Endpoint FIFO Architecture of EZ-USB FX1/FX2™

Abstract

This application note describes the FIFO architecture of the EZ-USB FX1™, the full speed USB microcontroller and the EZ-USB FX2™ (and FX2LP™), the high-speed USB microcontroller. The purpose of this application note is to help the user understand the very basics of the FX1/FX2 and get familiar with the terminologies used while describing the data flow in FX1/FX2. The application note addresses and discusses the following:

- Three modes of operation of the FX1/FX2
- Endpoint Configuration and Multiple Buffering
- Three Domains that form the basic component of the FIFO architecture
- Arming and committing endpoint buffers
- Endpoint operation in manual vs. auto mode

The application note covers the various domains of the FIFO architecture, improving data rate using the multiple buffering scheme, using the part in port I/O, slave FIFO or the GPIF mode. It also provides information about the use of the endpoint FIFOs in manual or auto mode. Furthermore, terminologies like 'arming' and 'committing' endpoint packets when referring to an IN/OUT transfer are explained.

It is not the intention of this application note to illustrate usage of the part in either of operational modes. The application note does not cover details on setting up for data transfers. After having reviewed this application note, the reader is expected to understand the data flow for and IN and OUT transfer and the interaction between the various domains of the FX1/FX2 FIFO architecture that define the data paths via IN and OUT endpoint.

In this application note we will only refer to the FX1. The same terminologies and concept is also used in the FX2. The FX2 has identical FIFO architecture and the same number of endpoints as the FX1.

Modes of Operation

Although some FX1-based devices may use the FX1's CPU to process USB data directly (*Port I/O Mode*), most applications use the FX1 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 FX1's CPU* — through the FX1'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 FX1's General Programmable Interface (GPIF) can act as an *internal* master when the FX1 is connected to external logic which doesn't include a standard FIFO interface (GPIF mode). The FIFOs can also be controlled by an external master. When the FIFOs are controlled by an external master, the FX1 is said to be in "Slave FIFO". The FX1 can be set in either one of three modes: Port I/O mode, GPIF mode or the slave FIFO mode. The mode can be set by setting bits 0 and 1 of the IFCONFIG register. Further information on the IFCONFIG register setting can be found in section 15.5.2 of the Technical Reference Manual. On power-up the part defaults to Ports mode: the IFCONFIG register bits 0 and 1 default to 00b.

Ports Mode

In this mode IFCONFIG[1..0] is set to 00b. The FX1 device can be used as a data sink or a data source. The 8051 may fill the IN endpoint with data and have the core send the packet to the host. The 8051 may access and process the data packet received by the core when the host issues an OUT token. In this port mode of operation, no peripheral device is wired to the FX1. 8051 can source data on an IN as well as OUT endpoint. Finally in this Ports I/O mode all the port I/O pins are available for general purpose I/O.

GPIF Mode

In this mode IFCONFIG[1..0] is set to 10b. The GPIF state machine is the master to the FIFO's. In GPIF mode, some of the port pins are not available for general purpose usage as they are dedicated to the GPIF engine. The external peripheral is a slave to the FX1. GPIF will read/write from /to the external peripheral which is wired to the FX1 device via PORTB and (if 16 bit bus width) PORTD. Hence PORTB and (if 16 bit bus width) PORTD are not available for general purpose usage in the GPIF mode of operation. PORTB and PORTD are dedicated to function as GPIF data bus, FD[15..0].

Slave FIFO Mode

In this mode IFCONFIG[1..0] is set to 11b. The endpoint FIFOs are slave to the external peripheral device wired to the FX1. In slave FIFO mode, some of the port pins are not available for general purpose usage as they are dedicated to the slave FIFO control signals. The slave FIFO control signals SLWR, SLRD, SLOE, SLCS, PKTEND, FIFOADR0 and FIFOADR1 are inputs to the FX1 device, whereas FLAGA, FLAGB, FLAGC, FLAGD, are outputs to the peripheral indicating the status of the FIFO. The external peripheral wired to the FX1 is the master and will access the FIFOs via PORTB and (16 bit bus width) PORTD. Hence PORTB and (if 16 bit bus width) PORTD are not available for general purpose usage. PORTB and PORTD are dedicated to function as FIFO data bus, FD[15..0].

The following table shows the different port pin functionality in the three different modes. Functionality of the pins shown in bold type in *Table 1* do not change with IFCONFIG register setting. They are shown in this table for completeness.

