# MoBL-USB™ FX2LP18

# Technical Reference Manual

Document # 001-11981 Rev. *B

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone (USA): 800.858.1810
Phone (Intnl): 408.943.2600
http://www.cypress.com

**EXHIBIT 2033**
LG Elecs. v. Cypress Semiconductor
IPR2014-01396, U.S. Pat. 6,249,825

## Copyrights

Copyright © 2002–2011 Cypress Semiconductor Corporation. All rights reserved.

Cypress, the Cypress Logo, MoBL-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. SmartMedia is a trademark of Toshiba Corporation. All other product or company names used in this manual may be trademarks, registered trademarks, or servicemarks of their respective owners.

## Disclaimer

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.

The MoBL-USB™ FX2LP18 Technical Reference Manual, Version 1.0 provides information for the CY7C68053.



Exhibit 2033 - Page 02 of 346

# Contents Overview

| | |
|---|---|
| 1. Introducing MoBL-USB™ FX2LP18 | 15 |
| 2. Endpoint Zero | 33 |
| 3. Enumeration and ReNumeration™ | 51 |
| 4. Interrupts | 59 |
| 5. Memory | 77 |
| 6. Power Management | 83 |
| 7. Resets | 89 |
| 8. Access to Endpoint Buffers | 93 |
| 9. Slave FIFOs | 107 |
| 10. General Programmable Interface | 135 |
| 11. CPU Introduction | 189 |
| 12. Instruction Set | 197 |
| 13. Input/Output | 203 |
| 14. Timers/Counters and Serial Interface | 217 |
| 15. Registers | 237 |
| Appendix A. Descriptors for Full-Speed Mode | 311 |
| Appendix B. Descriptors for High-Speed Mode | 319 |
| Appendix C. Device Register Summary | 327 |




Exhibit 2033 - Page 03 of 346

Exhibit 2033 - Page 04 of 346

# Contents

**1. Introducing MoBL-USB™ FX2LP18 15**

1.1 Introduction....15
1.2 An Introduction to USB....15
1.3 The USB Specification....16
1.4 Host Is Master....16
1.5 USB Direction....16
1.6 Tokens and PIDs....17
1.6.1 Receiving Data from the Host....18
1.6.2 Sending Data to the Host....18
1.7 USB Frames....18
1.8 USB Transfer Types....18
1.8.1 Bulk Transfers....18
1.8.2 Interrupt Transfers....19
1.8.3 Isochronous Transfers....19
1.8.4 Control Transfers....19
1.9 Enumeration....20
1.9.1 Full-Speed / High-Speed Detection....20
1.10 The Serial Interface Engine....20
1.11 ReNumeration™....21
1.12 MoBL-USB FX2LP18 Architecture....21
1.13 MoBL-USB FX2LP18 Features Summary....22
1.14 MoBL-USB FX2LP18 Integrated Microprocessor....23
1.15 MoBL-USB FX2LP18 Block Diagram....24
1.16 Package....25
1.16.1 56-Pin Package....25
1.16.2 Signals Available....25
1.17 Package Diagram....27
1.18 MoBL-USB FX2LP18 Endpoint Buffers....28
1.19 External FIFO Interface....29
1.20 MoBL-USB FX2LP18 Part Number....31
1.21 Document History....32

**2. Endpoint Zero 33**

2.1 Introduction....33
2.2 Control Endpoint EP0....33
2.3 USB Requests....36
2.3.1 Get Status....37
2.3.2 Set Feature....39
2.3.3 Clear Feature....41
2.3.4 Get Descriptor....41
2.3.4.1 Get Descriptor-Device....43
2.3.4.2 Get Descriptor-Device Qualifier....44
2.3.4.3 Get Descriptor-Configuration....44



**Exhibit 2033 - Page 05 of 346**

2.3.4.4 Get Descriptor-String ....... 44
2.3.4.5 Get Descriptor-Other Speed Configuration ....... 45
2.3.5 Set Descriptor ....... 45
2.3.5.1 Set Configuration ....... 47
2.3.6 Get Configuration ....... 47
2.3.7 Set Interface ....... 47
2.3.8 Get Interface ....... 48
2.3.9 Set Address ....... 48
2.3.10 Sync Frame ....... 49
2.3.11 Firmware Load ....... 49

**3. Enumeration and ReNumeration™ 51**

3.1 Introduction ....... 51
3.2 MoBL-USB FX2LP18 Startup ....... 51
3.3 The Default USB Device ....... 52
3.4 ‘C2’ EEPROM Boot-load Data Format ....... 53
3.5 EEPROM Configuration Byte ....... 54
3.6 The RENUM Bit ....... 55
3.7 MoBL-USB FX2LP18 Response to Device Requests (RENUM=0) ....... 55
3.8 MoBL-USB FX2LP18 Vendor Request for Firmware Load ....... 56
3.9 How the Firmware ReNumerates ....... 57
3.10 Multiple ReNumerations™ ....... 57

**4. Interrupts 59**

4.1 Introduction ....... 59
4.2 SFRs ....... 59
4.2.1 803x/805x Compatibility ....... 62
4.3 Interrupt Processing ....... 62
4.3.1 Interrupt Masking ....... 62
4.3.1.1 Interrupt Priorities ....... 63
4.3.2 Interrupt Sampling ....... 63
4.3.3 Interrupt Latency ....... 64
4.4 USB-Specific Interrupts ....... 64
4.4.1 Resume Interrupt ....... 64
4.4.2 USB Interrupts ....... 64
4.4.2.1 SUTOK, SUDAV Interrupts ....... 68
4.4.2.2 SOF Interrupt ....... 68
4.4.2.3 Suspend Interrupt ....... 68
4.4.2.4 USB RESET Interrupt ....... 68
4.4.2.5 HISPEED Interrupt ....... 68
4.4.2.6 EP0ACK Interrupt ....... 68
4.4.2.7 Endpoint Interrupts ....... 69
4.4.2.8 In-Bulk-NAK (IBN) Interrupt ....... 69
4.4.2.9 EPxPING Interrupt ....... 69
4.4.2.10 ERRLIMIT Interrupt ....... 69
4.4.2.11 EPxISOERR Interrupt ....... 69
4.5 USB-Interrupt Autovectors ....... 69
4.5.1 USB Autovector Coding ....... 71
4.6 I2C™ Bus Interrupt ....... 72
4.7 FIFO/GPIF Interrupt (INT4) ....... 73
4.8 FIFO/GPIF-Interrupt Autovectors ....... 74
4.8.1 FIFO/GPIF Autovector Coding ....... 75



**Exhibit 2033 - Page 06 of 346**

5. Memory 77
5.1 Introduction ....77
5.2 Internal Data RAM ....77
5.2.1 The Lower 128 ....78
5.2.2 The Upper 128 ....78
5.2.3 SFR (Special Function Register) Space ....78
5.3 External Program Memory and External Data Memory ....78
5.4 MoBL-USB FX2LP18 Memory Map ....79
5.5 On Chip Data Memory at 0xE000 – 0xFFFF ....81

6. Power Management 83
6.1 Introduction ....83
6.2 USB Suspend ....85
6.2.1 SUSPEND Register ....85
6.3 Wakeup/Resume ....86
6.3.1 Wakeup Interrupt ....87
6.4 USB Resume (Remote Wakeup) ....88
6.4.1 WU2 Pin ....88

7. Resets 89
7.1 Introduction ....89
7.2 Hard Reset ....89
7.3 Releasing the CPU Reset ....90
7.3.1 RAM Download ....90
7.3.2 EEPROM Load ....90
7.4 CPU Reset Effects ....91
7.5 USB Bus Reset ....91
7.6 MoBL-USB FX2LP18 Disconnect ....91
7.7 Reset Summary ....92

8. Access to Endpoint Buffers 93
8.1 Introduction ....93
8.2 MoBL-USB FX2LP18 Large and Small Endpoints ....93
8.3 High-Speed and Full-Speed Differences ....93
8.4 How the CPU Configures the Endpoints ....94
8.5 CPU Access to MoBL-USB FX2LP18 Endpoint Data ....95
8.6 CPU Control of MoBL-USB FX2LP18 Endpoints ....96
8.6.1 Registers That Control EP0, EP1IN, and EP1OUT ....96
8.6.1.1 EP0CS ....96
8.6.1.2 EP0BCH and EP0BCL ....97
8.6.1.3 USBIE and USBIRQ ....97
8.6.1.4 EP01STAT ....98
8.6.1.5 EP1OUTCS ....98
8.6.1.6 EP1OUTBC ....98
8.6.1.7 EP1INCS ....98
8.6.1.8 EP1INBC ....99
8.6.2 Registers That Control EP2, EP4, EP6, EP8 ....99
8.6.2.1 EP2468STAT ....99
8.6.2.2 EP2ISOINPKTS, EP4ISOINPKTS, EP6ISOINPKTS, EP8ISOINPKTS ....100
8.6.2.3 EP2CS, EP4CS, EP6CS, EP8CS ....100
8.6.2.4 EP2BCH:L, EP4BCH:L, EP6BCH:L, EP8BCH:L ....101



Exhibit 2033 - Page 07 of 346

8.6.3 Registers That Control All Endpoints ....... 102
8.6.3.1 IBNIE, IBNIRQ, NAKIE, NAKIRQ ....... 102
8.6.3.2 EPIE, EPIRQ ....... 103
8.6.3.3 USBERRIE, USBERRIRQ, ERRCNTLIM, CLRERRCNT ....... 103
8.6.3.4 TOGCTL ....... 104
8.7 The Setup Data Pointer ....... 104
8.7.1 Transfer Length ....... 105
8.7.2 Accessible Memory Spaces ....... 105
8.8 Autopointers ....... 105

**9. Slave FIFOs** **107**

9.1 Introduction ....... 107
9.2 Hardware ....... 107
9.2.1 Slave FIFO Pins ....... 108
9.2.2 FIFO Data Bus (FD) ....... 109
9.2.3 Interface Clock (IFCLK) ....... 110
9.2.4 FIFO Flag Pins (FLAGA, FLAGB, FLAGC, FLAGD) ....... 111
9.2.5 Control Pins (SLOE, SLRD, SLWR, PKTEND, FIFOADR[1:0]) ....... 112
9.2.6 Slave FIFO Chip Select ....... 114
9.2.7 Implementing Synchronous Slave FIFO Writes ....... 114
9.2.8 Implementing Synchronous Slave FIFO Reads ....... 117
9.2.9 Implementing Asynchronous Slave FIFO Writes ....... 119
9.2.10 Implementing Asynchronous Slave FIFO Reads ....... 121
9.3 Firmware ....... 122
9.3.1 Firmware FIFO Access ....... 122
9.3.2 EPx Memories ....... 123
9.3.3 Slave FIFO Programmable-Level Flag ....... 124
9.3.4 Auto-In / Auto-Out Modes ....... 124
9.3.5 CPU Access to OUT Packets, AUTOOUT = 1 ....... 126
9.3.6 CPU Access to OUT Packets, AUTOOUT = 0 ....... 126
9.3.7 CPU Access to IN Packets, AUTOIN = 1 ....... 129
9.3.8 Access to IN Packets, AUTOIN=0 ....... 131
9.3.9 Auto-In / Auto-Out Initialization ....... 132
9.3.10 Auto-Mode Example: Synchronous FIFO IN Data Transfers ....... 133
9.3.11 Auto-Mode Example: Asynchronous FIFO IN Data Transfers ....... 134
9.4 Switching Between Manual-Out and Auto-Out ....... 134

**10. General Programmable Interface** **135**

10.1 Introduction ....... 135
10.1.1 Typical GPIF Interface ....... 137
10.2 Hardware ....... 138
10.2.1 The External GPIF Interface ....... 138
10.2.2 Default GPIF Pins Configuration ....... 139
10.2.3 Six Control OUT Signals ....... 139
10.2.3.1 Control Output Modes ....... 139
10.2.4 Six Ready IN signals ....... 139
10.2.5 Nine GPIF Address OUT Signals ....... 139
10.2.6 Three GSTATE OUT Signals ....... 139
10.2.7 8/16-Bit Data Path, WORDWIDE = 1 (default) and WORDWIDE = 0 ....... 139
10.2.8 Byte Order for 16-bit GPIF Transactions ....... 140
10.2.9 Interface Clock (IFCLK) ....... 140
10.2.10 Connecting GPIF Signal Pins to Hardware ....... 141
10.2.11 Example GPIF Hardware Interconnect ....... 141



