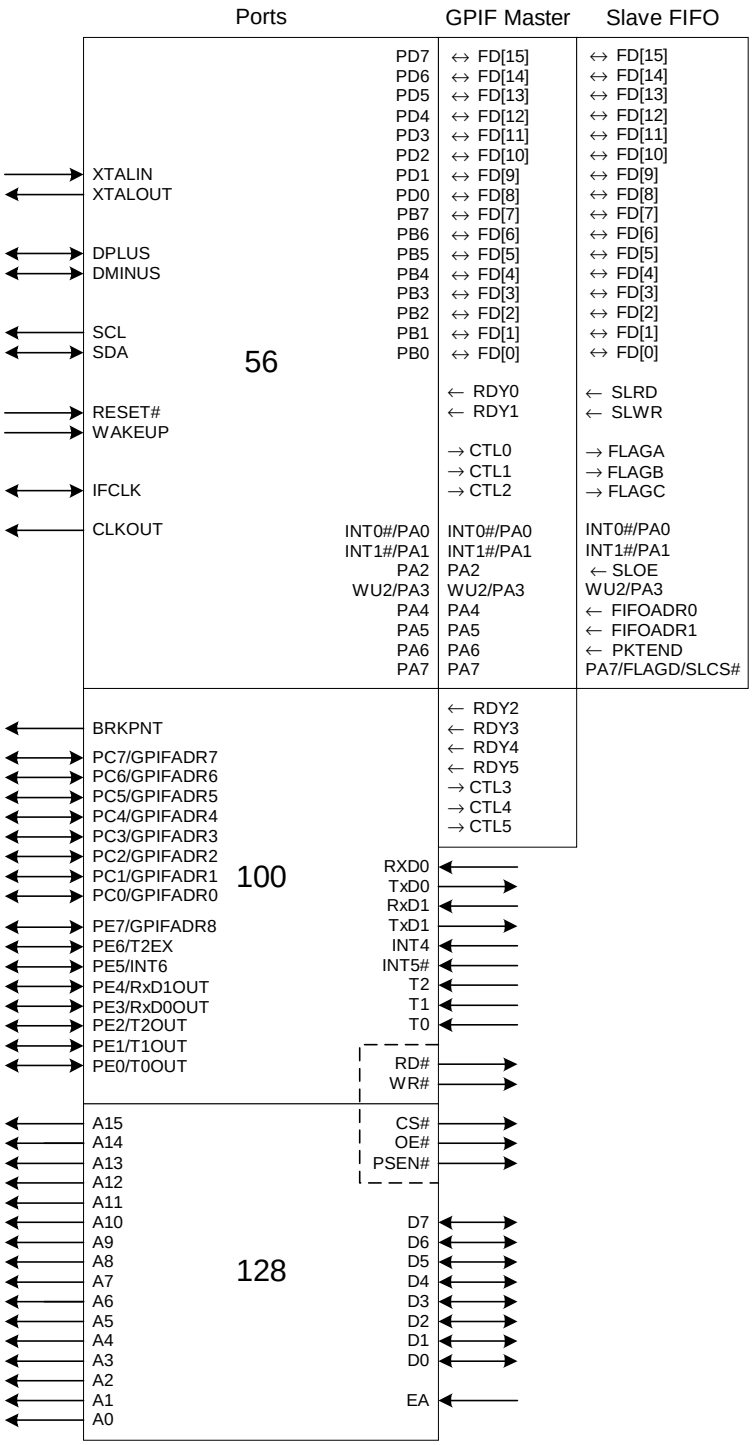


Figure 1-11. Signals for the Three FX2 Package Types

1.17 Package Diagrams

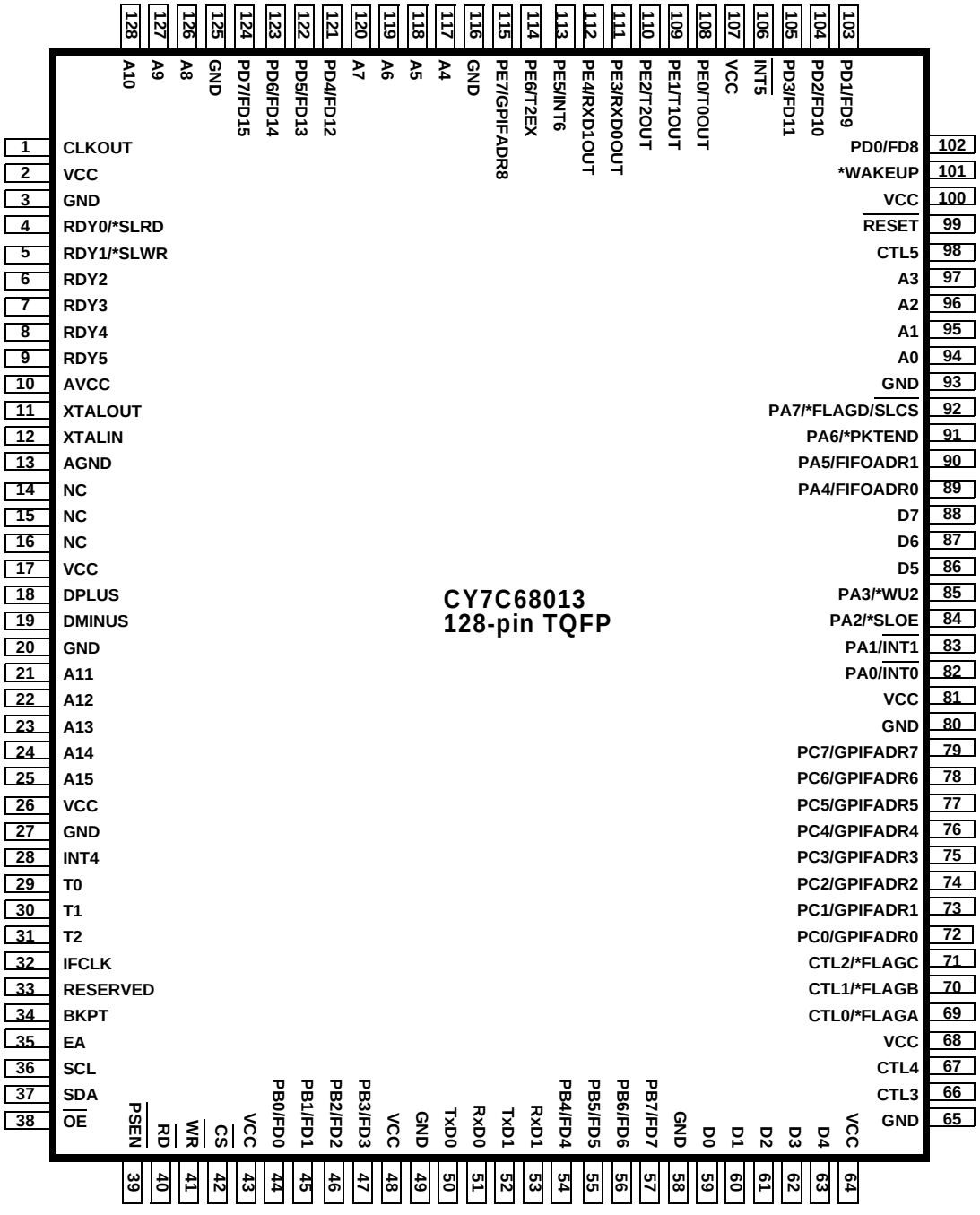


Figure 1-12. CY7C68013-128 TQFP Pin Assignment

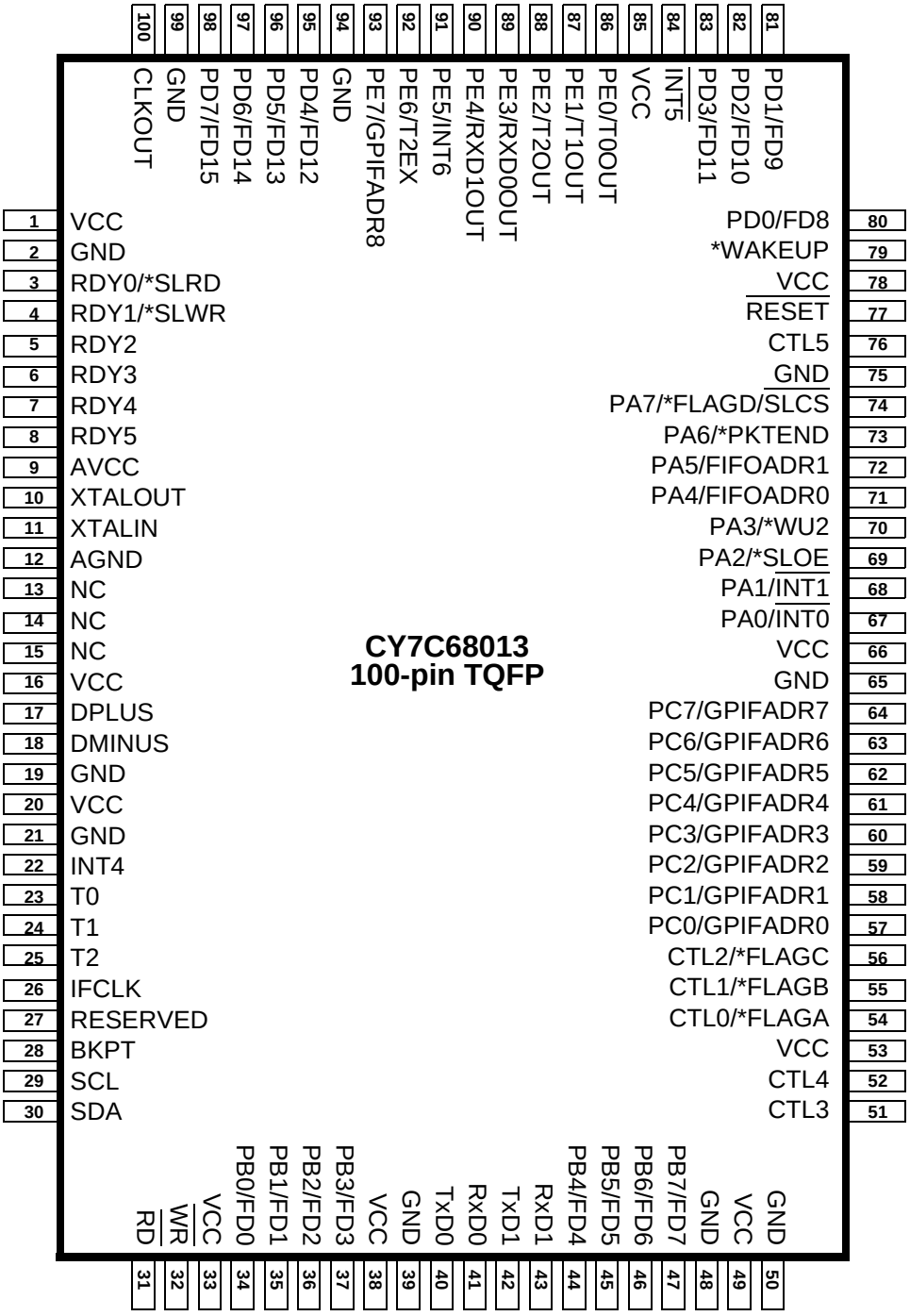


Figure 1-13. CY7C68013-100 TQFP Pin Assignment

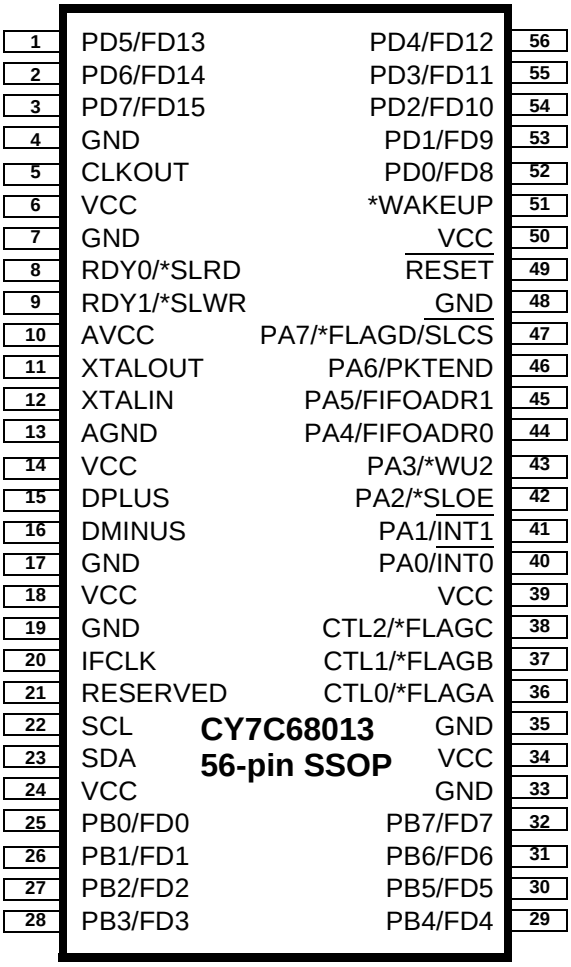


Figure 1-14. CY7C68013-56 SSOP Pin Assignment

1.18 FX2 Endpoint Buffers

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 or fills with USB data bytes. The host selects a device endpoint by sending a 4-bit address and a direction bit. Therefore, USB can uniquely address 32 endpoints, IN0 through IN15 and OUT0 through OUT15.

From the FX2's point of view, an endpoint is a buffer full of bytes received or held for transmission over the bus. The FX2 reads host data from an OUT endpoint buffer, and writes data for transmission to the host to an IN endpoint buffer.

FX2 contains three 64-byte endpoint buffers, plus 4 Kilobytes of buffer space that can be configured various ways, as indicated by Figure 1-15. The three 64-byte buffers are common to all configurations.

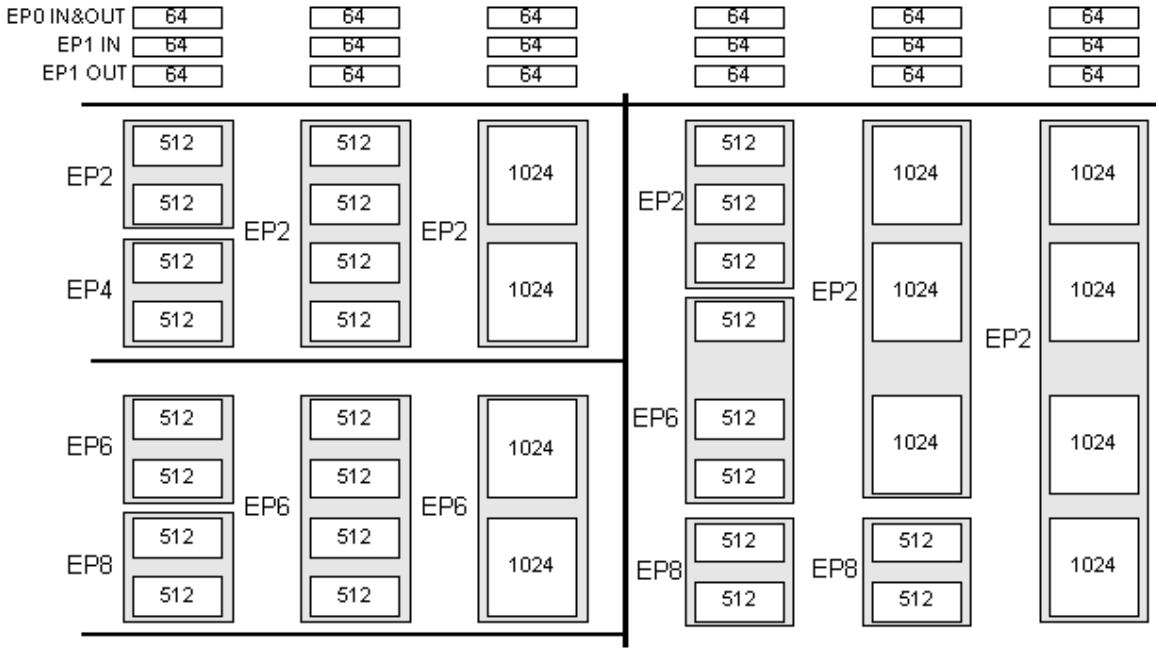


Figure 1-15. FX2 Endpoint Buffers

The three 64-byte buffers are designated EP0, EP1IN and EP1OUT. EP0 is the default CONTROL endpoint, a bidirectional endpoint that uses a single 64-byte buffer for both IN and OUT data. FX2 firmware reads or fills the EP0 buffer when the (optional) data stage of a CONTROL transfer is required.

The eight SETUP bytes in a CONTROL transfer do not appear in the 64-byte EP0 endpoint buffer. Instead, to simplify programming, the FX2 automatically stores the eight SETUP bytes in a separate buffer (SETUPDAT, at 0xE6B8-0xE6BF).

EP1IN and EP1OUT use separate 64 byte buffers. FX2 firmware can configure these endpoints as BULK, INTERRUPT or ISOCHRONOUS. These endpoints, as well as EP0, are accessible only by FX2 firmware. This is in contrast to the large endpoint buffers EP2, EP4, EP6 and EP8, which are designed to move high bandwidth data directly on and off chip without firmware intervention.

Endpoints 2, 4, 6 and 8 are the large, high bandwidth, data moving endpoints. They can be configured various ways to suit bandwidth requirements. The shaded boxes in Figure 1-15 enclose the buffers to indicate double, triple, or quad buffering. Double buffering means that one packet of data can be filling or emptying with USB data while another packet (from the same endpoint) is being serviced by external interface logic. Triple buffering adds a third packet buffer to the pool, which can be used by either side (USB or interface) as needed. Quad buffering adds a fourth packet buffer. Multiple buffering can significantly improve USB bandwidth performance when the data supplying and consuming rates are similar, but bursty; it smooths out the bursts, reducing or eliminating the need for one side to wait for the other.

Endpoints 2, 4, 6 and 8 can be configured using the choices shown in Tab le1-2.

Table 1-2. Endpoint 2, 4, 6, and 8 Configuration Choices

Characteristic	Choices
Direction	IN, OUT
Type	Bulk, Interrupt, Isochronous
Buffering	Double, Triple, Quad

When the FX2 operates at full speed (12 Mbits/sec), some or all of the endpoint buffer bytes shown in Figure 1-15 may be employed, depending on endpoint type. *Regardless of the physical buffer size, the endpoint buffer accommodates only one full-speed packet*.

For example, if EP2 is used as a full-speed BULK endpoint, the maximum number of bytes (maxPacketSize) it can accommodate is 64, even though the physical buffer size is 512 or 1024 bytes (it makes sense, therefore, to configure full-speed BULK endpoints as 512 bytes rather than 1024, so that fewer unused bytes are wasted). An ISOCHRONOUS full speed endpoint, on the other hand, could fully use either a 512- or 1024-byte buffer.

1.19 External FIFO Interface

The large data FIFOs (endpoints 2, 4, 6 and 8) in the FX2 are designed to move high speed (480 Mbits/sec) USB data on and off chip without introducing any bandwidth bottlenecks. They accomplish this goal by implementing the following features:

1. Direct interface with outside logic, with the FX2's CPU out of the data path.
2. "Quantum FIFO" architecture instantaneously moves ("commits") packets between the USB and the FIFOs.
3. Versatile interfaces: Slave FIFO (external master) or GPIF (internal master), synchronous or asynchronous clocking, internal or external clocks, etc.

The firmware sets switches to configure the outside FIFO interface, and then generally does not participate in moving the data into and out of the FIFOs.

To understand the "Quantum FIFO", it is necessary to refer to two data domains, the *USB domain* and the *Interface domain*. Each domain is independent, allowing different clocks and logic to handle its data.

The USB domain is serviced by the SIE, which receives and delivers FIFO data packets over the two-wire USB bus. The USB domain is clocked using a reference derived from the 24 MHz crystal attached to the FX2 chip.

The Interface domain loads and unloads the endpoint FIFOs. An external device such as a DSP or ASIC can supply its own clock to the FIFO interface, or the FX2's internal interface clock (IFCLK) can be supplied to the interface.

The classic solution to the problem of reconciling two different and independent clocks is to use a FIFO. The FX2's FIFOs have an unusual property: They're *Quantum* FIFOs, which means that data is committed to the FIFOs in USB-size packets, rather than one byte at a time. This is invisible to the outside interface, since it services the FIFOs just like any ordinary FIFO (i.e., by checking full and empty flags). The only minor difference is that when an empty flag goes from 1 (empty) to 0 (not empty), the number of bytes in the FIFO jumps to a USB packet size, rather than just one byte.

FX2 Quantum FIFOs may be moved between data domains almost instantaneously. The Quantum nature of the FIFOs also simplifies error recovery. If endpoint data were continuously clocked into an interface FIFO, some of the packet data might have already been clocked out by the time an error is detected at the end of a USB packet. By switching FIFO data between the domains in USB-packet-size blocks, each USB packet can be error-checked (and retried, if necessary) before it's committed to the other domain.

Figures 1-16 and 1-17 illustrate the two methods by which external logic interfaces to the endpoint FIFOs EP2, EP4, EP6 and EP8.

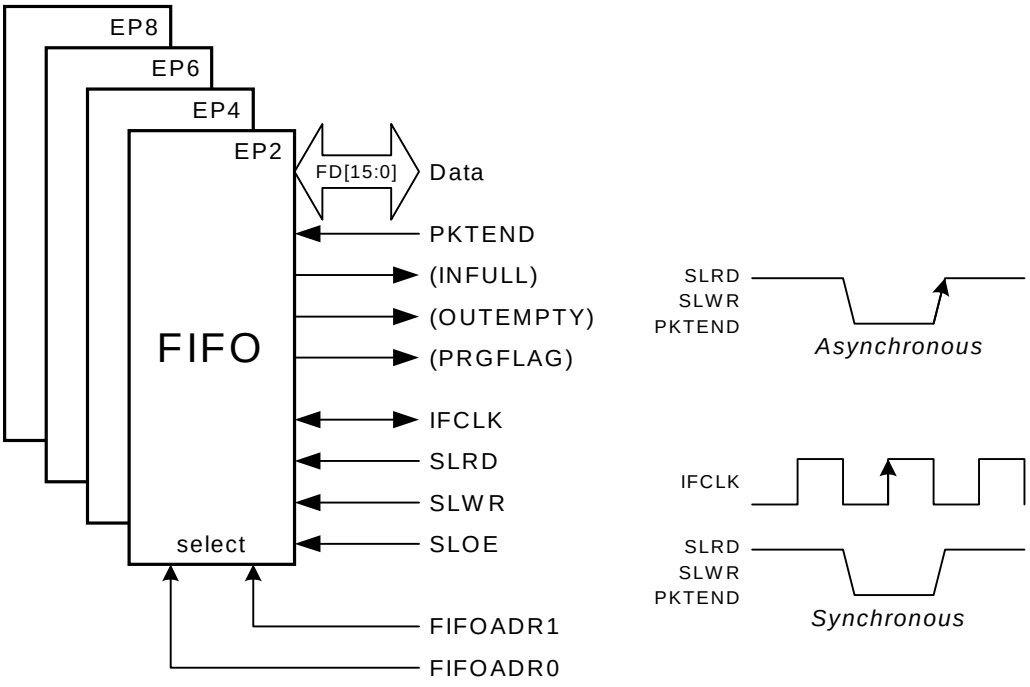


Figure 1-16. FX2 FIFOs in "Slave FIFO" Mode

Figure 1-16 illustrates the outside-world view of the FX2 data FIFOs configured as "Slave FIFOs". The outside logic supplies a clock, responds to the FIFO flags, and clocks FIFO data in and out using the strobe signals. Optionally, the outside logic may use the internal FX2 Interface Clock (IFCLK) as its reference clock.

Three FIFO flags are shown in parentheses in Figure1-16 because they actually are called FLAGA-FLAGD in the pin diagram (there are four flag pins). Using configuration bits, various FIFO flags can be assigned to these general-purpose flag pins. The names shown in parentheses illustrate typical uses for these configurable flags. The Programmable Level Flag (PRGFLAG) can be set to any value to indicate degrees of FIFO "fullness". The outside interface selects one of the four FIFOs using the FIFOADR pins, and then clocks the 16-bit FIFO data using the SLRD (Slave Read) and SLWR (Slave Write) signals. PKTEND is used to dispatch a non-full IN packet to USB.

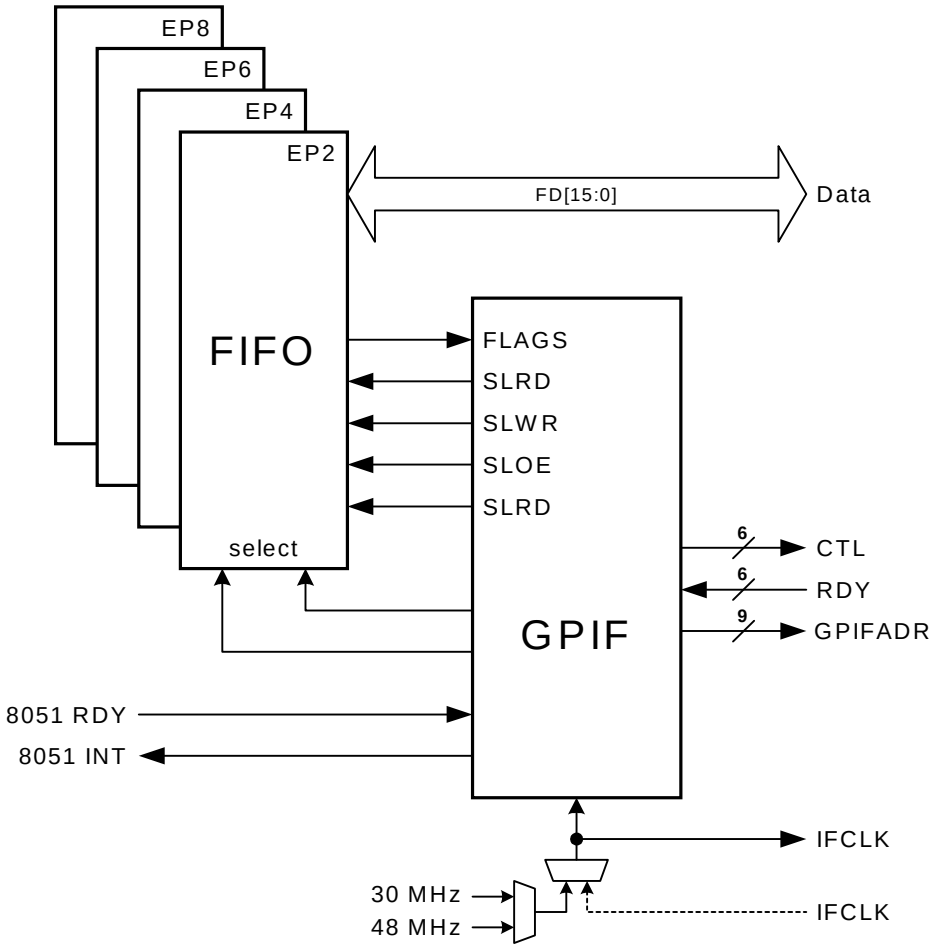


Figure 1-17. FX2 FIFOs in "GPIF Master" Mode

External systems that connect to the FX2 FIFOs must provide control circuitry to select FIFOs, check flags, clock data, etc. FX2 contains a sophisticated control unit (the General Programmable Interface, or GPIF) which can replace this external logic. In the "GPIF Master" FIFO mode, (Figure 1-17), the GPIF reads the FIFO flags, controls the FIFO strobes, and presents a user-customizable interface to the outside world. The GPIF runs at a very high speed (up to 48 MHz clock rate) so that it can develop high-resolution control waveforms. It can be clocked from one of two internal sources (30 or 48 MHz) or from an external clock.

Control (CTL) signals are programmable waveform outputs, and ready (RDY) signals are input pins that can be tested for conditions that cause the GPIF to pause and resume operation, imple-

menting “wait states”. GPIFADR pins present a 9-bit address to the interface that may be incremented as data is transferred. The 8051 INT signal is a ‘hook’ that can signal the FX2’s CPU in the middle of a transaction; GPIF operation resumes once the CPU asserts its own 8051 RDY signal. This ‘hook’ permits great flexibility in the generation of GPIF waveforms.

1.20 EZ-USB FX2 Product Family

The EZ-USB FX2 family is available in various pinouts to serve different system requirements and costs.

Table 1-3. EZ-USB FX2 Family

Part Number	Package	Ram	ISO Support	I/O	Bus Width	Data/Address Bus
CY7C68013-56PVC	56-pin SSOP	8 KBytes	Yes	24	8/16 Bits	No
CY7C68013-100AC	100-pin TQFP	8 KBytes	Yes	40	8/16 Bits	No
CY7C68013-128AC	128-pin TQFP	8 KBytes	Yes	40	8/16 Bits	8051 Address/Data Bus

Chapter 2 Endpoint Zero

2.1 Introduction

Endpoint zero has special significance in a USB system. It is a CONTROL endpoint, and it is required by every USB device. The USB host uses special SETUP tokens to signal transfers that deal with device control; only CONTROL endpoints accept these special tokens.

The USB host sends a suite of standard device requests over endpoint zero. These standard requests are fully defined in Chapter 9 of the *USB Specification*. This chapter describes how the FX2 chip handles endpoint zero requests.

The FX2 provides extensive hardware support for handling endpoint-zero operations; this chapter describes those operations and the FX2 resources that simplify the firmware which handles them.

Endpoint zero is the only CONTROL endpoint supported by the FX2. CONTROL endpoints are *bi-directional*, so the FX2 provides a single 64-byte buffer, EP0BUF, which firmware handles exactly like a bulk endpoint buffer for the data stages of a CONTROL transfer. A second 8-byte buffer called SETUPDAT, which is unique to endpoint zero, holds data that arrives in the SETUP stage of a CONTROL transfer. This relieves the FX2 firmware of the burden of tracking the three CONTROL transfer phases (SETUP, DATA, and STATUS). The FX2 also generates separate interrupt requests for the various transfer phases, further simplifying code.

Endpoint zero is always enabled and accessible by the USB host.

2.2 Control Endpoint EP0

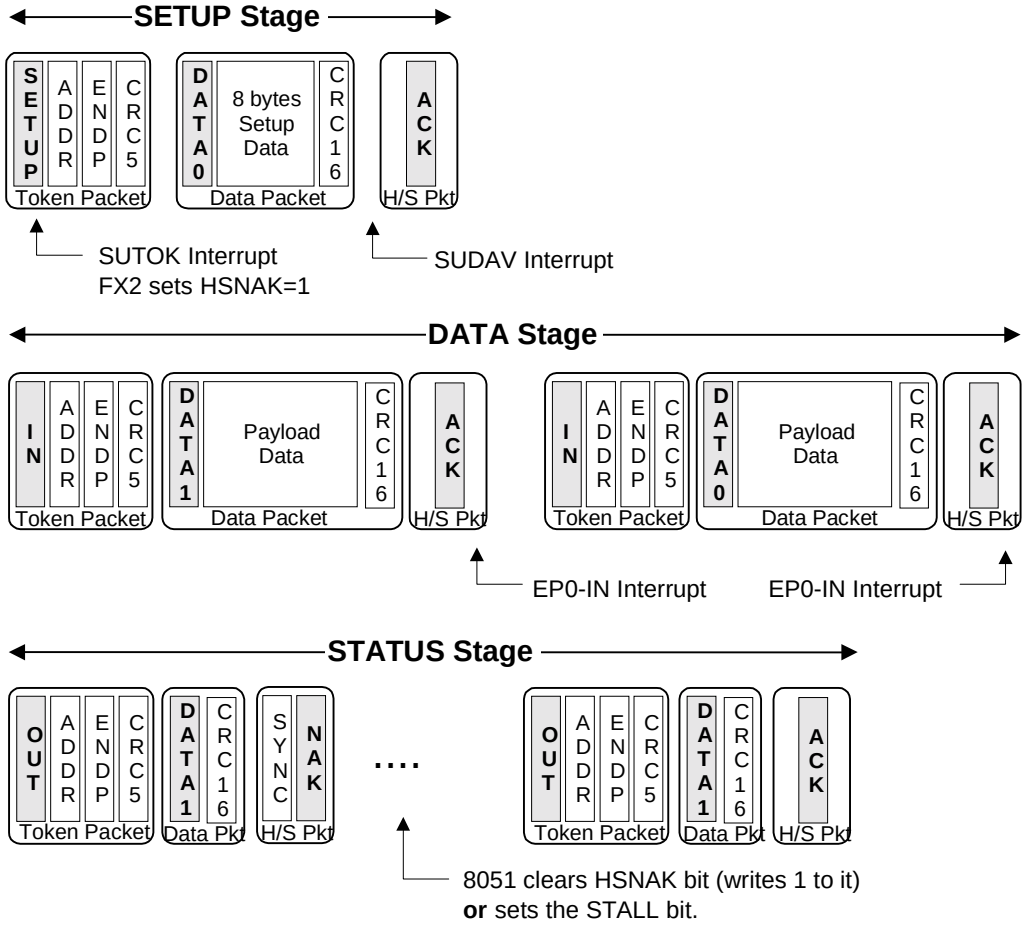


Figure 2-1. A USB Control Transfer (With Data Stage)

Endpoint zero accepts a special SETUP packet, which contains an 8-byte data structure that provides host information about the CONTROL transaction. CONTROL transfers include a final STATUS phase, constructed from standard PIDs (IN/OUT, DATA1, and ACK/NAK).

Some CONTROL transactions include all required data in their 8-byte SETUP Data packet. Other CONTROL transactions require more OUT data than will fit into the eight bytes, or require IN data from the device. These transactions use standard bulk-like transfers to move the data. Note in Figure 2-1 that the DATA Stage looks exactly like a bulk transfer. As with BULK endpoints, the endpoint zero byte count registers must be loaded to ACK each data transfer stage of a CONTROL transfer.

The STATUS stage consists of an empty data packet with the opposite direction of the data stage, or an IN if there was no data stage. This empty data packet gives the device a chance to ACK or NAK the entire CONTROL transfer.

The HSNAK bit holds off the completion of a CONTROL transfer until the device has had time to respond to a request. For example, if the host issues a Set_Interface Request, the FX2 firmware performs various housekeeping chores such as adjusting internal modes and re-initializing endpoints. During this time, the host issues handshake (STATUS stage) packets to which the FX2 automatically responds with NAKs, indicating “busy.” When the firmware completes its housekeeping operations, it clears the HSNAK bit (*by writing 1 to it*), which instructs the FX2 to ACK the STATUS stage, terminating the transfer. This handshake prevents the host from attempting to use an interface before it’s fully configured.

To perform an endpoint stall for the DATA or STATUS stage of an endpoint zero transfer (the SETUP stage can never stall), firmware must set both the STALL and HSNAK bits for endpoint zero.

Some CONTROL transfers do not have a DATA stage. Therefore, the code that processes the SETUP data should check the length field in the SETUP data (in the 8-byte buffer at SETUPDAT) and arm endpoint zero for the DATA phase (by loading EP0BCH:L) only if the length field is non-zero.

Two interrupts provide notification that a SETUP packet has arrived, as shown in Figur e2-2.

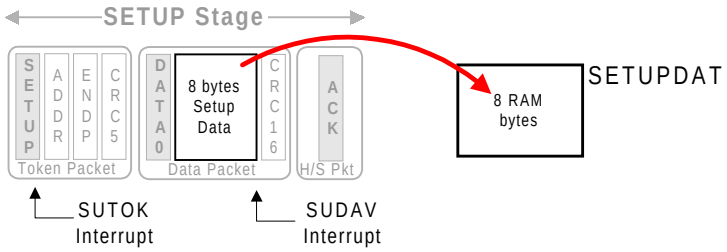


Figure 2-2. Two Interrupts Associated with EP0 CONTROL Transfers

The FX2 asserts the SUTOK (Setup Token) interrupt request when it detects the SETUP token at the beginning of a CONTROL transfer. This interrupt is normally used for debug only.

The FX2 asserts the SUDAV (Setup Data Available) interrupt request when the eight bytes of SETUP data have been received error-free and transferred to the SETUPDAT buffer. The FX2 automatically takes care of any retries if it finds errors in the SETUP data. These two interrupt request bits must be cleared by firmware.

Firmware responds to the SUDAV interrupt request by either directly inspecting the eight bytes at SETUPDAT or by transferring them to a local buffer for further processing. Servicing the SETUP data should be a high priority, since the USB Specification stipulates that CONTROL transfers

must always be accepted and never NAK'd. It is possible, therefore, that a CONTROL transfer could arrive while the firmware is still servicing a previous one. In this case, the earlier CONTROL transfer service should be aborted and the new one serviced. The SUTOK interrupt gives advance warning that a new CONTROL transfer is about to overwrite the eight SETUPDAT bytes.

If the firmware stalls endpoint zero (by setting the STALL and HSNAK bits to 1), the FX2 automatically clears the stall bit when the next SETUP token arrives.

Like all FX2 interrupt requests, the SUTOK and SUDAV bits can be directly tested and cleared by the firmware (*cleared by writing 1*) even if their corresponding interrupts are disabled.

Figure 2-3 shows the FX2 registers that are associated with CONTROL transactions over EP0.