Table 1. FX1 Interface Signals

IFCONFIG[1..0] 00b Ports Mode	IFCONFIG[1..0] 01b GPIF Mode	IFCONFIG[1..0] 11b Slave FIFO Mode
PD7	FD[15]	FD[15]
PD6	FD[14]	FD[14]
PD5	FD[13]	FD[13]
PD4	FD[12]	FD[12]
PD3	FD[11]	FD[11]
PD2	FD[10]	FD[10]
PD1	FD[9]	FD[9]
PD0	FD[8]	FD[8]
PB7	FD[7]	FD[7]
PB6	FD[6]	FD[6]
PB5	FD[5]	FD[5]
PB4	FD[4]	FD[4]
PB3	FD[3]	FD[3]
PB2	FD[2]	FD[2]
PB1	FD[1]	FD[1]
PB0	FD[0]	FD[0]
unused	RDY0	SLRD
unused	RDY1	SLWR
unused	CTL0	FLAGA
unused	CTL1	FLAGB
unused	CTL2	FLAGC
INT#0/PA0	**INT#0/PA0**	**INT#0/PA0**
INT1#/PA1	**INT1#/PA1**	**INT1#/PA1**
PA2	PA2	SLOE
WU2/PA3	**WU2/PA3**	**WU2/PA3**
PA4	PA4	FIFOADR0
PA5	PA5	FIFOADR1
PA6	PA6	PKTEND
PA7	PA7	PA7/FLAGD

Figure 1 below shows the three modes of operation of the FX1. This figure shows the main logic blocks of the FX1. In general, usually a commercial application would either use the slave FIFO mode or the GPIF mode to transfer data between the host and the external peripheral device wired to the FX1. In "Ports" mode, all the I/O pins are general purpose I/O ports. The GPIF state machine and the slave FIFO logic although existing, they are not 'actively' used when in Port I/O mode. "GPIF master" mode and "Slave FIFO mode" uses the PORTB and PORTD pins as a 16-bit data interface to the four FX1 endpoint FIFOs EP2, EP4, EP6 and EP8. If using the FX1 as a standalone device (no external peripheral wired to it), it should be configured to be in ports I/O mode.

Note. EZ-USB® and EZ-USB FX users redesigning the same application with ES-USB FX1.

If an existing full speed application:

- does not have any external peripheral wired to the EZ-USB or the EZ-USB FX, the FX1 may be set in ports I/O mode for implementing the same application.
- has an external peripheral wired to the EZ-USB AND the design does NOT use the fast transfer feature to move data between the external peripheral and the endpoint FIFO, the FX1 may be set in Ports Mode to design the same application.
- has an external peripheral wired to the EZ-USB AND the design uses the fast transfer feature to move data between the external peripheral and the endpoint FIFO, the slave FIFO mode may be used to design the same application using EZ-USB FX1.
- uses the GPIF interface of the EZ-USB FX device, the FX1 in GPIF mode can be used to design the same application
- uses the slave FIFO interface of the EZ-USB FX device, the slave FIFO mode may be used to design the same application using EZ-USB FX1.

The above are suggested ways to redesign the same EZ-USB/FX application with the FX1. There may be several alternate ways to redesign the same EZ-USB/FX application using the FX1

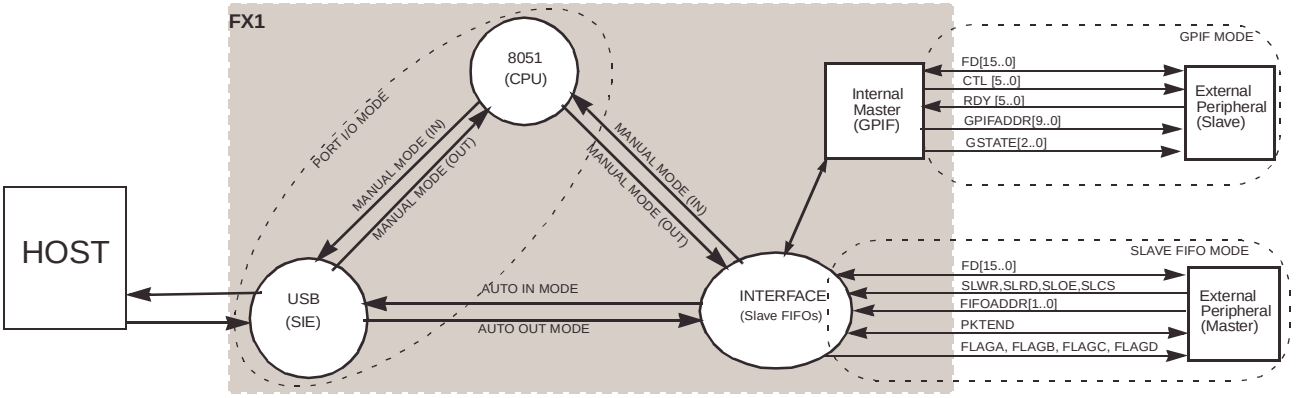

Figure 1. Three Modes of Operation in FX1/FX2

Endpoint Configuration and Multiple Buffering

This section discusses the different endpoints available in the FX1 and how to configure an endpoint. It also provides information on the restrictions to be aware of while configuring and accessing different type of endpoints. Finally the multiple buffering scheme used to maximize throughput is briefly discussed.