**Exhibit 2033 - Page 08 of 346**

10.3 Programming the GPIF Waveforms ....142
10.3.1 The GPIF Registers ....142
10.3.2 Programming GPIF Waveforms ....143
10.3.2.1 The GPIF IDLE State ....143
10.3.2.2 Defining States ....144
10.3.3 Re-Executing a Task Within a DP State ....147
10.3.4 State Instructions ....150
10.3.4.1 Structure of the Waveform Descriptors ....153
10.3.4.2 Terminating a GPIF Transfer ....154
10.4 Firmware ....155
10.4.1 Single-Read Transactions ....161
10.4.2 Single-Write Transactions ....166
10.4.3 FIFO-Read and FIFO-Write (Burst) Transactions ....170
10.4.3.1 Transaction Counter ....170
10.4.3.2 Reading the Transaction-Count Status in a DP State ....170
10.4.4 GPIF Flag Selection ....170
10.4.5 GPIF Flag Stop ....170
10.4.5.1 Performing a FIFO-Read Transaction ....171
10.4.6 Firmware Access to IN Packets, (AUTOIN=1) ....176
10.4.7 Firmware Access to IN Packets, (AUTOIN = 0) ....177
10.4.7.1 Performing a FIFO-Write Transaction ....179
10.4.8 Firmware Access to OUT Packets, (AUTOOUT=1) ....184
10.4.9 Firmware access to OUT packets, (AUTOOUT = 0) ....185
10.5 UDMA Interface ....187
10.6 ECC Generation ....187

**11. CPU Introduction 189**

11.1 Introduction ....189
11.2 8051 Enhancements ....190
11.3 Performance Overview ....190
11.4 Software Compatibility ....191
11.5 803x/805x Feature Comparison ....191
11.6 MoBL-USB FX2LP18/DS80C320 Differences ....192
11.6.1 Serial Ports ....192
11.6.2 Timer 2 ....192
11.6.3 Timed Access Protection ....192
11.6.4 Watchdog Timer ....192
11.6.5 Power Fail Detection ....192
11.6.6 Port IO ....192
11.6.7 Interrupts ....192
11.7 MoBL-USB FX2LP18 Register Interface ....193
11.8 MoBL-USB FX2LP18 Internal RAM ....193
11.9 IO Ports ....193
11.10 Interrupts ....194
11.11 Power Control ....194
11.12 Special Function Registers ....195
11.13 Reset ....195

**12. Instruction Set 197**

12.1 Introduction ....197
12.1.1 Instruction Timing ....200
12.1.2 Stretch Memory Cycles (Wait States) ....200



Exhibit 2033 - Page 09 of 346

12.1.3 Dual Data Pointers ....201
12.1.4 Special Function Registers ....201

13. Input/Output 203

13.1 Introduction ....203
13.2 IO Ports ....203
13.3 IO Port Alternate Functions ....206
13.3.1 Port A Alternate Functions ....207
13.3.2 Port B and Port D Alternate Functions ....208
13.3.3 Port C Alternate Functions ....209
13.3.4 Port E Alternate Functions ....210
13.4 I²C Bus Controller ....212
13.4.1 Interfacing to I²C Peripherals ....212
13.4.2 Registers ....213
13.4.2.1 Control Bits ....214
13.4.2.2 Status Bits ....214
13.4.3 Sending Data ....215
13.4.4 Receiving Data ....215
13.5 EEPROM Boot Loader ....216

14. Timers/Counters and Serial Interface 217

14.1 Introduction ....217
14.2 Timers/Counters ....217
14.2.1 803x/805x Compatibility ....217
14.2.2 Timers 0 and 1 ....218
14.2.2.1 Mode 0, 13-Bit Timer/Counter — Timer 0 and Timer 1 ....218
14.2.2.2 Mode 1, 16-Bit Timer/Counter — Timer 0 and Timer 1 ....219
14.2.2.3 Mode 2, 8-Bit Counter with Auto-Reload — Timer 0 and Timer 1 ....220
14.2.2.4 Mode 3, Two 8-Bit Counters — Timer 0 Only ....220
14.2.3 Timer Rate Control ....221
14.2.4 Timer 2 ....222
14.2.4.1 Timer 2 Mode Control ....222
14.2.5 Timer 2 The 6-Bit Timer/Counter Mode ....223
14.2.5.1 Timer 2 The 16-Bit Timer/Counter Mode with Capture ....223
14.2.6 Timer 2 16-Bit Timer/Counter Mode with Auto-Reload ....224
14.2.7 Timer 2 Baud Rate Generator Mode ....224
14.3 Serial Interface ....225
14.3.1 803x/805x Compatibility ....226
14.3.2 High-Speed Baud Rate Generator ....226
14.3.3 Mode 0 ....227
14.3.4 Mode 1 ....230
14.3.4.1 Mode 1 Baud Rate ....230
14.3.4.2 Mode 1 Transmit ....232
14.3.5 Mode 1 Receive ....232
14.3.6 Mode 2 ....234
14.3.6.1 Mode 2 Transmit ....234
14.3.6.2 Mode 2 Receive ....234
14.3.7 Mode 3 ....236



**Exhibit 2033 - Page 10 of 346**

15. Registers 237

15.1 Introduction.....237
15.1.1 Example Register Format.....237
15.1.2 Other Conventions.....237
15.2 Special Function Registers (SFR).....238
15.3 About SFRs.....239
15.4 GPIF Waveform Memories.....245
15.4.1 GPIF Waveform Descriptor Data.....245
15.5 General Configuration Registers.....245
15.5.1 CPU Control and Status.....245
15.5.2 Interface Configuration (Ports, GPIF, slave FIFOs).....246
15.5.3 Slave FIFO FLAGA-FLAGD Pin Configuration.....248
15.5.4 FIFO Reset.....249
15.5.5 Breakpoint, Breakpoint Address High, Breakpoint Address Low.....250
15.5.6 230 Kbaud Clock (T0, T1, T2).....250
15.5.7 Slave FIFO Interface Pins Polarity.....251
15.5.8 Chip Revision ID.....251
15.5.9 Chip Revision Control.....252
15.5.10 GPIF Hold Time.....253
15.6 Endpoint Configuration.....253
15.6.1 Endpoint 1-OUT/Endpoint 1-IN Configurations.....253
15.6.2 Endpoint 2, 4, 6 and 8 Configuration.....254
15.6.3 Endpoint 2, 4, 6 and 8/Slave FIFO Configuration.....255
15.6.4 Endpoint 2, 4, 6, 8 AUTOIN Packet Length (High/Low).....256
15.6.5 Endpoint 2, 4, 6, 8/Slave FIFO Programmable-Level Flag (High/Low).....258
15.6.5.1 IN Endpoints.....262
15.6.5.2 OUT Endpoints.....263
15.6.6 Endpoint 2, 4, 6, 8 ISO IN Packets per Frame.....264
15.6.7 Force IN Packet End.....266
15.6.8 Force OUT Packet End.....266
15.7 Interrupts.....267
15.7.1 Endpoint 2, 4, 6, 8 Slave FIFO Flag Interrupt Enable/Request.....267
15.7.2 IN-BULK-NAK Interrupt Enable/Request.....268
15.7.3 Endpoint Ping-NAK/IBN Interrupt Enable/Request.....269
15.7.4 USB Interrupt Enable/Request.....270
15.7.5 Endpoint Interrupt Enable/Request.....271
15.7.6 GPIF Interrupt Enable/Request.....271
15.7.7 USB Error Interrupt Enable/Request.....272
15.7.8 USB Error Counter Limit.....272
15.7.9 Clear Error Count.....273
15.7.10 INT 2 (USB) Autovector.....273
15.7.11 INT 4 (Slave FIFOs and GPIF) Autovector.....273
15.7.12 INT 2 and INT 4 Setup.....274
15.8 Input/Output Registers.....274
15.8.1 I/O PORTA Alternate Configuration.....274
15.8.2 I/O PORTC Alternate Configuration.....274
15.8.3 I/O PORTE Alternate Configuration.....275
15.8.4 I2C Bus Control and Status.....275
15.8.5 I2C Bus Data.....276
15.8.6 I2C Bus Control.....277
15.8.7 AUTOPOINTERs 1 and 2 MOVX Access.....277




Exhibit 2033 - Page 11 of 346

15.9 ECC Control and Data Registers ....278
15.9.1 ECC Features....278
15.9.2 ECC Implementation ....278
15.9.3 ECC Check/Correct....279
15.9.3.1 ECC Configuration....281
15.9.3.2 ECC Reset....281
15.9.3.3 ECC1 Byte 0 ....281
15.9.3.4 ECC1 Byte 1 ....281
15.9.3.5 ECC1 Byte 2 ....282
15.9.3.6 ECC2 Byte 0 ....282
15.9.3.7 ECC2 Byte 1 ....282
15.9.3.8 ECC2 Byte 2 ....282
15.10 UDMA CRC Registers....283
15.11 USB Control ....284
15.11.1 USB Control and Status ....284
15.11.2 Enter Suspend State ....284
15.11.3 Wakeup Control and Status....285
15.11.4 Data Toggle Control....286
15.11.5 USB Frame Count High....286
15.11.6 USB Frame Count Low ....286
15.11.7 USB Microframe Count ....287
15.11.8 USB Function Address....287
15.12 Endpoints ....287
15.12.1 Endpoint 0 (Byte Count High)....287
15.12.2 Endpoint 0 Control and Status (Byte Count Low) ....288
15.12.3 Endpoint 1 OUT and IN Byte Count ....288
15.12.4 Endpoint 2 and 6 Byte Count High ....288
15.12.5 Endpoint 4 and 8 Byte Count High ....288
15.12.6 Endpoint 2, 4, 6, 8 Byte Count Low....289
15.12.7 Endpoint 0 Control and Status....289
15.12.8 Endpoint 1 OUT/IN Control and Status....290
15.12.9 Endpoint 2 Control and Status....291
15.12.10 Endpoint 4 Control and Status....291
15.12.11 Endpoint 6 Control and Status....292
15.12.12 Endpoint 8 Control and Status....292
15.12.13 Endpoint 2 and 4 Slave FIFO Flags ....293
15.12.14 Endpoint 6 and 8 Slave FIFO Flags ....293
15.12.15 Endpoint 2 Slave FIFO Byte Count High....293
15.12.16 Endpoint 6 Slave FIFO Total Byte Count High ....294
15.12.17 Endpoint 4 and 8 Slave FIFO Byte Count High....294
15.12.18 Endpoint 2, 4, 6, 8 Slave FIFO Byte Count Low....294
15.12.19 Setup Data Pointer High and Low Address ....295
15.12.20 Setup Data Pointer Auto....295
15.12.21 Setup Data - 8 Bytes ....296
15.13 General Programmable Interface....296
15.13.1 GPIF Waveform Selector....296
15.13.2 GPIF Done and Idle Drive Mode ....297
15.13.3 CTL Outputs ....297
15.13.4 GPIF Address High ....298
15.13.5 GPIF Address Low ....298
15.13.6 GPIF Flowstate Registers....299
15.13.7 GPIF Transaction Count Bytes....304
15.13.8 Endpoint 2, 4, 6, 8 GPIF Flag Select....305




**Exhibit 2033 - Page 12 of 346**

15.13.9 Endpoint 2, 4, 6, and 8 GPIF Stop Transaction ....305
15.13.10 Endpoint 2, 4, 6, and 8 Slave FIFO GPIF Trigger ....306
15.13.11 GPIF Data High (16-Bit Mode) ....306
15.13.12 Read/Write GPIF Data LOW and Trigger Transaction ....306
15.13.13 Read GPIF Data LOW, No Transaction Trigger ....306
15.13.14 GPIF RDY Pin Configuration ....307
15.13.15 GPIF RDY Pin Status ....307
15.13.16 Abort GPIF Cycles ....307
15.14 Endpoint Buffers ....308
15.14.1 EP0 IN-OUT Buffer ....308
15.14.2 Endpoint 1-OUT Buffer ....308
15.14.3 Endpoint 1-IN Buffer ....308
15.14.4 Endpoint 2/Slave FIFO Buffer ....309
15.14.5 512-byte Endpoint 4/Slave FIFO Buffer ....309
15.14.6 512/1024-byte Endpoint 6/Slave FIFO Buffer ....309
15.14.7 512-byte Endpoint 8/Slave FIFO Buffer ....309
15.15 Synchronization Delay ....310

**Appendix A. Descriptors for Full-Speed Mode** **311**

**Appendix B. Descriptors for High-Speed Mode** **319**

**Appendix C. Device Register Summary** **327**

Register Summary ....329

**Index** **341**

Exhibit 2033 - Page 14 of 346

# 1. Introducing MoBL-USB™ FX2LP18

## 1.1 Introduction

The Universal Serial Bus (USB) has gained wide acceptance as the connection method of choice for 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 latest generation of the USB specification, 'USB 2.0,' extends the original specification to include:

- A new 'high-speed' 480 Mbps signaling rate, a 40× improvement over USB 1.1's 'full- speed' rate of 12 Mbps.
- Full backward and forward compatibility with USB 1.1 devices and cables.
- A new hub architecture that can provide multiple 12 Mbps downstream ports for USB 1.1 devices.