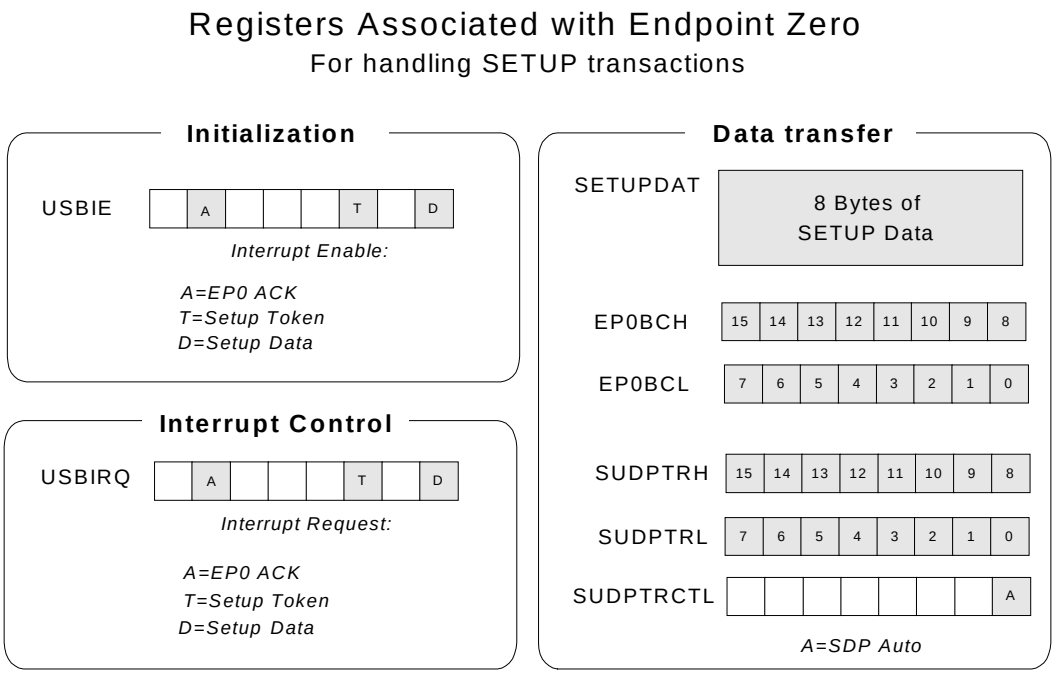


Figure 2-3. Registers Associated with EP0 Control Transfers

These registers augment those associated with normal bulk transfers over endpoint zero, which are described in *Chapter 8, "Access to Endpoint Buffers"*.

Two bits in the USBIE (USB Interrupt Enable) register enable the SETUPToken (SUTOK) and SETUP Data Available interrupts. The actual interrupt-request bits are in the USBIRQ (USB Interrupt Requests) register.

The FX2 transfers the eight SETUP bytes into eight bytes of RAM at SETUPDAT. A 16-bit pointer, SUDPTRH:L, provides hardware assistance for handling CONTROL IN transfers, in particular the *Get Descriptor* requests described later in this chapter.

2.3 USB Requests

The *Universal Serial Bus Specification Version 2.0, Chapter 9, "USB Device Framework"* defines a set of *Standard Device Requests*. When the firmware is in control of endpoint zero (RENUM=1), the FX2 handles only one of these requests (*Set Address)* automatically; it relies on the firmware to support all of the others. The firmware acts on device requests by decoding the eight bytes contained in the SETUP packet and available at SETUPDAT. Table2-1 defines these eight bytes.

Table 2-1. The Eight Bytes in a USB SETUP Packet

Byte	Field	Meaning
0	bmRequestType	Request Type, Direction, and Recipient.
1	bRequest	The actual request (see Tabl e2-2).
2	wValueL	16-bit value, varies according to bRequest.
3	wValueH	
4	wIndexL	16-bit field, varies according to bRequest.
5	wIndexH	
6	wLengthL	Number of bytes to transfer if there is a data phase.
7	wLengthH	

The **Byte** column in the previous table shows the byte offset from SETUPDAT. The **Field** column shows the different bytes in the request, where the “bm” prefix means bit-map, “b” means byte [8 bits, 0-255], and “w” means word [16 bits, 0-65535].

Table 2-2 shows the different values defined for bRequest, and how the firmware should respond to each request. The remainder of this chapter describes each of the requests inTabl e2-2 in detail.

Table 2-2 applies when RENUM=1, signifying that the firmware, rather than the FX2 hardware, handles device requests

.

Table 2-2. How the Firmware Handles USB Device Requests (RENUM=1)

bRequest	Name	FX2 Action	Firmware Response
0x00	Get Status	SUDAV Interrupt	Supply RemWU, SelfPwr or Stall Bits
0x01	Clear Feature	SUDAV Interrupt	Clear RemWU, SelfPwr or Stall Bits
0x02	(reserved)	none	Stall EP0
0x03	Set Feature	SUDAV Interrupt	Set RemWU, SelfPwr or Stall Bits
0x04	(reserved)	none	Stall EP0
0x05	Set Address	Update FNADDR Register	none
0x06	Get Descriptor	SUDAV Interrupt	Supply table data over EP0-IN
0x07	Set Descriptor	SUDAV Interrupt	Application dependent
0x08	Get Configuration	SUDAV Interrupt	Send current configuration number
0x09	Set Configuration	SUDAV Interrupt	Change current configuration
0x0A	Get Interface	SUDAV Interrupt	Supply alternate setting No. from RAM
0x0B	Set Interface	SUDAV Interrupt	Change alternate setting No.
0x0C	Sync Frame	SUDAV Interrupt	Supply a frame number over EP0-IN
Vendor Requests			
0xA0 (Firmware Load)		Upload / Download RAM	---
0xA1 - 0xAF		SUDAV Interrupt	Reserved by Cypress Semiconductor
All except 0xA0		SUDAV Interrupt	Depends on application

In the ReNumerated condition (RENUM=1), the FX2 passes all USB requests except *Set Address* to the firmware via the SUDAV interrupt.

The FX2 implements one vendor-specific request: “Firmware Load,” 0xA0 (the bRequest value of 0xA0 is valid only if byte 0 of the request, bmRequestType, is also “x10xxxxx,” indicating a vendor-specific request.) The load request is valid at all times, so the load feature may be used even after ReNumeration. If your application implements vendor-specific USB requests, and you do *not* wish to use the Firmware Load feature, be sure to refrain from using the bRequest value 0xA0 for your custom requests. The Firmware Load feature is fully described in *Chapter 3, "Enumeration and ReNumeration™"*.

To avoid future incompatibilities, vendor requests 0xA0-0xAF are reserved by Cypress Semiconductor.

2.3.1 Get Status

The USB Specification defines three USB status requests. A fourth request, to an interface, is declared in the spec as “reserved.” The four status requests are:

- Remote Wakeup (Device request)
- Self-Powered (Device request)
- Stall (Endpoint request)
- Interface request (reserved)

The FX2 automatically asserts the SUDAV interrupt to tell the firmware to decode the SETUP packet and supply the appropriate status information.

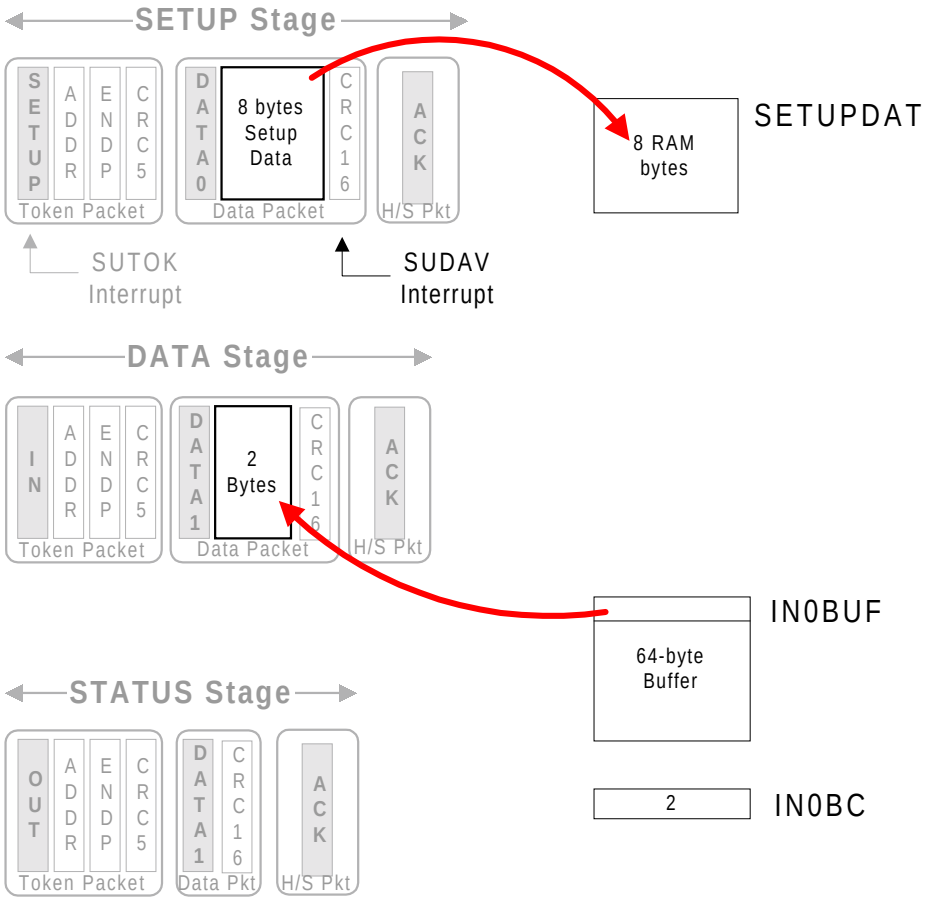


Figure 2-4. Data Flow for a Get_Status Request

As Figure 2-4 illustrates, the firmware responds to the SUDAV interrupt by decoding the eight bytes the FX2 has copied into RAM at SETUPDAT. The firmware answers a *Get Status* request (bRequest=0) by loading two bytes into the EP0BUF buffer and loading the byte count register EP0BCH:L with the value 0x0002. The FX2 then transmits these two bytes in response to an IN token. Finally, the firmware clears the HSNAK bit (*by writing 1 to it*), which instructs the FX2 to ACK the status stage of the transfer.

The following tables show the eight SETUP bytes for *Get Status* Requests.

Table 2-3. Get Status-Device (Remote Wakeup and Self-Powered Bits)

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x80**	IN, Device	
1	bRequest	**0x00**	"Get Status"	*Load two bytes into EP0BUF:*
2	wValueL	0x00		
3	wValueH	0x00		*Byte 0 : bit 0 = Self-Powered*
4	wIndexL	0x00		*: bit 1 = Remote Wakeup*
5	wIndexH	0x00		*Byte 1 : zero*
6	wLengthL	**0x02**	Two bytes requested	
7	wLengthH	**0x00**		

Get Status-Device queries the state of two bits, "Remote Wakeup" and "Self-Powered". The Remote Wakeup bit indicates whether or not the device is currently enabled to request remote wakeup (remote wakeup is explained in *Chapter 6, "Power Management"*). The Self-Powered bit indicates whether or not the device is self-powered (as opposed to USB bus-powered).

The firmware returns these two bits by loading two bytes into EP0BUF, then loading a byte count of 0x0002 into EP0BCH:L.

Table 2-4. Get Status-Endpoint (Stall Bits)

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x82**	IN, Endpoint	*Load two bytes into EP0BUF:*
1	bRequest	**0x00**	"Get Status"	*Byte 0 : bit 0 = Stall Bit for EP(n)*
2	wValueL	0x00		*Byte 1 : zero*
3	wValueH	0x00		
4	wIndexL	**EP**	0x00-0x08: OUT0-OUT8	
5	wIndexH	0x00	0x80-0x88: IN0-IN8	
6	wLengthL	**0x02**	Two bytes requested	
7	wLengthH	**0x00**		

Each endpoint has a STALL bit in its EPxCS register. If this bit is set, any request to the endpoint returns a STALL handshake rather than ACK or NAK. The *Get Status-Endpoint* request returns the STALL state for the endpoint indicated in byte 4 of the request. Note that bit 7 of the endpoint number EP (byte 4) specifies direction (0 = OUT, 1 = IN).

Endpoint zero is a CONTROL endpoint, which by USB definition is *bi-directional*. Therefore, it has only one stall bit.

About STALL

The USB STALL handshake indicates that something unexpected has happened. For instance, if the host requests an invalid alternate setting or attempts to send data to a non-existent endpoint, the device responds with a STALL handshake over endpoint zero instead of ACK or NAK.

Stalls are defined for all endpoint types except ISOCHRONOUS, which does not employ handshakes. Every FX2 bulk endpoint has its own stall bit. The firmware sets the stall condition for an endpoint by setting the STALL bit in the endpoint's EPxCS register. The host tells the firmware to set or clear the stall condition for an endpoint using the *Set Feature/Stall* and *Clear Feature/Stall* Requests.

The device might decide to set the stall condition on its own, too. In a routine that handles endpoint zero device requests, for example, when an undefined or non-supported request is decoded, the firmware should stall EP0.

Once the firmware stalls an endpoint, it should not remove the stall until the host issues a *Clear Feature/Stall* Request. An exception to this rule is endpoint 0, which reports a stall condition only for the current transaction and then automatically clears the stall condition. This prevents endpoint 0, the default CONTROL endpoint, from locking out device requests.

Table 2-5. Get Status-Interface

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x81**	IN, Endpoint	*Load two bytes into EP0BUF:*
1	bRequest	**0x00**	"Get Status"	*Byte 0 : zero*
2	wValueL	0x00		*Byte 1 : zero*
3	wValueH	0x00		
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	**0x02**	Two bytes requested	
7	wLengthH	**0x00**		

Get Status/Interface is easy: the firmware returns two zero bytes through EP0BUF and clears the HSNAK bit (*by writing 1 to it*). The requested bytes are shown as "Reserved (reset to zero)" in the USB Specification.

2.3.2 Set Feature

Set Feature is used to enable remote wakeup or stall an endpoint. No data stage is required.

Table 2-6. Set Feature-Device (Set Remote Wakeup Bit)

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x00**	OUT, Device	*Set the Remote Wakeup Bit*
1	bRequest	**0x03**	"Set Feature"	
2	wValueL	**0x01**	Feature Selector:	
3	wValueH	0x00	Remote Wakeup	
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	0x00		
7	wLengthH	0x00		

The only *Set Feature/Device* request presently defined in the USB Specification is to set the remote wakeup bit. This is the same bit reported back to the host as a result of a *Get Status-Device* request (Table 2-3). The host uses this bit to enable or disable remote wakeup by the device.

Table 2-7. Set Feature-Endpoint (Stall)

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x02**	OUT, Endpoint	*Set the STALL bit for the*
1	bRequest	**0x03**	"Set Feature"	*indicated endpoint:.*
2	wValueL	**0x00**	Feature Selector:	
3	wValueH	0x00	STALL	
4	wIndexL	**EP**	0x00-0x08: OUT0-OUT8	
5	wIndexH	0x00	0x80-0x88: IN0-IN8	
6	wLengthL	0x00		
7	wLengthH	0x00		

The only *Set Feature/Endpoint* request presently defined in the USB Specification is to stall an endpoint. The firmware should respond to this request by setting the STALL bit in the EPxCS register for the indicated endpoint EP (byte 4 of the request). The firmware can either stall an endpoint on its own or in response to the device request. Endpoint stalls are cleared by the host *Clear Feature/Stall* request.

The firmware should respond to the *Set Feature/Stall* request by performing the following tasks:

1. Set the STALL bit in the indicated endpoint's EPxCS register.
2. Reset the data toggle for the indicated endpoint.

3. Restore the stalled endpoint to its default condition, ready to send or accept data after the stall condition is removed by the host (via a *Clear Feature/Stall* request). For EP1 IN, for example, firmware should clear the BUSY bit in the EP1CS register; for EP1OUT, firmware should load any value into the EP1 byte-count register.
4. Clear the HSNAK bit in the EP0CS register (*by writing 1 to it*) to terminate the *Set Feature/Stall* CONTROL transfer.

Step 3 is also required whenever the host sends a *Set Interface* request.

Data Toggles

The FX2 automatically maintains the endpoint toggle bits to ensure data integrity for USB transfers. Firmware should directly manipulate these bits only for a very limited set of circumstances:

- *Set Feature/Stall*
- *Set Configuration*
- *Set Interface*

2.3.3 Clear Feature

Clear Feature is used to disable remote wakeup or to clear a stalled endpoint.

Table 2-8. Clear Feature-Device (Clear Remote Wakeup Bit)

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x00**	OUT, Device	*Clear the remote wakeup bit.*
1	bRequest	**0x01**	“Clear Feature”	
2	wValueL	**0x01**	Feature Selector:	
3	wValueH	0x00	Remote Wakeup	
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	0x00		
7	wLengthH	0x00		

Table 2-9. Clear Feature-Endpoint (Clear Stall)

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x02**	OUT, Endpoint	*Clear the STALL bit for the indicated endpoint.*
1	bRequest	**0x01**	"Clear Feature"	
2	wValueL	**0x00**	Feature Selector:	
3	wValueH	0x00	STALL	
4	wIndexL	**EP**	0x00-0x08: OUT0-OUT8	
5	wIndexH	0x00	0x80-0x88: IN0-IN8	
6	wLengthL	0x00		
7	wLengthH	0x00		

If the USB device supports remote wakeup (reported in its descriptor table when the device enumerates), the *Clear Feature/Remote Wakeup* request disables the wakeup capability.

The *Clear Feature/Stall* removes the stall condition from an endpoint. The firmware should respond by clearing the STALL bit in the indicated endpoint's EPxCS register.

2.3.4 Get Descriptor

During enumeration, the host queries a USB device to learn its capabilities and requirements using *Get Descriptor* requests. Using tables of *descriptors*, the device sends back (over EP0-IN) such information as what device driver to load, how many endpoints it has, its different configurations, alternate settings it may use, and informative text strings about the device.

The FX2 provides a special *Setup Data Pointer* to simplify firmware service for Get_Descriptor requests. The firmware loads this 16-bit pointer with the starting address of the requested descriptor, clears the HSNAK bit (*by writing 1 to it*), and the FX2 transfers the entire descriptor.

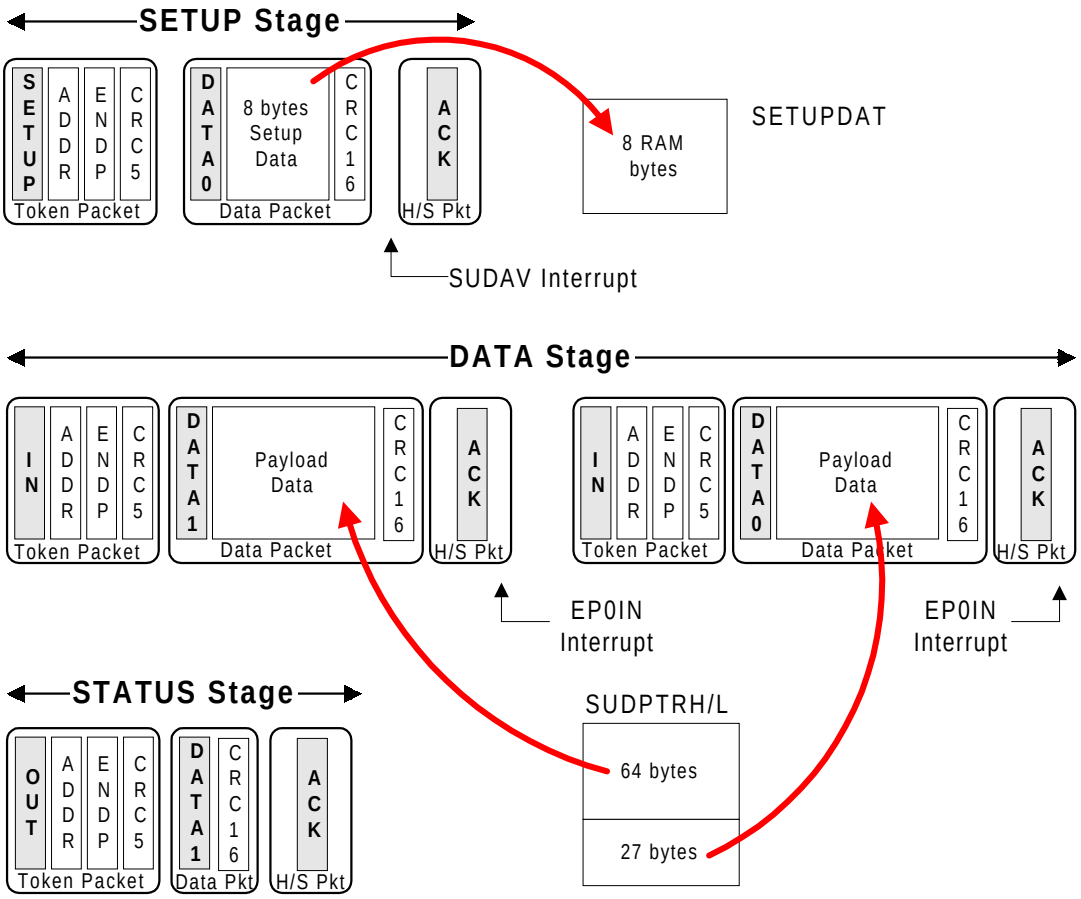


Figure 2-5. Using Setup Data Pointer (SUDPTR) for Get_Descriptor Requests

Figure 2-5 illustrates use of the Setup Data Pointer. This pointer is implemented as two registers, SUDPTRH and SUDPTRL. Most *Get Descriptor* requests involve transferring more data than fits into one packet. In the Figure 2-5 example, the descriptor data consists of 91 bytes.

The CONTROL transaction starts in the usual way, with theFX2 automatically transferring the eight bytes from the SETUP packet into RAM at SETUPDAT, then asserting the SUDAV interrupt request. The firmware decodes the *Get Descriptor* request, and responds by clearing the HSNAK bit (*by writing 1 to it*), and then loading the SUDPTRH:L registers with the address of the requested descriptor. Loading the SUDPTRL register causes the FX2 to automatically respond to two IN transfers with 64 bytes and 27 bytes of data using SUDPTR as a base address, and then to respond to the STATUS stage with an ACK.

The usual endpoint-zero interrupts SUDAV and EP0IN remain active during this automated transfer, so firmware will normally disables these interrupts because the transfer requires no firmware intervention.

Three types of descriptors are defined: Device, Configuration, and String.

2.3.4.1 Get Descriptor-Device

Table 2-10. Get Descriptor-Device

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x80**	IN, Device	*Set SUDPTR H:L to start of Device Descriptor table in RAM.*
1	bRequest	**0x06**	"Get Descriptor"	
2	wValueL	0x00		
3	wValueH	**0x01**	Descriptor Type: Device	
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	**LenL**		
7	wLengthH	**LenH**		

As illustrated in Figure 2-5, the firmware loads the 2-byte SUDPTR with the starting address of the Device Descriptor table. When SUDPTRL is loaded, the FX2 automatically performs the following operations:

1. Reads the requested number of bytes for the transfer from bytes 6 and 7 of the SETUP packet (**LenL** and **LenH** in Table 2-10).
2. Reads the requested descriptor's length field to determine the actual descriptor length.
3. Sends the smaller of (a) the requested number of bytes or (b) the actual number of bytes in the descriptor, over EP0BUF using the Setup Data Pointer as a data table index. This constitutes the second phase of the three-phase CONTROL transfer. The FX2 packetizes the data into multiple data transfers as necessary.
4. Automatically checks for errors and re-transmits data packets if necessary.
5. Responds to the third (handshake) phase of the CONTROL transfer to terminate the operation.

The Setup Data Pointer can be used for any *Get Descriptor* request (e.g., *Get Descriptor-String*).

It can also be used for vendor-specific requests. If bytes 6 and 7 of those requests contain the number of bytes in the transfer (see Step 1, above), the Setup Data Pointer works automatically, as it does for Get Descriptor requests; if bytes 6 and 7 don't contain the length of the transfer, the length can be loaded explicitly (see the SDPAUTO paragraphs of Section 8.7, "The Setup Data Pointer").

It is possible for the firmware to do *manual* CONTROL transfers by directly loading the EP0BUF buffer with the various packets and keeping track of which SETUP phase is in effect. This is a good USB training exercise, but not necessary due to the hardware support built into the FX2 for CONTROL transfers.

For DATA stage transfers of fewer than 64 bytes, moving the data into the EP0BUF buffer and then loading the EP0BCH:L registers with the byte count would be equivalent to loading the Setup

Data Pointer. However, this would waste bandwidth because it requires byte transfers into the EP0BUF Buffer; using the Setup Data Pointer doesn't.

2.3.4.2 Get Descriptor-Device Qualifier

Table 2-11. Get Descriptor-Device Qualifier

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x80**	IN, Device	*Set SUDPTR H:L to start of the appropriate Device Qualifier Descriptor table in RAM.*
1	bRequest	**0x06**	"Get_Descriptor"	
2	wValueL	0x00		
3	wValueH	**0x06**	Descriptor Type: Device Qualifier	
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	**LenL**		
7	wLengthH	**LenH**		

The Device Qualifier descriptor is used only by devices capable of high-speed (480 Mbps) operation; it describes information about the device that would change if the device were operating at the other speed (i.e., if the device is currently operating at high speed, the device qualifier returns information about how it would operate at full speed and vice-versa).

Device Qualifier descriptors are handled just like Device descriptors; the firmware loads the appropriate descriptor address into SUDPTRH:L, then the FX2 does the rest.

2.3.4.3 Get Descriptor-Configuration

Table 2-12. Get Descriptor-Configuration

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x80**	IN, Device	*Set SUDPTR H:L to start of Configuration Descriptor table in RAM*
1	bRequest	**0x06**	"Get_Descriptor"	
2	wValueL	**CFG**	Configuration Number	
3	wValueH	**0x02**	Descriptor Type: Configuration	
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	**LenL**		
7	wLengthH	**LenH**		

2.3.4.4 Get Descriptor-String

Table 2-13. Get Descriptor-String

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x80**	IN, Device	*Set SUDPTR H:L to start of String Descriptor table in RAM.*
1	bRequest	**0x06**	"Get_Descriptor"	
2	wValueL	**STR**	String Number	
3	wValueH	**0x03**	Descriptor Type: String	
4	wIndexL	0x00	(Language ID L)	
5	wIndexH	0x00	(Language ID H)	
6	wLengthL	**LenL**		
7	wLengthH	**LenH**		

Configuration and String descriptors are handled similarly to Device descriptors. The firmware reads byte 2 of the SETUP data to determine which configuration or string is being requested, then loads the corresponding descriptor address into SUDPTRH:L. The FX2 does the rest.

2.3.4.5 Get Descriptor-Other Speed Configuration

Table 2-14. Get Descriptor-Other Speed Configuration

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x80**	IN, Device	*Set SUDPTR H:L to start of Other Speed Configuration Descriptor table in RAM.*
1	bRequest	**0x06**	"Get_Descriptor"	
2	wValueL	**CFG**	Other Speed Configuration Number	
3	wValueH	**0x07**	Descriptor Type: Other Speed Configuration	
4	wIndexL	0x00	(Language ID L)	
5	wIndexH	0x00	(Language ID H)	
6	wLengthL	**LenL**		
7	wLengthH	**LenH**		

The Other Speed Configuration descriptor is used only by devices capable of high-speed (480 Mbps) operation; it describes the configuration(s) of the device if it were operating at the other speed (i.e., if the device is currently operating at high speed, the Other Speed Configuration returns information about full-speed configuration and vice-versa).

Other Speed Configuration descriptors are handled just like Configuration descriptors; the firmware loads the appropriate descriptor address into SUDPTRH:L, then the FX2 does the rest.

2.3.5 Set Descriptor

Table 2-15. Set Descriptor-Device

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x00**	OUT, Device	*Read device descriptor data over EP0BUF.*
1	bRequest	**0x07**	“Set_Descriptor”	
2	wValueL	0x00		
3	wValueH	**0x01**	Descriptor Type: Device	
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	**LenL**		
7	wLengthH	**LenH**		

Table 2-16. Set Descriptor-Configuration

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x00**	OUT, Device	*Read configuration descriptor data over EP0BUF.*
1	bRequest	**0x07**	“Set_Descriptor”	
2	wValueL	0x00		
3	wValueH	**0x02**	Descriptor Type: Configuration	
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	**LenL**		
7	wLengthH	**LenH**		

Table 2-17. Set Descriptor-String

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x00**	IN, Device	*Read string descriptor data over EP0BUF.*
1	bRequest	**0x07**	"Get_Descriptor"	
2	wValueL	0x00	String Number	
3	wValueH	0x03	Descriptor Type: String	
4	wIndexL	0x00	(Language ID L)	
5	wIndexH	0x00	(Language ID H)	
6	wLengthL	**LenL**		
7	wLengthH	**LenH**		

The firmware handles *Set Descriptor* requests by clearing the HSNAK bit (*by writing 1 to it*), then reading descriptor data directly from the EP0BUF buffer. The FX2 keeps track of the number of byes transferred from the host into EP0BUF, and compares this number with the length field in bytes 6 and 7. When the proper number of bytes has been transferred, the FX2 automatically responds to the STATUS phase, which is the third and final stage of the CONTROL transfer.

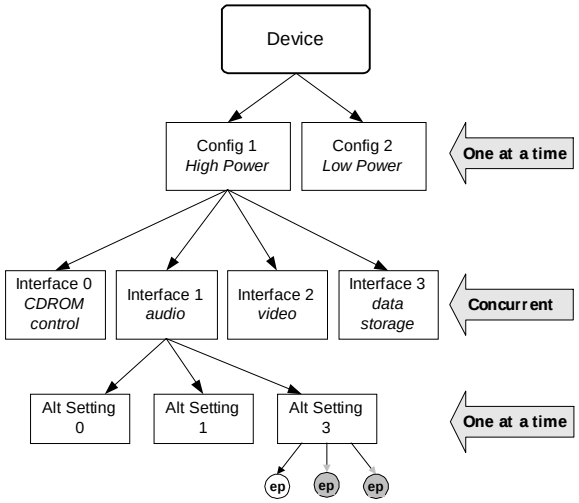
The firmware controls the flow of data in the Data Stage of a Control Transfer. After the firmware processes each OUT packet, it writes any value into the endpoint's byte count register to re-arm the endpoint.

Configurations, Interfaces, and Alternate Settings

A USB device has one or more **configurations**. Only one configuration is active at any time.

A configuration has one or more **interfaces**, all of which are concurrently active. Multiple interfaces allow different host-side device drivers to be associated with different portions of a USB device.

Each interface has one or more **alternate settings**. Each alternate setting has a collection of one or more endpoints.



This structure is a software model; the FX2 takes no action when these settings change. However, the firmware **must re-initialize endpoints** when the host changes configurations or interfaces alternate settings.

As far as the firmware is concerned, a *configuration* is simply a byte variable that indicates the current setting.

The host issues a *Set Configuration* request to select a configuration, and a *Get Configuration* request to determine the current configuration.

2.3.5.1 Set Configuration

Table 2-18. Set Configuration

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x00**	OUT, Device	*Read and store CFG, change configurations in firmware.*
1	bRequest	**0x09**	“Set Configuration”	
2	wValueL	**CFG**	Configuration Number	
3	wValueH	0x00		
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	0x00		
7	wLengthH	0x00		

When the host issues the *Set Configuration* request, the firmware saves the configuration number (byte 2, CFG, in Tabl e2-18), performs any internal operations necessary to support the configuration, and finally clears the HSNAK bit (*by writing 1 to it*) to terminate the *Set Configuration* CONTROL transfer.

After setting a configuration, the host issues Set Interface *commands to set up the various interfaces contained in the configuration.*

2.3.6 Get Configuration

Table 2-19. Get Configuration

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x80**	IN, Device	*Send CFG over EP0 after re-configuring.*
1	bRequest	**0x08**	“Get Configuration”	
2	wValueL	0x00		
3	wValueH	0x00		
4	wIndexL	0x00		
5	wIndexH	0x00		
6	wLengthL	**1**	LenL	
7	wLengthH	**0**	LenH	

When the host issues the *Get Configuration* request, the firmware returns the current configuration number. It loads the configuration number into EP0BUF, loads a byte count of one into EP0BCH:L, and finally clears the HSHAK bit (*by writing 1 to it*) to terminate the *Set Configuration* CONTROL transfer.

2.3.7 Set Interface

This confusingly-named USB command actually sets *alternate settings* for a specified interface.

USB devices can have multiple concurrent interfaces. For example, a device may have an audio system that supports different sample rates, and a graphic control panel that supports different languages. Each interface has a collection of endpoints. Except for endpoint 0, which each interface uses for device control, endpoints may not be shared between interfaces.

Interfaces may report alternate settings in their descriptors. For example, the audio interface may have setting 0, 1, and 2 for 8-KHz, 22-KHz, and 44-KHz sample rates. The panel interface may have settings 0 and 1 for English and Spanish. The *Set/Get Interface* requests select among the various alternate settings in an interface.

Table 2-20. Set Interface (Actually, Set Alternate Setting #AS for Interface #IF)

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x00**	OUT, Device	*Read and store byte 2 (AS) for Interface #IF, change setting for Interface #IF in firmware.*
1	bRequest	**0x0B**	"Set Interface"	
2	wValueL	**AS**	Alternate Setting Number	
3	wValueH	0x00		
4	wIndexL	**IF**	Interface Number	
5	wIndexH	0x00		
6	wLengthL	0x00		
7	wLengthH	0x00		

The firmware should respond to a *Set Interface* request by performing the following steps:

1. Perform the internal operation requested (such as adjusting a sampling rate).
2. Reset the data toggles for every endpoint in the interface.
3. Restore the endpoints to their default conditions, ready to send or accept data. For EP1 IN, for example, firmware should clear the BUSY bit in the EP1CS register; for EP1OUT, firmware should load any value into the EP1 byte-count register.
4. Clear the HSNAK bit (*by writing 1 to it*) to terminate the *Set Interface* CONTROL transfer.

2.3.8 Get Interface

Table 2-21. Get Interface (Actually, Get Alternate Setting #AS for interface #IF)

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x81**	IN, Device	*Send AS for Interface #IF over EP0.*
1	bRequest	**0x0A**	"Get Interface"	
2	wValueL	0x00		
3	wValueH	0x00		
4	wIndexL	**IF**	Interface Number	
5	wIndexH	0x00		
6	wLengthL	**1**	LenL	
7	wLengthH	**0**	LenH	

When the host issues the *Get Interface* request, the firmware simply returns the alternate setting for the requested interface IF and clears the HSNAK bit (*by writing 1 to it)*.

2.3.9 Set Address

When a USB device is first plugged in, it responds to device address 0 until the host assigns it a unique address using the *Set Address* request. The FX2 copies this device address into the FNADDR (Function Address) register, then subsequently responds only to requests to this address. This address is in effect until the USB device is unplugged, the host issues a USB Reset, or the host powers down.

The FNADDR register is read-only. Whenever the FX2 ReNumerates™ (see *Chapter 3, "Enumeration and ReNumeration™"*), it automatically resets FNADDR to zero, allowing the device to come back as *new*.

An FX2 program does not need to know the device address, because the FX2 automatically responds only to the host-assigned FNADDR value. The device address is readable only for debug/diagnostic purposes.

2.3.10 Sync Frame

Table 2-22. Sync Frame

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**0x82**	IN, Endpoint	*Send a frame number over EP0*
1	bRequest	**0x0C**	"Sync Frame"	*to synchronize endpoint #EP*
2	wValueL	0x00		
3	wValueH	0x00		
4	wIndexL	**EP**	Endpoint number	
5	wIndexH	0x00		
6	wLengthL	**2**	LenL	
7	wLengthH	**0**	LenH	

The *Sync Frame* request is used to establish a marker in time so the host and USB device can synchronize multi-frame transfers over isochronous endpoints.

Suppose an isochronous transmission consists of a repeating sequence of five 300-byte packets transmitted from host to device over EP8-OUT. Both host and device maintain sequence counters that count repeatedly from 1 to 5 to keep track of the packets inside a transmission. To start up in sync, both host and device need to reset their counts to "0" at the same time (in the same frame).

To get in sync, the host issues the *Sync Frame* request with EP=EP8OUT (0x08). The firmware responds by loading EP0BUF with a two-byte frame count for some future time; for example, the current frame plus 20. This marks frame "current+20" as the sync frame, during which both sides initialize their sequence counters to "0." The current frame count is always available in the USBFRAMEL and USBFRAMEH registers.

Multiple isochronous endpoints can be synchronized in this manner; the firmware can keep a separate internal sequence count for each endpoint.

About USB Frames

In full-speed mode (12 Mbps), the USB host issues an SOF (Start Of Frame) packet once every millisecond. Every SOF packet contains an 11-bit (mod-2048) frame number. The firmware services all isochronous transfers at SOF time, using a single SOF interrupt request and vector. If the FX2 detects a missing or garbled SOF packet, it can use an internal counter to generate the SOF interrupt automatically.