Endpoint Configuration

The FX1 can have up to a maximum of seven endpoints as listed in *Table 2* below.

Table 2. Endpoints Available in FX1

Endpoint	Direction	Type	Max Size	Buffering
EP0	IN and OUT bidirectional	Control	64	Single
EP1IN	IN	Bulk/ Interrupt/ Isochronous	64	Single
EP1OUT	OUT	Bulk/ Interrupt/ Isochronous	64	Single
EP2	IN or OUT (configurable)	Bulk/ Interrupt/ Isochronous	512/1024 (configurable)	Double, triple, quad
EP4	IN or OUT (configurable)	Bulk/ Interrupt/ Isochronous	512	Double
EP6	IN or OUT (configurable)	Bulk/ Interrupt/ Isochronous	512/1024 (configurable)	Double, triple, quad
EP8	IN or OUT (configurable)	Bulk/ Interrupt/ Isochronous	512	Double

EP1IN and EP1OUT have a fixed depth of 64 bytes. The external peripheral or the GPIF state machine cannot access these endpoints as FIFO buffers. In slave FIFO mode, SLWR/SLRD signals cannot be used to directly access these buffers.

Endpoints 2, 4, 6 and 8 are the large, high bandwidth (only applicable in FX2 when operating at high speed), data moving endpoints. They can be configured in various ways to suit bandwidth requirements. In slave FIFO mode, these endpoints can be accessed by the external peripheral directly using the slave FIFO control inputs (SLWR, SLRD, SLOE, SLCS, PKTEND). The specific FIFO being addressed is selected by setting FIFOADR[0..1] lines. Further information on these control signals can be found in chapter 9 of the Technical Reference Manual.

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 four large endpoints can be configured by setting the respective EPxCFG register (x= endpoint number). The register bits are defined below.

EP2CFG, EP6CFG

b7	b6	b5	b4	b3	b2	b1	b0
VALID	DIR	TYPE1	TYPE0	SIZE	0	BUF1	BUF0
R/W	R/W	R/W	R/W	R/W	R	R/W	R/W
x	x	x	x	x	x	x	x

EP4CFG, EP8CFG

b7	b6	b5	b4	b3	b2	b1	b0
VALID	DIR	TYPE1	TYPE0	0	0	0	0
R/W	R/W	R/W	R/W	R	R	R	R
x	x	x	x	x	x	x	x

Bit 7: VALID

Set VALID=1 to activate an endpoint, and VALID=0 to de-activate it. All FX2 endpoints default to valid. An endpoint whose VALID bit is 0 does not respond to any USB traffic.

Bit 6: DIR

Defines endpoint direction, 1=IN, 0 = OUT endpoint

Bit[5..4]: TYPE[1..0]

These bits define the endpoint type

0 0 : Invalid

0 1 : Isochronous

1 0 : Bulk (default)

1 1 : Interrupt

Bit 3: SIZE (not defined in EP4CFG and EP8CFG)

Defines endpoint FIFO buffer size, 1=1024, 0 = 512

Bit 2: Reserved. Defaults to 0

Bit[1..0]: BUF[1..0]. (not defined in EP4CFG and EP8CFG)

Defines the Buffering scheme

0 0 Quad Buffered

0 1 Invalid

1 0 Double Buffered

1 1 Triple Buffered

Since EP4 and EP8 have a fixed size of 512 and fixed buffering (double), bits 3 (SIZE), 0 and 1 (BUF0 and BUF1) are unused bits in the EP4CFG and EP8CFG register.

Each of these four large endpoints can only exist as multiple (atleast double) buffered endpoints. By default each endpoint is configured as double buffered 512 bytes deep. The shaded boxes in Figure2 on the following page 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.

In *Figure 2*, the buffer names and the number, 'BufferN' (where N= 1...4) does not have any significance as far as the order of the buffers are concerned. They are marked simply for enumerating the buffers. The 8051 has no knowledge as to which buffer is being addressed. The internal logic selects one of the available buffers and the 8051 can access only this buffer. As explained in section 8.5 (CPU Access to FX1 Endpoint Data) "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."

Actual FIFO Buffer Accessible by 8051

If an endpoint is configured as a bulk endpoint, even though physically the endpoint buffer is 512 bytes deep, the 8051 and the external master must treat it as 64 bytes deep only. As explained earlier, data is transferred in full speed packets only. The endpoints FIFOs are quantum FIFOs. The physical size of each of this quantum FIFO can be set using the SIZE bit in the EPxCFG register. Each full speed data packet is allocated a single quantum FIFO (size specified by the SIZE bit in EPxCFG). The 8051 should limit its FIFO access to a maximum depth that is defined by the smaller of the wMaxPacketSize (reported in the endpoint descriptor) and the size of the FIFO (set by the SIZE bit). For a bulk endpoint this is 64 bytes.

Note. Regardless of the physical buffer size, each endpoint buffer accommodates only one full-speed packet.