The Cypress Semiconductor MoBL-USB FX2LP18 offers single-chip USB 2.0 peripherals whose architecture has been designed to accommodate the higher data rates offered by USB 2.0. The MoBL-USB FX2LP18 device (CY7C68053) supports both full-speed and high-speed modes.

This introductory chapter begins with a brief USB tutorial to put USB and MoBL-USB FX2LP18 terminology into context. The remainder of the chapter briefly outlines the chip 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 automatically loads the device's driver into the operating system. When the device is unplugged, the operating system automatically logs it off and unloads its driver.
- USB devices do not use DIP switches, jumpers, or configuration programs. There is never an IRQ, DMA, memory, or IO 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.
- USB supports three speeds:
  - Low-Speed (1.5 Mbps) suitable for mice, keyboards and joysticks.
  - Full-Speed (12 Mbps) for devices like modems, speakers and scanners.
  - High-Speed (480 Mbps) for devices like hard disk drives, CD-ROMs, video cameras, and high-resolution scanners.

The Cypress Semiconductor MoBL-USB FX2LP18 supports the high bandwidth offered by the USB 2.0 High-Speed mode. The MoBL-USB FX2LP18 provides a highly-integrated solution for a USB peripheral device and 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 MoBL-USB FX2LP18 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.



**Exhibit 2033 - Page 15 of 346**

## 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 is easy to remember USB direction: OUT means from the host to the device and IN means from the device to the host. MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 sends data to the host by loading an IN endpoint buffer. Likewise, it receives host data from an OUT endpoint buffer.



Exhibit 2033 - Page 16 of 346

## 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

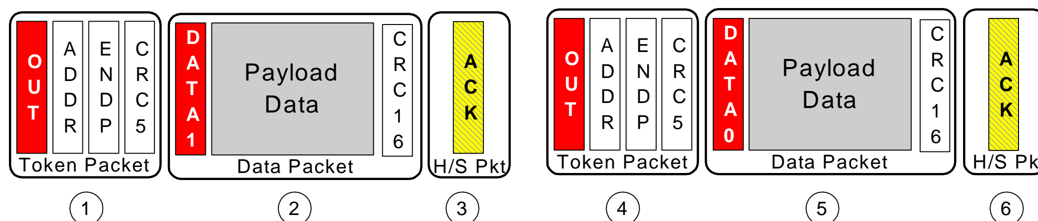



Figure 1-1 illustrates a USB OUT transfer. Host traffic is shown in solid shading; device traffic is shown crosshatched. Packet **1** is an OUT token indicated by the OUT PID. The OUT token signifies that data from the host is about to be transmitted over the bus. Packet **2** contains data as indicated 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 cannot 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 Figure 1-1 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 does not; a USB device indicates an error by not responding.



Exhibit 2033 - Page 17 of 346

- 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 does not understand a device request, that something went wrong on the peripheral end, or that the host tried to access a resource that was not 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 Mbps) USB transmission. The MoBL-USB FX2LP18 supports full-speed (12 Mbps) and high-speed (480 Mbps) 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 MoBL-USB FX2LP18 firmware or external logic can load data into an 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 endpoint. If the host never sends the IN token, the data remains in the endpoint buffer indefinitely.

## 1.7 USB Frames

The USB host provides a time base to all USB devices by transmitting a start-of-frame (SOF) 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 MoBL-USB FX2LP18 registers at any time.

At high-speed (480 Mbps), each one-millisecond frame is divided into eight 125-microsecond *micro-frames,* 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 MoBL-USB FX2LP18 provides a readable microframe counter.

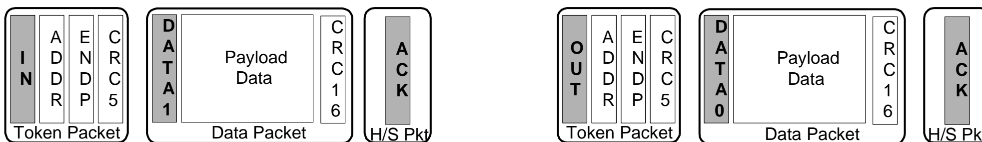
The MoBL-USB FX2LP18 can generate an interrupt request whenever it receives an SOF (once every millisecond at full-speed, or once every 125 microseconds at high-speed). This SOF interrupt can be used, for example, to service isochronous endpoint data.

## 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



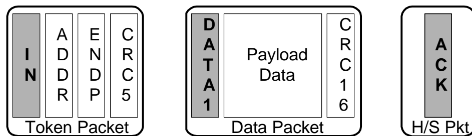

Bulk data is *bursty*, traveling in packets of 8, 16, 32 or 64 bytes at full-speed or 512 bytes at high- speed. Bulk data has guaranteed accuracy, due to an automatic retry mechanism for erroneous data. The host schedules bulk packets when there is available bus time. Bulk transfers are typically used for printer, scanner, or modem data. Bulk data has built-in flow control provided by handshake packets.



Exhibit 2033 - Page 18 of 346

### 1.8.2 Interrupt Transfers

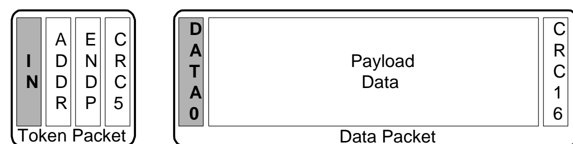
Figure 1-3. An Interrupt Transfer




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.

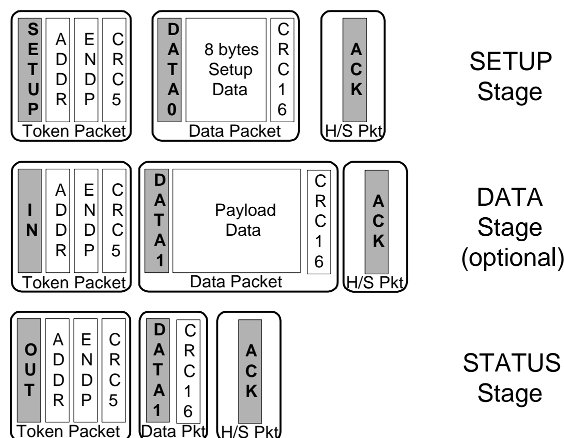
Time of delivery is the most important requirement for isochronous data. In every USB frame, a certain amount of USB bandwidth is allocated to isochronous transfers. To lighten the overhead, isochronous transfers have no handshake (ACK/NAK/STALL/NYET), and no retries; error detection is limited to a 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. For more details, refer to the Isochronous Transfers discussion in Chapter 5 of the USB specification.

### 1.8.4 Control Transfers

Figure 1-5. A Control Transfer




Control transfers configure and send commands to a device. Because they are so important, they employ the most extensive USB error checking. The host reserves a portion of each USB frame for Control transfers.



Exhibit 2033 - Page 19 of 346

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, and so on.

This sign-on process is called 'Enumeration'.

### 1.9.1 Full-Speed / High-Speed Detection

The USB Specification requires that high-speed (480 Mbps) devices must also be capable of enumerating at full-speed (12 Mbps). 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 Specification.

When connected to a full-speed host, the MoBL-USB FX2LP18 will enumerate as a full-speed device. When connected to a high-speed host, the chip automatically switches to high-speed mode. It does not support the low-speed mode (1.5 Mbps).

## 1.10 The Serial Interface Engine

Figure 1-6. What the SIE Does

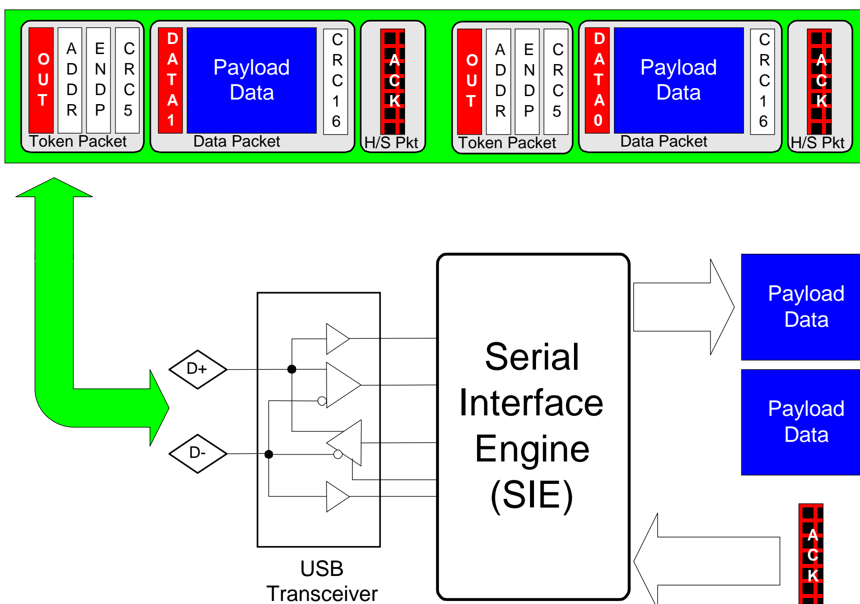





Exhibit 2033 - Page 20 of 346

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 MoBL-USB FX2LP18's SIE.

One of the most important features of the MoBL-USB FX2LP18 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 MoBL-USB FX2LP18's 'smart' SIE performs much more than the basic functions shown in Figure 1-6; it can perform a full enumeration by itself, which allows it 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 programmer, to make development easier and save code and processing time.

## 1.11 ReNumeration™

Because the MoBL-USB FX2LP18's configuration is 'soft,' one chip can take on the identities of multiple distinct USB devices.

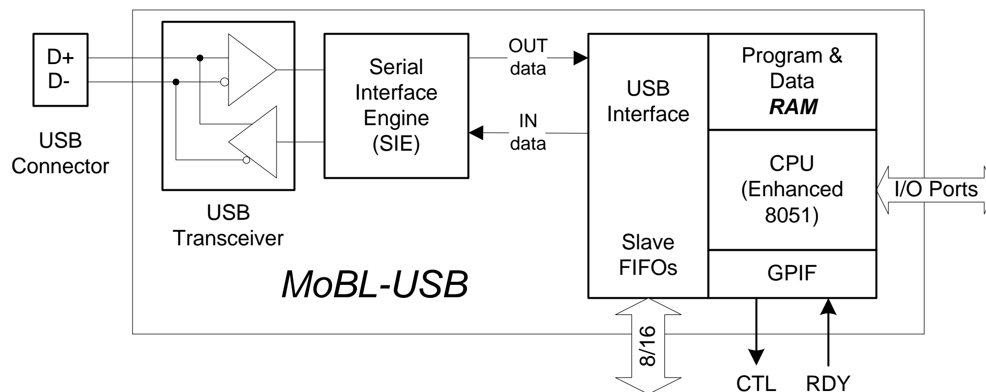
When first plugged into USB, the MoBL-USB FX2LP18 enumerates automatically and downloads firmware and USB descriptor tables over the USB cable. Next, it enumerates again, this time as a device defined by the downloaded information. This patented two-step process, called ReNumeration™, happens instantly when the device is plugged in, with no hint to the user that the initial download step has occurred.

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

The Enumeration and ReNumeration™ chapter on page 51 describes these processes in detail.

## 1.12 MoBL-USB FX2LP18 Architecture

Figure 1-7. MoBL-USB FX2LP18 56-pin Package Simplified Block Diagram




The MoBL-USB FX2LP18 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.



Exhibit 2033 - Page 21 of 346

The MoBL-USB FX2LP18 SIE operates at Full-Speed (12 Mbps) and High-Speed (480 Mbps) rates. To accommodate the increased bandwidth of USB 2.0, the MoBL-USB FX2LP18 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 MoBL-USB FX2LP18-based USB peripheral is two fold:

- 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 MoBL-USB FX2LP18's CPU is well-suited for handling host requests over the control endpoint. However, the data rates offered by USB 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. **Note** Instead, the CPU simply configures the interface, then 'gets out of the way' while the unified MoBL-USB FX2LP18 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 operate synchronously, or strobe signals to operate asynchronously.

Alternately, the FIFOs can be controlled by an internal MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 clock or an externally supplied clock.

The MoBL-USB FX2LP18's CPU is rich in features. Up to five IO 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 MoBL-USB FX2LP18 chip uses an enhanced SIE/USB interface which simplifies code by implementing much of the USB protocol. In fact, the MoBL-USB FX2LP18 can function as a full USB device even without firmware.

All MoBL-USB FX2LP18 chips can operate at a variable IO voltage from 1.8V to 3.3V. The core operates at 1.8V. The PHY and the oscillator operate at 3.3V. The variable IO voltage makes the MoBL-USB FX2LP18 ideal for applications like cell phones, PDAs, MP3 Players, and others.