In high-speed (480 Mbps) mode, each frame is divided into eight 125-microsecond microframes. Although the frame counter still increments only once per frame, the host issues an SOF every microframe. The host and device always synchronize on the zero-th microframe of the frame specified in the device's response to the *Sync Frame* request; there's no mechanism for synchronizing on any other microframe.

2.3.11 Firmware Load

The USB endpoint-zero protocol provides a mechanism for mixing vendor-specific requests with standard device requests. Bits 6:5 of the bmRequestType field are set to 00 for a standard device request and to 10 for a vendor request.

Table 2-23. Firmware Download

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**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	wLengthL	**LenL**	Number of bytes	
7	wLengthH	**LenH**		

Table 2-24. Firmware Upload

Byte	Field	Value	Meaning	Firmware Response
0	bmRequestType	**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**		

The FX2 responds to two endpoint-zero vendor requests, RAM Download and RAM Upload. These requests are active whether RENUM=0 or RENUM=1.

Because bit 7 of the first byte of the SETUP packet specifies direction, only one bRequest value (0xA0) is required for the upload and download requests. These RAM load commands are available to any USB device that uses the FX2 chip.

A host loader program will typically write 0x01 to the CPUCS register to put the FX2’s CPU into RESET, load all or part of the FX2’s internal RAM with code, then reload the CPUCS register with 0 to take the CPU out of RESET.

Chapter 3 Enumeration and ReNumeration™

3.1 Introduction

The FX2's configuration is *soft*: Code and data are stored in internal RAM, which can be loaded from the host over the USB interface. FX2-based USB peripherals can operate without ROM, EPROM, or FLASH memory, shortening production lead times and making firmware updates extremely simple.

To support this soft configuration, the FX2 is capable of enumerating as a USB device *without firmware*. This automatically-enumerated USB device (the *Default USB Device)* contains a set of interfaces and endpoints and can accept firmware downloaded from the host.

Two separate Default USB Devices actually exist, one for enumeration as a full speed (12 Mbits/sec) device, and the other for enumeration as a high speed (480 Mbits/sec) device. The FX2 automatically performs the speed-detect protocol and chooses the proper Default USB Device. The two sets of Default USB Device descriptors are shown in Appendices A and B.

Once the Default USB Device enumerates, it downloads firmware and descriptor tables from the host into the FX2's on-chip RAM. The FX2 then begins executing the downloaded code, which electrically simulates a physical disconnect/connect from the USB and causes the FX2 to enumerate again as a second device, this time taking on the USB personality defined by the downloaded code and descriptors. This patented secondary enumeration process is called "ReNumeration™."

An FX2 register bit called RENUM controls whether device requests over endpoint zero are handled by firmware or automatically by the Default USB Device. When RENUM=0, the Default USB Device handles the requests automatically; when RENUM=1, they must be handled by firmware.

3.2 FX2 Startup Modes

When the FX2 comes out of reset, it can act in various ways to establish itself as a USB device. FX2 power-on behavior depends on several factors:

1. If no off-chip memory (either on the I²C-compatible bus or on the address/data bus) is connected to the FX2, it enumerates as the Default USB Device, with descriptors and VID / PID / DID supplied by hardwired internal logic (Table 3-3). RENUM is set to 0, indicating that the Default USB Device automatically handles device requests.
2. If an EEPROM containing custom VID / PID / DID values is attached to the FX2's SCL and SDA pins, FX2 also enumerates as the Default USB Device as above, but it substitutes the VID / PID / DID values from the EEPROM for its internal values. The EEPROM must contain the value 0xC0 in its first byte to indicate this mode to FX2, so this mode is called a "C0 Load". As above, RENUM is automatically set to 0, indicating that the Default USB Device automatically handles device requests. A 16-byte EEPROM is sufficiently large for a C0 Load.
3. If an EEPROM containing FX2 firmware is attached to the SCL and SDA pins, the firmware is automatically loaded from the EEPROM into the FX2's on-chip RAM, and then the CPU is taken out of reset to execute this boot-loaded code. In this case, the VID / PID / DID values are encapsulated in the firmware; the RENUM bit is automatically set to 1 to indicate that the firmware, not the Default USB Device, handles device requests. The EEPROM must contain the value 0xC2 in its first byte to indicate this mode to FX2, so this mode is called a "C2 Load". *Although the FX2 can perform C2 Loads from EEPROMs as large as 64KB, code can only be downloaded to the 8K of on-chip RAM.*
4. If a Flash, EPROM, or other memory is attached to the FX2's address/data bus (128-pin package only) *and* a properly formatted EEPROM meeting the requirements above is *not* present, *and* the EA pin is tied high (indicating that the FX2 starts code execution at 0x0000 from off-chip memory), the FX2 begins executing firmware from the off-chip memory. In this case, the VID / PID / DID values are encapsulated in the firmware; the RENUM bit is automatically set to 1 to indicate that the firmware, not internal FX2 logic, handles device requests.

Case (2) is the most frequently used mode when soft operation is desired, since the VID/PID values from EEPROM always bind the device to the appropriate host driver while allowing FX2 firmware to be easily updated. In this case, the host first uses the FX2 Default USB Device to download firmware, then the host takes the CPU out of reset so that it can execute the downloaded code. Section 3.8, "FX2 Vendor Request for Firmware Load" describes the USB *Vendor Request* that the FX2 supports for code download and upload.

The Default USB Device is fully characterized in Appendices A and B, which list the built-in FX2 descriptor tables for full-speed and high-speed enumeration, respectively. Studying these Appendices in conjunction with Tables 3-1 and 3-2 is an excellent way to learn the structure of USB descriptors.

3.3 The Default USB Device

The Default USB Device consists of a single USB configuration containing one interface (interface 0) and alternate settings 0, 1, 2 and 3. The endpoints and MaxPacketSizes reported for this device are shown in Table 3-1 (full speed) and Table 3-2 (high speed). Note that alternate setting zero consumes no interrupt or isochronous bandwidth, as recommended by the USB Specification.

Table 3-1. Default Full-speed Alternate Settings

Alternate Setting	0	1	2	3
ep0	64	64	64	64
ep1out	0	64 bulk	64 int	64 int
ep1in	0	64 bulk	64 int	64 int
ep2	0	64 bulk out (2x)	64 int out (2x)	64 iso out (2x)
ep4	0	64 bulk out (2x)	64 bulk out (2x)	64 bulk out (2x)
ep6	0	64 bulk in (2x)	64 int in (2x)	64 iso in (2x)
ep8	0	64 bulk in (2x)	64 bulk in (2x)	64 bulk in (2x)
Note: “0” means “not implemented”, “2x” means double buffered.				

Table 3-2. Default High-speed Alternate Settings

Alternate Setting	0	1	2	3
ep0	64	64	64	64
ep1out	0	512 bulk	64 int	64 int
ep1in	0	512 bulk	64 int	64 int
ep2	0	512 bulk out (2x)	512 int out (2x)	512 iso out (2x)
ep4	0	512 bulk out (2x)	512 bulk out (2x)	512 bulk out (2x)
ep6	0	512 bulk in (2x)	512 int in (2x)	512 iso in (2x)
ep8	0	512 bulk in (2x)	512 bulk in (2x)	512 bulk in (2x)
Note: “0” means “not implemented”, “2x” means double buffered.				

Although the physical size of the EP1 endpoint buffer is 64 bytes, it is reported as a 512-byte buffer for high-speed alternate setting 1. This maintains compatibility with the USB 2.0 specification, which allows only 512-byte bulk endpoints. If you use this default alternate setting (for testing, for example), be sure to limit EP1 packet sizes to 64 bytes.

When FX2 logic establishes the Default USB Device shown in Table 3-1 orTabl e3-2, it also sets the various endpoint configuration bits to match the descriptor data. For example, bulk endpoints 2, 4, and 6 are implemented in the Default USB Device, so the FX2 logic sets the corresponding EPVAL (Endpoint Valid) bits.

Chapter 8 "Access to Endpoint Buffers" contains a detailed explanation of the EPVAL bits.

3.4 EEPROM Boot-load Data Formats

This section describes three EEPROM boot-load scenarios and the EEPROM data formats that support them. The three scenarios are:

- No EEPROM, or EEPROM with invalid boot data
- “C0” EEPROM (load custom VID / PID / DID only)
- “C2” EEPROM (load firmware to on-chip RAM)

3.4.1 No EEPROM or Invalid EEPROM

In the simplest scenario, either no serial EEPROM is present on the I²C-compatible bus or an EEPROM is present, but its first byte is neither 0xC0 nor 0xC2. In this case, descriptor data is supplied by hardwired internal FX2 tables. The FX2 enumerates as the *Default USB Device*, with the ID bytes shown in Table 3-3.

Pull-up resistors are required on the SCL/SDA pins even if no device is connected. The resistors are required to allow FX2 logic to detect the “No EEPROM / Invalid EEPROM” condition.

Table 3-3. FX2 Device Characteristics, No EEPROM / Invalid EEPROM

Vendor ID	0x04B4 (Cypress Semiconductor/)
Product ID	0x8613 (EZ-USB FX2)
Device Release	0xXXYY (depends on revision)

The USB host queries the FX2 Default USB Device during enumeration, reads its device descriptor, and uses the IDs in Table 3-3 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 / Invalid EEPROM” scenario is the simplest configuration, and also the most limiting. This configuration must only be used for code development, utilizing Cypress software tools matched to the ID values in Table 3-3; *no USB peripheral based on the FX2 may use this configuration*.

3.4.2 Serial EEPROM Present, First Byte is 0xC0

Table 3-4. “C0 Load” Format

EEPROM Address	Contents
0	**0xC0**
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	Configuration byte

If, at power-on reset, the FX2 detects an EEPROM connected to its I²C-compatible bus with the value **0xC0** at address 0, the FX2 automatically copies the Vendor ID (VID), Product ID (PID), and Device ID (DID) from the EEPROM (Table 3-4) into internal storage. The FX2 then supplies these EEPROM bytes to the host as part of its response to the host’s *Get_Descriptor-Device* request (these six bytes replace only the VID / PID / DID bytes in the Default USB Device descriptor). This causes a host driver matched to the VID / PID / DID values in the EEPROM to be loaded by the host OS.

After initial enumeration, that host driver downloads FX2 firmware and USB descriptor data into the FX2’s RAM and starts the CPU. The FX2 then ReNumerates™ as a custom device. At that point, the host may load a new driver, bound to the just-loaded VID / PID / DID.

The eighth EEPROM byte contains configuration bits that control the following:

- I²C-compatible bus speed. *Default is 100 KHz*.
- Disconnect polarity. *Default is for FX2 to come out of reset connected to USB*.

FX2 firmware can change the I²C-compatible bus speed using control-register bits, so an EEPROM is not required in order to override the default setting. However, the firmware cannot modify the disconnect polarity; if it’s desired for the FX2 to come out of reset disconnected from USB, a “C0” or “C2” EEPROM must be connected.

Section 3.5 "EEPROM Configuration Byte" contains a full description of the configurations bits.

3.4.3 Serial EEPROM Present, First Byte is 0xC2

If, at power-on reset, the FX2 detects an EEPROM connected to its I²C-compatible with the value **0xC2** at address 0, the FX2 loads the EEPROM data into RAM. It also sets the RENUM bit to 1, causing device requests to be handled by the firmware instead of the Default USB Device. The "C2 Load" EEPROM data format is shown in Table3-5.

Table 3-5. "C2 Load" Format

EEPROM Address	Contents
0	**0xC2**
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	Configuration byte
8	Length H
9	Length L
10	Start Address H
11	Start Address L
---	Data Block

---	Length H
---	Length L
---	Start Address H
---	Start Address L
---	Data Block

---	0x80
---	0x01
---	0xE6
---	0x00
last	00000000

The first byte indicates a "C2 load", which instructs the FX2 to copy the EEPROM data into RAM. The FX2 reads the next six bytes (VID / PID / DID) even though they aren't used by most C2-Load applications. The eighth byte (byte 7) is the configuration byte described in the previous section.

Bytes 1-6 of a C2 EEPROM can be loaded with VID / PID / DID bytes if it is desired at some point to run the firmware with RENUM = 0 (i.e., FX2 logic handles device requests), using the EEPROM VID / PID / DID rather than the development-only VID / PID / DID values shown in Table 3-3.

One or more data records follow, starting at EEPROM address 8. Each data record consists of a 10-bit Length field (0-1023) which indicates the number of bytes in the following data block, a 13-bit Start Address (0-0x1FFF) for the data block, and the data block itself.

The last data record, *which must always consist of a single-byte load of 0x00 to the CPUCS register at 0xE600*, is marked with a "1" in the most-significant bit of the Length field. Only the least-significant bit (8051RES) of this byte is writable by the download; that bit is set to zero to bring the CPU out of reset.

*Serial EEPROM data can be loaded only into these three **on-chip** RAM spaces:*

- *Program / Data RAM at 0x0000-0x1FFF*
- *Data RAM at 0xE000-0xE1FF*
- *The CPUCS register at 0xE600 (only bit 0, 8051RES, is EEPROM-loadable).*

General-Purpose Use of the I²C-Compatible Bus

The FX2's I²C-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 CPU is up and running, firmware can access the I²C-compatible controller for general-purpose use. This makes a wide range of standard I²C peripherals available to an FX2-based system.

Other I²C devices can be attached to the SCL and SDA lines as long as there is no address conflict with the serial EEPROM described in this chapter. Chapter 13, "Input/Output" describes the general-purpose nature of the I²C-compatible interface.

3.5 EEPROM Configuration Byte

The configuration byte is valid for both EEPROM load formats (C0 and C2) and has the following format:

Configuration

b7	b6	b5	b4	b3	b2	b1	b0
0	**DISCON**	**0**	**0**	**0**	**0**	**0**	**400KHz**

Figure 3-1. EEPROM Configuration Byte

Bit 6: DISCON *USB Disconnect*

A USB hub or host detects attachment of a full-speed device by sensing a high level on the D+ wire. A USB device provides this high level using a 1500-ohm resistor between D+ and 3.3V (the D+ line is normally low, pulled down by a 15 K-ohm resistor in the hub or host). The 1500-ohm resistor is internal to the FX2.

The FX2 accomplishes ReNumeration by selectively driving or floating the 3.3V supply to its internal 1500-ohm resistor. When the supply is floated, the host no longer “sees” the FX2; it appears to have been disconnected from the USB. When the supply is then driven, the FX2 appears to have been newly-connected to the USB. From the host’s point of view, the FX2 can be disconnected and re-connected to the USB, without ever *physically* disconnecting.

The “connect state” of FX2 is controlled by a register bit called DISCON (USBCS.3), which defaults to 0, or “connected”. This default may be overridden by setting the **DISCON** bit in the EEPROM configuration byte to 1. This bit is copied into the USBCS.3 bit before the CPU is taken out of reset. Once the CPU is running, firmware can modify this bit.

Bit 0: 400KHz *I²C-compatible bus speed*

0: 100 KHz

1: 400 KHz

If 400KHZ=0, the I²C-compatible bus operates at approximately 100 KHz. If 400KHZ=1, the I²C-compatible bus operates at approximately 400 KHz. This bit is copied to I²CCTL.0, whose default value is 0, or “100 KHz”. Once the CPU is running, firmware can modify this bit.

3.6 The RENUM Bit

An FX2 control bit called “RENUM” (ReNumerated) determines whether USB device requests over endpoint zero are handled by the Default USB Device or by FX2 firmware. At power-on reset, the RENUM bit (USBCS.1) is zero, indicating that the Default USB Device will automatically handle USB device requests. Once firmware has been downloaded to the FX2 and the CPU is running, it can set RENUM=1 so that subsequent device requests will be handled by the downloaded firmware and descriptor tables. *Chapter 2, "Endpoint Zero"* describes how the firmware handles device requests while RENUM=1.

If a 128-pin FX2 is using off-chip code memory at 0x0000 and there is no boot EEPROM to supply a custom Vendor ID and Product ID, the FX2 automatically sets the RENUM bit to 1 so that device requests are always handled by the firmware and descriptor tables in the off-chip memory. The FX2 also sets RENUM=1 after a “C2 load” if the EA pin is low. In this case, firmware execution begins in internal RAM using the code loaded from the EEPROM, with the firmware handling all USB requests.

Another Use for the Default USB Device

The Default USB Device is established at power-on to set up a USB device capable of downloading firmware into the FX2’s RAM. Another useful feature of the Default USB Device is that FX2 code can be written to support the already-configured generic USB device. Before bringing the CPU out of reset, the FX2 automatically enables certain endpoints and reports them to the host via descriptors. By utilizing the Default USB Device (i.e., by keeping RENUM=0), the firmware can, with very little code, perform meaningful USB transfers that use these pre-configured endpoints. This accelerates the USB learning curve.

3.7 FX2 Response to Device Requests (RENUM=0)

Table 3-6 shows how the Default USB Device responds to endpoint zero device requests when RENUM=0.

Table 3-6. How the Default USB Device Handles EP0 Requests When RENUM=0

bRequest	Name	FX2 Response
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)
0x0C	Sync Frame	None
Vendor Requests		
0xA0	Firmware Load	Upload/Download RAM
0xA1-0xAF	Reserved	Reserved by Cypress Semiconductor
all other		None

A USB host enumerates by issuing *Set_Address*, *Get_Descriptor*, and *Set_Configuration* (to 1) requests (the *Set_Address* and *Get_Address* requests are used **only** during enumeration). After enumeration, the Default USB Device will respond to the following device requests from the host:

- 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 FX2 RAM

3.8 FX2 Vendor Request for Firmware Load

Prior to ReNumeration, the host downloads data into the FX2's internal RAM. The host can access two **on-chip** FX2 RAM spaces — Program / Data RAM at 0x0000-0x1FFF and Data RAM at 0xE000-0xE1FF — which it can download or upload whether the CPU is in reset or running: These two RAM spaces may also be boot-loaded by a "C2" EEPROM connected to the I²C-compatible bus. The host may also write to the CPUCS register to put the CPU in or out of reset.

Off-chip RAM (on the 128-pin FX2's address/data bus) cannot be uploaded or downloaded by the host via the "Firmware Load" vendor request.

The USB Specification provides for *vendor-specific requests* to be sent over endpoint zero. The FX2 uses this feature to transfer data between the host and FX2 RAM. The FX2 automatically responds to two "Firmware Load" requests, as shown inTabl e3-7 and Table 3-8.

Table 3-7. Firmware Download

Byte	Field	Value	Meaning	FX2 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 3-8. Firmware Upload

Byte	Field	Value	Meaning	FX2 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 upload and download requests are always handled by the FX2, ***regardless*** *of the state of the RENUM bit.*

The bRequest value 0xA0 is reserved for this purpose. It should never be used for another vendor request. Cypress Semiconductor also reserves bRequest values 0xA1 through 0xAF; devices should not use these bRequest values.

A host loader program will typically write 0x01 to the CPUCS register to put the CPU into RESET, load all or part of the FX2 RAM with firmware, then reload the CPUCS register with 0 to take the CPU out of RESET. The CPUCS register (at 0xE600) is the only FX2 register that can be written using the Firmware Download command.

3.9 How the Firmware ReNumerates

Two control bits in the USBCS (USB Control and Status) register control the ReNumeration™ process: DISCON and RENUM.

USBCS **USB Control and Status** **E680**

b7	b6	b5	b4	b3	b2	b1	b0
HSM	**0**	**0**	**0**	**DISCON**	**NOSYNSOF**	**RENUM**	**SIGRSUME**
R/W	R	R	R	R/W	R/W	R/W	R/W
0	0	0	0	0	1	0	0

Figure 3-2. USB Control and Status Register

To simulate a USB disconnect, the firmware sets DISCON to 1. To reconnect, the firmware clears DISCON to 0.

Before reconnecting, the firmware sets or clears the RENUM bit to indicate whether the firmware or the Default USB Device will handle device requests over endpoint zero: if RENUM=0, the Default USB Device will handle device requests; if RENUM=1, the firmware will.

3.10 Multiple ReNumerations™

FX2 firmware 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.

Chapter 4 Interrupts

4.1 Introduction

The EZ-USB FX2's interrupt architecture is an enhanced and expanded version of the standard 8051's. The FX2 responds to the interrupts shown in Table 4-1; interrupt sources that are not present in the standard 8051 are shown in **bold** type.

.

Table 4-1. FX2 Interrupts

FX2 Interrupt	Source	Interrupt Vector	Natural Priority
IE0	$\overline{\text{INT0}}$ Pin	0x0003	1
TF0	Timer 0 Overflow	0x000B	2
IE1	$\overline{\text{INT1}}$ Pin	0x0013	3
TF1	Timer 1 Overflow	0x001B	4
RI_0 & TI_0	USART0 Rx & Tx	0x0023	5
TF2	**Timer 2 Overflow**	0x002B	6
Resume	**WAKEUP / WU2 Pin or USB Resume**	0x0033	0
RI_1 & TI_1	**USART1 Rx & Tx**	0x003B	7
USBINT	**USB**	0x0043	8
I²CINT	**I²C-Compatible Bus**	0x004B	9
IE4	**GPIF / FIFOs / $\overline{\text{INT4}}$ Pin**	0x0053	10
IE5	**INT5 Pin**	0x005B	11
IE6	**INT6 Pin**	0x0063	12

The **Natural Priority** column in Table 4-1 shows the FX2 interrupt priorities. As explained in *Chapter 14, "Timers/Counters and Serial Interface"*, the FX2 can assign each interrupt to a high or low priority group. The FX2 resolves priorities within the groups using the natural priorities.

4.2 SFRs

The following SFRs are associated with interrupt control:

- IE - SFR 0xA8 (Table 4-2)
- IP - SFR 0xB8 (Table 4-3)
- EXIF - SFR 0x91 (Table 4-4)
- EICON - SFR 0xD8 (Table 4-5)
- EIE - SFR 0xE8 (Table 4-6)
- EIP - SFR 0xF8 (Table 4-7)

The IE and IP SFRs provide interrupt enable and priority control for the standard interrupt unit, as with the standard 8051. Additionally, these SFRs provide control bits for the Serial Port 1 interrupt.

The EXIF, EICON, EIE and EIP Registers provide flags, enable control, and priority control.

Table 4-2. IE Register — SFR 0xA8

Bit	Function
IE.7	**EA** - Global interrupt enable. Controls masking of all interrupts except USB wakeup (resume). EA = 0 disables all interrupts except USB wakeup. When EA = 1, interrupts are enabled or masked by their individual enable bits.
IE.6	**ES1** - Enable Serial Port 1 interrupt. ES1 = 0 disables Serial port 1 interrupts (TI_1 and RI_1). ES1 = 1 enables interrupts generated by the TI_1 or RI_1 flag.
IE.5	**ET2** - Enable Timer 2 interrupt. ET2 = 0 disables Timer 2 interrupt (TF2). ET2=1 enables interrupts generated by the TF2 or EXF2 flag.
IE.4	**ES0** - Enable Serial Port 0 interrupt. ES0 = 0 disables Serial Port 0 interrupts (TI_0 and RI_0). ES0=1 enables interrupts generated by the TI_0 or RI_0 flag.
IE.3	**ET1** - Enable Timer 1 interrupt. ET1 = 0 disables Timer 1 interrupt (TF1). ET1=1 enables interrupts generated by the TF1 flag.
IE.2	**EX1** - Enable external interrupt 1. EX1 = 0 disables external interrupt 1 (INT1). EX1=1 enables interrupts generated by the $\overline{\text{INT1}}$ pin.
IE.1	**ET0** - Enable Timer 0 interrupt. ET0 = 0 disables Timer 0 interrupt (TF0). ET0=1 enables interrupts generated by the TF0 flag.
IE.0	**EX0** - Enable external interrupt 0. EX0 = 0 disables external interrupt 0 (INT0). EX0=1 enables interrupts generated by the $\overline{\text{INT0}}$ pin.

Table 4-3. IP Register — SFR 0xB8

Bit	Function
IP.7	**Reserved**. Read as 1.
IP.6	**PS1** - Serial Port 1 interrupt priority control. PS1 = 0 sets Serial Port 1 interrupt (TI_1 or RI_1) to low priority. PS1 = 1 sets Serial port 1 interrupt to high priority.
IP.5	**PT2** - Timer 2 interrupt priority control. PT2 = 0 sets Timer 2 interrupt (TF2) to low priority. PT2 = 1 sets Timer 2 interrupt to high priority.
IP.4	**PS0** - Serial Port 0 interrupt priority control. PS0 = 0 sets Serial Port 0 interrupt (TI_0 or RI_0) to low priority. PS0 = 1 sets Serial Port 0 interrupt to high priority.
IP.3	**PT1** - Timer 1 interrupt priority control. PT1 = 0 sets Timer 1 interrupt (TF1) to low priority. PT1 = 1 sets Timer 1 interrupt to high priority.
IP.2	**PX1** - External interrupt 1 priority control. PX1 = 0 sets external interrupt 1 (INT1) to low priority. PT1 = 1 sets external interrupt 1 to high priority.
IP.1	**PT0** - Timer 0 interrupt priority control. PT0 = 0 sets Timer 0 interrupt (TF0) to low priority. PT0 = 1 sets Timer 0 interrupt to high priority.
IP.0	**PX0** - External interrupt 0 priority control. PX0 = 0 sets external interrupt 0 (INT0) to low priority. PX0 = 1 sets external interrupt 0 to high priority.

Table 4-4. EXIF Register — SFR 0x91

Bit	Function
EXIF.7	**IE5** - External Interrupt 5 flag. IE5 = 1 indicates a falling edge was detected at the INT5 pin. IE5 must be cleared by software. Setting IE5 in software generates an interrupt, if enabled.
EXIF.6	**IE4** - GPIF/FIFO/External Interrupt 4 flag. The “INT4” interrupt is internally connected to the FIFO/GPIF interrupt by default; it can optionally function as External Interrupt 4 on the 100- and 128-pin FX2. When configured as External Interrupt 4, IE4 indicates that a rising edge was detected at the INT4 pin. IE4 must be cleared by software. Setting IE4 in software generates an interrupt, if enabled.
EXIF.5	**I²CINT** - I²C-Compatible Bus Interrupt flag. I²CINT = 1 indicates an I²C-Compatible Bus interrupt. I²CINT must be cleared by software. Setting I²CINT in software generates an interrupt, if enabled.
EXIF.4	**USBINT** - USB Interrupt flag. USBINT = 1 indicates an USB interrupt. USBINT must be cleared by software. Setting USBINT in software generates an interrupt, if enabled.
EXIF.3	**Reserved**. Read as 1.
EXIF.2-0	**Reserved**. Read as 0.

Table 4-5. EICON Register — SFR 0xD8

Bit	Function
EICON.7	**SMOD1** - Serial Port 1 baud rate doubler enable. When SMOD1 = 1, the baud rate for Serial Port 1 is doubled.
EICON.6	**Reserved**. Read as 1.
EICON.5	**ERESI** - Enable Resume interrupt. ERESI = 0 disables the Resume interrupt. ERESI = 1 enables interrupts generated by the resume event.
EICON.4	**RESI** - Wakeup interrupt flag. RESI = 1 indicates a false-to-true transition was detected at the WAKEUP or WU pin, or that USB activity has resumed from the suspended state. RESI must be cleared by software before exiting the interrupt service routine, otherwise the interrupt will immediately be reasserted. Setting RESI = 1 in software generates a wakeup interrupt, if enabled.
EICON.3	**INT6** - External interrupt 6. When INT6 = 1, the INT6 pin has detected a low to high transition. INT6 must be cleared by software. Setting this bit in software generates an INT6 interrupt, if enabled.
EICON.2-0	**Reserved**. Read as 0.

Table 4-6. EIE Register — SFR 0xE8

Bit	Function
EIE.7-5	Reserved. Read as 1.
EIE.4	EX6 - Enable external interrupt 6. EX6 = 0 disables external interrupt 6 (INT6). EX6 = 1 enables interrupts generated by the INT6 pin.
EIE.3	EX5 - Enable external interrupt 5. EX5 = 0 disables external interrupt 5 (INT5). EX5 = 1 enables interrupts generated by the $\overline{\text{INT5}}$ pin.
EIE.2	EX4 - Enable external interrupt 4. EX4 = 0 disables external interrupt 4 (INT4). EX4 = 1 enables interrupts generated by the INT4 pin or by the FIFO/GPIF Interrupt.
EIE.1	EI²C - Enable I²C-Compatible Bus interrupt (I²CINT). EI²C = 0 disables the I²C-Compatible Bus interrupt. EI²C = 1 enables interrupts generated by the I²C-Compatible Bus controller.
EIE.0	EUSB - Enable USB interrupt (USBINT). EUSB = 0 disables USB interrupts. EUSB = 1 enables interrupts generated by the USB Interface.

Table 4-7. EIP Register — SFR 0xF8

Bit	Function
EIP.7-5	**Reserved**. Read as 1.
EIP.4	**PX6** - External interrupt 6 priority control. PX6 = 0 sets external interrupt 6 (INT6) to low priority. PX6 = 1 sets external interrupt 6 to high priority.
EIP.3	**PX5** - External interrupt 5 priority control. PX5 = 0 sets external interrupt 5 ($\overline{\text{INT5}}$) to low priority. PX5=1 sets external interrupt 5 to high priority.
EIP.2	**PX4** - External interrupt 4 priority control. PX4 = 0 sets external interrupt 4 (INT4 / GPIF / FIFO) to low priority. PX4=1 sets external interrupt 4 to high priority.
EIP.1	**PI²C** - I²CINT priority control. PI²C = 0 sets I²C-Compatible Bus interrupt to low priority. PI²C=1 sets I²C-Compatible Bus interrupt to high priority.
EIP.0	**PUSB** - USBINT priority control. PUSB = 0 sets USB interrupt to low priority. PUSB=1 sets USB interrupt to high priority.

4.2.1 803x/805x Compatibility

The implementation of interrupts is similar to that of the Dallas Semiconductor DS80C320. Table 4-8 summarizes the differences in interrupt implementation between the Intel 8051, the Dallas Semiconductor DS80C320, and the FX2.

Table 4-8. Summary of Interrupt Compatibility

Feature	Intel 8051	Dallas DS80C320	Cypress FX2
Power Fail Interrupt	Not implemented	Internally generated	Replaced with RESUME Interrupt
External Interrupt 0	Implemented	Implemented	Implemented
Timer 0 Interrupt	Implemented	Implemented	Implemented
External Interrupt 1	Implemented	Implemented	Implemented
Timer 1 Interrupt	Implemented	Implemented	Implemented
Serial Port 0 Interrupt	Implemented	Implemented	Implemented
Timer 2 Interrupt	Not implemented	Implemented	Implemented
Serial Port 1 Interrupt	Not implemented	Implemented	Implemented
External Interrupt 2	Not implemented	Implemented	Replaced with autovectored USB Interrupt
External Interrupt 3	Not implemented	Implemented	Replaced with I²C-Compatible Bus Interrupt
External Interrupt 4	Not implemented	Implemented	Replaced by autovectored FIFO/GPIF Interrupt. Can be configured as External Interrupt 4 on 100- and 128-pin FX2 only.
External Interrupt 5	Not implemented	Implemented	Implemented
Watchdog Timer Interrupt	Not implemented	Internally generated	Replaced with External Interrupt 6
Real-time Clock Interrupt	Not implemented	Implemented	Not implemented

4.3 Interrupt Processing

When an enabled interrupt occurs, the FX2 completes the instruction it's currently executing, then vectors to the address of the interrupt service routine (ISR) associated with that interrupt (see Table 4-9). The FX2 executes the ISR to completion unless another interrupt of higher priority occurs. Each ISR ends with a `RETI` (return from interrupt) instruction. After executing the `RETI`, the FX2 continues executing firmware at the instruction following the one which was executing when the interrupt occurred.

The FX2 always completes the instruction in progress before servicing an interrupt. If the instruction in progress is `RETI`, or a write access to any of the IP, IE, EIP, or EIE SFRs, the FX2 completes one additional instruction before servicing the interrupt.

4.3.1 Interrupt Masking

The EA Bit in the IE SFR (IE.7) is a global enable for all interrupts except the RESUME (USB wakeup) interrupt, which is always enabled. When EA = 1, each interrupt is enabled or masked by its individual enable bit. When EA = 0, all interrupts are masked except the USB wakeup interrupt.

Table 4-9 provides a summary of interrupt sources, flags, enables, and priorities.

Table 4-9. Interrupt Flags, Enables, Priority Control, and Vectors

Interrupt	Description	Interrupt Request Flag	Interrupt Enable	Assigned Priority Control	Natural Priority	Interrupt Vector
RESUME	Resume interrupt	EICON.4	EICON.5	*Always Highest*	0 (highest)	0x0033
INT0	External interrupt 0	TCON.1	IE.0	IP.0	1	0x0003
TF0	Timer 0 interrupt	TCON.5	IE.1	IP.1	2	0x000B
INT1	External interrupt 1	TCON.3	IE.2	IP.2	3	0x0013
TF1	Timer 1 interrupt	TCON.7	IE.3	IP.3	4	0x001B
TI_0 or RI_0	Serial port 0 transmit or receive interrupt	SCON0.1 (TI.0) SCON0.0 (RI_0)	IE.4	IP.4	5	0x0023
TF2 or EXF2	Timer 2 interrupt	T2CON.7 (TF2) T2CON.6 (EXF2)	IE.5	IP.5	6	0x002B
TI_1 or RI_1	Serial port 1 transmit or receive interrupt	SCON1.1 (TI_1) SCON1.0 (RI_1)	IE.6	IP.6	7	0x003B
USBINT	Autovectored USB interrupt	EXIF.4	EIE.0	EIP.0	8	0x0043
I²CINT	I²C-Compatible Bus interrupt	EXIT.5	EIE.1	EIP.1	9	0x004B
INT4	Autovectored FIFO / GPIF or External interrupt 4	EXIF.6	EIE.2	EIP.2	10	0x0053
INT5	External interrupt 5	EXIF.7	EIE.3	EIP.3	11	0x005B
INT6	External interrupt 6	EICON.3	EIE.4	EIP.4	12	0x0063

4.3.1.1 Interrupt Priorities

There are two stages of interrupt priority: assigned interrupt level and natural priority. Assigned priority is set by FX2 firmware; natural priority is as shown in Table 4-9, and is fixed.

The assigned interrupt level (highest, high, or low) takes precedence over natural priority. The RESUME (USB wakeup) interrupt always has highest assigned priority and is the only interrupt that can have highest assigned priority. All other interrupts can be assigned either high or low priority.

In addition to an assigned priority level (high or low), each interrupt also has a natural priority, as listed in Table 4-9. *Simultaneous* interrupts with the same assigned priority level (for example, both high) are resolved according to their natural priority. For example, if INT0 and INT1 are both assigned high priority and both occur simultaneously, INT0 takes precedence due to its higher natural priority.

Once an interrupt is being serviced, only an interrupt of higher *assigned* priority level can interrupt the service routine. That is, an ISR for a low-assigned-level interrupt can only be interrupted by a high-assigned-level interrupt. An ISR for a high-assigned-level interrupt can only be interrupted by the RESUME interrupt.

4.3.2 Interrupt Sampling

The internal timers and serial ports generate interrupts by setting the interrupt flag bits shown in Table 4-9. These interrupts are sampled once per instruction cycle (i.e., once every 4 CLKOUT cycles).

INT0 and INT1 are both active low and can be programmed to be either edge-sensitive or level-sensitive, through the IT0 and IT1 bits in the TCON SFR. When ITx = 0, INTx is level-sensitive and the FX2 sets the IEx flag when the $\overline{\text{INTx}}$ pin is sampled low. When ITx = 1, INTx is edge-sensitive and the FX2 sets the IEx flag when the $\overline{\text{INTx}}$ pin is sampled high then low on consecutive samples.

The remaining five interrupts (INT 4-6, USB & I²C-Compatible Bus interrupts) are edge-sensitive only. INT6 and INT4 are active high and INT5 is active low.

To ensure that edge-sensitive interrupts are detected, the interrupt pins should be held in each state for a minimum of one instruction cycle (4 CLKOUT cycles). Level-sensitive interrupts are not latched; their pins must remain asserted until the interrupt is serviced.

4.3.3 Interrupt Latency

Interrupt response time depends on the current state of the FX2. The fastest response time is 5 instruction cycles: 1 to detect the interrupt, and 4 to perform the `LCALL` to the ISR.

The maximum latency is 13 instruction cycles. This 13-cycle latency occurs when the FX2 is currently executing a `RETI` instruction followed by a `MUL` or `DIV` instruction. The 13 instruction cycles in this case are: 1 to detect the interrupt, 3 to complete the `RETI`, 5 to execute the `DIV` or `MUL`, and 4 to execute the `LCALL` to the ISR.

This13-instruction-cycle latency excludes autovector latency for the USB and FIFO/GPIF interrupts (see Sections 4.5 and 4.8). Autovectoring adds a fixed 4 instruction cycles, so the maximum latency for an autovectored USB or FIFO/GPIF interrupt is 13 + 4 = 17 instruction cycles.

4.4 USB-Specific Interrupts

The FX2 provides 28 USB-specific interrupts. One, “Resume”, has its own dedicated interrupt; the other 27 share the “USB” interrupt.

4.4.1 Resume Interrupt

After the FX2 has entered its idle state, it responds to an external signal on its WAKEUP/WU2 pins or resumption of USB bus activity by restarting its oscillator and resuming firmware execution.