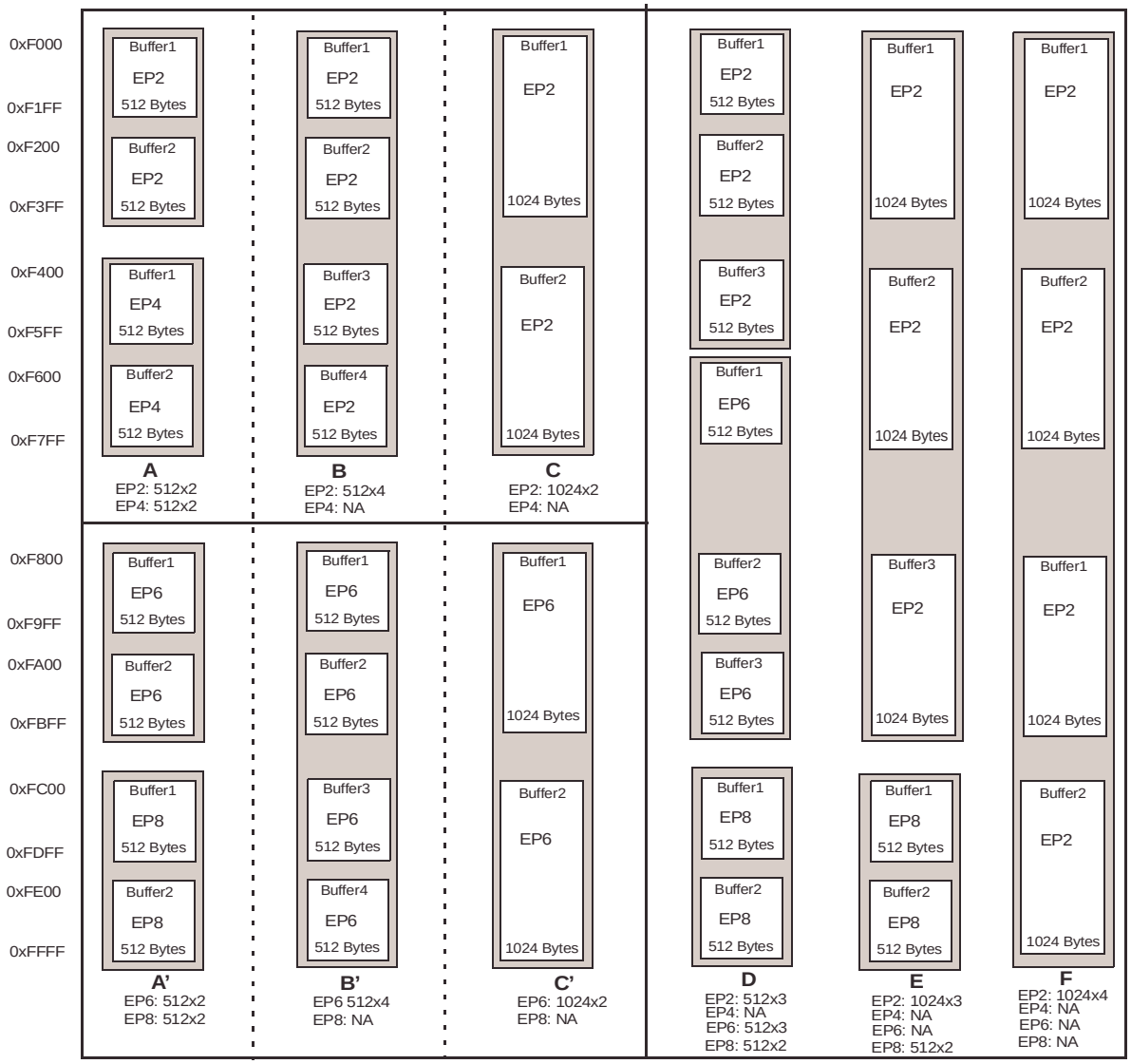

Figure 2. FX1/FX2 Endpoint Buffers

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. (as set by the SIZE bit). It makes sense, therefore, to configure full-speed BULK endpoints as 512 bytes rather than 1024, so that fewer unused bytes are wasted. For a quad buffered 512 bytes endpoint, only 64 bytes of each 512 bytes FIFO buffer is used for data.

An ISOCHRONOUS full speed endpoint, on the other hand, could fully use either a 512- or 1024-byte buffer. If the wMaxPacketSize is set to greater than 512 bytes, it makes sense to configure the endpoint as 1024 bytes deep. If the wMaxPacketSize is set to less than 512 bytes, it is more efficient to configure the endpoint as 512 bytes only, there by leaving less unused buffer space.

Table 3 lists all the possible endpoint configurations. Refer to *Figure 2* for the endpoint configuration names. Section 1.18 and chapter 8 of the Technical Reference Manual provides further details on the possible grouping and configuration of the endpoint buffers, which is also illustrated in *Figure 2*

In each of the endpoint configuration shown below, the 8051 or the external master must avoid accessing an endpoint that is not available..