## 1.13 MoBL-USB FX2LP18 Features Summary

The MoBL-USB FX2LP18 chips include the following features:

- Low power consumption enabling bus-powered designs.
- An on-chip 480 Mbps transceiver, a PLL and SIE—the entire USB physical layer (PHY).
- Double-, triple- and quad-buffered endpoint FIFOs accommodate the 480 Mbps USB 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 IO 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 MoBL-USB FX2LP18 FIFOs.
- 1.8V core operation, 1.8V-3.3V IO operation
- ECC Generation based on the SmartMedia™ standard.



Exhibit 2033 - Page 22 of 346

The MoBL-USB FX2LP18 offers single-chip USB 2.0 peripheral solutions. Unlike designs that use an external PHY, the MoBL-USB FX2LP18 integrates everything on one chip, eliminating costly high pin-count packages and the need to route high-speed signals between chips.

## 1.14 MoBL-USB FX2LP18 Integrated Microprocessor

The MoBL-USB FX2LP18's CPU uses on-chip RAM as program and data memory. The Memory chapter on page 77, describes the various internal/external memory options.

The CPU communicates with the SIE using a set of registers occupying on-chip RAM addresses 0xE500-0xE6FF. These registers are grouped and described by function in individual chapters of this reference manual and summarized in register order. See chapter “Registers” on page 237.

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 MoBL-USB FX2LP18'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 IO ports, USARTs, and an I²C bus master controller. These resources are described in the Input/Output chapter on page 203, and the Timers/Counters and Serial Interface chapter on page 217.

It's important to recognize that the MoBL-USB FX2LP18 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.

The MoBL-USB FX2LP18 chips add eight interrupt sources to the standard 8051 interrupt system:

- INT2: USB Interrupt
- INT3: I²C 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 MoBL-USB FX2LP18 chips provide 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 MoBL-USB FX2LP18 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. The Interrupts chapter on page 59 describes the MoBL-USB FX2LP18 interrupt system.



**Exhibit 2033 - Page 23 of 346**

## 1.15 MoBL-USB FX2LP18 Block Diagram

Figure 1-8. MoBL-USB FX2LP18 Block Diagram

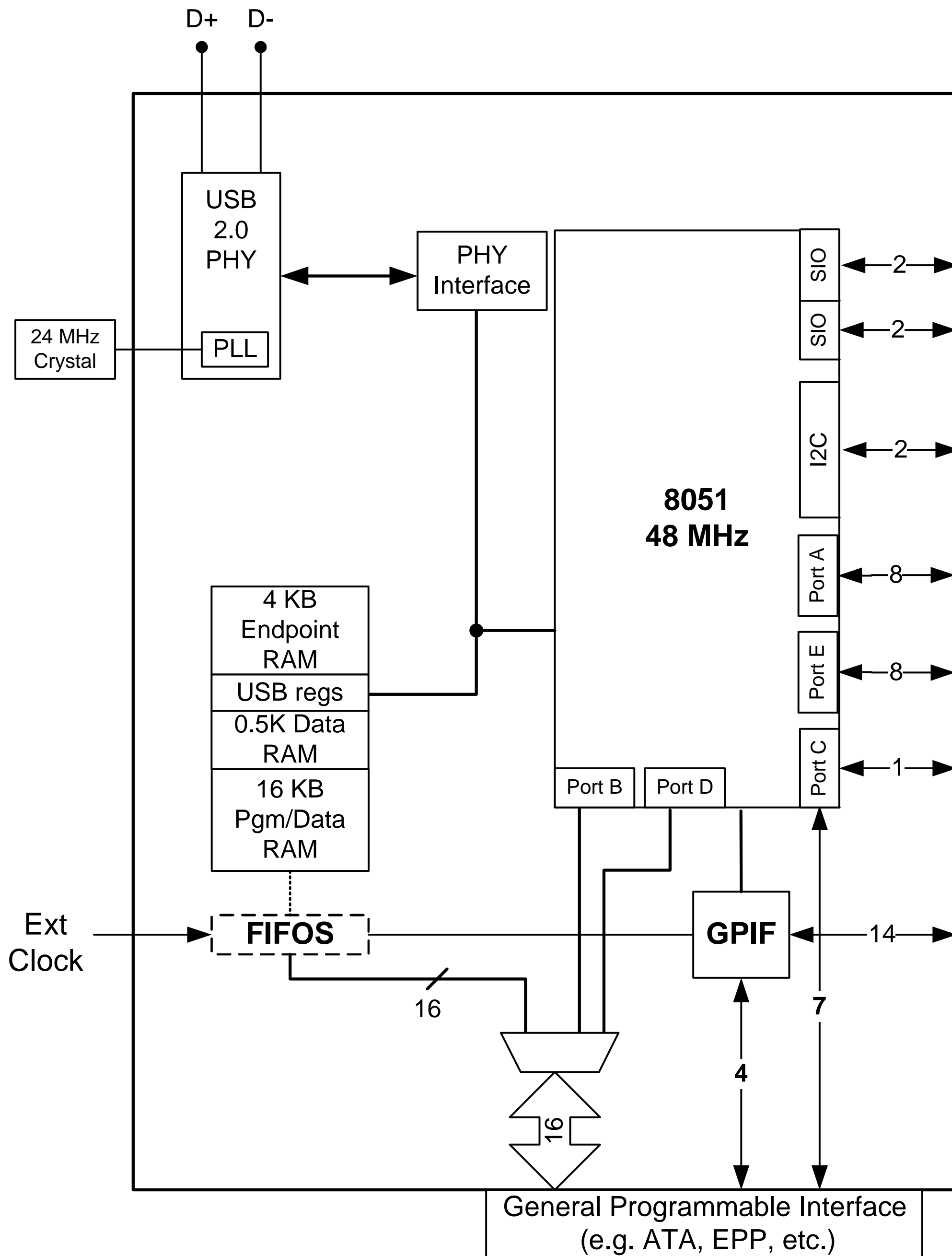




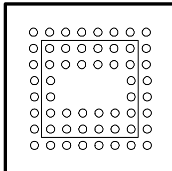

Exhibit 2033 - Page 24 of 346

## 1.16 Package

MoBL-USB FX2LP18 is currently available in one 56-pin VFBGA package. It can be made available as a 100-pin package option (VFBGA, TQFP, and so on). For more information, contact Cypress Sales.

Figure 1-9. 56-pin VFBGA Package

56-pin VFBGA
5x5x1
mm



### 1.16.1 56-Pin Package

Twenty-four general-purpose IO pins (ports A, B, and D) are available. Sixteen of these IO pins can be configured as the 16-bit data interface to the MoBL-USB FX2LP18's internal high-speed 16-bit FIFOs, which can be used to implement low cost, high-performance interfaces such as ATAPI, UTOPIA, EPP, and so on. The 56-pin package has the following:

- Three 8-bit IO ports: PORTA, PORTB, and PORTD
- $I^2C$™ 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

A 100-pin package (currently not available) would provide the following additional functionalities:

- Two additional 8-bit IO 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 INT5#)
- Eight additional control signals multiplexed onto PORTE
- Nine GPIF address lines, multiplexed onto PORTC (eight) and PORTE (one)
- RD# and WR# signals which may be used as read and write strobes for PORTC

### 1.16.2 Signals Available

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

Figure 1-10 shows a logical diagram of the signals available. 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 IO pins are general-purpose IO ports.

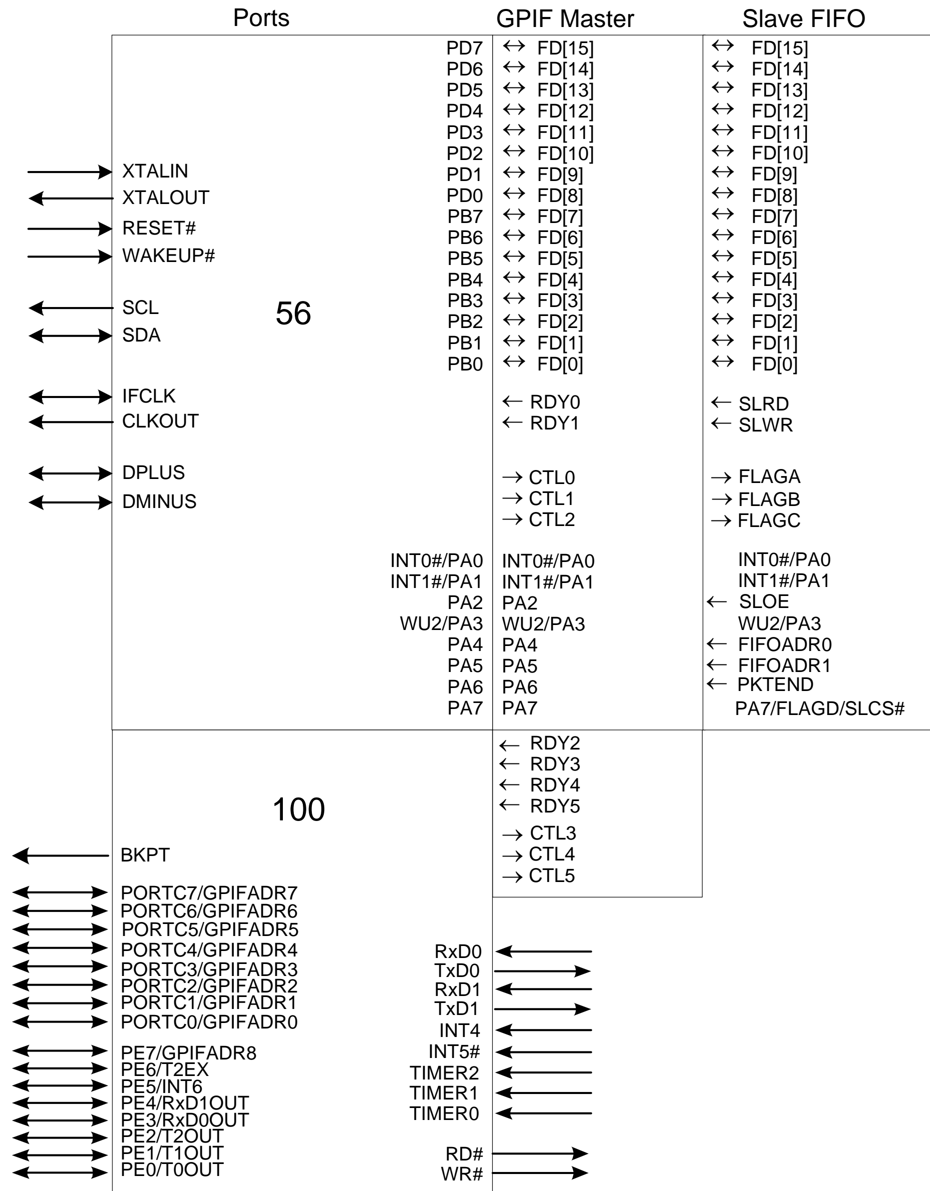
'GPIF master' mode uses the PORTB and PORTD pins as a 16-bit data interface to the four endpoint FIFOs EP2, EP4, EP6, and EP8. In this 'master' mode the FIFOs are controlled by the internal GPIF, a programmable waveform generator that responds to FIFO status flags, drives timing signals using its CTL 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) are available in the 56-pin package.

In the 'Slave FIFO' mode, external logic or an external processor interfaces directly to the MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 pins. The external master can be asynchronous or synchronous and it may supply its own independent clock to the MoBL-USB FX2LP18 interface.