Chapter 6, "Power Management" describes suspend/resume signaling in detail, and presents an example which uses the Wakeup Interrupt.

4.4.2 USB Interrupts

Table 4-10 shows the 27 USB requests that share the USB Interrupt. Figur e4-1 shows the USB Interrupt logic; the bottom IRQ, EP8ISOERR, is expanded in the diagram to show the logic which is associated with each USB interrupt request.

Table 4-10. Individual USB Interrupt Sources

Priority	INT2VEC Value	Source	Notes
1	00	SUDAV	SETUP Data Available
2	04	SOF	Start of Frame (or microframe)
3	08	SUTOK	Setup Token Received
4	0C	SUSPEND	USB Suspend request
5	10	USB RESET	Bus reset
6	14	HISPEED	Entered high speed operation
7	18	EP0ACK	FX2 ACK'd the CONTROL Handshake
8	1C	reserved	
9	20	EP0-IN	EP0-IN ready to be loaded with data
10	24	EP0-OUT	EP0-OUT has USB data
11	28	EP1-IN	EP1-IN ready to be loaded with data
12	2C	EP1-OUT	EP1-OUT has USB data
13	30	EP2	IN: buffer available. OUT: buffer has data
14	34	EP4	IN: buffer available. OUT: buffer has data
15	38	EP6	IN: buffer available. OUT: buffer has data
16	3C	EP8	IN: buffer available. OUT: buffer has data
17	40	IBN	IN-Bulk-NAK (any IN endpoint)
18	44	reserved	
19	48	EP0PING	EP0 OUT was Pinged and it NAK'd
20	4C	EP1PING	EP1 OUT was Pinged and it NAK'd
21	50	EP2PING	EP2 OUT was Pinged and it NAK'd
22	54	EP4PING	EP4 OUT was Pinged and it NAK'd
23	58	EP6PING	EP6 OUT was Pinged and it NAK'd
24	5C	EP8PING	EP8 OUT was Pinged and it NAK'd
25	60	ERRLIMIT	Bus errors exceeded the programmed limit
26	64	reserved	
27	68	reserved	
28	6C	reserved	
29	70	EP2ISOERR	ISO EP2 OUT PID sequence error
30	74	EP4ISOERR	ISO EP4 OUT PID sequence error
31	78	EP6ISOERR	ISO EP6 OUT PID sequence error
32	7C	EP8ISOERR	ISO EP8 OUT PID sequence error

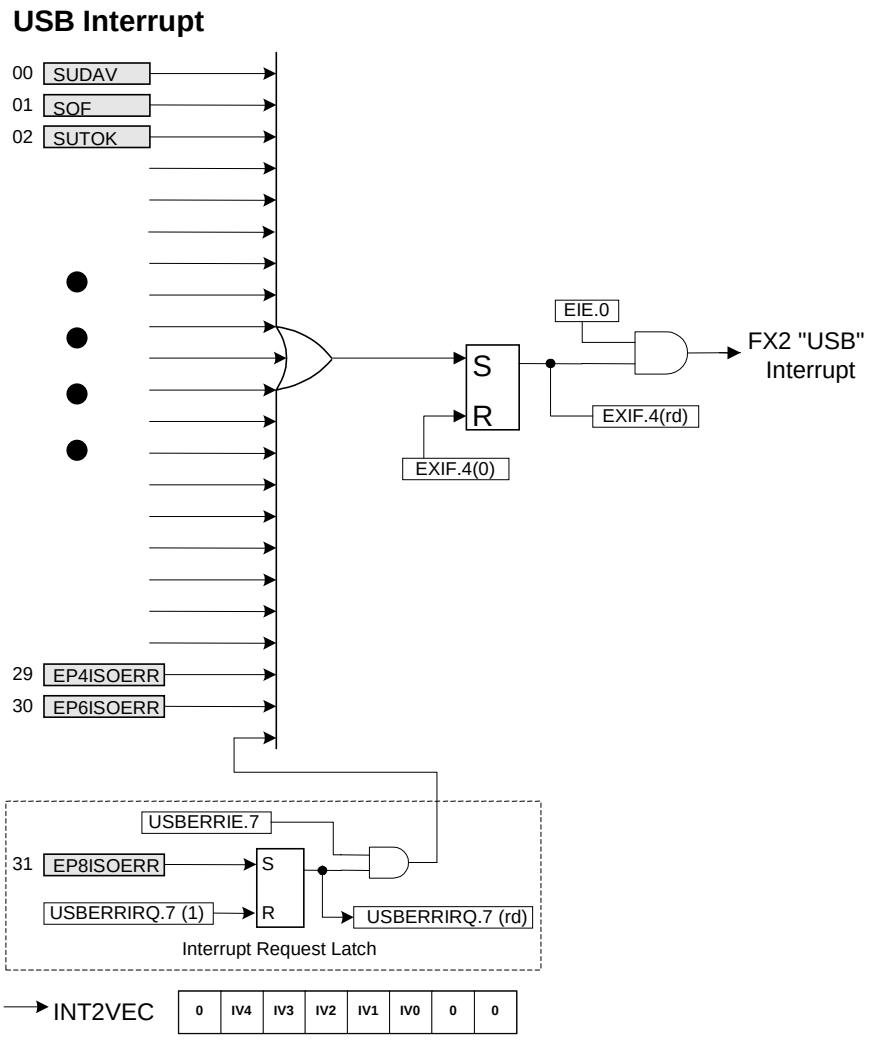


Figure 4-1. USB Interrupts

Referring to the logic inside the dotted lines, each USB interrupt source has an interrupt request latch. IRQ bits are set automatically by the FX2; firmware clears an IRQ bit by writing a “1” to it. The output of each latch is ANDed with an Interrupt Enable Bit and then ORed with all the other USB Interrupt request sources.

The FX2 prioritizes the USB interrupts and constructs an Autovector, which appears in the INT2VEC register. The interrupt vector values IV[4:0] are shown to the left of the interrupt sources (shaded boxes); 0 is the highest priority, 31 is the lowest. If two USB interrupts occur simultaneously, the prioritization affects which one is first indicated in the INT2VEC register.

If Autovectoring is enabled, the INT2VEC byte replaces the contents of address 0x0045 in the FX2's program memory. This causes the FX2 to automatically vector to a different address for each USB interrupt source. This mechanism is explained in detail in Section 4.5. *"USB-Interrupt Autovectors."*

Due to the OR gate in Figure 4-1, assertion of any of the individual USB interrupt sources sets the FX2's "main" USB Interrupt request bit (EXIF.4). This main USB interrupt is enabled by setting EIE.0 to 1.

To clear the main USB interrupt request, firmware clears the EXIF.4 bit to 0.

After servicing a USB interrupt, FX2 firmware clears the individual USB source's IRQ bit by setting it to 1. If any other USB interrupts are pending, the act of clearing the IRQ bit causes the FX2 to generate another pulse for the highest-priority pending interrupt. If more than one is pending, each is serviced in the priority order shown in Figur e4-1, starting with SUDAV (priority 00) as the highest priority, and ending with EP8ISOERR (priority 31) as the lowest.

*The main USB interrupt request is cleared by **clearing** the EXIF.4 bit **to 0**; each individual USB interrupt is cleared by **setting** its IRQ bit **to 1**.*

Important

It is important in any USB Interrupt Service Routine (ISR) to clear the main USB Interrupt **before** clearing the individual USB interrupt request latch. This is because as soon as the individual USB interrupt is cleared, any pending USB interrupt will immediately try to generate another main USB Interrupt. If the main USB IRQ bit has not been previously cleared, the pending interrupt will be lost.

Figure 4-2 illustrates a typical USB ISR for endpoint 2-IN.

```
USB_ISR:  push  dps
          push  dpl
          push  dph
          push  dpl1
          push  dph1
          push  acc
;
          mov   a,EXIF           ; FIRST clear the USB (INT2) interrupt request
          clr   acc.4
          mov   EXIF,a           ; Note:  EXIF reg is not bit-addressable
;
          mov   dptr,#USBERRIRQ  ; now clear the USB interrupt request
          mov   a,#10000000b     ; use EP8ISOERR as example
          movx  @dptr,a
;
; (service the interrupt here)
;
          pop   acc
          pop   dph1
          pop   dpl1
          pop   dph
          pop   dpl
          pop   dps
;
          reti
```

Figure 4-2. The Order of Clearing Interrupt Requests is Important

The registers associated with the individual USB interrupt sources are described in *Chapter 15, "Registers"* and Section 8.6, "CPU Control of FX2 Endpoints". Each interrupt source has an enable (IE) and a request (IRQ) bit. Firmware sets the IE bit to 1 to enable the interrupt. The FX2 sets an IRQ bit to 1 to request an interrupt, and the firmware clears an IRQ bit by writing a “1” to it.

4.4.2.1 SUTOK, SUDAV Interrupts

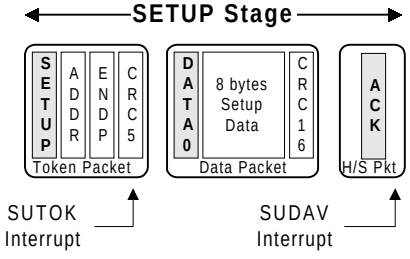


Figure 4-3. SUTOK and SUDAV Interrupts

SUTOK and SUDAV are supplied to the FX2 by CONTROL endpoint zero. The first portion of a USB CONTROL transfer is the SETUP stage shown in Figure 4-3 (a full CONTROL transfer is shown in Figure 2-1). When the FX2 decodes a SETUP packet, it asserts the SUTOK (SETUP Token) Interrupt Request. After the FX2 has received the eight bytes error-free and copied them into the eight internal registers at SETUPDAT, it asserts the SUDAV Interrupt Request.

Firmware responds to the SUDAV Interrupt by reading the eight SETUP data bytes in order to decode the USB request (*Chapter 2, "Endpoint Zero"*).

The SUTOK Interrupt is provided to give advance warning that the eight register bytes at SETUPDAT are about to be overwritten. It is useful for debug and diagnostic purposes.

4.4.2.2 SOF Interrupt

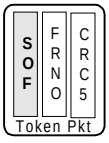


Figure 4-4. A Start Of Frame (SOF) Packet

A USB Start-of-Frame Interrupt Request is asserted when the host sends a Start of Frame (SOF) packet. SOFs occur once per millisecond in full-speed (12 Mbits/sec) mode, and once every 125 microseconds in high-speed (480 Mbits/sec) mode.

When the FX2 receives an SOF packet, it copies the eleven-bit frame number (FRNO in Figu re4-4) into the USBFRAMEH:L registers and asserts the SOF Interrupt Request. All isochronous endpoint data is generally serviced via the SOF Interrupt.

4.4.2.3 Suspend Interrupt

If the FX2 detects a “suspend” condition from the host, it asserts the SUSP (Suspend) Interrupt Request. A full description of Suspend-Resume signaling appears in *Chapter 6, "Power Management".*

4.4.2.4 USB RESET Interrupt

The USB host signals a bus reset by driving both D+ and D- low for at least 10 ms. When the FX2 detects the onset of USB bus reset, it asserts the URES Interrupt Request.

4.4.2.5 HISPEED Interrupt

This interrupt is asserted when the host grants high-speed (480 Mbps) access to the FX2.

4.4.2.6 EP0ACK Interrupt

This interrupt is asserted when the FX2 has acknowledged the STATUS stage of a CONTROL transfer on endpoint 0.

4.4.2.7 Endpoint Interrupts

These interrupts are asserted when an endpoint requires service.

For an OUT endpoint, the interrupt request signifies that OUT data has been sent from the host, validated by the FX2, and is in the endpoint buffer memory.

For an IN endpoint, the interrupt request signifies that the data previously loaded by the FX2 into the IN endpoint buffer has been read and validated by the host, making the IN endpoint buffer ready to accept new data.

Table 4-11. Endpoint Interrupts

EP0-IN	EP0-IN ready to be loaded with data (BUSY bit 1-to-0)
EP0-OUT	EP0-OUT has received USB data (BUSY bit 1-to-0)
EP1-IN	EP1-IN ready to be loaded with data (BUSY bit 1-to-0)
EP1-OUT	EP1-OUT has received USB data (BUSY bit 1-to-0)
EP2	IN: Buffer available (Empty Flag 1-to-0) OUT: Buffer has received USB data (Empty Flag 0-to-1)
EP4	IN: Buffer available (Empty Flag 1-to-0) OUT: Buffer has received USB data (Empty Flag 0-to-1)
EP6	IN: Buffer available (Empty Flag 1-to-0) OUT: Buffer has received USB data (Empty Flag 0-to-1)
EP8	IN: Buffer available (Empty Flag 1-to-0) OUT: Buffer has received USB data (Empty Flag 0-to-1)

4.4.2.8 In-Bulk-NAK (IBN) Interrupt

When the host sends an IN token to any IN endpoint which does not have data to send, the FX2 automatically NAKs the IN token and asserts this interrupt.

4.4.2.9 EPxPING Interrupt

These interrupts are active only during high speed (480 Mbits/sec) operation.

USB 2.0 improves the USB 1.1 bus bandwidth utilization by implementing a PING-NAK mechanism for OUT transfers. When the host wishes to send OUT data to an endpoint, it first sends a PING token to see if the endpoint is ready (i.e. if it has an empty buffer). If a buffer is not available, the FX2 returns a NAK handshake. PING-NAK transactions continue to occur until an OUT buffer is available, at which time the FX2 answers a PING with an ACK handshake and the host sends the OUT data to the endpoint.

The EPxPING interrupt is asserted when the host PINGs an endpoint and the FX2 responds with a NAK because no endpoint buffer memory is available.

4.4.2.10 ERRLIMIT Interrupt

This interrupt is asserted when the USB error-limit counter has exceeded the preset error limit threshold. See Section 8.6.3.3 for full details.

4.4.2.11 EPxISOERR Interrupt

These interrupts are asserted when an ISO data PID is missing or arrives out of sequence, or when an ISO packet is dropped because no buffer space is available (to receive an OUT packet) or no data is available to be sent (from an IN buffer).

4.5 USB-Interrupt Autovectors

The main USB interrupt is shared by 27 interrupt sources. To save the code and processing time which normally would be required to identify the individual USB interrupt source, the FX2 provides a second level of interrupt vectoring, called *Autovectoring*. When a USB interrupt is asserted, the FX2 pushes the program counter onto its stack then jumps to address 0x0043, where it expects to find a “jump” instruction to the USB Interrupt service routine.

The FX2 jump instruction is encoded as follows:

Table 4-12. FX2 JUMP Instruction

Address	Op-Code	Hex Value
0x0043	LJMP	0x02
0x0044	AddrH	0xHH
0x0045	AddrL	0xLL

If Autovectoring is enabled (AV2EN=1 in the INTSETUP register), the FX2 substitutes its INT2VEC byte (see Table 4-10) for the byte at address 0x0045. Therefore, if the high byte (“page”) of a jump-table address is preloaded at location 0x0044, the automatically-inserted INT2VEC byte at 0x0045 will direct the jump to the correct address out of the 27 addresses within the page.

As shown in Table 4-13, the jump table contains a series of jump instructions, one for each individual USB Interrupt source’s ISR.

Table 4-13. A Typical USB-Interrupt Jump Table

Table Offset	Instruction
0x00	LJMP SUDAV_ISR
0x04	LJMP SOF_ISR
0x08	LJMP SUTOK_ISR
0x0C	LJMP SUSPEND_ISR
0x10	LJMP USBRESET_ISR
0x14	LJMP HISPEED_ISR
0x18	LJMP EP0ACK_ISR
0x1C	LJMP SPARE_ISR
0x20	LJMP EP0IN _ISR
0x24	LJMP EP0OUT_ISR
0x28	LJMP EP1IN _ISR
0x2C	LJMP EP1OUT_ISR
0x30	LJMP EP2_ISR
0x34	LJMP EP4_ISR
0x38	LJMP EP6_ISR
0x3C	LJMP EP8_ISR
0x40	LJMP IBN_ISR
0x44	LJMP SPARE_ISR
0x48	LJMP EP0PING_ISR
0x4C	LJMP EP1PING_ISR
0x50	LJMP EP2PING_ISR
0x54	LJMP EP4PING_ISR
0x58	LJMP EP6PING_ISR
0x5C	LJMP EP8PING_ISR
0x60	LJMP ERRLIMIT_ISR
0x64	LJMP SPARE_ISR
0x68	LJMP SPARE_ISR
0x6C	LJMP SPARE_ISR
0x70	LJMP EP2ISOERR_ISR
0x74	LJMP EP2ISOERR_ISR
0x78	LJMP EP2ISOERR_ISR
0x7C	LJMP EP2ISOERR_ISR

4.5.1 USB Autovector Coding

To employ autovectoring for the USB interrupt:

1. Insert a jump instruction at 0x0043 to a table of jump instructions to the various USB interrupt service routines. Make sure the jump table starts on a 0x0100-byte page boundary.
2. Code the jump table with jump instructions to each individual USB interrupt service routine. This table has two important requirements, arising from the format of the INT2VEC Byte (zero-based, with the 2 LSBs set to 0):
   - It must begin on a page boundary (address 0xnn00)
   - The jump instructions must be four bytes apart.
3. The interrupt service routines can be placed anywhere in memory.
4. Write initialization code to enable the USB interrupt (INT2) and Autovectoring.

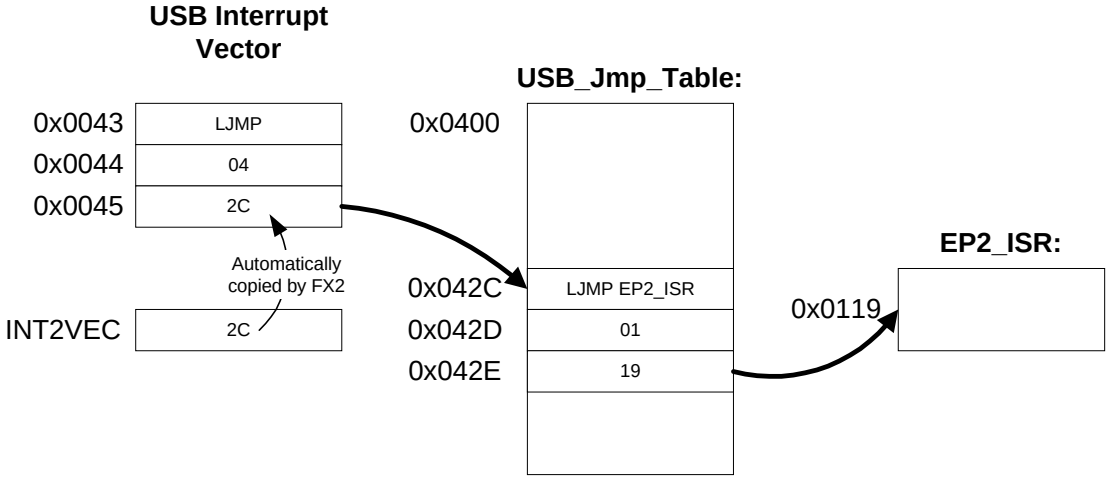


Figure 4-5. The USB Autovector Mechanism in Action

Figure 4-5 illustrates an ISR that services endpoint 2. When endpoint 2 requires service, the FX2 asserts the USB interrupt request, vectoring to location 0x0043.

The jump instruction at this location, which was originally coded as “LJMP 04**00**”, becomes “LJMP 04**2C**” because the FX2 automatically inserts **2C**, the INT2VEC value for EP2 (Table 4-13).

The FX2 jumps to 0x042C, where it executes the jump instruction to the EP2 ISR, arbitrarily located for this example at address 0x0119.

Once the FX2 vectors to 0x0043, initiation of the endpoint-specific ISR takes only eight instruction cycles.

4.6 I²C-Compatible Bus Interrupt

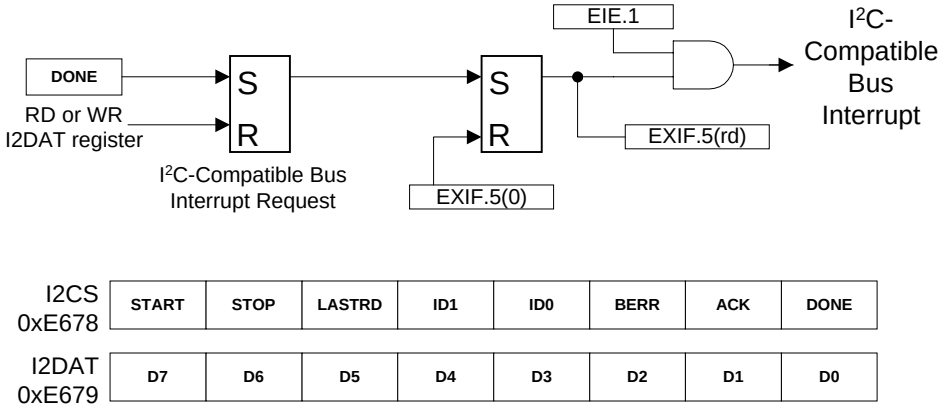


Figure 4-6. I²C-Compatible Bus Interrupt-Enable Bits and Registers

Chapter 13, "Input/Output" describes the interface to the FX2's I²C-Compatible Bus controller. The FX2 uses two registers, I2CS (Control and Status) and I2DAT (Data), to transfer data over the bus.

An I²C-Compatible Bus Interrupt is asserted whenever one of the following occurs:

- The DONE Bit (I2CS.0) makes a zero-to-one transition, signalling that the bus controller is ready for another command.
- The STOP bit (I2CS.6) makes a one-to-zero transition.

To enable the "Done" interrupt source, set EIE.1 to 1; to additionally enable the "Stop" interrupt source, set STOPIE to 1. If both interrupts are enabled, the interrupt source may be determined by checking the DONE and STOP Bits in the I2CS register.

To reset the Interrupt Request, write a zero to EXIF.5. Any firmware read or write to the I2DAT or I2CS register also automatically clears the Interrupt Request.

While the I²C-Compatible Bus controller is generating the "stop" condition, it ignores accesses to the I2CS and I2DAT registers. Firmware should therefore check the STOP Bit for zero before writing new data to I2CS or I2DAT.

4.7 FIFO/GPIF Interrupt (INT4)

Just as the USB Interrupt is shared among 27 individual USB-interrupt sources, the FIFO/GPIF interrupt is shared among 14 individual FIFO/GPIF sources.

The FIFO/GPIF Interrupt, like the USB Interrupt, can employ autovectoring. Table 4-14 shows the priority and INT4VEC values for the 14 FIFO/GPIF interrupt sources.

Table 4-14. Individual FIFO/GPIF Interrupt Sources

Priority	INT4VEC Value	Source	Notes
1	80	EP2PF	Endpoint 2 Programmable Flag
2	84	EP4PF	Endpoint 4 Programmable Flag
3	88	EP6PF	Endpoint 6 Programmable Flag
4	8C	EP8PF	Endpoint 8 Programmable Flag
5	90	EP2EF	Endpoint 2 Empty Flag
6	94	EP4EF	Endpoint 4 Empty Flag
7	98	EP6EF	Endpoint 6 Empty Flag
8	9C	EP8EF	Endpoint 8 Empty Flag
9	A0	EP2FF	Endpoint 2 Full Flag
10	A4	EP4FF	Endpoint 4 Full Flag
11	A8	EP6FF	Endpoint 6 Full Flag
12	AC	EP8FF	Endpoint 8 Full Flag
13	B0	GPIFDONE	GPIF Operation Complete (See *Chapter 10, "General Programmable Interface (GPIF)"*)
14	B4	GPIFWF	GPIF Waveform (See *Chapter 10, "General Programmable Interface (GPIF)"*)

When FIFO/GPIF interrupt sources are asserted, the FX2 prioritizes them and constructs an Autovector, which appears in the INT4VEC register; 0 is the highest priority, 14 is the lowest. If two FIFO/GPIF interrupts occur simultaneously, the prioritization affects which one is first indicated in the INT4VEC register. If Autovectoring is enabled, the INT4VEC byte replaces the contents of address 0x0055 in the FX2's program memory. This causes the FX2 to automatically vector to a different address for each FIFO/GPIF interrupt source. This mechanism is explained in detail in Section 4.8 *"FIFO/GPIF-Interrupt Autovectors"*.

Important

It is important in any FIFO/GPIF Interrupt Service Routine (ISR) to clear the main INT4 Interrupt **before** clearing the individual FIFO/GPIF interrupt request latch. This is because as soon as the individual FIFO/GPIF interrupt is cleared, any pending FIFO/GPIF interrupt will immediately try to generate another main FIFO/GPIF Interrupt. If the main INT4 IRQ bit has not been previously cleared, the pending interrupt will be lost.

The registers associated with the individual FIFO/GPIF interrupt sources are described in *Chapter 15, "Registers"* and Section 8.6, "CPU Control of FX2 Endpoints". Each interrupt source has an enable (IE) and a request (IRQ) bit. Firmware sets the IE bit to 1 to enable the interrupt. The FX2 sets an IRQ bit to 1 to request an interrupt, and the firmware clears an IRQ bit by setting it to 1.

*The main FIFO/GPIF interrupt request is cleared by **clearing** the EXIF.6 bit **to 0**; each individual FIFO/GPIF interrupt is cleared by **setting** its IRQ bit **to 1**.*

4.8 FIFO/GPIF-Interrupt Autovectors

The main FIFO/GPIF interrupt is shared by 14 interrupt sources.To save the code and processing time which normally would be required to sort out the individual FIFO/GPIF interrupt source, the FX2 provides a second level of interrupt vectoring, called *Autovectoring*. When a FIFO/GPIF interrupt is asserted, the FX2 pushes the program counter onto its stack then jumps to address 0x0053, where it expects to find a “jump” instruction to the FIFO/GPIF Interrupt service routine.

The FX2 jump instruction is encoded as follows:

Table 4-15. FX2 JUMP Instruction

Address	Op-Code	Hex Value
0x0053	LJMP	0x02
0x0054	AddrH	0xHH
0x0055	AddrL	0xLL

If Autovectoring is enabled (AV4EN=1 in the INTSETUP register), the FX2 substitutes its INT4VEC byte (see Table 4-14) for the byte at address 0x0055. Therefore, if the high byte (“page”) of a jump-table address is preloaded at location 0x0054, the automatically-inserted INT4VEC byte at 0x0055 will direct the jump to the correct address out of the 14 addresses within the page.

As shown in Table 4-16, the jump table contains a series of jump instructions, one for each individual FIFO/GPIF Interrupt source's ISR.

Table 4-16. A Typical FIFO/GPIF-Interrupt Jump Table

Table Offset	Instruction
0x80	LJMP EP2PF_ISR
0x84	LJMP EP4PF_ISR
0x88	LJMP EP6PF_ISR
0x8C	LJMP EP8PF_ISR
0x90	LJMP EP2EF_ISR
0x94	LJMP EP4EF_ISR
0x98	LJMP EP6EF_ISR
0x9C	LJMP EP8EF_ISR
0xA0	LJMP EP2FF_ISR
0xA4	LJMP EP4FF_ISR
0xA8	LJMP EP6FF_ISR
0xAC	LJMP EP8FF_ISR
0xB0	LJMP GPIFDONE_ISR
0xB4	LJMP GPIFWF_ISR

4.8.1 FIFO/GPIF Autovector Coding

To employ autovectoring for the FIFO/GPIF interrupt, perform the following steps:

1. Insert a jump instruction at 0x0053 to a table of jump instructions to the various FIFO/GPIF interrupt service routines. Make sure the jump table starts at a 0x0100-byte page boundary *plus 0x80*.
2. Code the jump table with jump instructions to each individual FIFO/GPIF interrupt service routine. This table has two important requirements, arising from the format of the INT4VEC byte (0x80-based, with the 2 LSBs set to 0); the two requirements are the following:
   - It must begin on a page boundary + 0x80 (address 0xnn80).
   - The jump instructions must be four bytes apart.
3. Place the interrupt service routines anywhere in memory.
4. Write initialization code to enable the FIFO/GPIF interrupt (INT4) and Autovectoring.

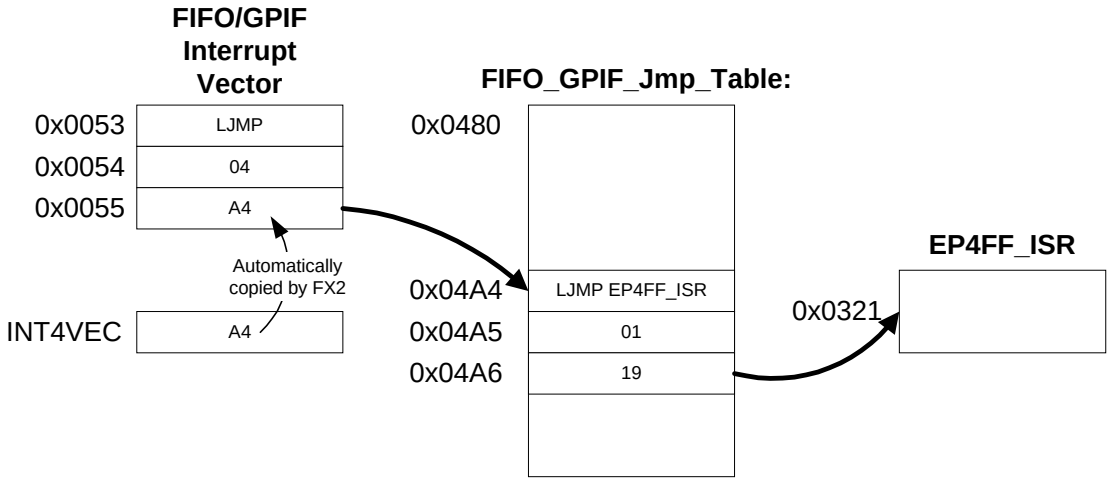


Figure 4-7. The FIFO/GPIF Autovector Mechanism in Action

Figure 4-7 illustrates an ISR that services EP4's Full Flag. When EP4 goes full, the FX2 asserts the FIFO/GPIF interrupt request, vectoring to location 0x0053.

The jump instruction at this location, which was originally coded as "LJMP 04**80**", becomes "LJMP 04**A4**" because the FX2 automatically inserts **A4**, the INT4VEC value for EP4FF (Table 4-13).

The FX2 jumps to 0x04A4, where it executes the jump instruction to the EP4FF ISR, arbitrarily located for this example at address 0x0321.

Once the FX2 vectors to 0x0053, initiation of the endpoint-specific ISR takes only eight instruction cycles.

Chapter 5 Memory

5.1 Introduction

Memory organization in the FX2 is similar, but not identical, to that of the standard 8051. There are three distinct memory areas: Internal Data Memory, External Data Memory, and External Program Memory. As will be explained below, *"External" memory is **not** necessarily external to the FX2 chip.*

5.2 Internal Data RAM

As shown in Figure 5-1, the FX2's Internal Data RAM is divided into three distinct regions: the "Lower 128", the "Upper 128", and "SFR Space". The Lower 128 and Upper 128 are general-purpose RAM; the SFR Space contains FX2 control and status registers.

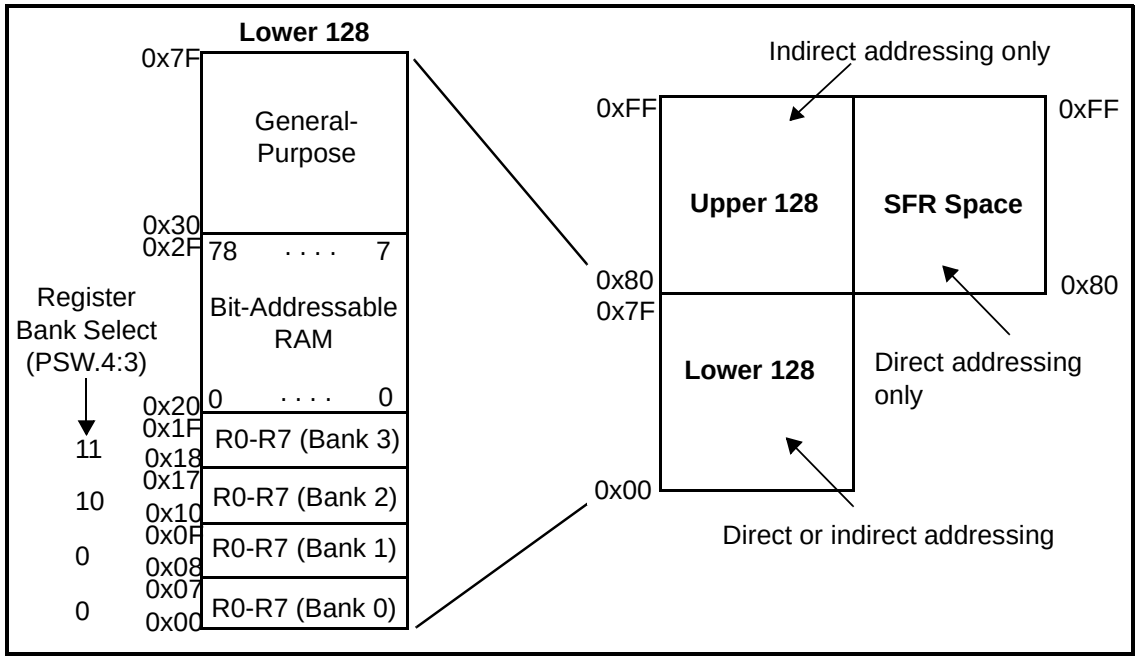


Figure 5-1. Internal Data RAM Organization

5.2.1 The Lower 128

The Lower 128 occupies Internal Data RAM locations 0x00-0x7F. All of the Lower 128 may be accessed as general-purpose RAM, using either *direct* or *indirect* addressing (for more information on the FX2 addressing modes, see *Chapter 12 "Instruction Set"*).

Two segments of the Lower 128 may additionally be accessed in other ways.

- Locations 0x00-0x1F comprise four banks of 8 registers each, numbered R0 through R7. The current bank is selected via the “register-select” bits (RS1:RS0) in the PSW special-function register; code which references registers R0-R7 will access them only in the currently-selected bank.
- Locations 0x20-0x2F are bit-addressable. Each of the 128 bits in this segment may be individually addressed, either by its bit address (0x00 to 0x7F) or by reference to the byte which contains it (0x20.0 to 0x2F.7).

5.2.2 The Upper 128

The Upper 128 occupies Internal Data RAM locations 0x80-0xFF; all 128 bytes may be accessed as general-purpose RAM, but only by using *indirect* addressing (for more information on the FX2 addressing modes, see *Chapter 12 "Instruction Set"*).

Since the FX2’s stack is internally accessed using indirect addressing, it’s a good idea to put the stack in the Upper 128; this frees the more-efficiently-accessed Lower 128 for General-Purpose use.

5.2.3 SFR (Special Function Register) Space

The SFR Space, like the Upper 128, is accessed at Internal Data RAM locations 0x80-0xFF. The FX2 keeps SFR Space separate from the Upper 128 by using different addressing modes to access the two regions: SFRs may only be accessed using *direct* addressing, and the Upper 128 may only be accessed using *indirect* addressing.

The SFR Space contains FX2 control and status registers; an overview is in Section 11.12, "Special Function Registers (SFR)", and a full description of all the SFRs is in *Chapter 15 "Registers"*.

The sixteen SFRs at locations 0x80, 0x88, ..., 0xF0, 0xF8 are bit-addressable. Each of the 128 bits in these registers may be individually addressed, either by its bit address (0x80 to 0xFF) or by reference to the byte which contains it (e.g., 0x80.0, 0xC8.7, etc.).

5.3 External Program Memory and External Data Memory

The standard 8051 employs a Harvard architecture for its External memory; the program and data memories are physically separate. The FX2 uses a modified version of this memory model; *off-chip* program and data memories are separate, but the *on-chip* program and data memories are unified in a Von Neumann architecture. This allows the FX2's on-chip RAM to be loaded from an external source (USB or EEPROM, see *Chapter 3 "Enumeration and ReNumeration™"*), then used as program memory.

Standard 8051

The standard 8051 has separate address spaces for program and data memory; it can address 64K of read-only program memory at addresses 0x0000-0xFFFF, and another 64K of read/write data memory, *also* at addresses 0x0000-0xFFFF. The standard 8051 keeps the two memory spaces separate by using different bus signals to access them; the read strobe for program memory is PSEN (Program Store Enable), and the read and write strobes for data memory are RD and WR. The 8051 generates PSEN strobes for instruction fetches and for the MOVC (move code memory into the accumulator) instruction; it generates RD and WR strobes for all data-memory accesses. In a standard 8051 application, an external 64K ROM chip (enabled by the 8051's PSEN signal) might be used for program memory and an external 64K RAM chip (enabled by the 8051's RD and WR signals) might be used for data memory.

In the standard 8051, all program memory is read-only.

FX2

The FX2 has 8K of on-chip RAM (the "Main RAM") at addresses 0x0000-0x1FFF, and 512 bytes of on-chip RAM (the "Scratch RAM") at addresses 0xE000-0xE1FFF. Although this RAM is physically located inside the chip, it's addressed by FX2 firmware as *External* memory, just as though it were in an external RAM chip.

Some systems use only this on-chip RAM, with no off-chip memory. In those systems, the RD and PSEN strobes are automatically combined for accesses to addresses below 0x2000, so the Main RAM is accessible as *both* data and program memory. The RD and PSEN strobes are *not* combined for the Scratch RAM; Scratch RAM is accessible as data memory only.