Table 3. Possible FX1 Endpoints Configuration

Endpoint Configuration (Refer to *Figure 1*)	Number of Endpoints Available	Available Endpoint	Size (Bytes)	Buffering (x2, x3, x4)
AA’	4	EP2	512	x2
		EP4	512	x2
		EP6	512	x2
		EP8	512	x2
BB’	2 (EP4 and EP8 not available)	EP2	512	x4
		EP6	512	x4
CC’	2 (EP4 and EP8 not available)	EP2	1024	x2
		EP6	1024	x2
AB’	3 (EP8 not available)	EP2	512	x2
		EP4	512	x2
		EP6	512	x4
AC’	3 (EP8 not available)	EP2	512	x2
		EP4	512	x2
		EP6	1024	x2
BA’	3 (EP4 not available)	EP2	512	x4
		EP6	512	x2
		EP8	512	x2
BC’	2 (EP4 and EP8 not available)	EP2	512	x4
		EP6	1024	x2
CA’	3 (EP4 not available)	EP2	1024	x2
		EP6	512	x2
		EP8	512	x2
CB’’	2 (EP4 and EP8 not available)	EP2	1024	x2
		EP6	512	x4
D	3 (EP4 not available)	EP2	512	x3
		EP6	512	x3
		EP8	512	x2
E	2 (EP4 and EP6 not available)	EP2	1024	x3
		EP8	512	x2
F	1 (EP4, EP6 and EP8 not available)	EP2	1024	x4

Multiple Buffering

The FX1 endpoint FIFO architecture uses the concept of "multiple buffering" to help minimize the delay involved in waiting for buffer space before being able to reuse the endpoint. This section elaborates the purpose and use of multi-buffering while configuring the endpoints.

The advantage of having endpoint multi buffered is that even though the data armed is still in transfer over the USB (maybe waiting for the host to send IN token for an IN endpoint), the endpoint will not appear full as there is another buffer available. So as far as the 8051 is concerned, it can start/continue (re)filling the buffer. Meanwhile the host may send an IN token to which the core will respond with the data packet. Once the core receives an ACK from the host, the buffer will be released back to the 8051 for reuse. This multi-buffering feature allows the 8051 to save time. Rather than waiting for the busy/FF bit to be cleared it can start filling another buffer while one is being 'worked on' by the USB core (SIE).

As far as the 8051is concerned, there is no difference in addressing an endpoint that is single buffered or double/triple/quad buffered. Buffering only allows the 8051 to access a buffer space while the other is in transition over USB. Endpoint buffering has no effect on the depth (which is set by the SIZE bit in the EPxCFG register) of the FIFO. The depth of the FIFO still is 512 bytes (or 1024 bytes if the SIZE bit is set) and the 8051 must not address an index greater than 511(or 1023 bytes if the SIZE bit is set) while accessing the buffer. Internal logic will basically handle the ping ponging of the buffers automatically.

Note. None of the four large endpoints of the FX1 can be configured to be 'single' buffered. The large endpoints have to be atleast double buffered for it to be usable.

As already stated earlier, "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.

Once the data packet is committed (covered in the next section), the internal logic will assign the next 512 bytes buffer as the "active buffer." This is all done automatically by the FX2 internal logic and is not visible to the 8051. The 8051 and the external peripheral must treat the buffer as 512 bytes only. In slave FIFO mode, the external master must avoid writing to a full FIFO or reading from an empty FIFO. The external master can use the FIFO full/empty/programmable flag to monitor the state of the FIFO and decide when to stop accessing the FIFO.

Endpoint FIFO Architecture

The endpoint architecture of the EZ-USB FX1 (and the FX2) is quiet different than what was used in the EZ-USB and the EZ-USB FX. FX1(and FX2) follows a "Quantum FIFO" architecture which involves moving control of the data packets from one domain to the other. This allows moving the data packet instantaneously between the USB and the FIFOs, without 8051 intervention. The CPU simply configures the interface, then "gets out of the way" while the unified FX1 FIFOs move the data directly between the USB and the external interface. This section discusses the different domains in the EZ-USB FX1 and the data flow/control among the various domains.

To understand the "Quantum FIFO", it is necessary to refer to the three data domains, the *USB domain,* the *8051 domain* and the *Interface domain* (also known as *Peripheral Domain*). *Figure 3* displays the interaction of these domains. Each domain is independent, allowing different clocks and logic to handle its data. At a specific time the data access is controlled by one (or two) of the these three domains. The control of the data buffer is shifted from one domain to the other, defining the data path taken by the packet, as summarized below:

OUT transfer:

SIE -->8051-->Peripheral

IN transfer:

Peripheral-->8051-->SIE

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 FX1 chip.