**Exhibit 2033 - Page 25 of 346**

Figure 1-10. Signals for the MoBL-USB FX2LP18 Package



## 1.17 Package Diagram

Figure 1-11. CY7C68053 56-pin VFBGA Pin Assignment

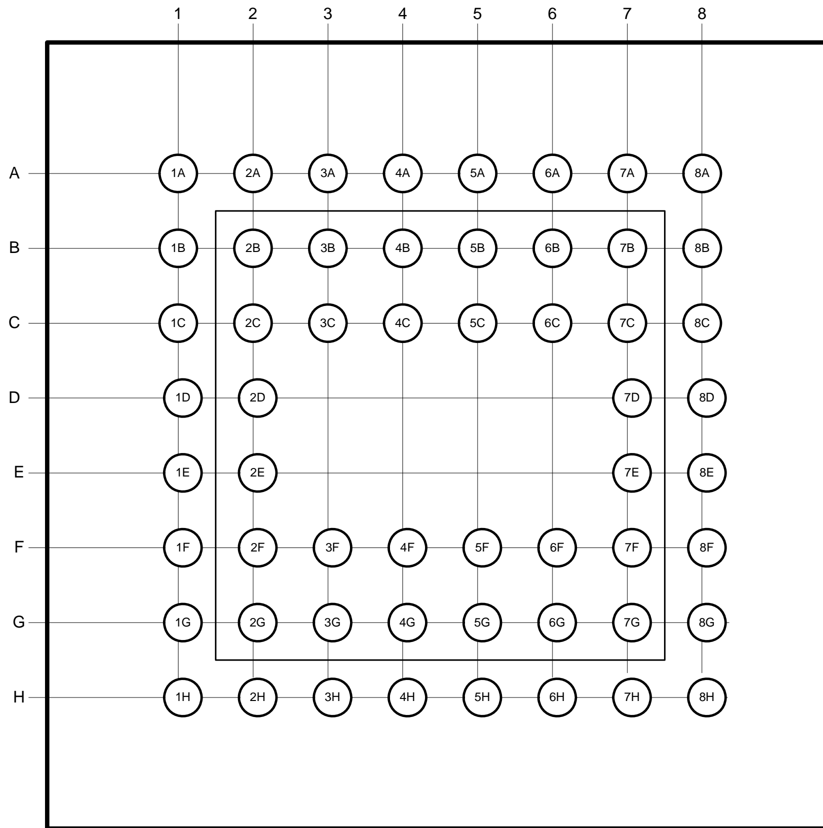





Exhibit 2033 - Page 27 of 346

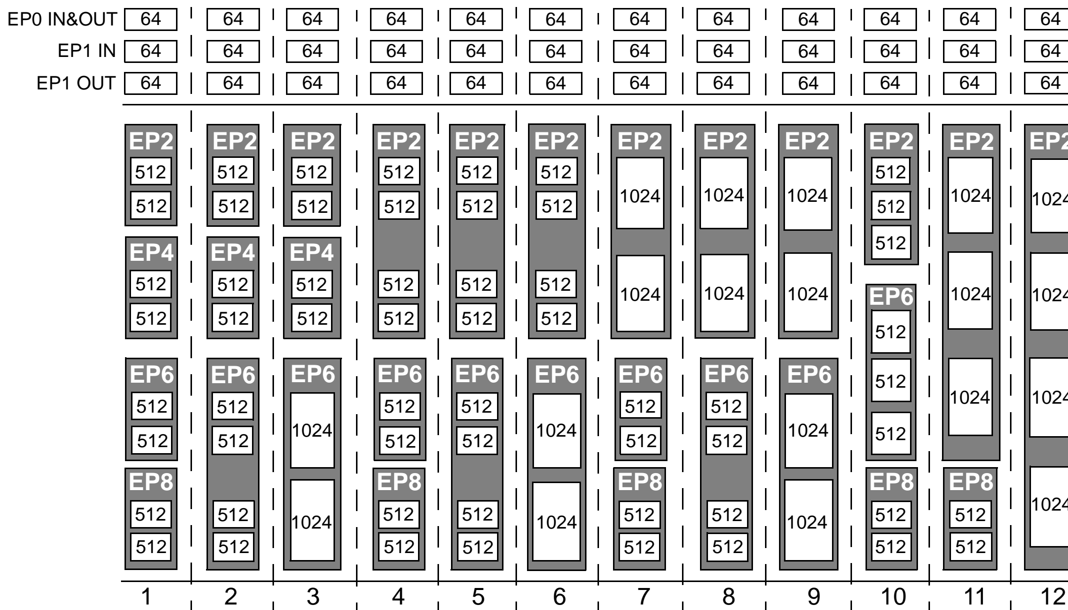
## 1.18 MoBL-USB FX2LP18 Endpoint Buffers

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 MoBL-USB FX2LP18's point of view, an endpoint is a buffer full of bytes received or held for transmission over the bus. The MoBL-USB FX2LP18 reads host data from an OUT endpoint buffer and writes data for transmission to the host to an IN endpoint buffer.

MoBL-USB FX2LP18 contains three 64-byte endpoint buffers plus 4 KB of buffer space that can be configured 12 ways, as indicated in Figure 1-12. The three 64-byte buffers are common to all configurations.

Figure 1-12. MoBL-USB FX2LP18 Endpoint Buffers




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

**Note** The eight SETUP bytes in a CONTROL transfer do not appear in the 64-byte EP0 endpoint buffer. Instead, to simplify programming, the MoBL-USB FX2LP18 automatically stores the eight SETUP bytes in a separate buffer (SETUPDAT, at 0xE6B8-0xE6BF).

EP1IN and EP1OUT use separate 64 byte buffers. MoBL-USB FX2LP18 firmware can configure these endpoints as BULK or INTERRUPT. These endpoints, as well as EP0, are accessible only by 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 the 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 1-12 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.



Exhibit 2033 - Page 28 of 346

Endpoints 2, 4, 6, and 8 can be configured using the choices shown in Table 1-2.

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 MoBL-USB FX2LP18 operates at full-speed (12 Mbps), some or all of the endpoint buffer bytes shown in Figure 1-12 may be employed, depending on endpoint type. **Note** Regardless of the physical buffer size, each 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 MoBL-USB FX2LP18 are designed to move high speed (480 Mbps) USB data on and off the chip without introducing any bandwidth bottlenecks. They accomplish this goal by implementing the following features:

1. Interfaces directly with outside logic, with the MoBL-USB FX2LP18'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, and so on.

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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18'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 MoBL-USB FX2LP18'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 (that is, 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.

MoBL-USB FX2LP18 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.

Figure 1-13 on page 30 and Figure 1-14 on page 31 illustrate the two methods by which external logic interfaces to the endpoint FIFOs EP2, EP4, EP6, and EP8.



Exhibit 2033 - Page 29 of 346

Figure 1-13. MoBL-USB FX2LP18 FIFOs in Slave FIFO Mode

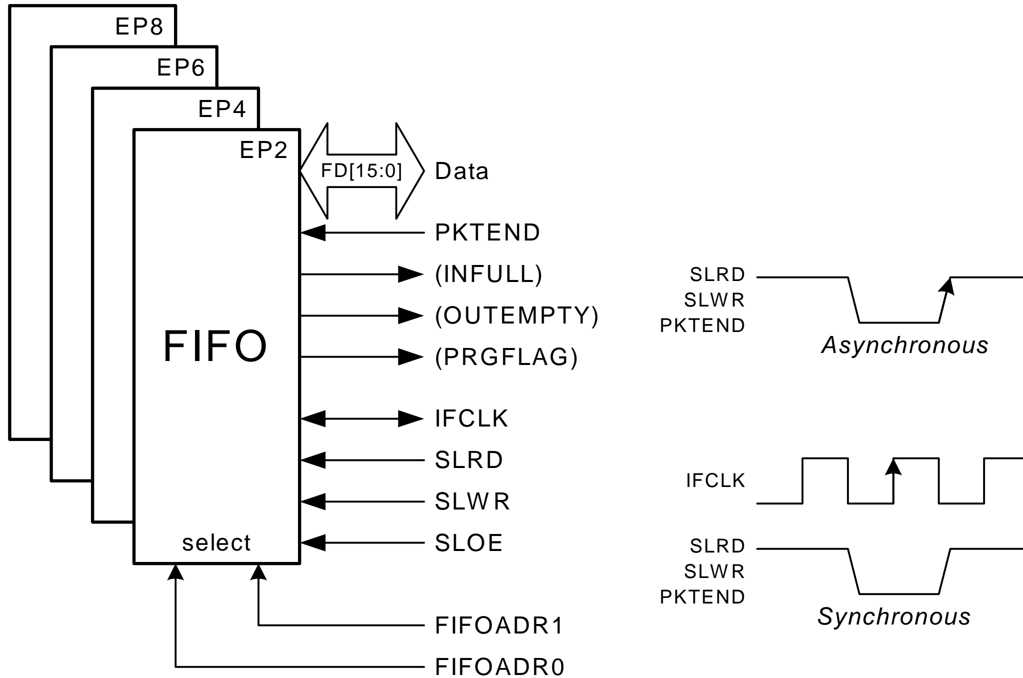


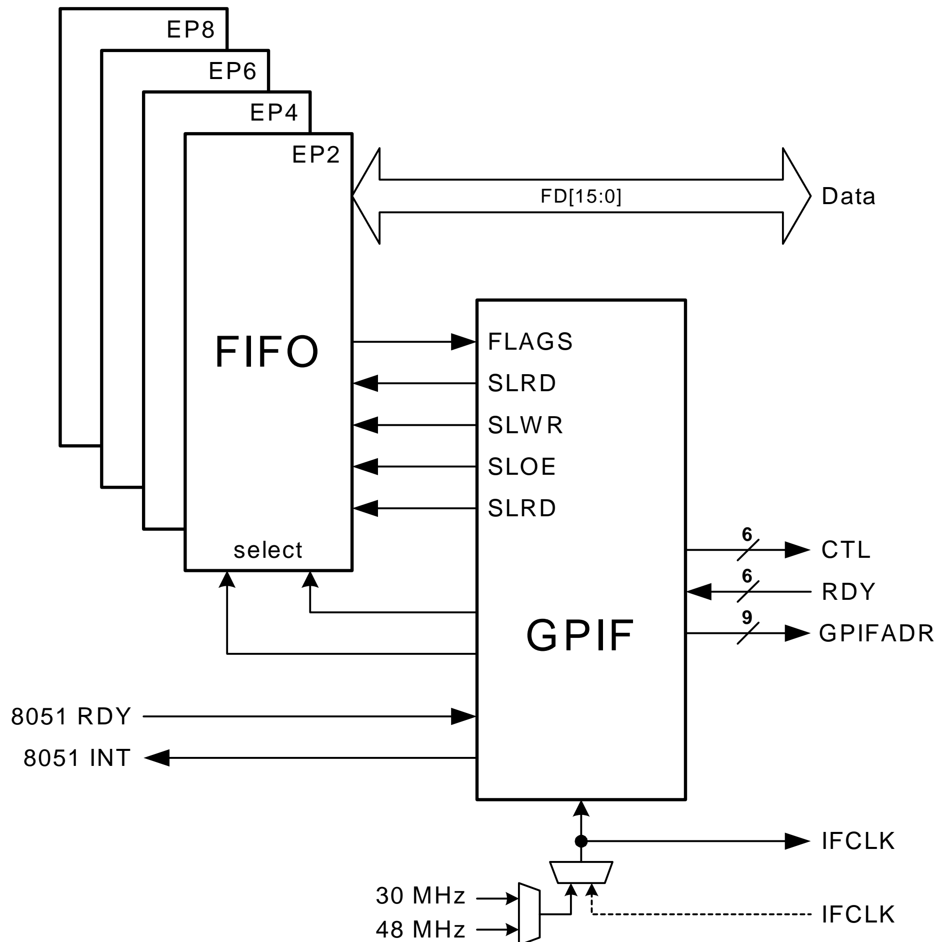

Figure 1-13 illustrates the outside-world view of the MoBL-USB FX2LP18 data FIFOs configured as Slave FIFOs. The outside logic supplies a clock, responds to the FIFO flags, and clocks FIFO data in and out using the strobe signals. Optionally, the outside logic may use the internal MoBL-USB FX2LP18 Interface Clock (IFCLK) as its reference clock.

Three FIFO flags are shown in parentheses in Figure 1-13 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 short (less than max packet size) IN packet to USB.



Exhibit 2033 - Page 30 of 346

Figure 1-14. MoBL-USB FX2LP18 FIFOs in GPIF Master Mode




External systems that connect to the MoBL-USB FX2LP18 FIFOs must provide control circuitry to select FIFOs, check flags, clock data, and so on. The MoBL-USB FX2LP18 contains a sophisticated control unit (the General Programmable Interface, or GPIF) which can replace this external logic. In the GPIF Master FIFO mode (Figure 1-14), 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, implementing '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 MoBL-USB FX2LP18'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 MoBL-USB FX2LP18 Part Number

Table 1-3. MoBL-USB FX2LP18 Part Number (Full-speed and High-speed)

| Part Number | Package | RAM | ISO Support | I/O |
|---|---|---|---|---|
| CY7C68053-56BAXI | 56 VFBGA – Lead-Free | 16 kBytes | Yes | 24 |

## 1.21 Document History

This section is a chronicle of the *MoBL-USB™ FX2LP18 Technical Reference Manual*.

MoBL-USB™ FX2LP18 Technical Reference Manual History

| Release Date | Version | Originator | Description of Change |
|---|---|---|---|
| 01/05/2007 | ** | ARI | This is a new manual. Version 1.0 was printed and released without going through doc control. This is version 1.1; it has an index. |
| 09/30/10 | *A | DSG | Added bit 7 functionality to section 3.5 EEPROM Configuration Byte.<br>Fixed typo in section 9.2.4 (FLAG-FLAGC to FLAGA-FLAGC)<br>Updated Bit 2 BERR Description in Section 13.4.2.1 and 15.8.4.<br>Updated Note in section 8.4. Added Note in section 15.6.2.<br>Updated Bit 3 AUTOIN Description in section 15.6.3.<br>Added Contents Overview. |
| 01/18/2011 | *B | DSG | Sunset ECN - No content change |



Exhibit 2033 - Page 32 of 346

# 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 MoBL-USB FX2LP18 chip handles endpoint zero requests.

The MoBL-USB FX2LP18 provides extensive hardware support for handling endpoint-zero operations; this chapter describes those operations and the resources that simplify the firmware that handles them.