Although it's technically accurate to say that the Main RAM *data* memory is writable while the Main RAM *program* memory is not, it's a distinction without a difference. The Main RAM is accessible both as program memory and data memory, so writing to Main RAM data memory is equivalent to writing to Main RAM program memory at the same address.

The Scratch RAM is never accessible as program memory.

The FX2 also reserves 7.5K (0xE200-0xFFFF) of the data-memory address space for control/status registers and endpoint buffers (see *Section 5.6, "On-Chip Data Memory at 0xE000-0xFFFF"*).

Note that *only* the data-memory space is reserved; *program* memory in the 0xE000-0xFFFF range is not reserved, so the 128-pin FX2 can access off-chip program memory in that range.

5.3.1 56- and 100-pin FX2

The 56- and 100-pin FX2 chips have no facility for adding off-chip program or data memory. Therefore, the Main RAM must serve as both program and data memory. To accomplish this, the FX2 reads the Main RAM using the logical OR of the PSEN and RD strobes. It is the responsibility of the system designer to ensure that the program- and data-memory spaces do not overlap; with most C compilers, this is done by using linker directives that place the code and data modules into separate areas.

5.3.2 128-pin FX2

It is possible to add off-chip program and data memory to the 128-pin FX2; the organization of that memory depends on the state of the EA (External Access) pin.

EA = 0

The Main RAM is accessible both as program and data memory, just as in the 56- and 100-pin FX2.

To avoid conflict with the Main RAM, the pins which control access to off-chip memory (the $\overline{RD}$, $\overline{WR}$, $\overline{CS}$, $\overline{OE}$, and $\overline{PSEN}$ pins) are inactive whenever the FX2 accesses addresses 0x0000-0x1FFF. This allows a 64K memory chip (data and/or program) to be added without requiring additional external logic to inhibit access to the lower 8K of that chip. Note that the PSEN and RD signals are available on separate pins, so the program and data spaces *outside* the FX2 are not combined as they are *inside* the FX2.

When code in the range 0x0000-0x1FFF is fetched from the on-chip RAM, the $\overline{PSEN}$ pin is not asserted; when code is fetched from program memory in the range 0x2000-0xFFFF, the $\overline{PSEN}$ pin *is* asserted.

EA = 1

All program memory is off-chip; all on-chip RAM, including the Main RAM, is data memory only.

The FX2 reads all on-chip RAM using only the RD strobe; the combining of RD and PSEN is disabled, so the on-chip RAM becomes data memory only. All program memory is off-chip; accesses to the lower 8K of off-chip program memory are not inhibited.

Any code fetch will assert the $\overline{PSEN}$ pin.

After a power-on-reset, the FX2 immediately begins executing code at address 0x0000 in the off-chip program memory, rather than waiting for an EEPROM load or USB code download to complete (see *Chapter 7 "Resets"* for a full description of the FX2 resets).

5.4 FX2 Memory Maps

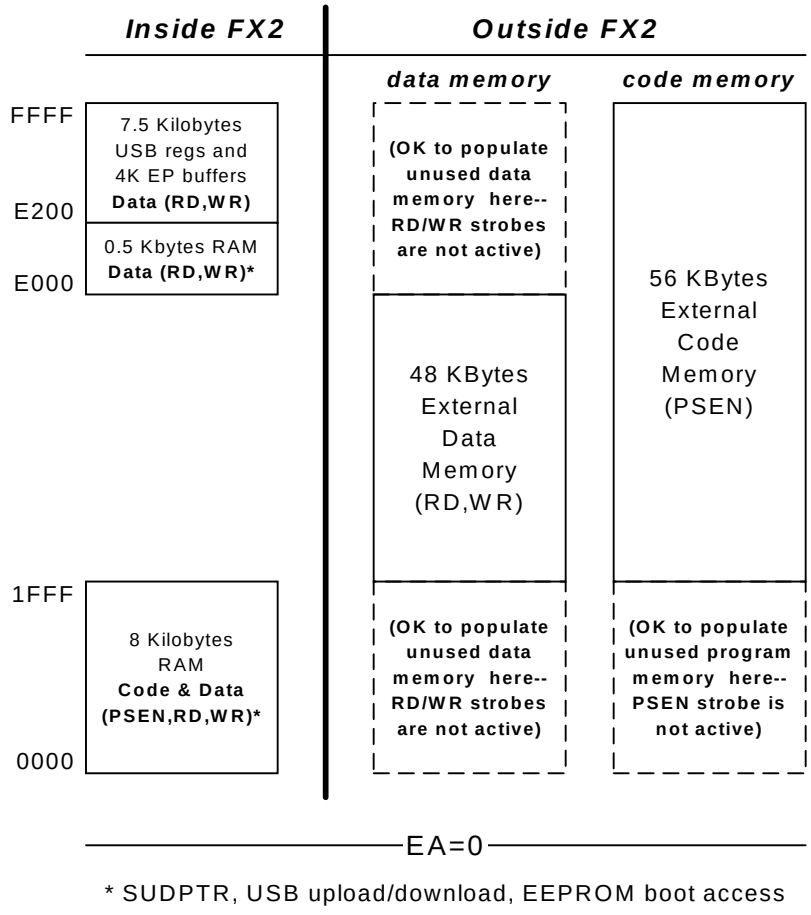


Figure 5-2. FX2 External Program/Data Memory Map, EA=0

Figure 5-2 illustrates the memory map of the 128-pin FX2 with off-chip program and data memory.

The 56- and 100-pin FX2 chips cannot access off-chip memory; the entire memory map for those chips is illustrated on the left side of Figur e5-2, in the “Inside FX2” column.

On-chip FX2 memory consists of three RAM regions:

- 0x0000-0x1FFF (Main RAM)
- 0xE000-0xE1FF (Scratch RAM)
- 0xE200-0xFFFF (Registers/Buffers)

The 8K “Main RAM” occupies code-memory (PSEN) and data-memory (RD/WR) addresses 0x0000-0x1FFF.

The 512-byte “Scratch RAM” occupies data-memory (RD/WR) addresses 0xE000-0xE1FF.

7.5K of control/status registers and endpoint buffers occupy data-memory (RD/WR) addresses 0xE200-0xFFFF.

When off-chip memory is connected to the FX2, it fills in the gaps not occupied by on-chip FX2 RAM. Since the lower 8K of memory is occupied by on-chip program/data memory and the upper 8K is occupied by on-chip data memory, the off-chip memory cannot be active in these regions. Nevertheless, it’s still safe to *populate* those regions with off-chip memory, as the following paragraphs explain.

The middle column of Figure 5-2 indicates FX2 data memory (activated by the RD and WR strobes) and the right-most column indicates FX2 code memory (activated by PSEN).

The “middle” 48K of the data-memory space may be filled with off-chip memory, since it does not conflict with the upper and lower 8K of on-chip FX2 data memory. To allow a 64K RAM to be connected to the FX2, the FX2 gates its RD and WR strobes to exclude the top and bottom 8K for off-chip accesses. Therefore, a 64K RAM can be connected to FX2, and the top and bottom 8K of it are automatically disabled.

Likewise, when a 64K *code* memory (PSEN strobe) is attached to the FX2 (when EA = 0), the lower 8K is automatically excluded for off-chip code fetches, avoiding conflict with the on-chip code memory inside FX2.

The asterisks in Figures 5-2 and 5-3 indicate memory regions that may be accessed using three special FX2 resources:

- Setup Data Pointer (see Section 8.7)
- Upload or download via USB (see Section 3.8)
- Code boot from an I²C-compatible EEPROM (see Section 13.5 and Section 3.4)

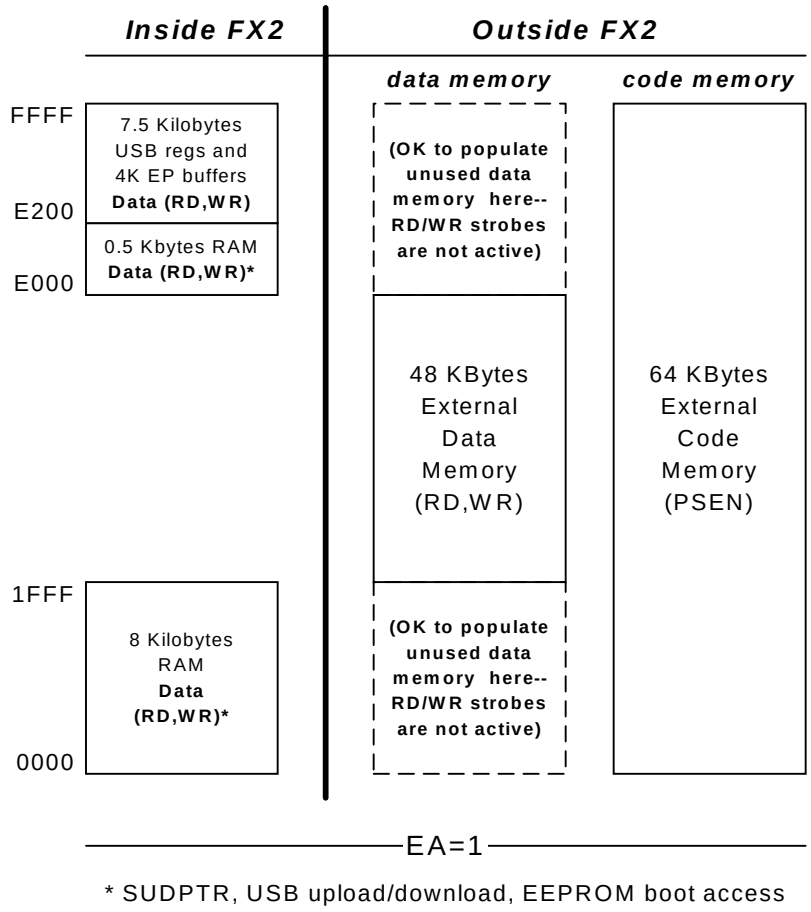


Figure 5-3. FX2 External Program/Data Memory Map, EA=1

Figure 5-3 illustrates the 128-pin FX2 memory map when the EA pin is tied high. *The only difference from Figure 5-2 is that the Main RAM is data memory only, instead of combined code/data memory.* This allows an off-chip code memory to contain all of the FX2 firmware. In this configuration, the FX2 can begin executing code from off-chip memory immediately after power-on-reset.

FX2 code execution begins at address 0x0000, where the reset vector is located.

Off-chip *data* memory is partially disabled just as in Figure 5-2, ensuring that off-chip data memory does not conflict with on-chip data RAM.

*Be careful to check the access time of external Flash or other code memory in this mode. The FX2 can stretch its RD and WR strobes to compensate for slow **data** memories, but it does not have the capability to stretch its PSEN signal to allow for slow **code** memories. At 48 MHz, an external code-memory chip must have an access time of approximately 44 ns or shorter (access-time parameters are given in the CY7C68013 data sheet).*

5.5 "Von-Neumannizing" Off-Chip Program and Data Memory

The 128-pin FX2 package provides a 16-bit address bus, an 8-bit data bus, and memory control signals PSEN, RD, and WR. These signals are used to expand the FX2's External Program and/or External Data memory.

As described in the previous section, the FX2 gates the $\overline{RD}$ and $\overline{WR}$ signals to exclude selection of off-chip data memory in the range occupied by the on-chip memory. The PSEN signal is also available on a pin for connection to off-chip code memory.

In some systems, it may be desirable to combine off-chip program and data memory, just as the FX2 combines its on-chip program/data Main RAM. These systems must logically OR the PSEN and RD strobes to qualify the off-chip memory's chip enable and output enable signals. To save the external logic which would normally be needed, FX2 provides two additional control signals, $\overline{CS}$ and $\overline{OE}$. The equations for these active-low signals are:

$$\overline{CS} = \overline{RD + WR + PSEN}$$

$$\overline{OE} = \overline{RD + PSEN}$$

Because the RD, WR, and PSEN signals are already qualified by the addresses allocated to off-chip memory, the added strobes $\overline{CS}$ and $\overline{OE}$ strobes are active only when the FX2 accesses off-chip memory.

5.6 On-Chip Data Memory at 0xE000-0xFFFF

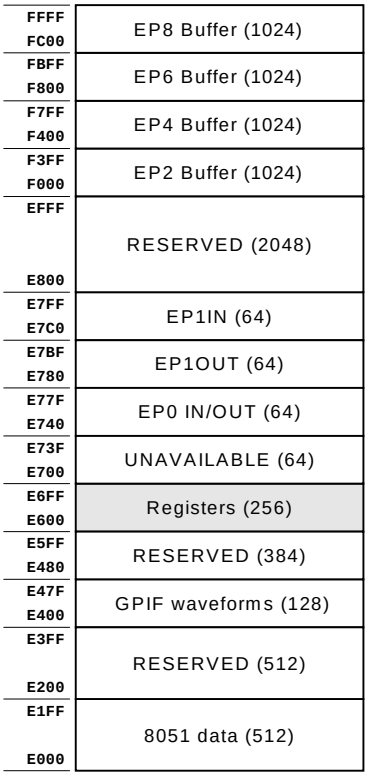


Figure 5-4. On-Chip Data Memory at 0xE000-0xFFFF

Figure 5-4 shows the memory map for on-chip data RAM at 0xE000-0xFFFF.

512 bytes of Scratch RAM is available at 0xE000-0xE1FF. This is data RAM only; code cannot be run from it. The 128 bytes at 0xE400-0xE47F hold the four waveform descriptors for the GPIF, described in Chapter 10. The shaded area from 0xE600-0xE6FF contains FX2 control and status registers.

Memory blocks 0xE200-0xE3FF, 0xE480-0xE5FF, 0xE700-0xE73F, and 0xE800-0xEFFF) are reserved; they must not be used for data storage.

The remaining RAM contains the endpoint buffers. These buffers are accessible either as addressable data RAM (via the 'MOVX' instruction) or as a FIFO (via the Autopointer, described in Section 8.8).

Chapter 6 Power Management

6.1 Introduction

The USB host can *suspend* a device to put it into a power-down mode. When the USB signals a SUSPEND operation, the FX2 goes through a sequence of steps to allow the firmware first to turn off external power-consuming subsystems, and then to enter a low-power mode by turning off the FX2's oscillator. Once suspended, the FX2 is awakened either by resumption of USB bus activity or by assertion of one of its two WAKEUP pins (provided that they're enabled). This chapter describes the suspend-resume mechanism.

It is important to understand the distinction between 'suspend', 'resume', 'idle', and 'wakeup'.

- **SUSPEND** is a request—indicated by a 3-millisecond "J" state on the USB bus—from the USB host/hub to the device. This request is usually sent by the host when *it* enters a low-power "suspended" state. USB devices are required to enter a low power state in response to this request.

  The FX2 also provides a register called SUSPEND; writing any value to it will allow the FX2 to enter the suspended state even when a SUSPEND condition doesn't exist on the bus.

- **RESUME** is a signal from the device to the host, requesting that the host be taken out of its low-power "suspended" mode. RESUME can be signaled only by a USB device that has reported (via its Configuration Descriptor) that it supports this "remote wakeup" feature, and only if the host has enabled remote wakeup from that device.

- **Idle** is an FX2 low-power state. FX2 firmware initiates this mode by setting bit 0 of the PCON (Power Control) register.To meet the stringent USB suspend current specification, the FX2's oscillator must be stopped; after the PCON.0 bit is set, the oscillator will stop if a) a SUSPEND condition exists on the bus or the SUSPEND register has been written to, and b) the two WAKEUP pins are either disabled or false. The FX2 exits the **Idle** state when it receives a Wakeup Interrupt.

- **Wakeup** is the mechanism which restarts the FX2 oscillator and asserts an interrupt to force the FX2 to exit the Idle state and resume code execution. The FX2 recognizes three wakeup sources: one from the USB itself (when bus activity resumes) and two from device pins (WAKEUP and WU2).

The FX2 enters and exits its **Idle** state independent of USB activity; in other words, the FX2 can enter the **Idle** state at any time, even when not connected to USB. The **Idle** state is “hooked into” the USB SUSPEND-RESUME mechanism using interrupts. An interrupt is automatically generated when the USB goes inactive for 3 milliseconds; FX2 firmware may respond to that interrupt by entering the **Idle** state to reduce power. If the FX2 is in the **Idle** state, a Wakeup Interrupt is generated when one of the three Wakeup sources is asserted; the FX2 responds to that interrupt by exiting the **Idle** state and resuming code execution.

Once the FX2 is awake, its firmware may send a USB RESUME request by setting the SIGRSUME bit in the USBCS register (at 0xE680). Before sending the RESUME request, the device must have: a) reported remote-wakeup capability in its Configuration Descriptor, and b) been given permission (via a *Set Feature-Remote Wakeup* request from the host) to use that remote-wakeup capability. To be compliant with the USB Specification, firmware should wait 5 milliseconds after the wakeup interrupt, set the SIGRSUME bit, wait 10-15 milliseconds, then clear it.

Figure 6-1 illustrates the FX2 logic that implements USB suspend and resume. These operations are explained in the next sections.

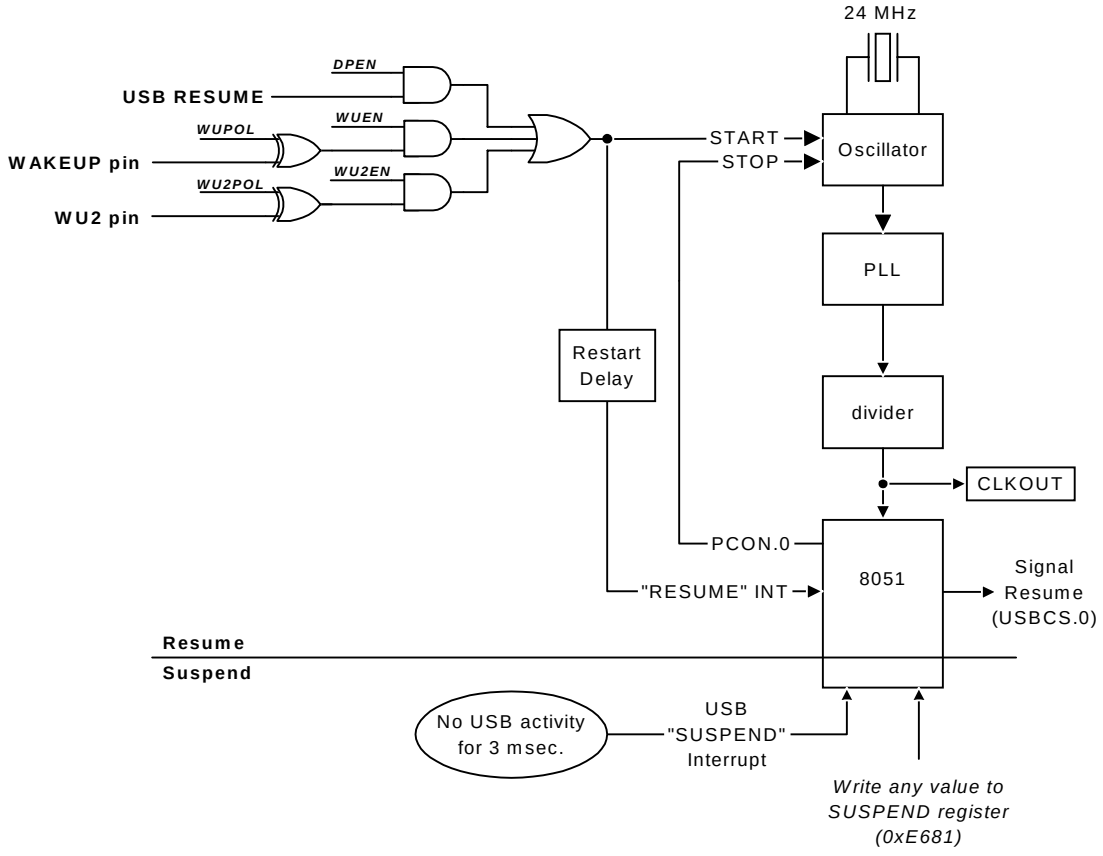


Figure 6-1. Suspend-Resume Control

6.2 USB Suspend

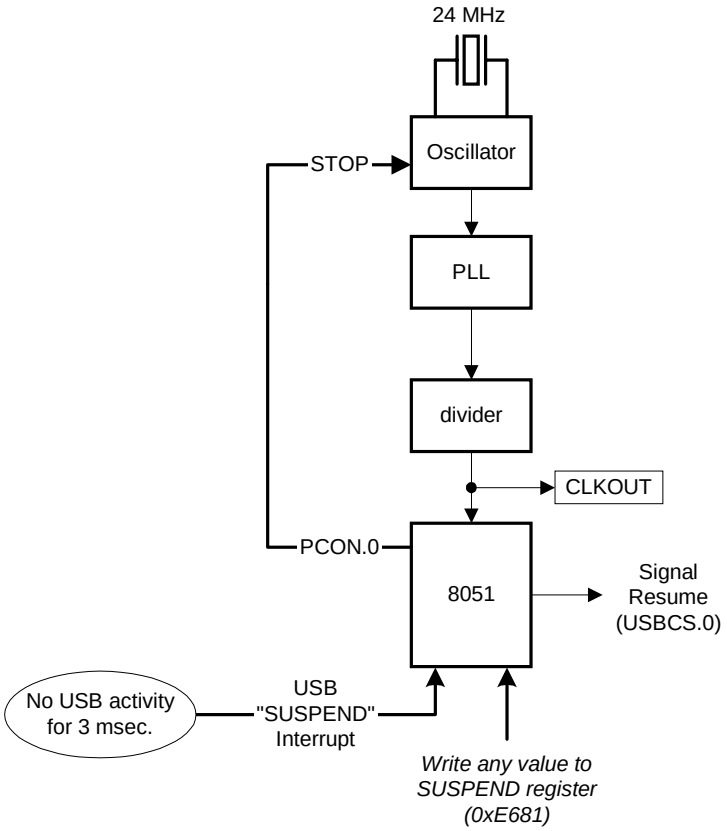


Figure 6-2. USB Suspend sequence

A USB device recognizes a SUSPEND request as three milliseconds of the bus-idle (“J”) state. When the FX2 detects this condition, it asserts the USB interrupt (INT2) and the SUSPEND interrupt autovector (vector #3).

If the CPU is in reset when a SUSPEND condition is detected on the bus, the FX2 will automatically turn off its oscillators (and keep the CPU in reset) until an enabled wakeup source is asserted.

*The bus-idle (“J”) state is **not** equivalent to the disconnected-from-USB state; the “J” state means that the voltage on D+ is higher than that on D-.*

FX2 firmware responds to the SUSPEND interrupt by taking the following actions:

1. Perform any necessary housekeeping such as shutting off external power-consuming devices.
2. Set bit 0 of the PCON register.

These actions put the FX2 into a low power ‘suspend’ state, as required by the USB Specification.

6.2.1 SUSPEND Register

FX2 firmware can force the chip into its low-power mode at any time, even without detecting a 3-millisecond “J” state on the USB bus. This “unconditional suspend” functionality is useful in applications which require the FX2 to enter its low-power mode even while disconnected from the USB bus.

To force the FX2 unconditionally to enter its low-power mode, firmware simply writes any value to the SUSPEND register (at 0xE681) before setting the PCON.0 bit.

6.3 Wakeup/Resume

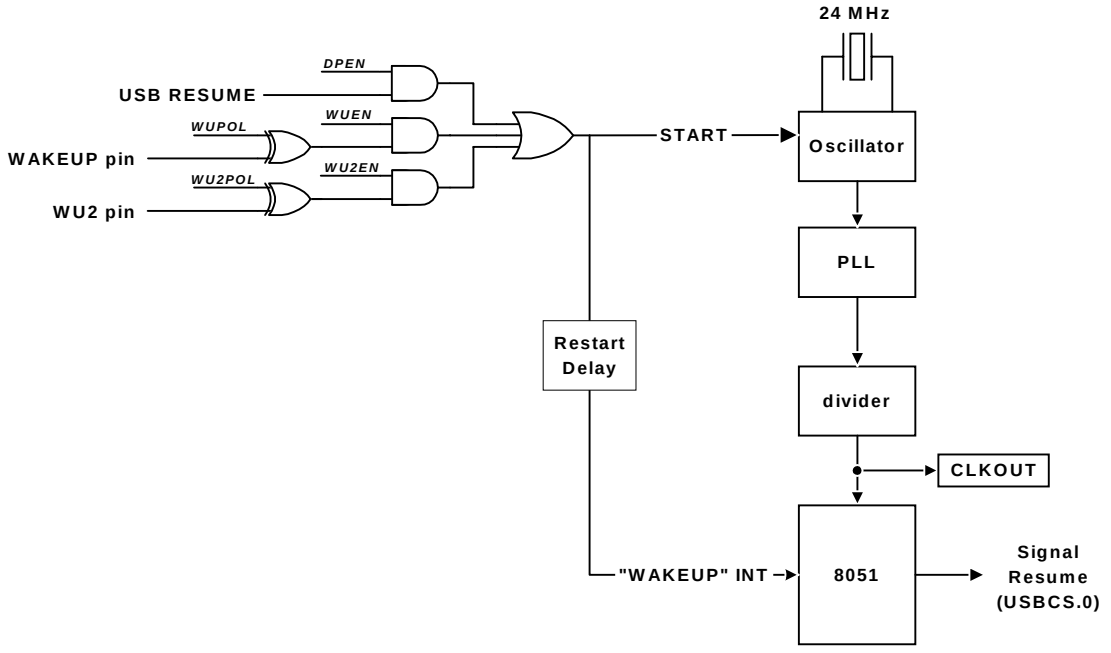


Figure 6-3. FX2 Wakeup/Resume sequence

Once in the low-power mode, there are three ways to wake up the FX2:

- USB activity on the FX2's DPLUS pin
- Assertion of the WAKEUP pin
- Assertion of the WU2 ("Wakeup 2") pin

These three wakeup sources may be individually enabled by setting the DPEN, WUEN, and WU2EN bits in the Wakeup Control register.

WAKEUPCS **Wakeup Control & Status** **E682**

b7	b6	b5	b4	b3	b2	b1	b0
WU2	**WU**	**WU2POL**	**WUPOL**	**0**	**DPEN**	**WU2EN**	**WUEN**
R/W	R/W	R/W	R/W	R	R/W	R/W	R/W
0	0	0	0	0	1	0	1

The polarities of the wakeup pins are set using the WUPOL and WU2POL bits; 0 is active low and 1 is active high.

Three bits in the WAKEUP register enable the three wakeup sources. DPEN stands for "DPLUS Enable" (DPLUS is one of the USB data lines; the other is DMINUS).

WUEN (Wakeup Enable) enables the WAKEUP pin, and WU2EN (Wakeup 2 Enable) enables the WU2 pin.

When the FX2 chip detects activity on DPLUS while DPEN is true, or a false-to-true transition on WAKEUP or WU2 while WUEN or WU2EN is true, it asserts the "wakeup" interrupt.

The status bits WU and WU2 indicate which of the wakeup pins caused the wakeup event. Asserting the wakeup pin (according to its programmed polarity) sets the corresponding bit. If the wakeup was caused by resumption of USB DPLUS activity, neither of these bits is set, leading to the conclusion that the third source, a USB bus reset, caused the wakeup event. FX2 firmware clears the WU and WU2 flags by writing "1" to them.

6.3.1 Wakeup Interrupt

When a wakeup event occurs, the FX2 restarts its oscillator and, after the PLL stabilizes, it generates an interrupt request. This applies whether or not the FX2 is connected to the USB. The Wakeup Interrupt is a dedicated interrupt, and is not shared by USBINT like most of the other individual USB interrupts.

The Wakeup Interrupt vector is at 0x33, and has the highest interrupt priority. It is enabled by EICON.5, and its IRQ flag is at EICON.4 (EICON is SFR 0xD8).

The Wakeup Interrupt Service Routine clears the interrupt request flag (using the 'bit clear' instruction, i.e. 'clr EICON.4'), and then executes a 'reti' (return from interrupt) instruction. This causes the FX2 to continue program execution at the instruction following the one that set PCON.0 to initiate the power-down operation.

> ***About the Wakeup Interrupt***
>
> The FX2 enters its idle state when it sets PCON.0 to 1. Although a standard 8051 exits the idle state when *any* interrupt occurs, the FX2 supports only the Wakeup Interrupt to exit the idle state.

If PCON.0 is set when no Suspend condition exists (i.e., the USB is not signaling "Suspend", and firmware hasn't written to the SUSPEND register), the Wakeup Interrupt will fire immediately.

6.4 USB Resume (Remote Wakeup)

USBCS **USB Control and Status** **7FD6**

b7	b6	b5	b4	b3	b2	b1	b0
HSM	-	-	-	DISCON	NOSYNSOF	RENUM	SIGRSUME

Figure 6-4. USB Control and Status register

Firmware sets the SIGRSUME bit to send a remote-wakeup request to the host. To be compliant with the USB Specification, the firmware should wait 5 milliseconds after the wakeup interrupt, set the SIGRSUME bit, wait 10-15 milliseconds, then clear it.

Holding either WAKEUP pin in its active state (as determined by the programmed polarity) inhibits the FX2 chip from turning off its oscillator in order to enter the 'suspend' state.

The Default USB Device does not support remote wakeup. This fact is reported at enumeration time in byte 7 of the built-in Configuration Descriptor (see Appendices A and B).

6.4.1 WU2 Pin

The WU2 function shares the general-purpose I/O pin PA3. Unlike other multi-purpose I/O pins that use configuration registers (PORTACFG, PORTBCFG and PORTCCFG) to select alternate

functions, the PA3 and WU2 functions are *simultaneously* active. However, the WU2 function has no effect unless enabled (by setting the WU2EN bit to 1). If WU2 is used as a wakeup pin, make sure to set PA3 as an input (OEA.3=0, the default state) to prevent PA3 from also driving the pin.

The dual nature of the PA3/WU2 pin allows the FX2 to enter the low-power mode, then periodically awaken itself. This is done by connecting an RC network to the PA3/WU2 pin; if the WU2 pin is set to the default polarity (active-high), the resistor is connected to 3.3V and the capacitor is connected to ground.

The firmware then performs the following steps:

1. Set W2POL to 1 for active-high polarity on the WU2 pin.
2. Set WU2EN to 1 to enable Wakeup 2.
3. Enable the wakeup interrupt by setting EICON.5=1.
4. Set PA3 to 0, then set OEA.3 to 1. This enables the PA3 output and drives the PA3/WU2 pin to ground, discharging the capacitor.
5. Set OEA.3 to 0. This floats the PA3/WU2 pin, allowing the resistor to begin charging the capacitor.
6. Write any value to the SUSPEND register, so the FX2 will unconditionally stop the oscillator when the firmware sets PCON.0.
7. Set PCON.0 to 1. This commands the FX2 to enter the **Idle** state.

After the capacitor charges to a logic high level, the wakeup interrupt triggers via the WU2 pin.

8. In the Wakeup interrupt service routine, clear EICON.4 (the wakeup interrupt request flag), then execute a ‘reti’ instruction. This resumes program execution at the instruction following the instruction in step 7.
9. At this point, the firmware can check for any tasks to perform; if none are required, it can then re-enter the **Idle** state starting at step 4.

By selecting a long time constant for the RC network attached to the WU2 pin, the FX2 chip can operate at extremely low average power, since the on/off (active/suspend) duty-cycle is very short.

Chapter 7 Resets

7.1 Introduction

The FX2 chip has two internal resets:

- *Power-On Reset (POR)*, controlled by the $\overline{\text{RESET}}$ pin, which puts the FX2 in a known state.
- *CPU Reset*, controlled by the FX2's USB Core logic. The CPU Reset is always asserted (i.e., the CPU is always held in reset) while the FX2's $\overline{\text{RESET}}$ pin is asserted.

Additionally, the USB Specification defines a *USB Bus Reset*, which is a condition on the bus initiated by the USB host in order to put every device's USB functions in a known state.

This chapter describes the effects of these three resets.

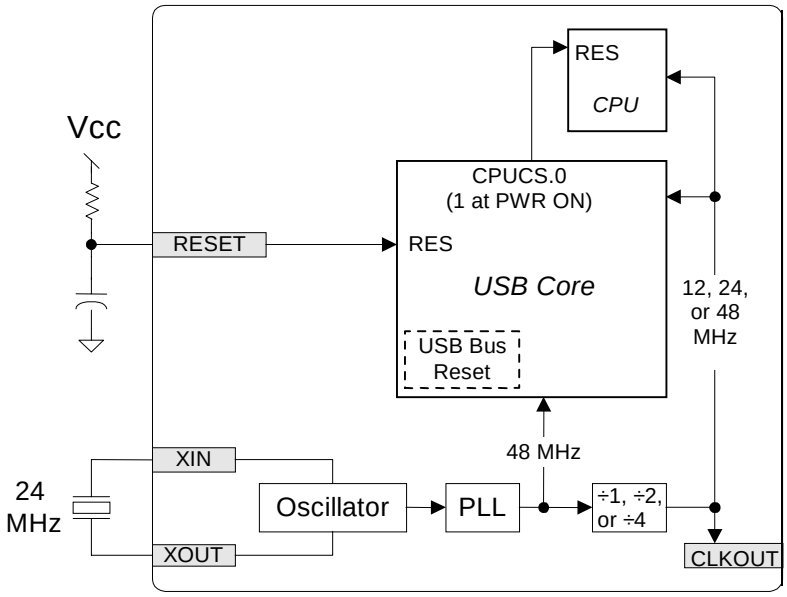


Figure 7-1. EZ-USB FX2 Resets

7.2 Power-On Reset (POR)

An active-low input pin (RESET) resets the FX2 chip. **Note that the term "Power-On Reset" refers to a reset initiated *either* by application of power *or* by assertion of the RESET pin.**

The RESET pin is normally connected to an external R-C network in order to ensure that, when power is first applied, the FX2 is held in reset until the operating parameters (Vcc voltage, crystal frequency, PLL frequency, etc.) stabilize. The recommended values for the R-C network are a 10K resistor to Vcc and a 1 µF capacitor to GND (see Figure 7-1). External logic can force a POR at any time by pulling the RESET pin low.

Whenever the RESET pin is asserted, the USB Core holds the CPU in reset.

The CLKOUT pin, crystal oscillator, and PLL are active as soon as power is applied. Once the CPU is out of reset, firmware may clear a control bit (CLKOE, CPUCS.1) to inhibit the CLKOUT output pin for EMI-sensitive applications that do not need this signal.

The CLKOUT signal is active while RESET is low. When RESET returns high, the activity on the CLKOUT pin depends on whether or not the FX2 is in the low-power "suspend" state; if it is, CLKOUT stops. Resumption of USB bus activity or assertion of the WAKEUP or WU2 pin (if enabled) restarts the CLKOUT signal.

The oscillator and PLL are unaffected by the state of the RESET pin.

Power-on default values for all FX2 register bits are shown in *Chapter 15, "Registers"*. At power-on reset:

- Endpoint data buffers and byte counts are uninitialized.
- The CPU clock speed is set to 12 MHz, the CPU is held in reset, and the CLKOUT pin is active.
- All port pins are configured as general-purpose input pins.
- USB interrupts are disabled and USB interrupt requests are cleared.
- Bulk IN and OUT endpoints are unarmed, and their stall bits are cleared. The FX2 will NAK IN and OUT tokens while the CPU is reset.
- Endpoint toggle bits are cleared to 0.
- The RENUM bit is cleared to 0. This means that the Default USB Device, not the firmware, will respond to USB device requests.
- The USB Function Address register is cleared to zero.
- The endpoints are configured for the Default USB Device.
- Interrupt autovectoring is turned off.
- Configuration Zero, Alternate Setting Zero is in effect.

7.3 Releasing the CPU Reset

Register bit CPUCS.0 resets the CPU. This bit is set to 1 at power-on, initially holding the CPU in reset. There are three ways that the CPUCS.0 bit can be cleared to 0, releasing the CPU from reset:

- By the host, as the final step of a RAM download.
- Automatically, at the end of an EEPROM load (assuming the EEPROM is correctly programmed).
- Automatically, when external ROM is used (EA=1) and no “C0” or “C2” EEPROM is present.

FX2 firmware cannot put the CPU into reset by setting CPUCS.0 to 1; to the firmware, that bit is read-only.

7.3.1 RAM Download

Once enumerated, the host can download code into the FX2 RAM using the “Firmware Load” vendor request (*Chapter 2, "Endpoint Zero"*). The last packet loaded writes 0x00 to the CPUCS register, which releases the CPU from reset.

7.3.2 EEPROM Load

Chapter 3, "Enumeration and ReNumeration™" describes the EEPROM boot loads in detail. At power-on, the FX2 checks for the presence of an EEPROM on its I²C-compatible bus. If found, it reads the first EEPROM byte. If it reads 0xC2 as the first byte, the FX2 downloads firmware from the EEPROM into internal RAM. The last operation in a “C2” Load writes 0x00 to the CPUCS register, which releases the CPU from reset.

After a “C2” Load, the FX2 sets the RENUM bit to 1, so the firmware will be responsible for responding to USB device requests.