The 8051 domain is the intermediate domain that allows the user to probe the USB packet before it is sent to its "destination": for an IN transfer the destination is the host and for an OUT transfer it is the external peripheral wired to the FX1

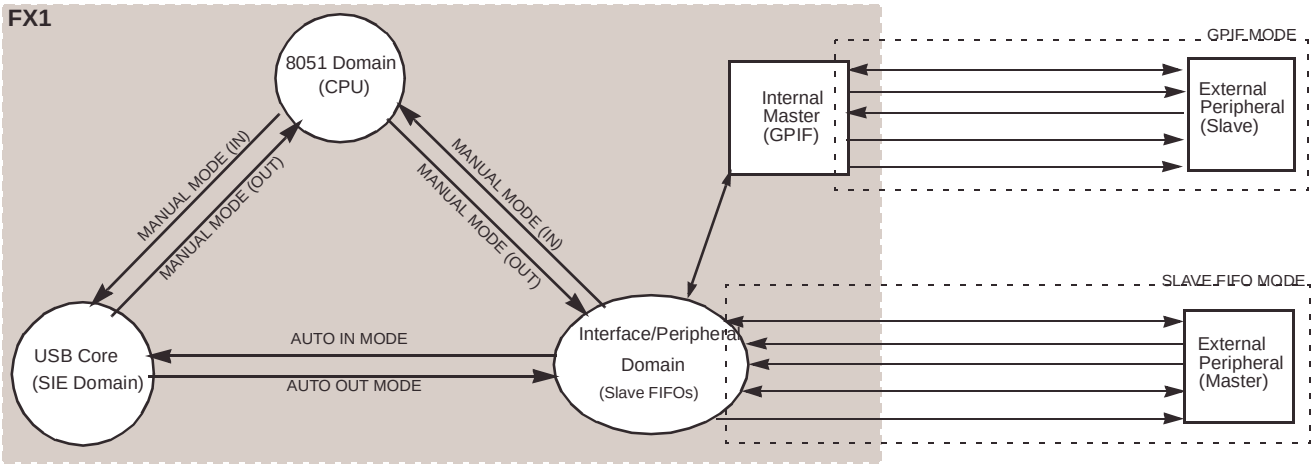

Figure 3. Interrelation of the FX1/FX2 Domains

The Interface domain can load and unload the endpoint FIFOs. An external device such as a DSP or ASIC (wired to the FX1) can supply its own clock to the FIFO interface, or the FX1'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 the FIFO. As mentioned earlier, the FX1'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. So a high data rate is attributed to this quantum nature of data transfer.

FX1 provides two options of handling the endpoint buffer once it is filled with the USB data. This depends on the mode of the endpoint FIFO which is set by bits 3 and 4 of the EPxFIFOCFG register (x=2,4,6,8). These modes include the auto mode and the manual mode. Depending on the mode, the 8051 domain may or may not be in the data path discussed earlier.

On power-up, OUT endpoint come up unarmed (covered in the next section) and are in the control of the 8051 domain only. IN endpoints also come up unarmed and can be accessed either by the 8051 or the external peripheral. Also, on power-up EP6 and EP8 are configured as double-buffered IN endpoints in *manual mode*, whereas EP2 and EP4 are configured as double-buffered OUT endpoints in *manual mode*.

In manual mode, the 8051 has the initial access to the data in the endpoint FIFO buffer. The control of the OUT endpoint buffer is shifted to the USB domain by arming the endpoint. When an OUT endpoint is armed it is under the control of the USB domain. Data send by the host is accepted by the core. The core ACKs the OUT transfer. Once the core has ACK'ed the OUT transfer, the 8051 gains access to the FIFO buffer and can modify the data as required. The 8051 can either move the data to interface domain or just ignore it. Once the data packet is committed to the interface domain, the 8051 loses access to the FIFO buffer and the interface domain now has access to the data buffer.

In the Interface domain, the FIFOs can be controlled by an external master or an internal master. External master either supplies a clock and read/write enable signals to operate synchronously, or strobe signals to operate asynchronously. Alternately, the FIFOs can be controlled by an internal FX1 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. Alternately, if using the part in slave FIFO mode, the external master may access the FIFO using various control signals. For further information on the various control signal input/outputs please refer to chapters 9 and 10 of the Technical Reference Manual. 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 FX1 clock or an externally supplied clock.

Figure 3 also illustrates the data path for manual and auto mode. In manual mode, the 8051 is involved in moving data between the USB domain and the interface domain. In Auto mode, *the CPU simply* configures the interface, then "gets out of the way" while the unified FX1 FIFOs move the data directly between the USB and the external interface.