Endpoint zero is the only CONTROL endpoint supported by the MoBL-USB FX2LP18. CONTROL endpoints are *bi-directional*, so the MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 firmware of the burden of tracking the three CONTROL transfer phases (SETUP, DATA, and STATUS). The MoBL-USB FX2LP18 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

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 on page 34 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.

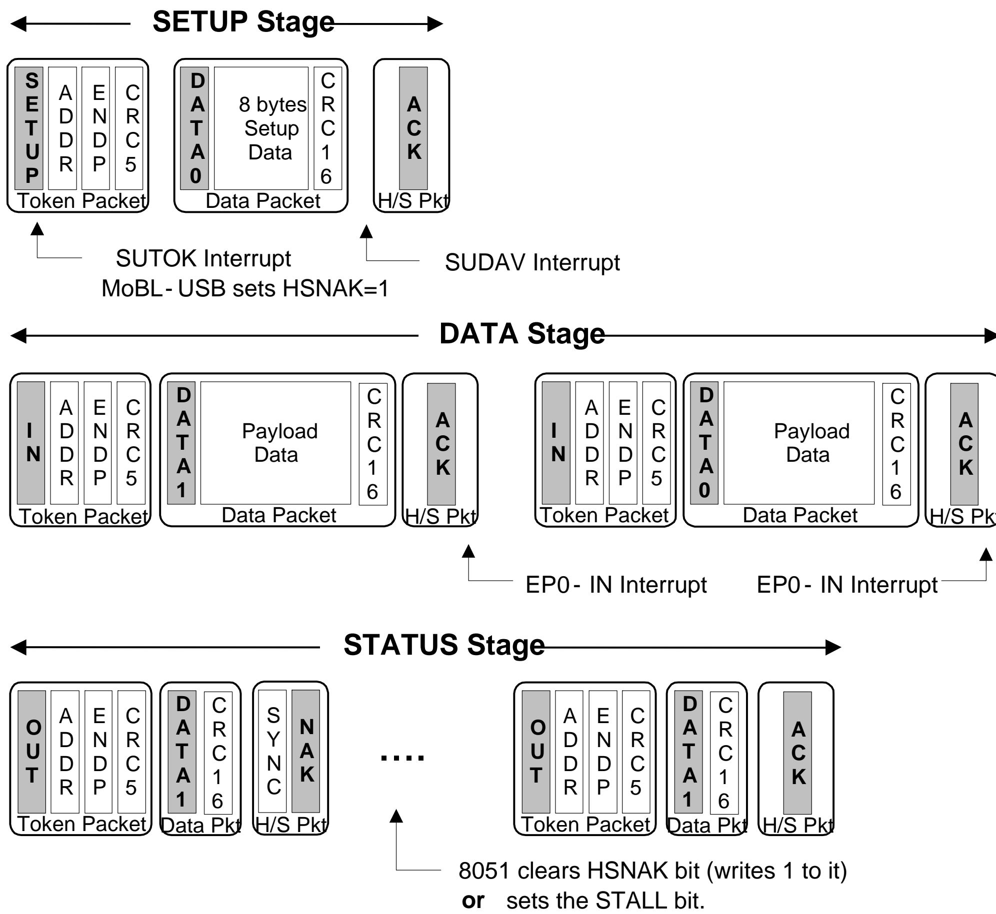
The HSNAK bit holds off the completion of a CONTROL transfer until the device has had time to respond to a request. For example, if the host issues a Set_Interface Request, the MoBL-USB FX2LP18 firmware performs various housekeeping chores such as adjusting internal modes and re-initializing endpoints. During this time, the host issues handshake (STATUS stage) packets to which the MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 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.



Exhibit 2033 - Page 33 of 346

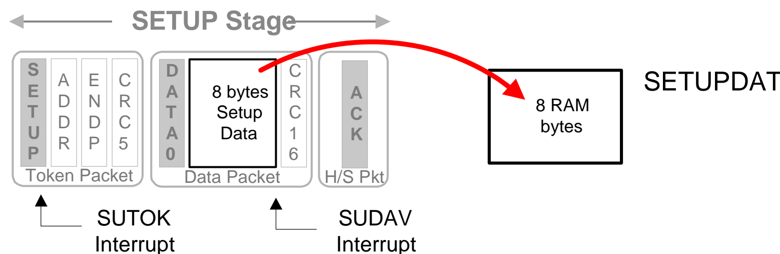
Figure 2-1. A USB Control Transfer (With Data Stage)




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 Figure 2-2.

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




The MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 automatically takes care of any retries if it finds errors in the SETUP data. These two interrupt request bits must be cleared by firmware.



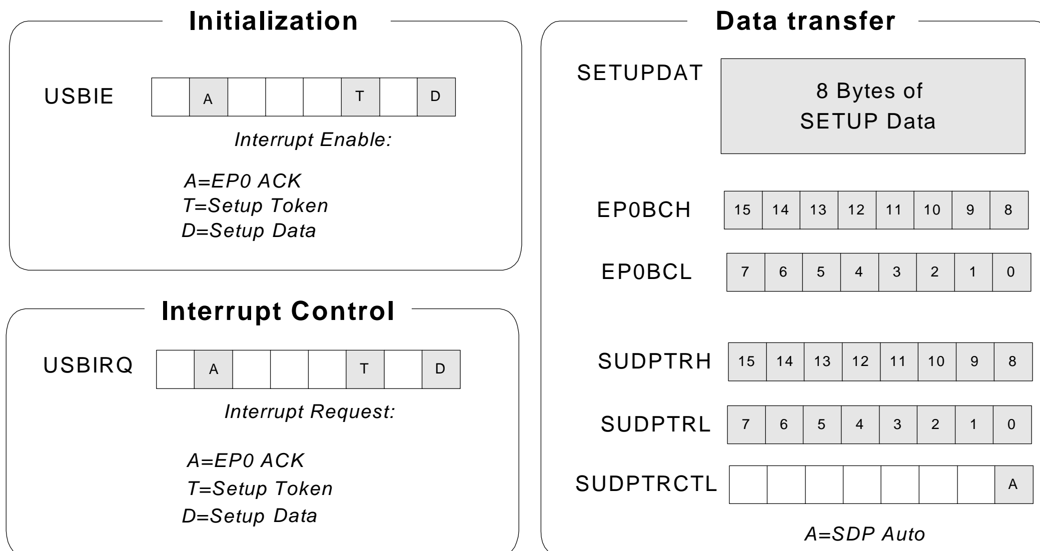
Exhibit 2033 - Page 34 of 346

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 MoBL-USB FX2LP18 automatically clears the stall bit when the next SETUP token arrives.

Like all MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 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, which are described in the Access to Endpoint Buffers chapter on page 93.

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

The MoBL-USB FX2LP18 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.



Exhibit 2033 - Page 35 of 346

## 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 MoBL-USB FX2LP18 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. Table 2-1 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 Table 2-2). |
| 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 in Table 2-2 in detail.

**Note** Table 2-2 applies when RENUM=1, signifying that the firmware, rather than the MoBL-USB FX2LP18 hardware, handles device requests.

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

| bRequest | Name | MoBL-USB FX2LP18 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 on-chip RAM | --- |
| *0xA1 - 0xAF* | | SUDAV Interrupt | Reserved by Cypress Semiconductor |
| *All except 0xA0* | | SUDAV Interrupt | Application dependent |

In the ReNumerated condition (RENUM=1), the MoBL-USB FX2LP18 passes all USB requests except *Set Address* to the firmware via the SUDAV interrupt.

The MoBL-USB FX2LP18 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 0xA0 firmware load request may be used even after ReNumeration, but is only valid while the 8051 is held in reset. If your application



Exhibit 2033 - Page 36 of 346

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 the Enumeration and ReNumeration™ chapter on page 51.

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

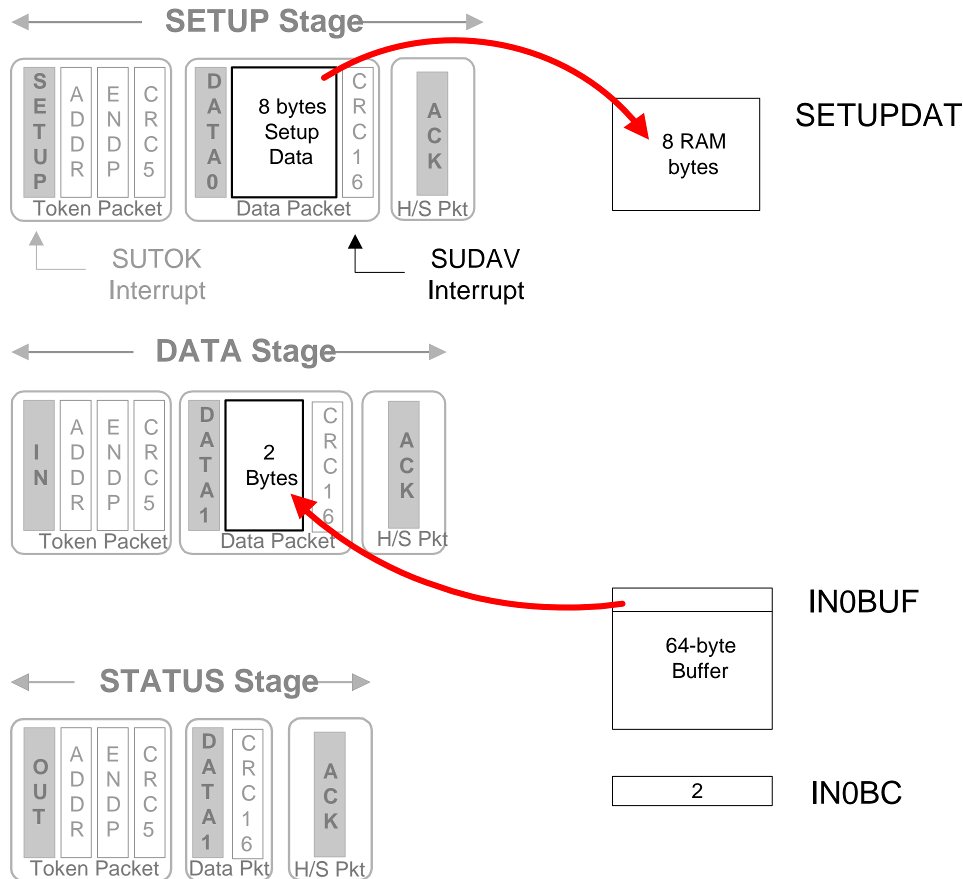
### 2.3.1 Get Status

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

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

The MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 to ACK the status stage of the transfer.



Exhibit 2033 - Page 37 of 346

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 | *Load two bytes into EP0BUF:* |
| 1 | bRequest | **0x00** | 'Get Status' | |
| 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 the Power Management chapter on page 83). 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 MoBL-USB FX2LP18 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.



**Exhibit 2033 - Page 38 of 346**

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, stall an endpoint, or put the device into a specific test mode. 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 | | |

This *Set Feature/Device* request sets 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 on page 38). The host uses this bit to enable or disable remote wakeup by the device.

Table 2-7. Set Feature-Device (Set TEST_MODE Feature)

| Byte | Field | Value | Meaning | Firmware Response |
|---|---|---|---|---|
| 0 | bmRequestType | **0x00** | OUT, Device | *ACK handshake phase* |
| 1 | bRequest | **0x03** | 'Set Feature' | |
| 2 | wValueL | **0x02** | Feature Selector: | |
| 3 | wValueH | 0x00 | TEST_MODE | |
| 4 | wIndexL | 0x00 | | |
| 5 | wIndexH | 0xnn | nn = specific test mode | |
| 6 | wLengthL | 0x00 | | |
| 7 | wLengthH | 0x00 | | |

This *Set Feature/Device* request sets the TEST_MODE feature. This request puts the device into a specific test mode, and power to the device must be cycled in order to exit test mode. The MoBL-USB FX2LP18 SIE handles this request automatically, but the firmware is responsible for acknowledging the handshake phase.



**Exhibit 2033 - Page 39 of 346**

Table 2-8. Set Feature-Endpoint (Stall)

| Byte | Field | Value | Meaning | Firmware Response |
|---|---|---|---|---|
| 0 | bmRequestType | **0x02** | OUT, Endpoint | Set the STALL bit for the indicated endpoint:. |
| 1 | bRequest | **0x03** | 'Set Feature' | |
| 2 | wValueL | **0x00** | Feature Selector: STALL | |
| 3 | wValueH | 0x00 | | |
| 4 | wIndexL | **EP** | 0x00-0x08: OUT0-OUT8 0x80-0x88: IN0-IN8 | |
| 5 | wIndexH | 0x00 | | |
| 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 MoBL-USB FX2LP18 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



Exhibit 2033 - Page 40 of 346

### 2.3.3 Clear Feature

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

Table 2-9. 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 2-10. Clear Feature-Endpoint (Clear Stall)

| Byte | Field | Value | Meaning | Firmware Response |
|---|---|---|---|---|
| 0 | bmRequestType | **0x02** | OUT, Endpoint | *Clear the STALL bit for the* |
| 1 | bRequest | **0x01** | 'Clear 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 | | |

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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 transfers the entire descriptor.