7.3.3 External ROM

The 128-pin FX2 can use off-chip program memory containing FX2 code and USB device descriptors, which include the VID/DID/PID bytes. Because such a system does not require an I²C-compatible EEPROM to supply the VID/DID/PID, the FX2 automatically releases the CPU from reset when:

- The EA pin is pulled high (indicating off-chip code memory), *and*

- No “C0/C2” EEPROM is detected on the I²C-compatible bus.

Under these conditions, the FX2 also sets the RENUM bit to 1, so the firmware will be responsible for responding to USB device requests.

7.4 CPU Reset Effects

The USB host may reset the CPU at any time by downloading the value 0x01 to the CPUCS register. The host might do this, for example, in preparation for loading code overlays, effectively magnifying the size of the internal FX2 RAM. For such applications, it is important to know the state of the FX2 chip during and after a CPU reset. In this section, this particular reset is called a “CPU Reset,” and should not be confused with the POR described in Section 7.2, *"Power-On Reset (POR)."* This discussion applies only to the condition in which the FX2 chip is powered, and the CPU is reset by the host setting the CPUCS.0 bit to 1.

The basic USB device configuration remains intact through a CPU reset. Endpoints keep their configuration, the USB Function Address remains the same, and the I/O ports retain their configurations and values. Stalled endpoints remain stalled, data toggles don’t change, and the RENUM bit is unaffected. The only effects of a CPU reset are as follows:

- USB (INT2) interrupts are disabled, but pending interrupt requests remain pending.
- When the CPU comes out of reset, pending interrupts are kept pending, but disabled. This gives the firmware writer the choice of acting on pre-reset USB events, or ignoring them by clearing the pending interrupt(s) before enabling INT2.
- The breakpoint condition (BREAKPT.3) is cleared.
- While the CPU is in reset, the FX2 will enter the Suspend state automatically if a “suspend” condition is detected on the bus.

7.5 USB Bus Reset

The host signals a USB Bus Reset by driving an SE0 state (both D+ and D- data lines low) for a minimum of 10 ms. The FX2 senses this condition, requests the USB Interrupt (INT2), and supplies the interrupt vector for a USB Reset. After a USB bus reset, the following occurs:

- Toggle bits are cleared to 0.
- The device address is reset to zero.
- If the Default USB Device is active, the USB configuration and alternate settings are reset to zero.
- The FX2 will renegotiate with the host for high-speed (480 Mbps) mode.

Note that the RENUM bit is unchanged after a USB bus reset. Therefore, if a device has ReNumerated™ and loaded a new personality, it retains the new personality through a USB bus reset.

7.6 FX2 Disconnect

Although not strictly a “reset,” the disconnect-reconnect sequence used for ReNumeration™ affects the FX2 in ways similar to the other resets. When the FX2 simulates a disconnect-reconnect, the following occurs:

- Endpoint STALL bits are cleared.
- Data toggles are reset to 0.
- The Function Address is reset to zero.
- If the Default USB Device is active, the USB configuration and alternate settings are reset to zero.

7.7 Reset Summary

Table 7-1. Effects of Various Resets on FX2 Resources (“—” means “no change”)

	RESET Pin	CPU Reset	USB Bus Reset	Disconnect
CPU Reset	Reset	n/a	—	—
IN Endpoints	Unarm	—	—	—
OUT Endpoints	Unarm	—	—	—
Breakpoint	0	0	—	—
Stall Bits	0	—	—	0
Interrupt Enables	0	0	—	—
Interrupt Requests	0	—	—	—
CLKOUT	Active	—	—	—
CPU Clock Speed	12 MHz	—	—	—
Data Toggles	0	—	0	0
Function Address	0	—	0	0
Default USB Device Configuration	0	—	0	0
Default USB Device Alternate Setting	0	—	0	0
RENUM Bit	0	—	—	—

Chapter 8 Access to Endpoint Buffers

8.1 Introduction

USB data enters and exits FX2 via endpoint buffers. In order to keep up with the high-speed 480 megabit/second transfer rates, *external logic usually reads and writes this data by direct connection to the endpoint FIFOs without any participation by the FX2's CPU*.

Chapter 9, "Slave FIFOs" and Chapter 10, "General Programmable Interface (GPIF)" give details about how external logic directly connects to the large endpoint FIFOs.

When an application requires the CPU to process the data as it flows between external logic and the USB — or when there *is* no external logic — firmware can access the endpoint buffers either as blocks of RAM or (using a special auto-incrementing pointer) as a FIFO.

Even when external logic or the built-in General Programmable Interface (GPIF) is handling high-bandwidth data transfers through the four large endpoint FIFOs without any CPU intervention, the firmware has certain responsibilities:

- Configure the endpoints.
- Respond to host requests on CONTROL endpoint zero.
- Control and monitor GPIF activity.
- Handle all application-specific tasks using its USARTs, counter-timers, interrupts, I/O pins, etc.

8.2 FX2 Large and Small Endpoints

FX2 endpoint buffers are divided into "small" and "large" groups. EP0 and EP1 are small, 64-byte endpoints which are accessible only by the CPU; they can't be connected directly to external logic.

EP2, EP4, EP6 and EP8 are large, configurable endpoints designed to meet the high-bandwidth requirements of USB 2.0. Although data normally flows through the large endpoint buffers under

control of the FIFO interfaces described in Chapters 9 and 10, the CPU can access the large endpoints if necessary.

8.3 High-Speed and Full-Speed Differences

FX2 operates at both full speed (12 Mbps) and high speed (480 Mbps). The data-payload-size and transfer-speed requirements differ between the two modes. FX2 architecture is optimized for high speed transfers:

- Instead of many small endpoint buffers, FX2 provides a reduced number of large buffers.
- FX2 provides double, triple or quad buffering on its large endpoints (EP2, 4, 6, and 8).
- *The CPU need not participate in high-bandwidth transfers.* Instead, dedicated FX2 logic and unified endpoint/interface FIFOs move data on and off the chip at USB 2.0 rates without any CPU intervention.

FX2 endpoint buffers appear to have different sizes depending on whether the FX2 is operating at full or high speed. This is due to the difference in maximum packet sizes allowed by the USB specification for the two modes, as illustrated by Table 8-1.

Table 8-1. Maximum Packet Sizes for USB 1.1 and 2.0

Transfer Type	Max Packet Size	
	USB 1.1	**USB 2.0**
CONTROL (EP0 only)	8,16,32,64	64
BULK	8,16,32,64	512
INTERRUPT	1-64	1-1024
ISOCHRONOUS	1-1023	1-1024

Although the EP2, EP4, EP6 and EP8 buffers are physically large, they appear as smaller buffers when the FX2 is operating at full speed to account for the smaller maximum packet sizes.

When operating at high speed, firmware can configure the large endpoints' size, type, and buffering; when operating at full speed, type and buffering are configurable but the maximum packet size is always fixed at 64 bytes for the non-isochronous types.

8.4 How the CPU Configures the Endpoints

Endpoints are configured via the six registers shown in Table 8-2.

Table 8-2. Endpoint Configuration Registers

Address	Name	Configurable Parameters
0xE610	**EP1OUTCFG**	valid, type[1] (always OUT, 64 bytes, single-buffered)
0xE611	**EP1INCFG**	valid, type[1] (always IN, 64 bytes, single-buffered)
0xE612	**EP2CFG**	valid, direction, type, size, buffering
0xE613	**EP4CFG**	valid, direction, type (always 512 double-buffered)
0xE614	**EP6CFG**	valid, direction, type, size, buffering
0xE615	**EP8CFG**	valid, direction, type (always 512 double-buffered)
Note 1: For EP1, “type” may be set to Interrupt or Bulk only.		

Chapter 15 gives full bit-level details for all registers.

Endpoint 0 does not require a configuration register since it is fixed as valid, IN/OUT, CONTROL, 64 bytes, single-buffered. EP0 uses a single 64-byte buffer both for IN and OUT transfers. EP1 uses separate 64 byte buffers for IN and OUT transfers.

Endpoints 2, 4, 6 and 8 handle the high bandwidth USB 2.0 transfers. Endpoints EP2 and EP6 are the most flexible endpoints, as they are configurable for size (512 or 1024 bytes) and depth of buffering (double, triple, or quad). Endpoints EP4 and EP8 are fixed at 512 bytes, double-buffered.

The bits in these registers control the following:

- **Valid**. Set to 1 (default) to enable the endpoint. A non-valid endpoint does not respond to host IN or OUT packets.
- **Type**. Two bits, TYPE1:0 (bits 5 and 4) set the endpoint type:
  - 00 = *invalid*
  - 01 = ISOCHRONOUS (EP2,4,6,8 only)
  - 10 = BULK (default)
  - 11 = INTERRUPT
- **Direction**. 1 = IN, 0 = OUT.
- **Buffering**. EP2 and EP6 only. Two bits, BUF1:0 control the depth of buffering:
  - 00 = quad

- 01 = *invalid*
- 10 = double (default)
- 11 = triple

“Buffering” refers to the number of RAM blocks available to the endpoint. With double buffering, for example, USB data can fill or empty an endpoint buffer at the same time that another packet from the same endpoint fills or empties from the external logic. This technique maximizes performance by saving each side, USB and external-logic interface, from waiting for the other side. Multiple buffering is most effective when the providing and consuming rates are comparable but bursty (as is the case with USB and many other interfaces, such as disk drives). Assigning more RAM blocks (triple and quad buffering) provides more “smoothing” of the bursty data rates. A simple way to determine the appropriate buffering depth is to start with the minimum, then increase it until no NAKs appear on the USB side and no wait states appear on the interface side.

8.5 CPU Access to FX2 Endpoint Data

Endpoint data is visible to the CPU at the addresses shown in Table 8-3. Whenever the application calls for endpoint buffers smaller than the physical buffer sizes shown in Tabl e8-3, the CPU accesses the endpoint data starting from the lowest address in the buffer. For example, if EP2 has a reported MaxPacketSize of 512 bytes, the CPU accesses the data in the lower portion of the EP2 buffer (i.e., from 0xF000 to 0xF1FF). Similarly, if the FX2 is operating in full speed mode (which dictates a maximum Bulk packet size of only 64 bytes), only the lower 64 bytes of the endpoint (i.e., 0xF000-0xF03F for EP2) will be used for Bulk data.

Table 8-3. Endpoint Buffers in RAM Space

Name	Address	Size (bytes)
EP0BUF	0xE740-0xE77F	64
EP1OUTBUF	0xE780-0xE7BF	64
EP1INBUF	0xE7C0-0xE7FF	64
EP2FIFOBUF	0xF000-0xF3FF	1024
EP4FIFOBUF	0xF400-0xF5FF	512
EP6FIFOBUF	0xF800-0xFBFF	1024
EP8FIFOBUF	0xFC00-0xFDFF	512

EP0BUF is for the (optional) data stage of a CONTROL transfer. The eight bytes of data from the CONTROL packet appear in a separate FX2 RAM buffer called SETUPDAT, at 0xE6B8-0xE6BF.

The CPU can only access the “active” buffer of a multiple-buffered endpoint. In other words, firmware must treat a quad-buffered 512-byte endpoint as being only 512 bytes wide, even though the quad-buffered endpoint actually occupies 2048 bytes of RAM. Also, when EP2 and EP6 are configured such that EP4 and/or EP8 are unavailable, the firmware must never attempt to access the buffers corresponding to those unavailable endpoints.

For example, if EP2 is configured for triple-buffered 1024-byte operation, the firmware should access EP2 only at 0xF000-0xF3FF. The firmware should not access the EP4 or EP6 buffers in this configuration, since they don't exist (the RAM space which they would normally occupy is used to implement the EP2 triple-buffering).

8.6 CPU Control of FX2 Endpoints

From the CPU's point of view, the "small" and "large" endpoints operate slightly differently, due to the multiple-packet buffering scheme used by the large endpoints.

The CPU uses internal registers to control the flow of endpoint data. Since the small endpoints EP0 and EP1 are programmed differently than the large endpoints EP2, EP4, EP6, and EP8, these registers fall into three categories:

- Registers that apply to the small endpoints (EP0, EP1IN, and EP1OUT)
- Registers that apply to the large endpoints (EP2, EP4, EP6, and EP8)
- Registers that apply to both sets of endpoints

8.6.1 Registers That Control EP0, EP1IN, and EP1OUT

Table 8-4. Registers that control EP0 and EP1

Address	Name	Function
0xE6A0	**EP0CS**	EP0 HSNAK, Busy, Stall
0xE68A 0xE68B	**EP0BCH** **EP0BCL**	EP0 Byte Count (MSB) EP0 Byte Count (LSB)
0xE65C 0xE65D	**USBIE** **USBIRQ**	EP0 Interrupt Enables EP0 Interrupt Requests
SFR 0xBA	**EP01STAT**	Endpoint 0 and 1 Status
0xE6A1	**EP1OUTCS**	EP1OUT Busy, Stall
0xE68D	**EP1OUTBC**	EP1OUT Byte Count
0xE6A2	**EP1INCS**	EP1IN Busy, Stall
0xE68F	**EP1INBC**	EP1IN Byte Count

8.6.1.1 EP0CS

Firmware uses this register to coordinate CONTROL transfers over endpoint 0. The EP0CS register contains three bits: **HSNAK**, **BUSY** and **STALL**.

HSNAK

HSNAK is automatically set to 1 whenever the SETUP token of a CONTROL transfer arrives. The FX2 logic automatically NAKs the STATUS (handshake) stage of the CONTROL transfer until the firmware clears the HSNAK bit *by writing "1" to it*. This mechanism gives the firmware a chance to hold off subsequent transfers until it completes the actions required by the CONTROL transfer.

Firmware must clear the HSNAK bit after servicing every CONTROL transfer.

BUSY

The read-only BUSY bit is relevant only for the data stage of a CONTROL transfer. BUSY=1 indicates that the endpoint is currently being serviced by USB, so firmware should not access the endpoint data.

BUSY is automatically cleared to 0 whenever the SETUP token of a CONTROL transfer arrives. The BUSY bit is set to 1 under different conditions for IN and OUT transfers.

For IN transfers, FX2 logic will NAK all IN0 tokens until the firmware has "armed" EP0 for IN transfers by writing to the EP0BCH:L Byte Count register, which sets BUSY=1 to indicate that firmware should not access the data. Once the endpoint data is sent and acknowledged, BUSY is automatically cleared to 0 and the EP0IN interrupt request bit is asserted. After BUSY is automatically cleared to 0, the firmware may refill the EP0IN buffer.

For OUT transfers, FX2 logic will NAK all OUT0 tokens until the firmware has "armed" EP0 for OUT transfers by writing any value to the EP0BCL register. BUSY is automatically set to 1 when the firmware writes to EP0BCL, and BUSY is automatically cleared to 0 after the data has been correctly received and ACK'd. When BUSY transitions to zero, the FX2 also generates an EP0OUT interrupt request.

The FX2's autovectored interrupt system automatically transfers control to the appropriate ISR (Interrupt Service Routine) for the endpoint requiring service. Chapter 4, "Interrupts" describes this mechanism.

STALL

Set STALL=1 to instruct the FX2 to return the STALL response to a CONTROL transfer. This is generally done when the firmware does not recognize an incoming USB request. According to the USB spec, endpoint zero must always accept transfers, so STALL is automatically cleared to 0 whenever a SETUP token arrives. If it's desired to stall a transfer and also clear HSNAK to 0 (by writing a 1 to it), the firmware should set STALL=1 first, in order to ensure that the STALL bit is set before the "acknowledge" phase of the CONTROL transfer can complete.

8.6.1.2 EP0BCH and EP0BCL

These are the byte count registers for bytes sent as the optional data stage of a CONTROL transfer. Although the EP0 buffer is only 64 bytes wide, the byte count registers are 16 bits wide to allow using the Setup Data Pointer to send USB IN data records that consist of multiple packets.

To use the Setup Data Pointer in its most-general mode, firmware clears the SUDPTR AUTO bit and writes the address of a data block into the Setup Data Pointer, then loads the EP0BCH:L registers with the total number of bytes to transfer. The FX2 automatically transfers the entire block, partitioning the data into MaxPacketSIze packets as necessary.

The Setup Data Pointer is the subject of Section 8.7.

For IN transfers *without* using the Setup Data Pointer, firmware loads data into EP0BUF, then writes the number of bytes to transfer into EP0BCH and EP0BCL. The packet is armed for IN transfer when the firmware writes to EP0BCL, so EP0BCH should always be loaded first. These transfers are always 64 bytes or less, so EP0BCH must be loaded with 0 (and EP0BCL must be in the range [0-64]). EP0BCH will hold that zero value until firmware overwrites it.

For EP0 OUT transfers, the byte count registers indicate the number of bytes received in EP0BUF. Byte counts for EP0 OUT transfers are always 64 or fewer, so EP0BCH is always zero after an OUT transfer. To re-arm the EP0 buffer for a future OUT transfer, the firmware simply writes any value to EP0BCL.

The EP0BCH register must be initialized on reset, since its power-on-reset state is undefined.

8.6.1.3 USBIE, USBIRQ

Three interrupts — SUTOK, SUDAV, and EP0ACK — are used to manage CONTROL transfers over endpoint zero. The individual enables for these three interrupt sources are in the USBIE register, and the interrupt-request flags are in the USBIRQ register.

Each of the three interrupts signals the completion of a different stage of a CONTROL transfer.

- **SUTOK** (“Setup Token”) asserts when FX2 receives the SETUP token.
- **SUDAV** (“Setup Data Available”) asserts when FX2 logic has loaded the eight bytes from the SETUP stage into the 8-byte buffer at SETUPDAT.
- **EP0ACK** (“Endpoint Zero Acknowledge”) asserts when the handshake stage has completed.

The SUTOK interrupt is not normally used; it is provided for debug and diagnostic purposes. Firmware generally services the CONTROL transfer by responding to the SUDAV interrupt, since this interrupt fires only after the 8 setup bytes are available for examination in the SETUPDAT buffer.

8.6.1.4 EP01STAT

The BUSY bits in EP0CS, EP1OUTCS, and EP1INCS (described later in this chapter) are replicated in this SFR; they are provided here in order to allow faster access (via the MOV instruction rather than MOVX) to those bits.

Three status bits are provided in the EP01STAT register; the status bits are the following:

- EP1INBSY: 1 = EP1IN is busy
- EP1OUTBSY: 1 = EP1OUT is busy
- EP0BSY: 1 = EP0 is busy

8.6.1.5 EP1OUTCS

This register is used to coordinate BULK or INTERRUPT transfers over EP1OUT. The EP1OUTCS register contains two bits, **BUSY** and **STALL**.

BUSY

This bit indicates when the firmware can read data from the Endpoint 1 OUT buffer. BUSY=1 means that the SIE “owns” the buffer, so firmware should not read (or write) the buffer. BUSY=0 means that the firmware may read from (or write to) the buffer. A 1-to-0 BUSY transition asserts the EP1OUT interrupt request, signaling that new EP1OUT data is available.

BUSY is automatically cleared to 0 after the FX2 verifies the OUT data for accuracy and ACKs the transfer. If a transmission error occurs, the FX2 automatically retries the transfer; error recovery is transparent to the firmware.

Firmware arms the endpoint for OUT transfers by writing any value to the byte count register EP1OUTBC, which automatically sets BUSY=1.

At power-on (or whenever a 0-to-1 transition occurs on the RESET pin), the BUSY bit is set to 0, so the FX2 will NAK all EP1OUT transfers until the firmware arms EP1OUT by writing any value to EP1OUTBC.

EZ-USB / EZ-USB FX Programmers:

The power-on state of all FX2 endpoint BUSY bits is zero, in contrast to EZ-USB and EZ-USB FX, whose BUSY bits for OUT endpoints default to one. This means that FX2 firmware must arm OUT endpoints prior to using them (EZ-USB and EZ-USB FX accept one OUT transfer before the OUT endpoint must be armed).

STALL

Firmware sets STALL=1 to instruct the FX2 to return the STALL PID (instead of ACK or NAK) in response to an EP1OUT transfer. The FX2 will continue to respond to EP1OUT transfers with the STALL PID until the firmware clears this bit.

8.6.1.6 EP1OUTBC

Firmware may read this 7-bit register to determine the number of bytes (0-64) in EP1OUTBUF.

Firmware writes any value to EP1OUTBC to arm an EP1OUT transfer.

8.6.1.7 EP1INCS

This register is used to coordinate BULK or INTERRUPT transfers over EP1IN. The EP1INCS register contains two bits, **BUSY** and **STALL**.

BUSY

This bit indicates when the firmware can load data into the Endpoint 1 IN buffer. BUSY=1 means that the SIE “owns” the buffer, so firmware should not write (or read) the buffer. BUSY=0 means that the firmware may write data into (or read from) the buffer. A 1-to-0 BUSY transition asserts the EP1IN interrupt request, signaling that the EP1IN buffer is free and ready to be loaded with new data.

The firmware schedules an IN transfer by loading up to 64 bytes of data into EP1INBUF, then writing the byte count register EP1INBC with the number of bytes loaded (0-64). Writing the byte count register automatically sets BUSY=1, indicating that the transfer over USB is pending. After the FX2 subsequently receives an IN token, sends the data, and successfully receives an ACK from the host, BUSY is automatically cleared to 0 to indicate that the buffer is ready to accept more data. This generates the EP1IN interrupt request, which signals that the buffer is again available.

At power-on, or whenever a 0-to-1 transition occurs on the RESET pin, the BUSY bit is set to 0, meaning that the FX2 will NAK all EP1IN transfers until the firmware arms the endpoint by writing the number of bytes to transfer into the EP1INBC register.

STALL

Firmware sets STALL=1 to instruct the FX2 to return the STALL PID (instead of ACK or NAK) in response to an EP1IN transfer. The FX2 will continue to respond to EP1IN transfers with the STALL PID until the firmware clears this bit.

8.6.1.8 EP1INBC

Firmware arms an IN transfer by loading this 7-bit register with the number of bytes (0-64) it has previously loaded into EP1INBUF.

8.6.2 Registers That Control EP2, EP4, EP6, EP8

In order to achieve the high transfer bandwidths required by USB 2.0's high-speed mode, the FX2's CPU should not participate in transfers to and from the "large" endpoints. Instead, those endpoints are usually connected directly to external logic (see *Chapter 9* and *Chapter 10* for details).

Some applications, however, may require the firmware to have at least some small amount of control over the large endpoints. For those applications, the FX2 provides the registers shown in Table 8-5.

Table 8-5. Registers that control EP2,EP4,EP6 and EP8

Address	Name	Function
SFR 0xAA	**EP2468STAT**	EP2, 4, 6, 8 empty/full
0xE648	**INPKTEND**	force end of IN packet
0xE640	**EP2ISOINPKTS**	ISO IN packets per frame or microframe
0xE6A3	**EP2CS**	npak, full, empty, stall
0xE690	**EP2BCH**	byte count (H)
0xE691	**EP2BCL**	byte count (L)
0xE641	**EP4ISOINPKTS**	ISO IN packets per frame or microframe
0xE6A4	**EP4CS**	npak, full, empty, stall
0xE694	**EP4BCH**	byte count (H)
0xE695	**EP4BCL**	byte count (L)
0xE642	**EP6ISOINPKTS**	ISO IN packets per frame/microframe
0xE6A5	**EP6CS**	npak, full, empty, stall
0xE698	**EP6BCH**	byte count (H)
0xE699	**EP6BCL**	byte count (L)
0xE643	**EP8ISOINPKTS**	ISO IN packets per frame/microframe
0xE6A6	**EP8CS**	npak, full, empty, stall
0xE69C	**EP8BCH**	byte count (H)
0xE69D	**EP8BCL**	byte count (L)

8.6.2.1 EP2468STAT

The Endpoint Full and Endpoint Empty status bits (described below, in Section 8.6.2.3) are replicated here in order to allow faster access by the firmware.

8.6.2.2 EP2ISOINPKTS, EP4ISOINPKTS, EP6ISOINPKTS, EP8ISOINPKTS

For high-speed (480 Mbps) ISOCHRONOUS IN endpoints only, the **INPPF1** and **INPPF0** bits in each of these registers determine the number of packets per microframe.

These registers do not affect full-speed (12 Mbps) operation; full-speed isochronous transfers are always fixed at one packet per frame.

Table 8-6. Isochronous IN Packets per Microframe, High-Speed Only

INPPF1	INPPF0	Packets
0	0	Invalid
0	1	1
1	0	2
1	1	3

8.6.2.3 EP2CS, EP4CS, EP6CS, EP8CS

Because the four large FX2 endpoints offer double, triple or quad buffering, a single BUSY bit is not sufficient to convey the state of these endpoint buffers. Therefore, these endpoints have multiple bits (NPAK, FULL, EMPTY) that can be inspected in order to determine the state of the endpoint buffers.

*Multiple-buffered endpoint data must be read or written **only** at the buffer addresses given in Table 8-3. The FX2 automatically switches the multiple buffers in and out of the single addressable buffer space.*

NPAK[2:0] (EP2, EP6) and **NPAK[1:0] (EP4, EP8)**

NPAK values have different interpretations for IN and OUT endpoints:

- ***OUT Endpoints:*** NPAK indicates the number of packets received over USB and ready for the firmware to read.
- ***IN Endpoints:*** NPAK indicates the number of IN packets committed to USB (i.e., loaded and armed for USB transfer), and thus *unavailable* to the firmware.

The NPAK fields differ in size to account for the depth of buffering available to the endpoints. Only double buffering is available for EP4 and EP8 (two NPAK bits), and up to quad buffering is available for EP2 and EP6 (three NPAK bits).

FULL

While FULL and EMPTY apply to transfers in both directions, “FULL” is more useful for IN transfers. It has the same meaning as “BUSY”, but applies to multiple-buffered IN endpoints. FULL=1 means that all buffers are committed to USB, and none are available for firmware access.

For IN transfers, FULL=1 means that all buffers are committed to USB, so firmware should not load the endpoint buffer with any more data. When FULL=1, NPAK will hold 2, 3 or 4, depending on the buffering depth (double, triple or quad). This indicates that all buffers are in use by the USB

transfer logic. As soon as one buffer becomes available, FULL will be cleared to 0 and NPAK will decrement by one, indicating that all but one of the buffers are committed to USB (i.e., one is available for firmware access). As IN buffers are transferred over USB, NPAK decrements to indicate the number still pending, until all are sent and NPAK=0.

EMPTY

While FULL and EMPTY apply to transfers in both directions, EMPTY is more useful for OUT transfers. EMPTY=1 means that the buffers are empty; all received packets (2, 3, or 4, depending on the buffering depth) have been serviced.

STALL

Firmware sets STALL=1 to instruct the FX2 to return the STALL PID (instead of ACK or NAK) in response to an IN or OUT transfer. The FX2 will continue to respond to IN or OUT transfers with the STALL PID until the firmware clears this bit.

8.6.2.4 EP2BCH:L, EP4BCH:L, EP6BCH:L, EP8BCH:L

Endpoints EP2 and EP6 have 11-bit byte count registers to account for their maximum buffer sizes of 1024 bytes. Endpoints EP4 and EP8 have 10-bit byte count registers to account for their maximum buffer sizes of 512 bytes.

The byte count registers function similarly to the EP0 and EP1 byte count registers:

- For an IN transfer, the firmware loads the byte count registers to arm the endpoint (if EPxBCH must be loaded, it should be loaded first, since the endpoint is armed when EPxBCL is loaded).
- For an OUT transfer, the firmware reads the byte count registers to determine the number of bytes in the buffer, then writes any value to the low byte count register to re-arm the endpoint. See the “Skip” section, below, for further details.

SKIP

Normally, the CPU interface and outside-logic interface to the endpoint FIFOs are independent, with separate sets of control bits for each interface. The AUTOOUT mode and the SKIP bit implement an “overlap” between these two domains. A brief introduction to the AUTOOUT mode is given below; full details appear in *Chapter 9, "Slave FIFOs."*

When outside logic is connected to the interface FIFOs, the normal data flow is for the FX2 automatically to commit OUT data packets to the outside interface FIFO as they become available. This ensures an uninterrupted flow of OUT data from the host to the outside world, and preserves the high bandwidth required by high speed mode.

In some cases, it may be desirable to insert a “hook” into this data flow, so that -- rather than the FX2 automatically committing the packets to the outside interface as they are received over USB, firmware receives an interrupt for every received OUT packet, then has the option to either commit

the packet to the outside interface (the “output FIFO”), or discard it. The firmware might, for example, inspect a packet header to make this skip/commit decision.

To enable this “hook”, the AUTOOUT bit is cleared to 0. If AUTOOUT = 0 and an OUT endpoint is re-armed by writing to its low byte-count register, the actual value written to the register becomes significant:

- If the SKIP bit (bit 7 of each EPxBCL register) is cleared to 0, the packet will be committed to the output FIFO and thereby made available to the FIFO’s master (either external logic or the internal GPIF).
- If the SKIP bit is set to 1, the just-received OUT packet will not be committed to the output FIFO for transfer to the external logic; instead, the packet will be ignored, its buffer will immediately be made available for the next OUT packet, and the output FIFO (and external logic) will never even “know” that it arrived.

The AUTOOUT bit appears in bit 4 of the Endpoint FIFO Configuration Registers EP2FIFOCFG, EP4FIFOCFG, EP6FIFOCFG and EP8FIFOCFG.

8.6.3 Registers That Control All Endpoints

Table 8-7. Registers that control all endpoints

0xE658	**IBNIE**	IN-BULK-NAK individual interrupt enables
0xE659	**IBNIRQ**	IN-BULK-NAK individual interrupt requests
0xE65A	**NAKIE**	PING plus combined IBN-interrupt enable
0xE65B	**NAKIRQ**	PING plus combined IBN-interrupt request
0xE65C	**USBIE**	SUTOK, SUDAV, EP0-ACK, SOF interrupt enables
0xE65D	**USBIRQ**	SUTOK, SUDAV, EP0-ACK, and SOF interrupt requests
0xE65E	**EPIE**	Endpoint interrupt enables
0xE65F	**EPIRQ**	Endpoint interrupt requests
0xE662	**USBERRIE**	USB error interrupt enables
0xE663	**USBERRIE**	USB error interrupt requests
0xE664	**ERRCNTLIM**	USB error counter and limit
0xE665	**CLRERRCNT**	Clear error count
0xE683	**TOGCTL**	EP0/EP1 data toggle

8.6.3.1 IBNIE, IBNIRQ, NAKIE, NAKIRQ

These registers contain the interrupt-enable and interrupt-request bits for two endpoint conditions, **IN-BULK-NAK** and **PING**.

IN-BULK-NAK (IBN)

When the host requests an IN packet from an FX2 BULK endpoint, the endpoint NAKs (returns the NAK PID) until the endpoint buffer is filled with data and armed for transfer, at which point the FX2 answers the IN request with data.

Until the endpoint is armed, a flood of IN-NAKs can tie up bus bandwidth. Therefore, if the IN endpoints aren't always kept full and armed, it may be useful to know when the host is "knocking at the door", requesting IN data.

The IN-BULK-NAK (IBN) interrupt provides this notification. The IBN interrupt fires whenever a BULK endpoint NAKs an IN request. The IBNIE/IBNIRQ registers contain individual enable and request bits per endpoint, and the NAKIE/NAKIRQ registers each contain a single bit, IBN, that is the OR'd combination of the individual bits in IBNIE/IBNIRQ, respectively.

Firmware enables an interrupt by setting the enable bit high, and clears an interrupt request bit by writing a 1 to it.

The FX2 interrupt system is described in detail in Chapter 4, "Interrupts."

The IBNIE register contains an individual interrupt-enable bit for each endpoint: EP0, EP1, EP2, EP4, EP6 and EP8. These bits are valid only if the endpoint is configured as a BULK or INTERRUPT endpoint. The IBNIRQ register similarly contains individual interrupt request bits for the 6 endpoints.

The IBN interrupt-service routine should take the following actions, in the order shown:

1. Clear the USB (INT2) interrupt request (by writing 0 to it).
2. Inspect the endpoint bits in IBNIRQ to determine which IN endpoint just NAK'd.
3. Take the required action (set a flag, arm the endpoint, etc.), then clear the individual IBN bit in IBNIRQ for the serviced endpoint (by writing 1 to it).
4. Repeat steps (2) and (3) for any other endpoints that require IBN service, until all IRQ bits are cleared.
5. Clear the IBN bit in the NAKIRQ register (by writing 1 to it).

Because the *IBN bit represents the OR'd combination of the individual IBN interrupt requests, it will not "fire" again until all individual IBN interrupt requests have been serviced and cleared.*

PING

PING is the “flip side” of IBN; it’s used for high speed (480 Mbits/sec) BULK OUT transfers.

When operating at full speed (USB 1.1 spec), every host OUT transfer consists of the OUT PID *and the endpoint data*, even if the endpoint is NAKing (not ready). While the endpoint is not ready, the host repeatedly sends all the OUT data; if it’s repeatedly NAK’d, bus bandwidth is wasted.

USB 2.0 introduced a new mechanism, called PING, that makes better use of bus bandwidth for “unready” BULK OUT endpoints.

At high speed (USB 2.0 spec), the host can “ping” a BULK OUT endpoint to determine if it is ready to accept data, *holding off the OUT data transfer until it can actually be accepted*. The host sends a PING token, and the FX2 responds with:

- An ACK to indicate that there is space in the OUT endpoint buffer
- A NAK to indicate “not ready, try later”.

The PING interrupts indicate that an FX2 BULK OUT endpoint returned a NAK in response to a PING.

PING only applies at high speed (480 Mbits/sec).

Unlike the IBN bits, which are combined into a single IBN interrupt for all endpoints, each BULK OUT endpoint has a separate interrupt (EP0PING, EP1PING, EP2PING, ..., EP8PING). Interrupt-enables for the individual interrupts are in the NAKIE register; the interrupt-requests are in the NAKIRQ register.

The interrupt service routine for the PING interrupts should perform the following steps, in the order shown:

1. Clear the INT2 interrupt request.
2. Take the action for the requesting endpoint.
3. Clear the appropriate EPxPING bit for the endpoint.

8.6.3.2 EPIE, EPIRQ

These registers are used to manage interrupts from the FX2 endpoints. In general, an interrupt request is asserted whenever the following occurs:

- An IN endpoint buffer becomes available for the CPU to load.
- An OUT endpoint has new data for the CPU to read.

For the small endpoints (EP0 and EP1IN/OUT), these conditions are synonymous with the end-point BUSY bit making a 1-to-0 transition (busy to not-busy). As with all FX2 interrupts, this one is enabled by writing a “1” to its enable bit, and the interrupt flag by writing a “1” to it.

***Do not** attempt to clear an IRQ bit by reading the IRQ register, ORing its contents with a bit mask (e.g. 00010000), then writing the contents back to the register. Since a “1” clears an IRQ bit, this clears **all** the asserted IRQ bits rather than just the desired one. Instead, simply write a single “1” (e.g., 00010000) to the register.*

8.6.3.3 USBERRIE, USBERRIRQ, ERRCNTLIM, CLRERRCNT

These registers are used to monitor the “health” of the USB connection between the FX2 and the host.

USBERRIE

This register contains the interrupt-enable bits for the “Isochronous Endpoint Error” interrupts and the “USB Error Limit” interrupt.

An “Isochronous Endpoint Error” occurs when the FX2 detects a PID sequencing error for a high-bandwidth, high-speed ISO endpoint.

USBERRIRQ

This register contains the interrupt flags for the “Isochronous Endpoint Error” interrupts and the “USB Error Limit” interrupt.

ERRCNTLIM

FX2 firmware sets the USB error limit to any value from 1 to 15 by writing that value to the lower nibble of this register; when that many USB errors (CRC errors, Invalid PIDs, garbled packets, etc.) have occurred, the “USB Error Limit” interrupt flag will be set. At power-on-reset, the error limit defaults to 4 (0100 binary).

The upper nibble of this register contains the current USB error count.

CLRERRCNT

Writing any value to this register clears the error count in the upper nibble of ERRCNTLIM. The lower nibble of ERRCNTLIM is not affected.

8.6.3.4 TOGCTL

As described in *Chapter 1, "Introducing EZ-USB FX2"* the host and device maintain a *data toggle* bit, which is toggled between data packet transfers. There are certain times when the firmware must reset an endpoint’s data toggle bit to 0:

- After a configuration changes (i.e., after the host issues a *Set Configuration* request).
- After an interface's alternate setting changes (i.e., after the host issues a *Set Interface* request).
- After the host sends a *Clear Feature - Endpoint Stall* request to an endpoint.

For the first two, the firmware must clear the data toggle bits for all endpoints contained in the affected interfaces. For the third, only one endpoint's data toggle bit is cleared.

The TOGCTL register contains bits to set or clear an endpoint data toggle bit, as well as to read the current state of a toggle bit.

At this writing, there is no known reason for firmware to set an endpoint toggle to "1". Also, since the FX2 handles all data toggle management, normally there is no reason to know the state of a data toggle. These capabilities are included in the TOGCTL register for completeness and debug purposes.