Depending on the domain that has access to the FIFO, different registers are used to query the FIFO status. The EPxxFIFOFLGS (x= 2, 4, 6, 8) register indicates the status of the FIFO under the control of the peripheral domain. EP2468STAT register is used to indicate the status of the FIFOs under the control of the 8051 domain. So basically if the FIFO is in the interface domain, in order to determine the status of the flags (EF, FF, PF) the 8051 must check the EPxxFIFOFLGS register (one in xdata space or the SFR). For an IN transfer, once data is committed to the host (data packet in USB domain) the 8051 can check the status of the endpoint FIFO probing the EP2468STAT register bits. The external peripheral may check the status of the FIFO in the interface domain by monitoring the FIFO flag pins: FLAGA, FL:AGB, FLAGC and FLAGD

The following section discusses this dataflow and the path taken for an IN and OUT transfer.

Data Flow

This section explains the data flow and the significance of arming and committing a data packet when referring to an OUT versus an IN endpoint.

In general, an endpoint is considered to be armed when the USB domain has control over the endpoint and the host can send data to the endpoint (OUT) or receive data from the endpoint (IN). An IN endpoint is considered armed when it has data to send to the host when the host sends an IN token. An OUT endpoint is considered to be armed when it can accept data from the host when the host sends an OUT token.

A data packet is considered to be committed when it is under the control of the "destination": for an IN transfer the destination is the USB domain, for an OUT transfer the destination is the peripheral domain. Although the essence is the same, "arming" and "committing" an endpoint has different meaning based on the endpoint direction and endpoint mode (auto or manual) as defined below.

IN endpoint:

Data path (manual mode)

host <---- USB core (SIE) <---- 8051 (CPU) <----- Interface

Data path (auto mode)

host <---- USB core (SIE) <---- Interface

Arming

Causing the core to be able to respond to the IN token (sent by the host) with data packet. Arming an endpoint essentially means shifting the control of the data packet to the USB domain. Once the endpoint is armed, USB core is in control of the packet. The 8051 may arm the endpoint by writing to the byte count register with the number of bytes that it would like to send to the host in that packet. The maximum number of bytes is limited by the smaller of what is defined in the

wMaxPacketSize field of the endpoint descriptor and the SIZE bit set in the EPxCFG register. An IN endpoint can be armed by the 8051 only.

Committing

Moving the control of the data packet from the 8051 domain (when in manual mode) or the interface (GPIF or slave FIFO) domain (when in auto mode) to the destination (the USB domain).

For an in endpoint the result of arming or committing a data packet is the same: USB domain is in control of the data packet. The 8051 may arm the endpoint or commit the packet to the USB domain. The interface domain commits the packet to the USB domain (can also use the word arm here, but typically, commit is used instead).

Once an IN endpoint is armed/committed, the only way to abort the transfer is by resetting the endpoint FIFO using the FIFORESET register.

OUT endpoint:

Data path (manual mode)

Host ----> USB core (SIE) ----> 8051 (CPU) -----> Interface

Data path (auto mode)

Host ----> USB core (SIE) ----> Interface

Arming

Causing the core to be able to accept data from the host when the host sends OUT tokens. Arming an OUT endpoint essentially means making buffer space available for the USB core (SIE) to accept data from the host. Once an OUT endpoint is armed, the USB core is in control of the packet buffer. The 8051 may arm the endpoint by writing to the byte count register with any value.

Committing

Causing the Interface domain to be in control of the data packet. In auto mode, when an 'armed' OUT endpoint receives data from the host and the core ACKs the transfer, the control of the data packet is automatically 'shifted' to the interface domain. This committed packet is now under the control of the interface domain (GPIF slave FIFO). In manual mode, when an 'armed' OUT endpoint receives data from the host and the core ACKs the transfer, the data packet is NOT moved to the interface domain. This (moving control of the data packet to the interface domain) needs to be done by the 8051.

8051 is in control of the data packet that was sent by the host to the OUT endpoint. It can 'commit' the packet to the interface domain by writing to the OUTPKTEND register or the bytecount register as explained in the Technical Reference Manual. The 8051 also has a choice of not moving this packet to the interface domain but rearming the endpoint buffer by simply writing to the byte count register with the SKIP bit set. If the SKIP bit is set the control of the data buffer is moved back to the USB domain, in effect allowing the core to accept new data.

For an OUT endpoint the result of arming or committing is NOT the same. Arming the endpoint enables the core to accept new data packet (data packet is in control of the USB domain). Committing causes the data packet to be in the control of the interface domain (the destination).

Once an OUT endpoint is armed/committed, the only way for the 8051 to abort (dis-arm) the transfer is by resetting the endpoint FIFO using the FIFORESET register.

Manual vs. Auto Mode

In order to meet the high-speed data throughput, the data packets can be directly moved from the USB domain to the Interface domain and vice-versa without the 8051's intervention. This is done when the FIFOs are configured for *auto mode*. *Figure 4* below illustrates the interrelation of the three domains of the FX1 in the manual and auto mode.