**Exhibit 2033 - Page 41 of 346**

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

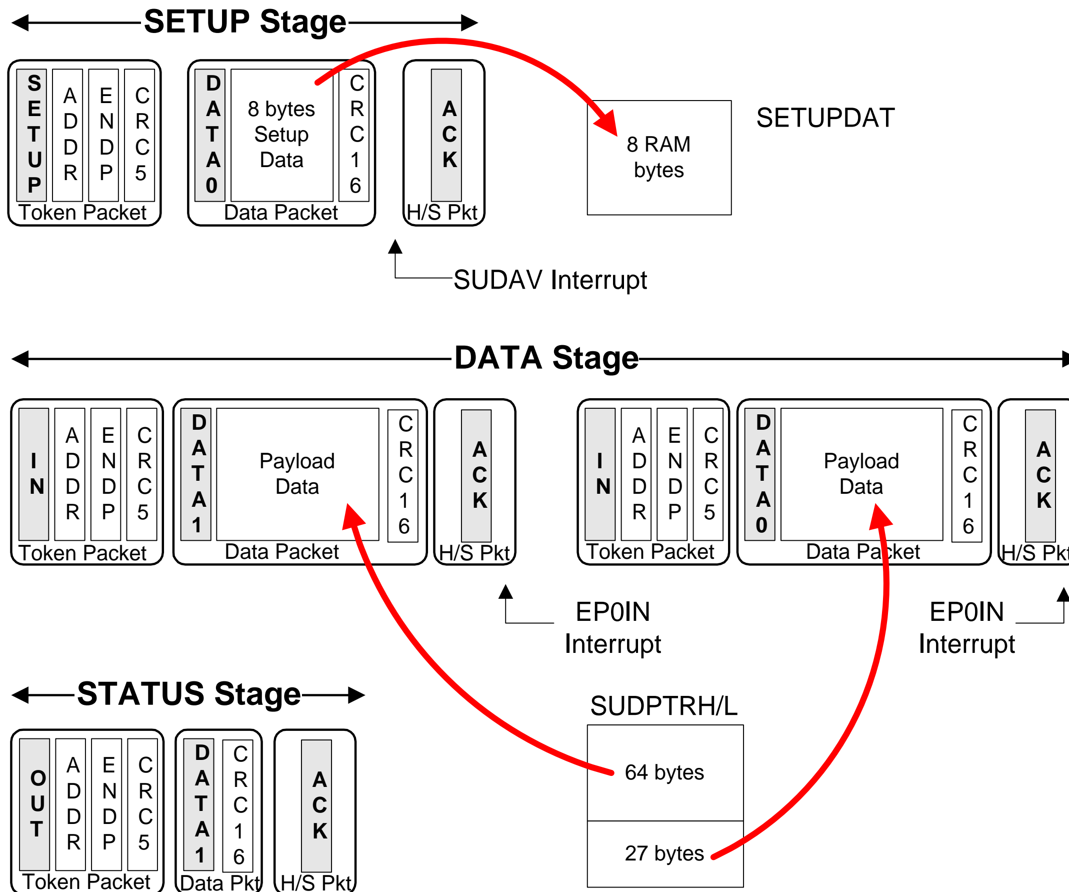



Figure 2-5 illustrates use of the Setup Data Pointer. This pointer is implemented as two registers, SUDPTRH and SUDPTRL. The base address of SUDPTRH:L must be word-aligned. Most *Get Descriptor* requests involve transferring more data than fits into one packet. In the Figure 2-5 example, the descriptor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

The CONTROL transaction starts in the usual way, with the MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 to automatically respond to two IN transfers with 64 bytes and 27 bytes of data using SUDPTRH:L 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 2-11. Get Descriptor-Device

| Byte | Field | Value | Meaning | Firmware Response |
|---|---|---|---|---|
| 0 | bmRequestType | **0x80** | IN, Device | *Set SUDPTR H:L to start of* |
| 1 | bRequest | **0x06** | 'Get Descriptor' | *Device Descriptor table in RAM.* |
| 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 on page 42, the firmware loads the 2 byte SUDPTRH:L with the starting address of the Device Descriptor table. The start address needs to be word-aligned. When SUDPTRL is loaded, the MoBL-USB FX2LP18 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 2-11).
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 MoBL-USB FX2LP18 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 (for example, *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 do not contain the length of the transfer, the length can be loaded explicitly (see the SDPAUTO paragraphs of section 8.7 The Setup Data Pointer on page 104).

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 MoBL-USB FX2LP18 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 does not.



**Exhibit 2033 - Page 43 of 346**

### 2.3.4.2 Get Descriptor-Device Qualifier

Table 2-12. Get Descriptor-Device Qualifier

| Byte | Field | Value | Meaning | Firmware Response |
|---|---|---|---|---|
| 0 | bmRequestType | **0x80** | IN, Device | *Set SUDPTR H:L to start of* |
| 1 | bRequest | **0x06** | 'Get Descriptor' | *the appropriate Device Qualifier* |
| 2 | wValueL | 0x00 | | *Descriptor table in RAM.* |
| 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 (for example, 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 (must be word-aligned) into SUDPTRH:L, then the MoBL-USB FX2LP18 does the rest.

### 2.3.4.3 Get Descriptor-Configuration

Table 2-13. Get Descriptor-Configuration

| Byte | Field | Value | Meaning | Firmware Response |
|---|---|---|---|---|
| 0 | bmRequestType | **0x80** | IN, Device | *Set SUDPTR H:L to start of* |
| 1 | bRequest | **0x06** | 'Get Descriptor' | *Configuration Descriptor table in* |
| 2 | wValueL | **CFG** | Configuration Number | *RAM* |
| 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 2-14. Get Descriptor-String

| Byte | Field | Value | Meaning | Firmware Response |
|---|---|---|---|---|
| 0 | bmRequestType | **0x80** | IN, Device | *Set SUDPTR H:L to start of* |
| 1 | bRequest | **0x06** | 'Get Descriptor' | *String Descriptor table in* |
| 2 | wValueL | **STR** | String Number | *RAM.* |
| 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 (must be word-aligned) into SUDPTRH:L. The MoBL-USB FX2LP18 does the rest.



**Exhibit 2033 - Page 44 of 346**

### 2.3.4.5 *Get Descriptor-Other Speed Configuration*

Table 2-15. Get Descriptor-Other Speed Configuration

| Byte | Field | Value | Meaning | Firmware Response |
|---|---|---|---|---|
| 0 | bmRequestType | **0x80** | IN, Device | *Set SUDPTR H:L to start of* |
| 1 | bRequest | **0x06** | ‘Get Descriptor’ | *Other Speed Configuration* |
| 2 | wValueL | **CFG** | Other Speed Configuration Number | *Descriptor table in RAM.* |
| 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 configurations of the device if it were operating at the other speed (for example, 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 (must be word-aligned) into SUDPTRH:L, then the MoBL-USB FX2LP18 does the rest.

## 2.3.5 Set Descriptor

Table 2-16. Set Descriptor-Device

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

Table 2-17. Set Descriptor-Configuration

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



**Exhibit 2033 - Page 45 of 346**

Table 2-18. Set Descriptor-String

| Byte | Field | Value | Meaning | Firmware Response |
|---|---|---|---|---|
| 0 | bmRequestType | **0x00** | IN, Device | *Read string descriptor data over EP0BUF.* |
| 1 | bRequest | **0x07** | 'Set 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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 automatically responds to the STATUS phase, which is the third and final stage of the CONTROL transfer.

**Note** 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 MoBL-USB FX2LP18 takes no action when these settings change. However, the firmware **must re-initialize endpoints and reset the data toggle** 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.

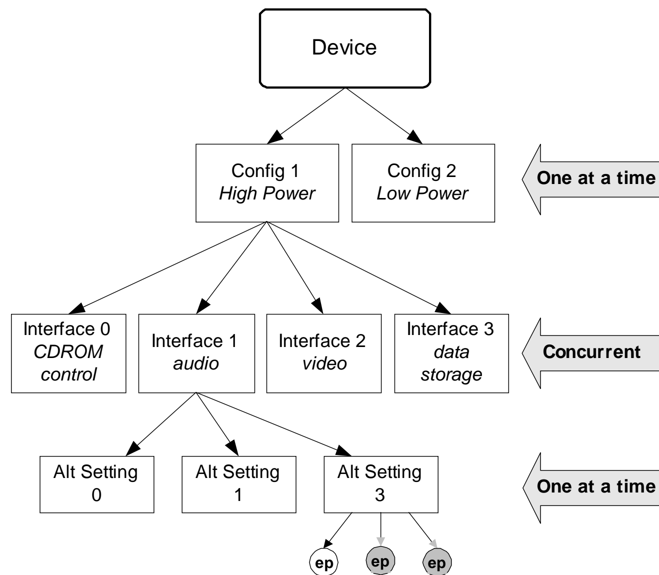





Exhibit 2033 - Page 46 of 346

#### 2.3.5.1 *Set Configuration*

Table 2-19. 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 Table 2-19), 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.

**Note** 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 2-20. 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 2-21. 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 2-22. 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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 ReNumerates™ (see Enumeration and ReNumeration™, on page 51), it automatically resets FNADDR to zero, allowing the device to come back as *new*.

A MoBL-USB FX2LP18 program does not need to know the device address, because the MoBL-USB FX2LP18 automatically responds only to the host-assigned FNADDR value. The device address is readable only for debug/diagnostic purposes.



Exhibit 2033 - Page 48 of 346

### 2.3.10 Sync Frame

Table 2-23. 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 MoBL-USB FX2LP18 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 micro-frames. 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 2-24. 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 2-25. 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 MoBL-USB FX2LP18 responds to two endpoint zero vendor requests, RAM Download and RAM Upload. These requests are active whether RENUM=0 or RENUM=1, but can only occur while the 8051 is held in reset. RAM Uploads can only occur on word boundaries (that is, the start address must be evenly divisible by 2). The same restriction does not apply to RAM Downloads.

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 MoBL-USB FX2LP18 chip.

A host loader program must write 0x01 to the CPUCS register to put the MoBL-USB FX2LP18's CPU into RESET, load all or part of the MoBL-USB FX2LP18's internal RAM with code, then reload the CPUCS register with 0 to take the CPU out of RESET.



Exhibit 2033 - Page 50 of 346

# 3. Enumeration and ReNumeration™

## 3.1 Introduction

The MoBL-USB FX2LP18's configuration is 'soft': Code and data are stored in internal RAM, which can be loaded from the host over the USB interface. MoBL-USB FX2LP18-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 MoBL-USB FX2LP18 is capable of enumerating as a USB device with minimal 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. However, at a minimum, an $I^2C$™ boot EEPROM is required (see "MoBL-USB FX2LP18 Startup" on page 51 for more details).

**Note** For the MoBL-USB FX2LP18, two separate Default USB Devices actually exist, one for enumeration as a full-speed (12 Mbps) device, and the other for enumeration as a high-speed (480 Mbps) device. The MoBL-USB FX2LP18 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 and the host downloads firmware and descriptor tables to the MoBL-USB FX2LP18, it then begins executing the downloaded code, which electrically simulates a physical disconnect/connect from the USB and causes the MoBL-USB FX2LP18 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™.'

A MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 Startup

During the power-up sequence, internal logic checks the I2C port for the connection of an EEPROM whose first byte is 0xC2. If an EEPROM containing firmware is attached to the I2C bus, the firmware is automatically loaded from the EEPROM into the MoBL-USB FX2LP18'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 MoBL-USB FX2LP18, so this mode is called a 'C2 Load'. **Note** Although the MoBL-USB FX2LP18 can perform C2 Loads from EEPROMs as large as 64 kB, code can only be downloaded to the 16K of on chip RAM.

It is important to note that the MoBL-USB FX2LP18 comes out of reset with the DISCON bit set. This means that the device is effectively disconnected from the USB bus. Firmware is required to connect to USB. So the device will not enumerate without an EEPROM (a C2 Load), to download firmware that at least connects the device to USB.

If no EEPROM is present on the I2C port, an external processor must emulate an I2C slave. The MoBL-USB FX2LP18 does not enumerate using internally stored descriptors (that is, Cypress' VID/PID/DID is not used for enumeration).

It is still possible to download firmware over USB once the initial C2 Load from EEPROM occurs, and the device is connected to USB by clearing the DISCON bit and indicates that the Default USB Device will handle USB requests by clearing the RENUM bit. In this case, the firmware loaded from the EEPROM can be as simple as a one line code to clear the DISCON bit and RENUM bit in the USBCS register.