TOGCTL **Data Toggle Control** **E683**

b7	b6	b5	b4	b3	b2	b1	b0
Q	**S**	**R**	**IO**	**EP3**	**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

A two-step process is employed to clear an endpoint data toggle bit to 0. First, writes the TOGCTL register with an endpoint address (EP3:EP0) plus a direction bit (IO). Then, keeping the endpoint and direction bits the same, write a "1" to the R (reset) bit. For example, to clear the data toggle for EP6 configured as an "IN" endpoint, write the following values sequentially to TOGCTL:

- 00010110
- 00110110

8.7 The Setup Data Pointer

The USB host sends device requests using CONTROL transfers over endpoint 0. Some requests require the FX2 to return data over EP0. During enumeration, for example, the host issues *Get Descriptor* requests that ask for the device's capabilities and requirements. The returned data can span many packets, so it must be partitioned into packet-sized blocks, then the blocks must be sent at the appropriate times (i.e., when the EP0 buffer becomes ready).

The Setup Data Pointer automates this process of returning IN data over EP0, simplifying the firmware.

*For the Setup Data Pointer to work properly, EP0's MaxPacketSize **must** be set to 64.*

Table 8-8 lists the registers which configure the Setup Data Pointer.

Table 8-8. Registers used to control the Setup Data Pointer

Address	Register Name	Function
0xE6B3	SUDPTRH	High address
0xE6B4	SUDPTRL	Low address
0xE6B5	SUDPTRCTL	SDPAUTO bit

To send a block of data, the block's starting address is loaded into SUDPTRH:L. The block length must previously have been set; the method for accomplishing this depends on the state of the SDPAUTO bit:

- **SDPAUTO = 0 (Manual Mode):** Used for general-purpose block transfers. Firmware writes the block length to EP0BCH:L.
- **SDPAUTO = 1 (Auto Mode):** Used for sending Device, Configuration, String, Device Qualifier, and Other Speed Configuration descriptors *only*. The block length is automatically read from the "length" field of the descriptor itself; no explicit loading of EP0BCH:L is necessary.

Writing to SUDPTRL starts the transfer; the FX2 automatically sends the entire block, packetizing as necessary.

For example, to answer a *Get Descriptor - Device* request, firmware sets SDPAUTO = 1, then loads the address of the device descriptor into SUDPTRH:L. The FX2 then automatically loads the EP0 data buffer with the required number of packets and transfers them to the host.

To command the FX2 to ACK the status (handshake) packet, the firmware clears the HSNAK bit (by writing 1 to it) before starting the Setup Data Pointer transfer.

If the firmware needs to know when the transaction is complete (i.e., sent and acknowledged), it can enable the EP0ACK interrupt before starting the Setup Data Pointer transfer.

*When SDPAUTO = 0, writing to EP0BCH:L only sets the block length; it does not arm the transfer (the transfer is armed by writing to SUDPTRL). Therefore, before performing an EP0 transfer which does **not** use the Setup Data Pointer (i.e., one which is meant to be armed by writing to EP0BCL), SDPAUTO **must** be set to 1.*

8.7.1 Transfer Length

When the host makes any EP0IN request, the FX2 respects the following two length fields:

- the requested number of bytes (from the last two bytes of the SETUP packet received from the host)
- the available number of bytes, supplied either as a length field in the actual descriptor (SDPAUTO=1) *or* in EP0BCH:L (SDPAUTO=0)

In accordance with the USB Specification, the FX2 sends the *smaller* of these two length fields.

8.7.2 Accessible Memory Spaces

The Setup Data Pointer can access data in either of two RAM spaces:

- On-chip Main RAM (8 KB at 0x0000-0x1FFF)
- On-chip Scratch RAM (512 bytes at 0xE000-0xE1FF)

The Setup Data Pointer cannot be used to access off-chip memory at any address.

8.8 Autopointers

Endpoint data is available to the CPU in RAM buffers (see Table 8-3). In some cases, it is faster for the firmware to access endpoint data as though it were in a FIFO register. The FX2 provides two special data pointers, called “Autopointers”, that automatically increment after each byte transfer. Using the Autopointers, firmware can access contiguous blocks of on- or off-chip data memory as a FIFO.

Each Autopointer is controlled by a 16-bit address register (AUTOPTRnH:L), a data register (XAUTODATn), and a control bit (APTRnINC). An additional control bit, APTREN, enables both Autopointers.

A read from (or write to) an Autopointer data register *actually* accesses the address pointed to by the corresponding Autopointer address register, which increments on every data-register access. To read or write a contiguous block of memory (for example, an endpoint buffer) using an Autopointer, load the Autopointer’s address register with the starting address of the block, then repeatedly read or write the Autopointer’s data register.

The AUTOPTRnH:L registers may be written or read at any time to determine the current Autopointer address.

Most of the Autopointer registers are in SFR Space for quick access; the data registers are available only in External Data space.

Table 8-9. Registers that control the Autopointers

Address	Register Name	Function
SFR 0xAF	**AUTOPTRSETUP**	Increment/freeze, off-chip access enable
SFR 0x9A	**AUTOPTR1H**	Address high
SFR 0x9B	**AUTOPTR1L**	Address low
0xE67B	**XAUTODAT1**	Data
SFR 0x9D	**AUTOPTR2H**	Address high
SFR 0x9E	**AUTOPTR2L**	Address low
0xE67C	**XAUTODAT2**	Data

The Autopointers are configured using three bits in the AUTOPTRSETUP register: one bit (APTREN) enables both autopointers, and two bits (one for each Autopointer, called APTR1INC and APTR2INC, respectively) control whether or not the address increments for every Autodata access.

Enabling the Autopointers has one side-effect: Any *code* access (an instruction fetch, for instance) from addresses 0xE67B and 0xE67C will return the AUTODATA values, rather than the code-memory values at these two addresses. This introduces a two-byte “hole” in the code memory.

*There is no two-byte hole in the **data** memory at 0xE67B:E67C; the hole only appears in the **program** memory.*

Chapter 9 Slave FIFOs

9.1 Introduction

Although some FX2-based devices may use the FX2's CPU to process USB data directly (see *Chapter 8 "Access to Endpoint Buffers"*), most will use the FX2 simply as a conduit between the USB and external data-processing logic (e.g., an ASIC or DSP, or the IDE controller on a hard disk drive).

In devices with external data-processing logic, USB data flows between the host and that external logic — *usually without any participation by the FX2's CPU* — through the FX2's internal *endpoint FIFOs*. To the external logic, these endpoint FIFOs look like most others; they provide the usual timing signals, handshake lines (full, empty, programmable-level), read and write strobes, output enable, etc.

These FIFO signals must, of course, be controlled by a FIFO "master". The FX2's General Programmable Interface (GPIF) can act as an *internal* master when the FX2 is connected to external logic which doesn't include a standard FIFO interface, or the FIFOs can be controlled by an external master. While its FIFOs are controlled by an external master, the FX2 is said to be in "Slave FIFO" mode.

Chapter 10, "General Programmable Interface (GPIF)," discusses the internal-master GPIF. This chapter provides details on the interface — both hardware and software — between the FX2's slave FIFOs and an *external* master.

9.2 Hardware

Figure 9-1 illustrates the four slave FIFOs. The figure shows the FIFOs operating in 16-bit mode, although they can also be configured for 8-bit operation.

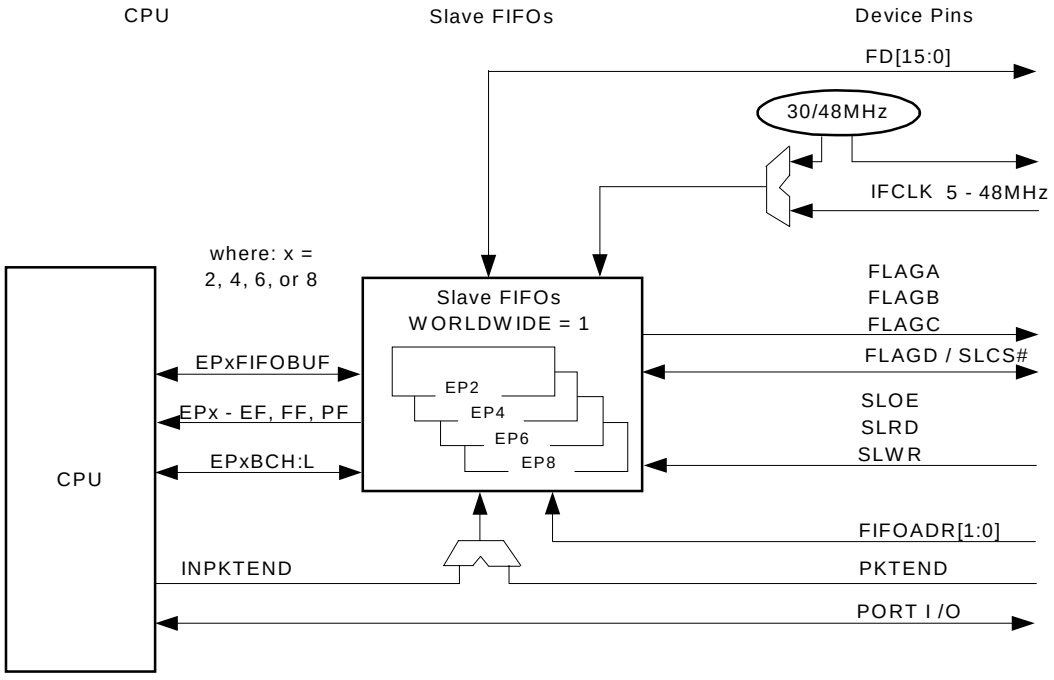


Figure 9-1. Slave FIFOs' Role in the FX2 System

Table 9-1 lists the registers associated with the slave-FIFO hardware. The registers are fully described in *Chapter 15, "Registers."*

Table 9-1. Registers Associated with Slave FIFO Hardware

IFCONFIG	EPxFIFOPFH/L
PINFLAGAB	PORTACFG
PINFLAGCD	INPKTEND
FIFORESET	EPxFLAGIE
FIFOPINPOLAR	EPxFLAGIRQ
EPxCFG	EPxFIFOBCH:L
EPxFIFOCFG	EPxFLAGS
EPxAUTOINLENH:L	EPxBUF

9.2.1 Slave FIFO Pins

The FX2 comes out of reset with its I/O pins configured in “Ports” mode, not “Slave FIFO” mode. To configure the pins for Slave FIFO mode, the IFCFG1:0 bits in the IFCONFIG register must be set to 11 (see Table 13-10, “IFCFG Selection of Port I/O Pin Functions" for details). When IFCFG1:0 = 11, the Slave FIFO interface pins are presented to the external master, as shown in Figure 9-2.

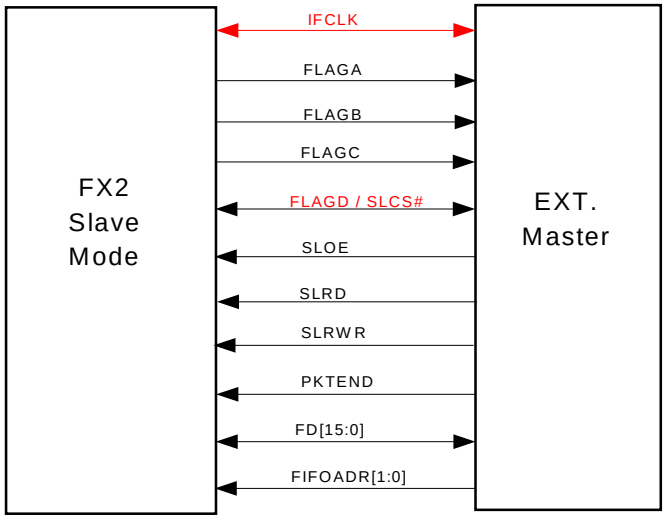


Figure 9-2. FX2 Slave Mode Full-Featured Interface Pins

External logic accesses the FIFOs through an 8- or 16-bit-wide data bus, FD. The data bus is bidirectional, with its output drivers controlled by the SLOE pin.

The FIFOADR[1:0] pins select which of the four FIFOs is connected to the FD bus.

In asynchronous mode (IFCONFIG.3 = 1), SLRD and SLWR are read and write strobes; in synchronous mode (IFCONFIG.3 = 0), SLRD and SLWR are enables for the IFCLK clock pin.

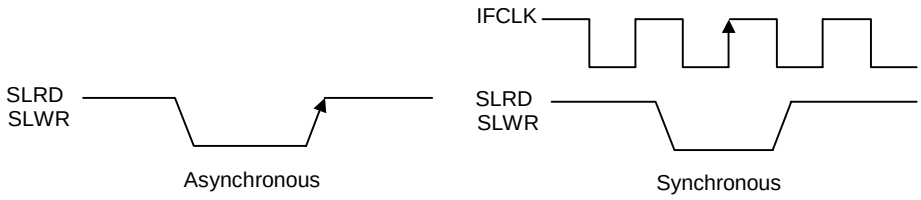


Figure 9-3. Asynchronous vs. Synchronous Timing Models

9.2.2 FIFO Data Bus (FD)

The FIFO data bus, FD[x:0], can be either 8 or 16 bits wide. The width is selected via each FIFO's WORDWIDE bit, (EPxFIFOCFG.0):

- WORDWIDE=0: 8-bit mode. FD[7:0] replaces Port B. See Figure 9-4.
- WORDWIDE=1: 16-bit mode. FD[15:8] replaces Port D and FD[7:0] replaces Port B. See Figure 9-5.

At power-on reset, the FIFO data bus defaults to 16-bit mode (WORDWIDE = 1) for all FIFOs.

In either mode, the FIFOADR[1:0] pins select which of the four FIFOs is internally connected to the FD pins.

*If **all** of the FIFOs are configured for 8-bit mode, Port D remains available for use as general-purpose I/O. If **any** FIFO is configured for 16-bit mode, Port D is unavailable for use as general-purpose I/O regardless of which FIFO is currently selected via the FIFOADR[1:0] pins.*

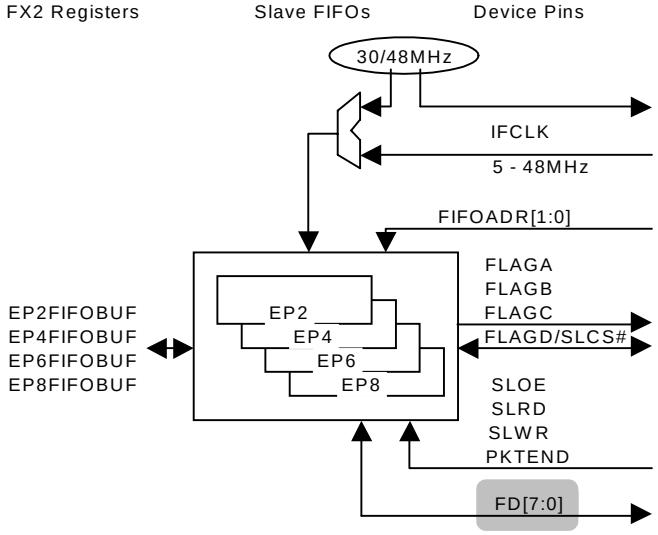


Figure 9-4. 8-bit Mode Slave FIFOs, WORDWIDE=0

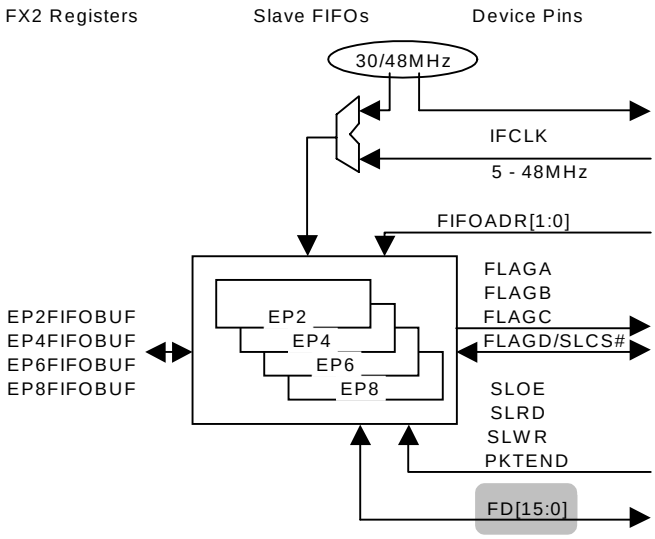


Figure 9-5. 16-bit Mode Slave FIFOs, WORDWIDE=1

9.2.3 Interface Clock (IFCLK)

The slave FIFO interface can be clocked from either an internal or an external source. The FX2's internal clock source can be configured to run at either 30 or 48 MHz, and it can optionally be output on the IFCLK pin. If the FX2 is configured to use an external clock source, the IFCLK pin can be driven at any frequency between 5 MHz and 48 MHz. On power-on reset, the FX2 defaults to the internal source at 48 MHz, normal polarity, with the IFCLK output disabled. See Figur e9-6.

IFCONFIG.7 selects between internal and external sources: 0 = external, 1 = internal.

IFCONFIG.6 selects between the 30- and 48-MHz internal clock: 0 = 30 MHz, 1 = 48 MHz. This bit has no effect when IFCONFIG.7 = 0.

IFCONFIG.5 is the output enable for the internal clock source: 0 = disable, 1 = enable. This bit has no effect when IFCONFIG.7 = 0.

IFCONFIG.4 inverts the polarity of the interface clock (whether it's internal or external): 0 = normal, 1 = inverted. IFCLK inversion can make it easier to interface the FX2 with certain external circuitry; Figure 9-7, for example, demonstrates the use of IFCLK inversion in order to ensure a long-enough setup time for reading the FX2's FIFO flags.

When IFCLK is configured as an input, the minimum frequency that can be applied to it is 5 MHz.

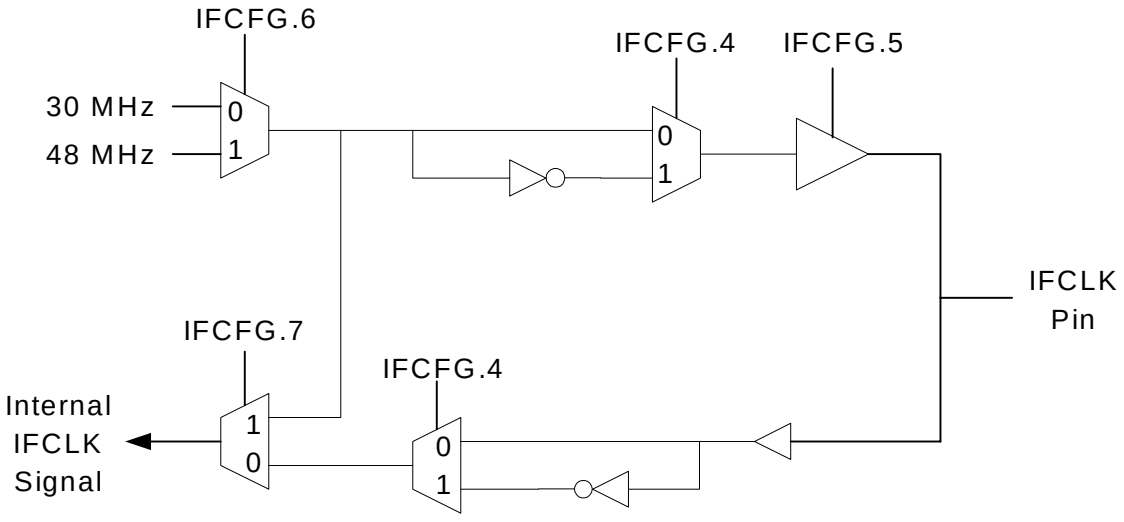


Figure 9-6. IFCLK Configuration

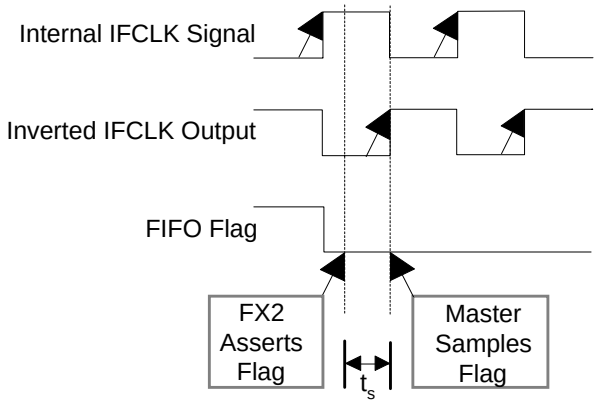


Figure 9-7. Satisfying Setup Timing by Inverting the IFCLK Output

9.2.4 FIFO Flag Pins (FLAGA, FLAGB, FLAGC, FLAGD)

Four pins — FLAGA, FLAGB, FLAGC, and FLAGD — report the status of the FX2's FIFOs; in addition to the usual "FIFO full" and "FIFO empty" signals, there is also a signal which indicates that a FIFO has filled to a user-programmable level. The external master typically monitors the "empty" flag of OUT endpoints and the "full" flag of IN endpoints; the "programmable-level" flag is

equally useful for either type of endpoint (it can, for instance, give advance warning that an OUT endpoint is almost empty or that an IN endpoint is almost full).

The FLAGA, FLAGB, and FLAGC pins can operate in either of two modes: *Indexed* or *Fixed*, as selected via the PINFLAGSAB and PINFLAGSCD registers. The FLAGD pin operates in Fixed mode only. Each pin is configured independently; some pins can be in Fixed mode while others are in Indexed mode. See Chapter 15, "Registers," for complete details.

Flag pins configured for Indexed mode report the status *of the FIFO currently selected by the FIFOADR[1:0] pins*. When configured for Indexed mode, FLAGA reports the “programmable-level” status, FLAGB reports the “full” status, and FLAGC reports the “empty” status.

Flag pins configured for Fixed mode report one of the three conditions *for a specific FIFO, regardless of the state of the FIFOADR[1:0] pins*. The condition and FIFO are user-selectable. For example, FLAGA could be configured to report FIFO2’s “empty” status, FLAGB to report FIFO4’s “empty” status, FLAGC to report FIFO4’s “programmable level” status, and FLAGD to report FIFO6’s “full” status.

The polarity of the “empty” and “full” flag pins defaults to active-low but may be inverted via the FIFOPINPOLAR register.

At power-on reset, the FIFO flags are configured for Indexed operation.

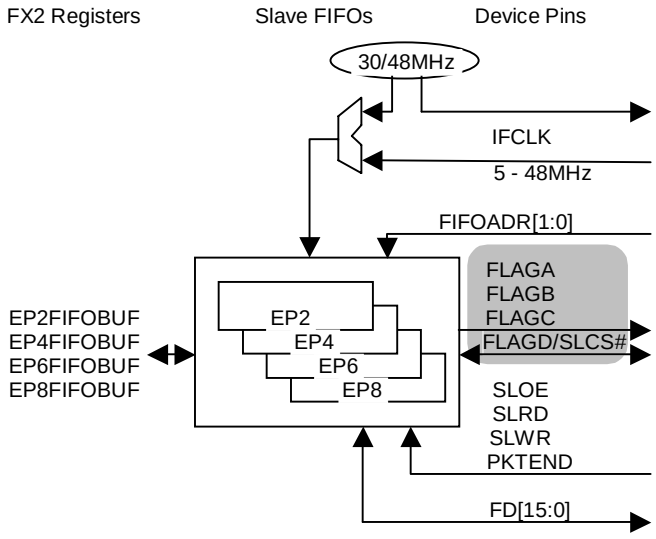


Figure 9-8. FLAGx

9.2.5 Control Pins (SLOE, SLRD, SLWR, PKTEND, FIFOADR[1:0])

The Slave FIFO "control" pins are SLOE (Output Enable), SLRD (Read), SLWR (Write), PKTEND (Packet End), and FIFOADR[1:0] (FIFO Select). "Read" and "Write" are from the external master's point of view; the external master reads from OUT endpoints and writes to IN endpoints. See Figure 9-9.

Read — SLOE and SLRD:

In synchronous mode (IFCONFIG.3 = 0), the FIFO pointer is incremented on each rising edge of IFCLK while SLRD is asserted. In asynchronous mode (IFCONFIG.3 = 1), the FIFO pointer is incremented on each asserted-to-deasserted transition of SLRD.

The SLOE pin enables the FD outputs.

By default, SLOE and SLRD are active-low; their polarities can be changed via the FIFOPINPOLAR register.

Write — SLWR:

In synchronous mode (IFCONFIG.3 = 0), data on the FD bus is written to the FIFO (and the FIFO pointer is incremented) on each rising edge of IFCLK while SLWR is asserted. In asynchronous mode (IFCONFIG.3 = 1), data on the FD bus is written to the FIFO (and the FIFO pointer is incremented) on each asserted-to-deasserted transition of SLWR.

By default, SLWR is active-low; its polarity can be changed via the FIFOPINPOLAR register.

FIFOADR[1:0]:

The FIFOADR[1:0] pins select which of the four FIFOs is connected to the FD bus (and, if the FIFO flags are operating in Indexed mode, they select which FIFO's flags are presented on the FLAGx pins):

Table 9-2. FIFO Selection via FIFOADR[1:0]

FIFOADR[1:0]	Selected FIFO
00	EP2
01	EP4
10	EP6
11	EP8

PKTEND:

An external master asserts the PKTEND pin to commit an IN packet to USB regardless of the packet's length. PKTEND is usually used when the master wishes to send a "short" packet (i.e., a packet smaller than the size specified in the EPxAUTOINLENH:L registers).

For example: Assume that EP4AUTOINLENH:L is set to the default of 512 bytes. If AUTOIN = 1, the external master can stream data to FIFO4 continuously, and (absent any bottlenecks in the data path) the FX2 will automatically commit a packet to USB whenever the FIFO fills with 512 bytes. If the master wants to send a stream of data whose length is not a multiple of 512, the last packet will *not* be automatically committed to USB because it's smaller than 512 bytes. To commit that last packet, the master can do one of two things: It can pad the packet with dummy data in order to make it exactly 512 bytes long, or it can write the short packet to the FIFO then assert the PKTEND pin.

If the FIFO is configured to allow zero-length packets (EPxFIFOCFG.2 = 1), asserting the PKTEND pin when the FIFO is empty will commit a zero-length packet.

By default, PKTEND is active-low; its polarity can be changed via the FIFOPINPOLAR register.

The PKTEND pin must not be asserted unless a buffer is available, even if only a zero-length packet is being committed. The "full" flag may be used to determine whether a buffer is available.

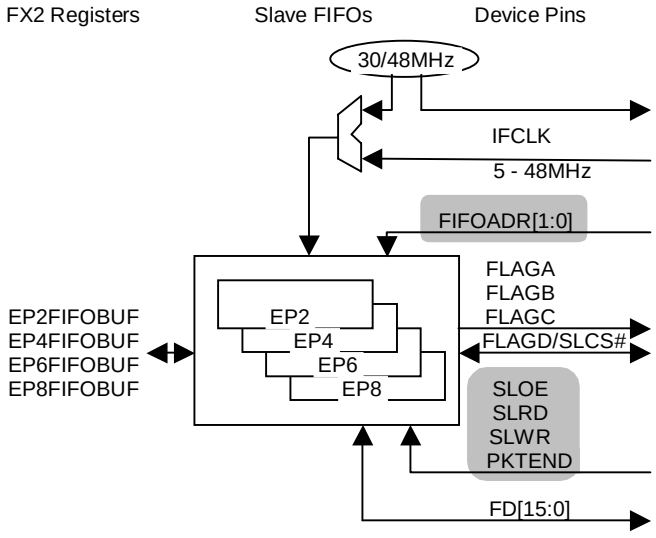


Figure 9-9. Slave FIFO Control Pins

9.2.6 Slave FIFO Chip Select ($\overline{SLCS}$)

The “Slave FIFO Chip Select” pin ($\overline{SLCS}$) is an alternate function of pin PA7; it’s enabled via the PORTACFG.6 bit (see Section 13.3.1, "Port A Alternate Functions").

The $\overline{SLCS}$ pin allows external logic to effectively remove the FX2 from the FIFO Data bus, in order to, for example, share that bus among multiple slave devices.

While the $\overline{SLCS}$ pin is pulled high by external logic, the FX2 floats its FD[x:0] pins and ignores the SLOE, SLRD, SLWR, and PKTEND pins.

9.2.7 Implementing Synchronous Slave FIFO Writes

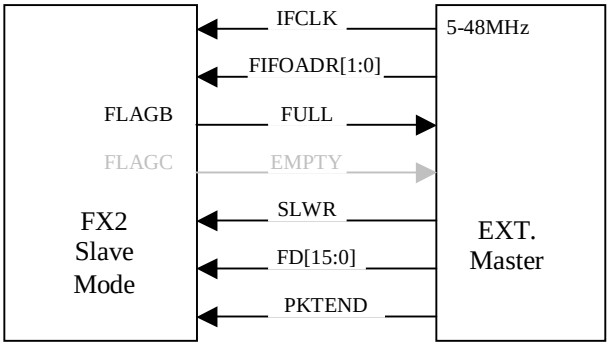


Figure 9-10. Interface Pins Example: Synchronous FIFO Writes

Typically, the sequence of events for the external master is:

IDLE: When write event occurs, transition to State 1.

STATE 1: Point to IN FIFO, assert FIFOADR[1:0], transition to State 2.

STATE 2: If FIFO-Full flag is false (FIFO not full), transition to State 3 else remain in State 2.

STATE 3: Drive data on the bus, assert SLWR for one IFCLK, transition to State 4.

STATE 4: If more data to write, transition to State 2 else transition to IDLE.

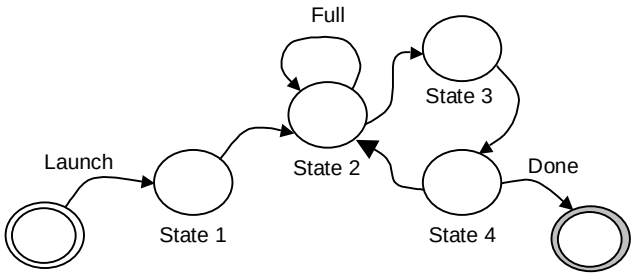


Figure 9-11. State Machine Example: Synchronous FIFO Writes

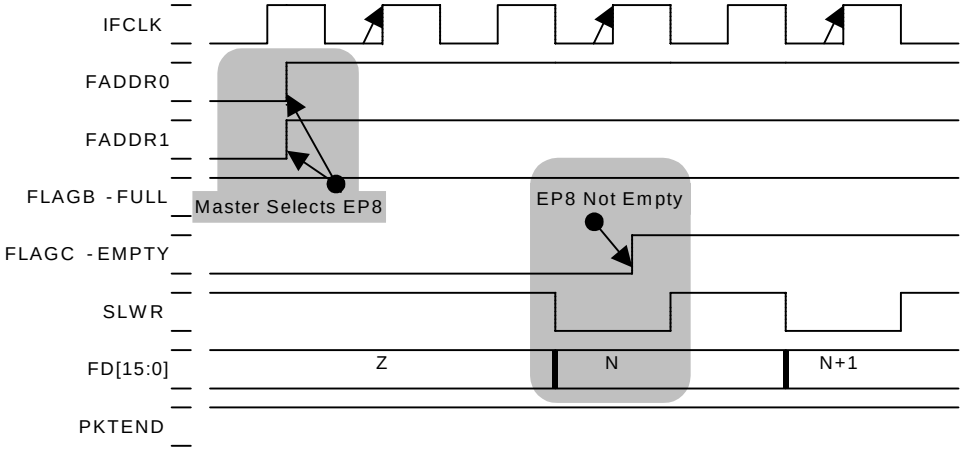


Figure 9-12. Timing Example: Synchronous FIFO Writes, Waveform 1

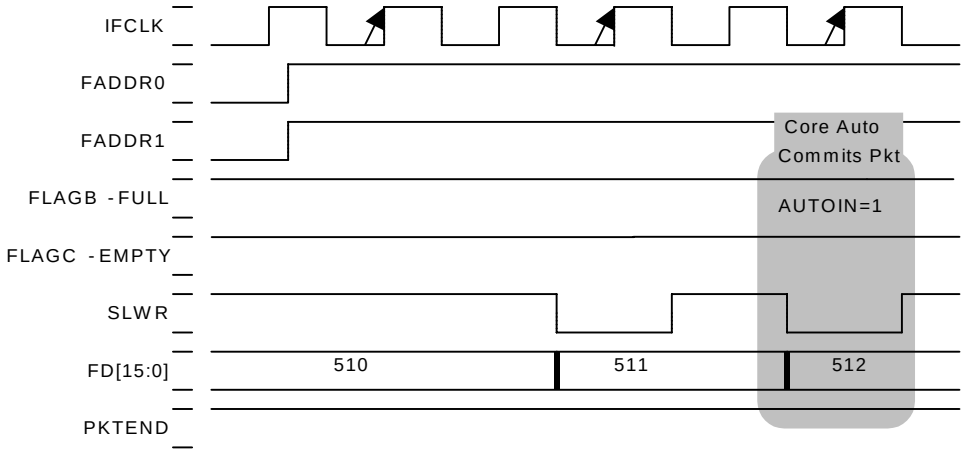


Figure 9-13. Timing Example: Synchronous FIFO Writes, Waveform 2

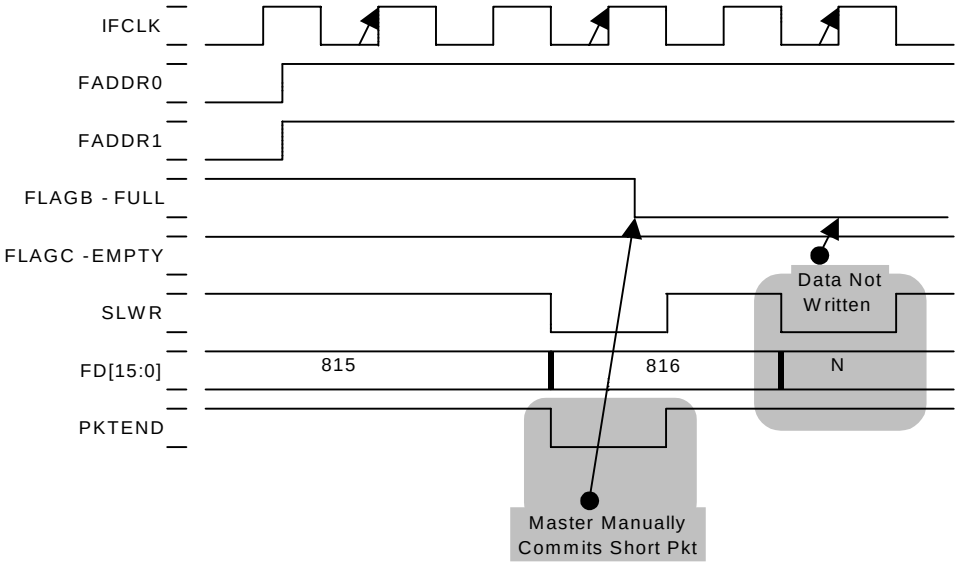


Figure 9-14. Timing Example: Synchronous FIFO Writes, Waveform 3, PKTEND Pin Illustrated

9.2.8 Implementing Synchronous Slave FIFO Reads

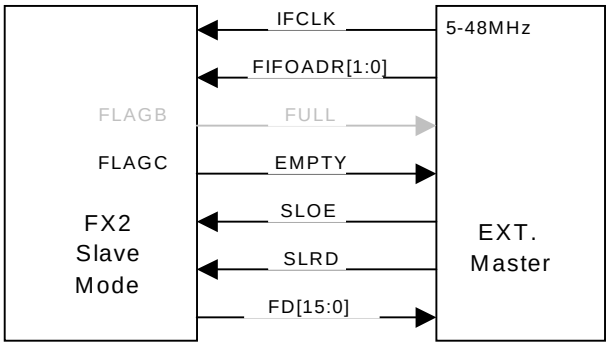


Figure 9-15. Interface Pins Example: Synchronous FIFO Reads

Typically, the sequence of events for the external master is:

IDLE: When read event occurs, transition to State 1.

STATE 1: Point to OUT FIFO, assert FIFOADR[1:0], transition to State 2.

STATE 2: Assert SLOE. If FIFO-Empty flag is false (FIFO not empty), transition to State 3 else remain in State 2.

STATE 3: Sample data on the bus, increment pointer by asserting SLRD for one IFCLK, de-assert SLOE, transition to State 4.

STATE 4: If more data to read, transition to State 2 else transition to IDLE.