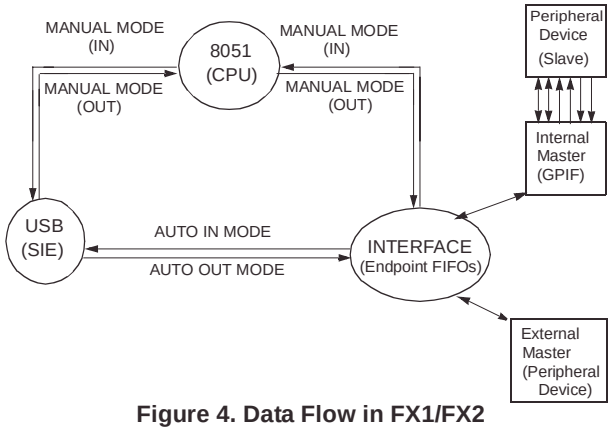

Figure 4. Data Flow in FX1/FX2

Manual Mode

If the 8051 sets bit 3 (AUTOIN for an IN endpoint) or bit 4 (AUTOOUT for an OUT endpoint) in the EPxFIFOCFG register to 0, the 8051 receives an interrupt on the buffer being filled with USB data and connection of the buffer to the endpoint is under 8051's control. This mode of operation is known as the manual mode. In this mode the 8051 is in charge of "committing" the data when the number of bytes in the buffer meets a desired level.

IN Transfer: Peripheral to Host

For an IN transfer, the FX2 provides two optional ways of committing the data once the endpoint IN buffer is filled. The 8051 may commit this packet by doing either of the following

1. Writing to the byte count register with the number of bytes to send in the specific packet.
2. Writing to the INKPKTEND register with the IN endpoint number.

Note.

1. Using the second method to commit data is faster as it involves writing to a single register. Using the byte count registers involves writing to two registers and hence more instruction cycles.
2. Regardless of the number of bytes in the IN endpoint, the external master may also commit the packet anytime using the PKTEND pin. PKTEND is usually used when the master wishes to send a "short" packet (a packet smaller than the size specified in the EPxAUTOINLEN register).

OUT Transfer: Host to Peripheral

In manual mode, the 8051 needs to write to the byte count register with the SKIP bit set to 0 in order to commit data to

the master. If the SKIP bit is set to 1, data is simply ignored (discarded). Writing to the byte count register automatically rearms the endpoint.

Auto Mode

If the 8051 sets bit 3 (AUTOIN for an IN endpoint) or bit 4 (AUTOOUT for an OUT endpoint) in the EPxFIFOCFG register to 1, data in the FIFO buffer (slave FIFO) is automatically and instantly connected to the endpoint FIFO. The endpoint FIFO flags and buffer counts (EPxFIFOBCH/L, EPXXFIFOFLAGS) simultaneously indicate the change in the FIFO status. In Auto mode, *the CPU simply* configures the interface, then “gets out of the way” while the unified FX1 FIFOs move the data directly between the USB and the external interface.

IN Transfer: Peripheral to Host

When the number of bytes in the FIFO reaches the value set in EPxAUTOINLEN, the control of the data packet is automatically shifted from the interface domain to the USB domain.

In this auto mode when EPxAUTOINLEN is specified, the external master can stream data continuously through the FIFO, over the USB to the host without the 8051’s intervention. If the number of bytes in the FIFO is less than what is specified in EPxAUTOINLEN, the packet being a short packet will not be committed automatically to the USB. This short packet needs to be committed manually. To commit this last packet manually,

1. have the external master pad the packet with dummy data in order to meet size specified in the EPxAUTOINLEN register
2. have the external master assert the PKTEND pin
3. have the 8051 write to the INPKTEND register with the endpoint number.

OUT Transfer: Host to Peripheral

In auto mode regardless of the number of bytes received from the host, the data packet is committed to the peripheral domain instantly as it is received from the host.

Conclusion

After having read this application note, the reader should be familiar with the FIFO architecture of the EZ-USB FX1. This application note can also be used by an EZ-USB FX2 user to understand the basics regarding the FIFO architecture and the data flow.

The EZ-USB FX2 shares the same FIFO architecture and endpoint configuration, and multiple buffering scheme. Being a high speed USB microcontroller, the EZ-USB FX2 can operate at full and high speed. When operating at high speed, the maximum packet size of an endpoint defined as bulk, can be up to 512 bytes. Hence each quantum FIFO can accommodate about 512 bytes, even if the physical buffer size is 1024 bytes (when the SIZE bit is set). It makes sense, therefore, to configure high speed (or full-speed) BULK endpoints of the EZZ-USB FX2 as 512 bytes rather than 1024, so that fewer bytes are left unused. When configured as 512 bytes and operating at high speed, the 8051 and the external master should not access the FIFO deeper than 512 bytes.

Further information on additional features of the EZ-USB FX2 (ex: high bandwidth endpoints) can be found in the EZ-USB FX2 Technical Reference Manual.

Approved AN4067 10/19/04 kkv