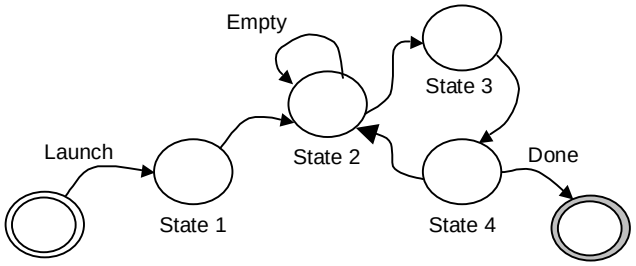


Figure 9-16. State Machine Example: Synchronous FIFO Reads

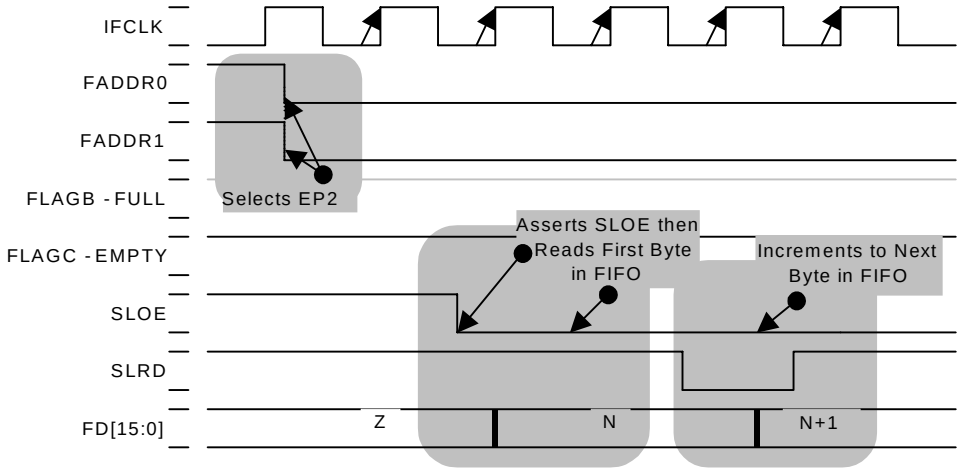


Figure 9-17. Timing Example: Synchronous FIFO Reads, Waveform 1

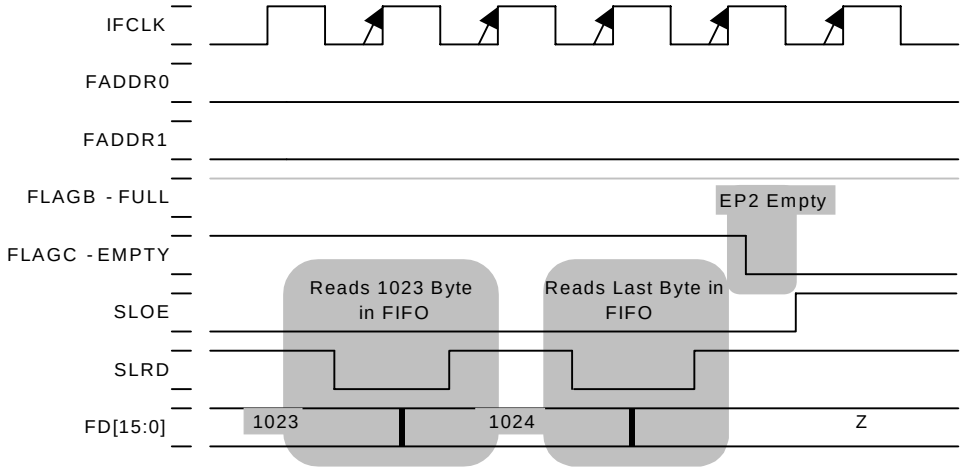


Figure 9-18. Timing Example: Synchronous FIFO Reads, Waveform 2, EMPTY Flag Illustrated

9.2.9 Implementing Asynchronous Slave FIFO Writes

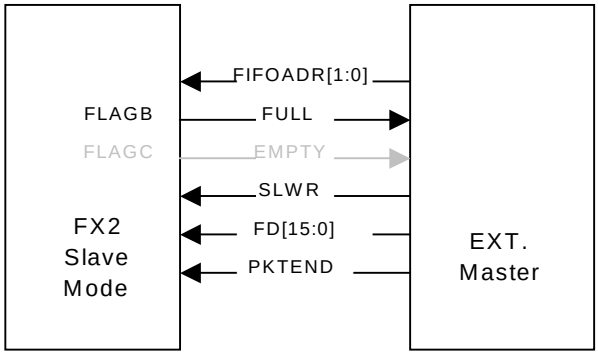


Figure 9-19. Interface Pins Example: Asynchronous FIFO Writes

Typically, the sequence of events for the external master is:

IDLE: When write event occurs, transition to State 1.

STATE 1: Point to IN FIFO, assert FIFOADR[1:0], transition to State 2.

STATE 2: If FIFO-Full flag is false (FIFO not full), transition to State 3 else remain in State 2.

STATE 3: Drive data on the bus, increment pointer by asserting then de-asserting SLWR, transition to State 4.

STATE 4: If more data to write, transition to State 2 else transition to IDLE.

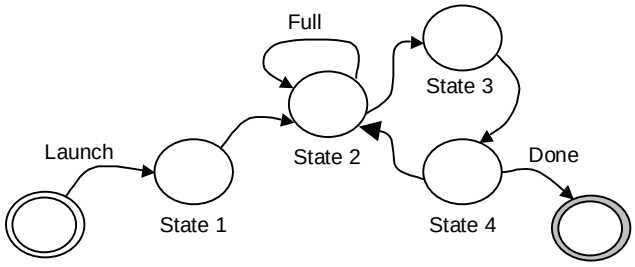


Figure 9-20. State Machine Example: Asynchronous FIFO Writes

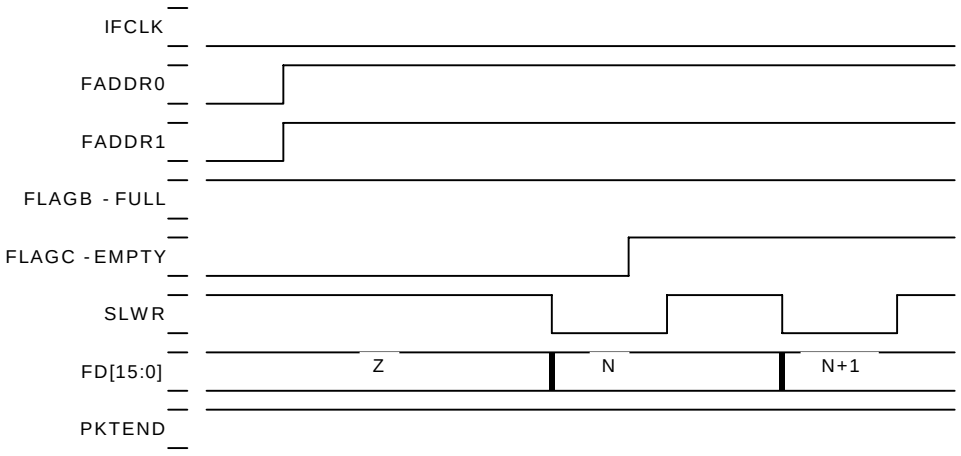


Figure 9-21. Timing Example: Asynchronous FIFO Writes

9.2.10 Implementing Asynchronous Slave FIFO Reads

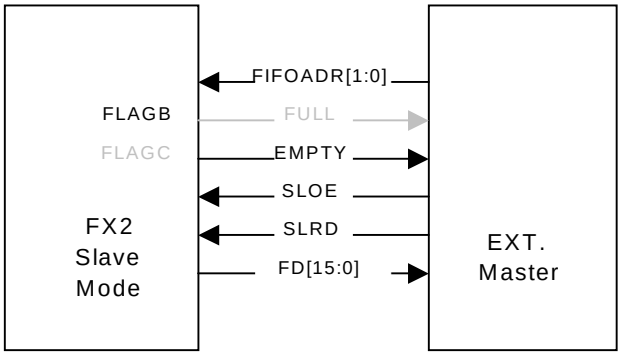


Figure 9-22. Interface Pins Example: Asynchronous FIFO Reads

Typically, the sequence of events for the external master is:

IDLE: When read event occurs, transition to State 1.

STATE 1: Point to OUT FIFO, assert FIFOADR[1:0], transition to State 2.

STATE 2: If Empty flag is false (FIFO not empty), transition to State 3 else remain in State 2.

STATE 3: Assert SLOE, assert SLRD, sample data on the bus, de-assert SLRD (increment pointer), de-assert SLOE, transition to State 4.

STATE 4: If more data to read, transition to State 2 else transition to IDLE.

.

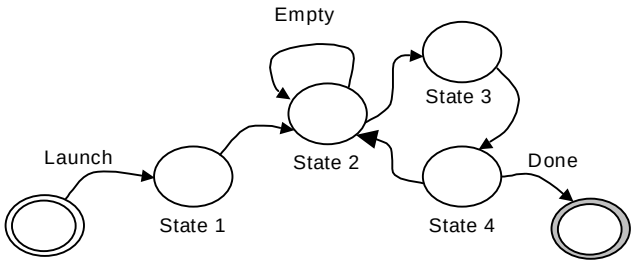


Figure 9-23. State Machine Example: Asynchronous FIFO Reads

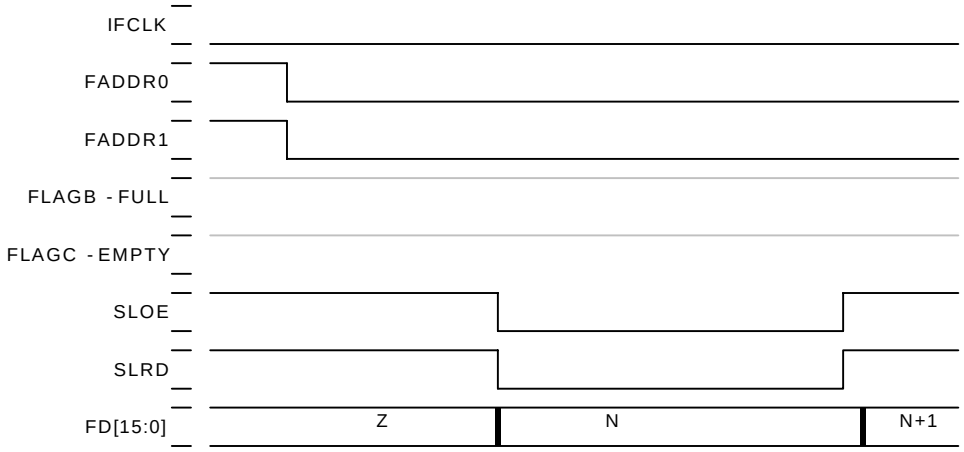


Figure 9-24. Timing Example: Asynchronous FIFO Reads

9.3 Firmware

This section describes the interface between FX2 firmware and the FIFOs. More information is available in *Chapter 8, "Access to Endpoint Buffers."*

Table 9-3. Registers Associated with Slave FIFO Firmware

EPxCFG	INPKTEND
EPxFIFOCFG	EPxFIFOIE
EPxAUTOINLENH/L	EPxFIFOIRQ
EPxFIFOPFH:L	INT2IVEC
EP2468STAT	INT4IVEC
EP24FIFOFLGS	INTSETUP
EP68FIFOFLGS	IE
EPxCS	IP
EPxFIFOFLGS	INT2CLR
EPxBCH:L	INT4CLR
EPxFIFOBCH:L	EIE
EPxFIFOBUF	EXIF
REVCTL (bits 0 and 1 *must* be initialized to 1 for operation as described in this chapter)	

9.3.1 Firmware FIFO Access

FX2 firmware can access the slave FIFOs using four registers in XDATA memory: EP2FIFOBUF, EP4FIFOBUF, EP6FIFOBUF, and EP8FIFOBUF. These registers can be read and written directly (using the MOVX instruction), or they can serve as sources and destinations for the dual Auto-pointer mechanism built into the EZ-USB FX2 (see Section 8.8. *"Autopointers"*).

Additionally, there are a number of FIFO control and status registers: Byte Count registers indicate the number of bytes in each FIFO; flag bits indicate FIFO fullness, mode bits control the various FIFO modes, etc.

This chapter focuses on the registers and bits which are specific to slave-FIFO operation; for a fuller description of all the FIFO registers, see *Chapter 8 "Access to Endpoint Buffers"* and *Chapter 15, "Registers."*

*For proper operation as described in this chapter, FX2 firmware **must** set the DYN_OUT and ENH_PKT bits (REVCTL.0 and REVCTL.1) to 1.*

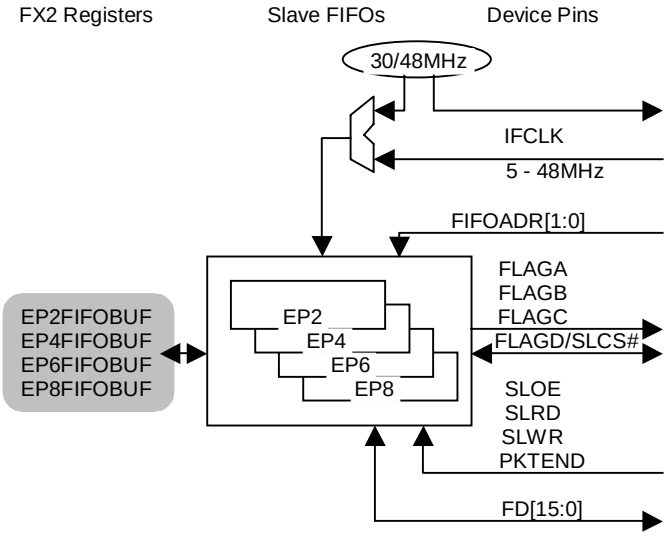


Figure 9-25. EPxFIFOBUF Registers

9.3.2 EPx Memories

The slave FIFOs connect external logic to the FX2's four endpoint memories (EP2, EP4, EP6, and EP8). These endpoint memories have the following programmable features:

1. Type can be either BULK, INTERRUPT, or ISOCHRONOUS.
2. Direction can be either IN or OUT.
3. For EP2 and EP6, size can be either 512 or 1024 bytes. EP4 and EP8 are fixed at 512 bytes.
4. Buffering can be 2x, 3x, or 4x for EP2 and EP6. EP4 and EP8 are fixed at 2x.
5. FX2 automatically commits endpoint data to and from the slave FIFO interface (AUTOIN=1, AUTOOUT=1).

At power-on-reset, these endpoint memories are configured as follows:

1. EP2 - Bulk OUT, 512 bytes/packet, 2x buffered.
2. EP4 - Bulk OUT, 512 bytes/packet, 2x buffered.
3. EP6 - Bulk IN, 512 bytes/packet, 2x buffered.
4. EP8 - Bulk IN, 512 bytes/packet, 2x buffered.

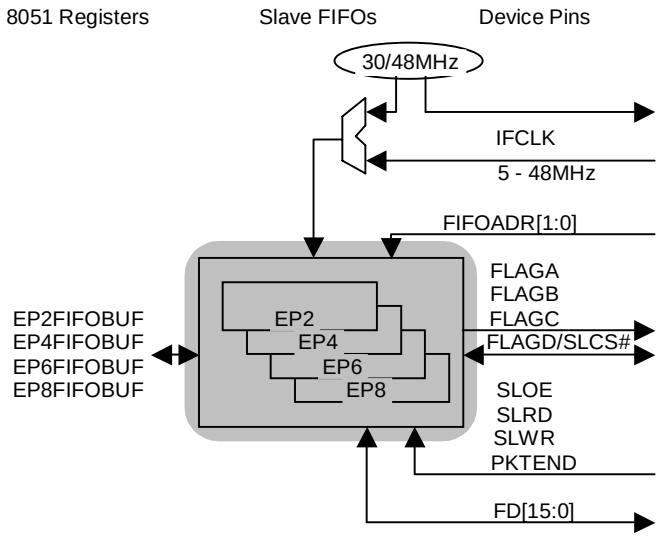


Figure 9-26. EPx Memories

9.3.3 Slave FIFO Programmable-Level Flag (PF)

Each FIFO's programmable-level flag (PF) asserts when the FIFO reaches a user-defined fullness threshold. That threshold is configured as follows:

1. For OUT packets: The threshold is stored in PFC12:0. The PF is asserted when the number of bytes *in the entire FIFO* is less than/equal to (DECIS=0) or greater than/equal to (DECIS=1) the threshold.
2. For IN packets, with PKTSTAT = 1: The threshold is stored in PFC9:0. The PF is asserted when the number of bytes written into *the current packet in the FIFO* is less than/equal to (DECIS=0) or greater than/equal to (DECIS=1) the threshold.
3. For IN packets, with PKTSTAT = 0: The threshold is stored in two parts: PKTS2:0 holds the number of committed packets, and PFC9:0 holds the number of bytes in the current packet. The PF is asserted when the FIFO is at or less full than (DECIS=0), or at or more full than (DECIS=1), the threshold.

By default, FLAGA is the Programmable-Level Flag (PF) for the endpoint currently pointed to by the FIFOADR[1:0] pins. For EP2 and EP4, the default endpoint configuration is BULK, OUT, 512, 2x, and the PF pin asserts when the entire FIFO has greater than/equal to 512 bytes. For EP6 and EP8, the default endpoint configuration is BULK, IN, 512, 2x, and the PF pin asserts when the entire FIFO has less than/equal to 512 bytes.

In other words, the default-configuration PFs for EP2 and EP4 assert when the FIFOs are half-full, and the default-configuration PFs for EP6 and EP8 assert when those FIFOs are half-empty.

See Chapter 15, "Registers," for full details.

9.3.4 Auto-In / Auto-Out Modes

The FX2 FIFOs can be configured to commit packets to/from USB automatically. For IN endpoints, Auto-In Mode allows the external logic to stream data into a FIFO continuously, with no need for it or the FX2 firmware to packetize the data or explicitly signal the FX2 to send it to the host. For OUT endpoints, Auto-Out Mode allows the host to continuously fill a FIFO, with no need for the external logic or FX2 firmware to handshake each incoming packet, arm the endpoint buffers, etc. See Figure 9-27.

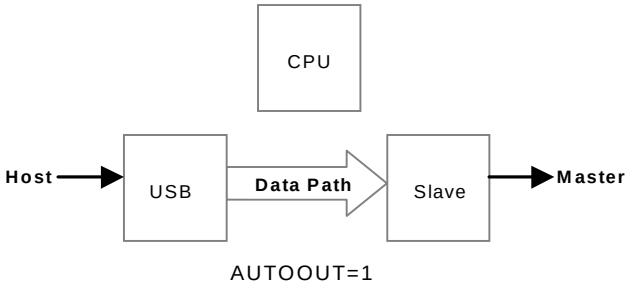


Figure 9-27. When AUTOOUT=1, OUT Packets are Automatically Committed

To configure an IN endpoint FIFO for Auto Mode, set the AUTOIN bit in the appropriate EPxFIFOCFG register to 1. To configure an OUT endpoint FIFO for Auto Mode, set the AUTOOUT bit in the appropriate EPxFIFOCFG register to 1. See Figures 9-28 and 9-29.

At power-on reset, all FIFOs default to Manual Mode (i.e., AUTOIN = 0 and AUTOOUT = 0).

```
TD_Init():
… … … … …
REVCTL = 0x03;       // MUST set REVCTL.0 and REVCTL.1 to 1
SYNCDELAY;
EP2CFG = 0xA2;       // EP2 is DIR=OUT, TYPE=BULK, SIZE=512, BUF=2x
SYNCDELAY;
FIFORESET = 0x80;    // Reset the FIFO
SYNCDELAY;
FIFORESET = 0x02;
SYNCDELAY;
FIFORESET = 0x00;
SYNCDELAY;
EP2FIFOCFG = 0x10;   // EP2 is AUTOOUT=1, AUTOIN=0, ZEROLEN=0, WORDWIDE=0
SYNCDELAY;
OUTPKTEND = 0x82;    // Arm both EP2 buffers to “prime the pump”
SYNCDELAY;
OUTPKTEND = 0x82;
… … … … …
```

Figure 9-28. TD_Init Example: Configuring AUTOOUT = 1

```
TD_Init():
… … … … …
REVCTL = 0x03;       // MUST set REVCTL.0 and REVCTL.1 to 1
SYNCDELAY;
SYNCDELAY;
EP8CFG = 0xE0;       // EP8 is DIR=IN, TYPE=BULK
SYNCDELAY;
FIFORESET = 0x80;    // Reset the FIFO
SYNCDELAY;
FIFORESET = 0x08;
SYNCDELAY;
FIFORESET = 0x00;
SYNCDELAY;
EP8FIFOCFG = 0x0C;   // EP8 is AUTOOUT=0, AUTOIN=1, ZEROLEN=1, WORDWIDE=0
SYNCDELAY;
EP8AUTOINLENH = 0x02;  // Auto-commit 512-byte packets
SYNCDELAY;
EP8AUTOINLENL = 0x00;
… … … … …
```

Figure 9-29. TD_Init Example: Configuring AUTOIN = 1

9.3.5 CPU Access to OUT Packets, AUTOOUT = 1

The FX2's CPU is not in the host-to-master data path when AUTOOUT = 1.To achieve the maximum USB 2.0 bandwidth, the host and master are directly connected, bypassing the CPU. Figure 9-30 shows that, in Auto-Out mode, data from the host is automatically committed to the FIFOs with no firmware intervention.

```
TD_Poll():
… … … … …
// no code necessary to xfr data from host to master!
// AUTOOUT=1 and SIZE=0 auto-commits packets
// in 512 byte chunks.
… … … … …
```

Figure 9-30. TD_Poll Example: No Code Necessary for OUT Packets When AUTOOUT=1

9.3.6 CPU Access to OUT Packets, AUTOOUT = 0

In some systems, it may be desirable to allow the FX2's CPU to participate in the transfer of data between the host and the slave FIFOs. To configure a FIFO for this "Manual-Out" mode, the AUTOOUT bit in the appropriate EPxFIFOCFG register must be cleared to 0 (see Figure 9-31).

```
TD_Init():
… … … … …
REVCTL = 0x03;       // MUST set REVCTL.0 and REVCTL.1 to 1
SYNCDELAY;
EP2CFG = 0xA2;       // EP2 is DIR=OUT, TYPE=BULK, SIZE=512, BUF=2x
SYNCDELAY;
FIFORESET = 0x80;    // Reset the FIFO
SYNCDELAY;
FIFORESET = 0x02;
SYNCDELAY;
FIFORESET = 0x00;
SYNCDELAY;
EP2FIFOCFG = 0x00;   // EP2 is AUTOOUT=0, AUTOIN=0, ZEROLEN=0, WORDWIDE=0
SYNCDELAY;
OUTPKTEND = 0x82;    // Arm both EP2 buffers to "prime the pump"
SYNCDELAY;
OUTPKTEND = 0x82;
… … … … …
```

Figure 9-31. TD_Init Example, Configuring AUTOOUT=0

As Illustrated in Figure 9-32, FX2 firmware can do one of three things when the FX2 is in Manual-Out mode and a packet is received from the host:

1. It can *commit* (pass to the FIFOs) the packet by writing OUTPKTEND with SKIP=0 (Figur e9-33).
2. It can *skip* (discard) the packet by writing OUTPKTEND with SKIP=1 (Figur e9-34).
3. It can *edit* the packet (or *source* an entire OUT packet) by writing to the FIFO buffer directly, then writing the length of the packet to EPxBCH:L. The write to EPxBCL commits the edited packet, so EPxBCL should be written *after* writing EPxBCH (Figure9-35).

In all cases, the OUT buffer automatically re-arms so it can receive the next packet.

See Section 8.6.2.4 for a detailed description of the SKIP bit.

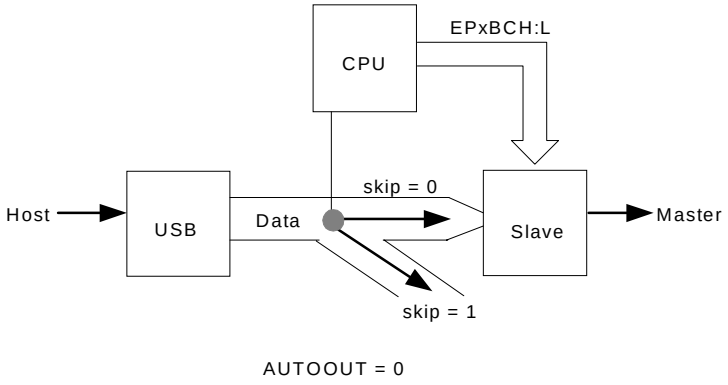


Figure 9-32. Skip, Commit, or Source (AUTOOUT=0)

```
TD_Poll():
… … … … …
if( !( EP2468STAT & 0x01 ) )
{ // EP2EF=0 when FIFO NOT empty, host sent packet
  OUTPKTEND = 0x02;  // SKIP=0, pass buffer on to master
}
… … … … …
```

Figure 9-33. TD_Poll Example, AUTOOUT=0, Commit Packet

```
TD_Poll():
… … … … …
if( !( EP2468STAT & 0x01 ) )
{ // EP2EF=0 when FIFO NOT empty, host sent packet
  OUTPKTEND = 0x82;  // SKIP=1, do NOT pass buffer on to master
}
… … … … …
```

Figure 9-34. TD_Poll Example, AUTOOUT=0, Skip Packet

```
TD_Poll():
… … … … …
if( EP24FIFOFLGS & 0x02 )
{
SYNCDELAY;                //
FIFORESET = 0x80;         // nak all OUT pkts. from host
SYNCDELAY;                //
FIFORESET = 0x02;         // advance all EP2 buffers to cpu domain
SYNCDELAY;                //
EP2FIFOBUF[0] = 0xAA;     // create newly sourced pkt. data
SYNCDELAY;                //
EP2BCH = 0x00;
SYNCDELAY;                //
EP2BCL = 0x01;            // commit newly sourced pkt. to interface fifo

// beware of "left over" uncommitted buffers

SYNCDELAY;                //
OUTPKTEND = 0x82;         // skip uncommitted pkt. (second pkt.)
// note: core will not allow pkts. to get out of sequence
SYNCDELAY;                //
FIFORESET = 0x00;         // release "nak all"
}
… … … … …
```

Figure 9-35. TD_Poll Example, AUTOOUT=0, Source

*If an uncommitted packet is in an OUT endpoint buffer when the FX2 is reset, that packet is **not** automatically committed to the master. To ensure that no uncommitted packets are in the endpoint buffers after a reset, the FX2 firmware's "endpoint initialization" routine should skip 2, 3, or 4 packets (depending on the buffering depth selected for the FIFO) by writing OUTPKTEND with SKIP=1. See Figure 9-36*.

```
TD_Init():
… … … … …
REVCTL = 0x03;      // MUST set REVCTL.0 and REVCTL.1 to 1
SYNCDELAY;
SYNCDELAY;
EP2CFG = 0xA2;      // EP2 is DIR=OUT, TYPE=BULK, SIZE=512, BUF=2x
SYNCDELAY;
EP2FIFOCFG = 0x00;  // EP2 is AUTOOUT=0, AUTOIN=0, ZEROLEN=0, WORDWIDE=0

// OUT endpoints do NOT come up armed
SYNCDELAY;
OUTPKTEND = 0x82;   // arm first buffer by writing OUTPKTEND w/skip=1
SYNCDELAY;
OUTPKTEND = 0x82;  // arm second buffer by writing OUTPKTEND w/skip=1
… … … … …
```

Figure 9-36. TD_Init Example, OUT Endpoint Initialization

9.3.7 CPU Access to IN Packets, AUTOIN = 1

Auto-In mode is similar to Auto-Out mode: When an IN FIFO is configured for Auto-In mode (by setting its AUTOIN bit to 1), data from the master is automatically packetized and committed to USB without any CPU intervention (see Figure 9-37).

```
TD_Poll():
… … … … …
// no code necessary to xfr data from master to host!
// AUTOIN=1 and EP8AUTOINLEN=512 auto commits packets
// in 512 byte chunks.
… … … … …
```

Figure 9-37. TD_Poll Example, AUTOIN = 1

Auto-In mode differs in one important way from Auto-Out mode: In Auto-Out mode, data (excluding data in short packets) is always auto-committed in 512- or 1024-byte packets; in Auto-In mode, the auto-commit packet size may be set to *any non-zero value* (with the single restriction, of course, that the packet size must be less than or equal to the size of the endpoint buffer). Each FIFO's Auto-In packet size is stored in its EPxAUTOINLENH:L register pair.

To *source* an IN packet, FX2 firmware can temporarily halt the flow of data from the external master (via a signal on a general-purpose I/O pin, typically), wait for an endpoint buffer to become available, create a new packet by writing directly to that buffer, then commit the packet to USB and release the external master. In this way, the firmware can insert its own packets in the data stream. See Figure 9-38, which illustrates data flowing directly between the master and the host, and Figure 9-39, which shows the firmware sourcing an IN packet. A firmware example appears in Figure 9-40.

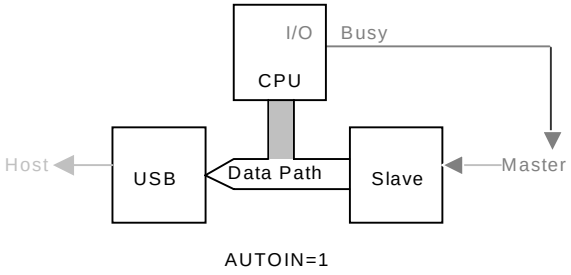


Figure 9-38. Master Writes Directly to Host, AUTOIN = 1

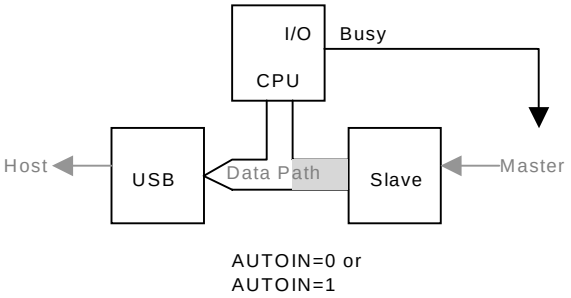


Figure 9-39. Firmware Intervention, AUTOIN = 0 or 1

```
TD_Poll():
… … … … …
if( source_pkt_event )
{  // 100-msec background timer fired
  if( holdoff_master( ) )
  {  // signaled “busy” to master successful
    while( !( EP68FIFOFLGS & 0x20 ) )
    {  // EP8EF=0, when buffer not empty
      ;  // wait ‘til host takes entire FIFO data
    }

    FIFORESET = 0x80;  // initiate the “source packet” sequence
    SYNCDELAY;
    FIFORESET = 0x06;
    SYNCDELAY;
    FIFORESET = 0x00;

    EP8FIFOBUF[ 0 ] = 0x02;  // <STX>, packet start of text msg
    EP8FIFOBUF[ 1 ] = 0x06;  // <ACK>
    EP8FIFOBUF[ 2 ] = 0x07;  // <HEARTBEAT>
    EP8FIFOBUF[ 3 ] = 0x03;  // <ETX>, packet end of text msg

    SYNCDELAY;
    EP8BCH = 0x00;
    SYNCDELAY;
    EP8BCL = 0x04;  // pass newly-sourced buffer on to host
  }
  else
  {
    history_record( EP8, BAD_MASTER );
  }
}
… … … … …
```

Figure 9-40. TD_Poll Example: Sourcing an IN Packet

9.3.8 Access to IN Packets, AUTOIN=0

In some systems, it may be desirable to allow the FX2's CPU to participate in every data-transfer between the external master and the host. To configure a FIFO for this "Manual-In" mode, the AUTOIN bit in the appropriate EPxFIFOCFG register must be cleared to 0.

In Manual-In mode, FX2 firmware can commit, skip, or edit packets sent by the external master, and it may also source packets directly. To commit a packet, firmware writes the endpoint number (with SKIP=0) to the INPKTEND register. To skip a packet, firmware writes the endpoint number with SKIP=1 to the INPKTEND register. To edit or source a packet, firmware writes to the FIFO buffer, then writes the packet length to EPxBCH and EPxBCl (in that order).

```
TD_Poll():
… … … … …
if( master_finished_longxfr( ) )
{  // master currently points to EP8, pins FIFOADR[1:0]=11
  if( !( EP68FIFOFLGS & 0x10 ) )
  { // EP8FF=0 when buffer available
    INPKTEND = 0x08;  // firmware commits EP8 packet
                      // by writing 8 to INPKTEND
    release_master( EP8 );
  }
}
… … … … …
```

Figure 9-41. TD_Poll Example, AUTOIN=0, Committing a Packet via INPKTEND

```
TD_Poll():
… … … … …
if( master_finished_longxfr( ) )
{  // master currently points to EP8, pins FIFOADR[1:0]=11
  if( !( EP68FIFOFLGS & 0x10 ) )
  { // EP8FF=0 when buffer available
    INPKTEND = 0x88;  // firmware skips EP8 packet
                      // by writing 0x88 to INPKTEND
    release_master( EP8 );
  }
}
… … … … …
```

Figure 9-42. TD_Poll Example, AUTOIN=0, Skipping a Packet via INPKTEND

```
TD_Poll():
… … … … …
if( master_finished_xfr( ) )
{  // modify the data
  EP8FIFOBUF[ 0 ] = 0x02;  // <STX>, packet start of text msg
  EP8FIFOBUF[ 7 ] = 0x03;  // <ETX>, packet end of text msg
  SYNCDELAY;
  EP8BCH = 0x00;
  SYNCDELAY;
  EP8BCL = 0x08;  // pass buffer on to host
}
… … … … …
```

Figure 9-43. TD_Poll Example, AUTOIN=0, Editing a Packet via EPxBCH:L

9.3.9 Auto-In / Auto-Out Initialization

Enabling Auto-In transfers between slave FIFO and endpoint

Typically, a FIFO is configured for Auto-In mode as follows:

1. Configure bits IFCONFIG[7:4] to define the behavior of the interface clock.
2. Set bits IFCFG1:0=11.
3. Reset the FIFOs.
4. Set bit EPxFIFOCFG.3=1.
5. Set the size via the EPxAUTOINLENH:L registers.

Enabling Auto-Out transfers between endpoint and slave FIFO

Typically, a FIFO is configured for Auto-Out mode as follows:

1. Configure bits IFCONFIG[7:4] to define the behavior of the interface clock.
2. Set bits IFCFG1:0=11.
3. Reset the FIFOs.
4. Set bit EPxFIFOCFG.4=1.

9.3.10 Auto-Mode Example: Synchronous FIFO IN Data Transfers

```
TD_Init():
REVCTL = 0x03;   // MUST set REVCTL.0 and REVCTL.1 to 1
SYNCDELAY;
FIFORESET = 0x80;  // reset all FIFOs
SYNCDELAY;
FIFORESET = 0x02;
SYNCDELAY;
FIFORESET = 0x04;
SYNCDELAY;
FIFORESET = 0x06;
SYNCDELAY;
FIFORESET = 0x08;
SYNCDELAY;
FIFORESET = 0x00;
SYNCDELAY;         // this defines the external interface to be the following:
IFCONFIG = 0x43;   // use IFCLK pin driven by external logic (5MHz to 48MHz)
                  // use slave FIFO interface pins driven sync by external master
EP8FIFOCFG = 0x0C; // this lets the FX2 auto commit IN packets, gives the
                   // ability to send zero length packets,
                   // and sets the slave FIFO data interface to 8-bits
EP8CFG = 0xE0;     // sets EP8 valid for IN's
                   // and defines the endpoint for 512 byte packets, 2x buffered
PINFLAGSAB = 0x00; // defines FLAGA as prog-level flag, pointed to by FIFOADR[1:0]
SYNCDELAY;         // FLAGB as full flag, as pointed to by FIFOADR[1:0]
PINFLAGSCD = 0x00; // FLAGC as empty flag, as pointed to by FIFOADR[1:0]
                   // won't generally need FLAGD

PORTACFG = 0x00;   // used PA7/FLAGD as a port pin, not as a FIFO flag
FIFOPINPOLAR = 0x00; // set all slave FIFO interface pins as active low

SYNCDELAY;
EP8AUTOINLENH = 0x02; // you can define these as you wish,
SYNCDELAY;            // to have the FX2 automatically limit IN's
EP8AUTOINLENL = 0x00;

SYNCDELAY;
EP8FIFOPFH = 0x82; // you can define the programmable flag (FLAGA)
SYNCDELAY;         // to be active at the level you wish
EP8FIFOPFL = 0x00;

SYNCDELAY;         // out endpoints do not POR (power-on reset) armed
EP2BCL = 0x80;     // since the defaults are double buffered we must
SYNCDELAY;         // write dummy byte counts twice
EP2BCL = 0x80;     // arm EP2OUT & EP4OUT by writing to the byte count w/skip.
SYNCDELAY;
EP4BCL = 0x80;
SYNCDELAY;
EP4BCL = 0x80;
```

```
TD_Poll():
// nothing! The FX2 is doing all the work of transferring packets
// from the external master sync interface to the endpoint buffer...
```

Figure 9-44. Code Example, Synchronous Slave FIFO IN Data Transfer

9.3.11 Auto-Mode Example: Asynchronous FIFO IN Data Transfers

The initialization code is exactly the same as for the synchronous-transfer example in Section 9.3.10, but with IFCLK configured for internal use at a rate of 48 MHz and the ASYNC bit set to 1. Figure 9-45 shows the one-line modification that's needed.

```
TD_Init( ):  // slight modification from our synchronous firmware example
IFCONFIG = 0xCB;
// this defines the external interface as follows:
// use internal IFCLK (48MHz)
// use slave FIFO interface pins asynchronously to external master
```

Figure 9-45. TD_Init Example, Asynchronous Slave FIFO IN Data Transfers

Code to perform the transfers is, as before, unnecessary; as Figure 9-46 illustrates.

```
TD_Poll( ):
// nothing!  The FX2 is doing all the work of transferring packets
// from the external master async interface to the endpoint buffer…
```

Figure 9-46. TD_Poll Example, Asynchronous Slave FIFO IN Data Transfers

9.4 Switching Between Manual-Out and Auto-Out

Because OUT endpoints are not automatically armed when the FX2 enters Auto-Out mode, the firmware can safely switch the FX2 between Manual-Out and Auto-Out modes without any need to flush or reset the FIFOs.