Section 3.8 MoBL-USB FX2LP18 Vendor Request for Firmware Load on page 56 describes the USB 'Vendor Request' that the MoBL-USB FX2LP18 supports for code download and upload.



**Exhibit 2033 - Page 51 of 346**

**Note** The Default USB Device is fully characterized in Appendices A and B, which list the built-in 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. | | | | |

**Note** 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 specification, which allows only 512-byte bulk endpoints. If you use this default alternate setting, do not send/receive EP1 packets larger than 64 bytes.



Exhibit 2033 - Page 52 of 346

## 3.4 ‘C2’ EEPROM Boot-load Data Format

This section describes the ‘C2’ EEPROM boot-load format.

If, at power-on reset, the MoBL-USB FX2LP18 detects an EEPROM connected to its I2C with the value **0xC2** at address zero, the MoBL-USB FX2LP18 loads the EEPROM data into on chip 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 Table 3-3.

Table 3-3. ‘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 MoBL-USB FX2LP18 to copy the EEPROM data into on chip RAM. The MoBL-USB FX2LP18 reads the next six bytes (VID / PID / DID) even though they are not used by most C2 Load applications. The eighth byte (byte 7) is the configuration byte described in the previous section.

**Note** 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 (for example, MoBL-USB FX2LP18 logic handles device requests), using the EEPROM VID / PID / DID

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 14-bit Start Address (0-0x3FFF) 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.



**Exhibit 2033 - Page 53 of 346**

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

- Program / Data RAM at 0x0000-0x3FFF
- Data RAM at 0xE000-0xE1FF
- The CPUCS register at 0xE600 (only bit 0, 8051RES, is EEPROM-loadable).

### General Purpose Use of the I2C Bus

The MoBL-USB FX2LP18's I2C 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 controller for general-purpose use. This makes a wide range of standard I2C peripherals available to an MoBL-USB FX2LP18-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. The Input/Output chapter on page 203 describes the general-purpose nature of the I2C interface.

## 3.5 EEPROM Configuration Byte

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

Figure 3-1. EEPROM Configuration Byte

| Configuration | | | | | | | |
|---|---|---|---|---|---|---|---|
| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
| **0** | **DISCON** | **0** | **0** | **0** | **0** | **0** | **400 kHz** |

| Bit | Name | Description |
|---|---|---|
| **7** | | The config byte of the .iic file must be changed so that the FX2 will start at full speed. The uVision project file contains the following iic generation line:<br><br>c:\cypress\usb\bin\hex2bix -c 0x80 -i -f 0xC2 -o bulkloop.iic bulkloop.hex<br><br>This line sets bit 7 of the config byte with the "-c 0x80". Setting bit 7 prevents the FX2 from entering high-speed mode on startup. |
| **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 MoBL-USB FX2LP18.<br><br>The MoBL-USB FX2LP18 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 MoBL-USB FX2LP18; it appears to have been disconnected from the USB. When the supply is then driven, the MoBL-USB FX2LP18 appears to have been newly-connected to the USB. From the host's point of view, the MoBL-USB FX2LP18 can be disconnected and re-connected to the USB, without ever *physically* disconnecting. |
| **0** | **400KHz** (I2C bus speed) | 0 100 kHz<br>1 400 kHz<br>If 400KHZ=0, the I2C bus operates at approximately 100 kHz. If 400KHZ=1, the I2C bus operates at approximately 400 kHz. This bit is copied to I2CTL.0, whose default value is 0, or 100 kHz. Once the CPU is running, firmware can modify this bit. |



Exhibit 2033 - Page 54 of 346

## 3.6 The RENUM Bit

A MoBL-USB FX2LP18 control bit called 'RENUM' (ReNumerated) determines whether USB device requests over endpoint zero are handled by the Default USB Device or by 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 MoBL-USB FX2LP18 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. The Endpoint Zero chapter on page 33 describes how the firmware handles device requests while RENUM=1.

### 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 MoBL-USB FX2LP18's RAM. Another useful feature of the Default USB Device is that code can be written to support the already-configured generic USB device. Before bringing the CPU out of reset, the MoBL-USB FX2LP18 automatically enables certain endpoints and reports them to the host via descriptors. By utilizing the Default USB Device (for example, 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 MoBL-USB FX2LP18 Response to Device Requests (RENUM=0)

Table 3-4 shows how the Default USB Device responds to endpoint zero device requests when RENUM=0.

Table 3-4. How the Default USB Device Handles EP0 Requests When RENUM=0

| bRequest | Name | MoBL-USB FX2LP18 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 | Sets TEST_MODE feature |
| 0x03 | Set Feature-Endpoint | Sets Stall bit for indicated EP |
| 0x04 | (reserved) | None |
| 0x05 | Set Address | Updates FNADDR 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 on chip RAM |
| 0xA1-0xAF | Reserved | Reserved by Cypress Semiconductor |
| all other | | None |



Exhibit 2033 - Page 55 of 346

A USB host enumerates by issuing *Set Address*, *Get Descriptor*, and *Set Configuration* (to 1) requests (the *Set Address* and *Get Descriptor* 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 on chip RAM

## 3.8 MoBL-USB FX2LP18 Vendor Request for Firmware Load

Prior to ReNumeration, the host downloads data into the MoBL-USB FX2LP18's internal RAM. The host can access two **on chip** RAM spaces — Program / Data RAM at 0x0000-0x3FFF and Data RAM at 0xE000-0xE1FF — **which it can download or upload only when the CPU is held in reset**. The host must write to the CPUCS register to put the CPU in or out of reset. These two RAM spaces may also be boot-loaded by a 'C2' EEPROM connected to the I2C bus.

The USB Specification provides for 'vendor-specific requests' to be sent over endpoint zero. The MoBL-USB FX2LP18 uses this feature to transfer data between the host and MoBL-USB FX2LP18 RAM. The MoBL-USB FX2LP18 automatically responds to two 'Firmware Load' requests, as shown in Table 3-5 and Table 3-6.

Table 3-5. Firmware Download

| Byte | Field | Value | Meaning | MoBL-USB FX2LP18 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 3-6. Firmware Upload

| Byte | Field | Value | Meaning | MoBL-USB FX2LP18 Response |
|---|---|---|---|---|
| 0 | bmRequest | **0xC0** | Vendor Request, IN | None required |
| 1 | bRequest | **0xA0** | 'Firmware Load' | |
| 2 | wValueL | **AddrL** | Starting Address (must be word-aligned) | |
| 3 | wValueH | **AddrH** | | |
| 4 | wIndexL | 0x00 | | |
| 5 | wIndexH | 0x00 | | |
| 6 | wLengthL | **LenL** | Number of Bytes | |
| 7 | wLengthH | **LenH** | | |

**Note** These upload and download requests are always handled by the MoBL-USB FX2LP18, **regardless** of the state of the RENUM bit. The upload start address **must** be word-aligned (that is, the start address must be evenly divisible by 2).

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 must write 0x01 to the CPUCS register to put the CPU into RESET, load all or part of the MoBL-USB FX2LP18 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 register that can be written using the Firmware Download command.



Exhibit 2033 - Page 56 of 346

## 3.9 How the Firmware ReNumerates

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

Figure 3-2. USB Control and Status Register

| 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 | 1 | 1 | 0 | 0 |

To simulate a USB connect, the firmware clears DISCON to 0.

The firmware also 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™

MoBL-USB FX2LP18 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.



Exhibit 2033 - Page 57 of 346

# 4. Interrupts

## 4.1 Introduction

The MoBL-USB FX2LP18's interrupt architecture is an enhanced and expanded version of the standard 8051's. The MoBL-USB FX2LP18 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. MoBL-USB FX2LP18 Interrupts

| MoBL-USB FX2LP18 Interrupt | Source | Interrupt Vector | Natural Priority |
|---|---|---|---|
| IE0 | INT0# Pin | 0x0003 | 1 |
| TF0 | Timer 0 Overflow | 0x000B | 2 |
| IE1 | 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 Bus** | 0x004B | 9 |
| **IE4** | **GPIF / FIFOs / INT4 Pin** | 0x0053 | 10 |
| **IE5** | **INT5# Pin** | 0x005B | 11 |
| **IE6** | **INT6 Pin** | 0x0063 | 12 |

The **Natural Priority** column in Table 4-1 shows the MoBL-USB FX2LP18 interrupt priorities. The MoBL-USB FX2LP18 can assign each interrupt to a high or low priority group; priorities are resolved within the groups using the natural priorities.

## 4.2 SFRs

The following SFRs are associated with interrupt control:

- IE - SFR 0xA8 (Table 4-2 on page 60)
- IP - SFR 0xB8 (Table 4-3 on page 60)
- EXIF - SFR 0x91 (Table 4-4 on page 60)
- EICON - SFR 0xD8 (Table 4-5 on page 61)
- EIE - SFR 0xE8 (Table 4-6 on page 61)
- EIP - SFR 0xF8 (Table 4-7 on page 61)

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.



Exhibit 2033 - Page 59 of 346

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 (IE1). EX1=1 enables interrupts generated by the 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 (IE0). EX0=1 enables interrupts generated by the 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 (IE1) 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 (IE0) 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. IE4 must be cleared by software. Setting IE4 in software generates an interrupt, if enabled. |
| EXIF.5 | **I2CINT** - I2C Bus Interrupt flag. I2CINT = 1 indicates an I2C 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. |



Exhibit 2033 - Page 60 of 346

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 WU2 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 IE6 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 (IE6). EX6 = 1 enables interrupts generated by the INT6 pin. |
| EIE.3 | EX5 - Enable external interrupt 5. EX5 = 0 disables external interrupt 5 (IE5). EX5 = 1 enables interrupts generated by the INT5# pin. |
| EIE.2 | EX4 - Enable external interrupt 4. EX4 = 0 disables external interrupt 4 (IE4). EX4 = 1 enables interrupts generated by the INT4 pin or by the FIFO/GPIF Interrupt. |
| EIE.1 | EI2C - Enable I2C Bus interrupt (I2CINT). EI2C = 0 disables the I2C Bus interrupt. EI2C = 1 enables interrupts generated by the I2C 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 (IE6) 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 (IE5) 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 Bus interrupt to low priority. PI2C=1 sets I2C 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. |



**Exhibit 2033 - Page 61 of 346**

### 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 MoBL-USB FX2LP18.

Table 4-8. Summary of Interrupt Compatibility

| Feature | Intel 8051 | Dallas DS80C320 | Cypress MoBL-USB FX2LP18 |
|---|---|---|---|
| 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 Bus Interrupt |
| External Interrupt 4 | Not implemented | Implemented | Replaced by autovectored FIFO/GPIF Interrupt. |
| 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 MoBL-USB FX2LP18 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 on page 63). The MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 continues executing firmware at the instruction following the one which was executing when the interrupt occurred.

**Note** The MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 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.



Exhibit 2033 - Page 62 of 346

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 |
| IE0 | External interrupt 0 | TCON.1 | IE.0 | IP.0 | 1 | 0x0003 |
| TF0 | Timer 0 interrupt | TCON.5 | IE.1 | IP.1 | 2 | 0x000B |
| IE1 | 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 Bus interrupt | EXIF.5 | EIE.1 | EIP.1 | 9 | 0x004B |
| IE4 | Autovectored FIFO / GPIF or External interrupt 4 | EXIF.6 | EIE.2 | EIP.2 | 10 | 0x0053 |
| IE5 | External interrupt 5 | EXIF.7 | EIE.3 | EIP.3 | 11 | 0x005B |
| IE6 | 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 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 (that is, 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 MoBL-USB FX2LP18 sets the IEx flag when the INTx# pin is sampled low. When ITx = 1, INTx# is edge-sensitive and the MoBL-USB FX2LP18 sets the IEx flag when the INTx# pin is sampled high then low on consecutive samples.

The remaining five interrupts (INT 4-6, USB & I2C 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.



**Exhibit 2033 - Page 63 of 346**

### 4.3.3 Interrupt Latency

Interrupt response time depends on the current state of the MoBL-USB FX2LP18. The fastest response time is five instruction cycles: one to detect the interrupt, and four to perform the `LCALL` to the ISR.

The maximum latency is 13 instruction cycles. This 13-cycle latency occurs when the MoBL-USB FX2LP18 is currently executing a `RETI` instruction followed by a `MUL` or `DIV` instruction. The 13 instruction cycles in this case are: one to detect the interrupt, three to complete the `RETI`, five to execute the `DIV` or `MUL`, and four to execute the `LCALL` to the ISR.

This 13-instruction-cycle latency excludes autovector latency for the USB and FIFO/GPIF interrupts (see sections 4.5 USB-Interrupt Autovectors on page 69 and 4.8 FIFO/GPIF-Interrupt Autovectors on page 74), and any instructions required to perform housekeeping, as shown in Figure 4-2. Autovectoring adds a fixed four 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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 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.

The Power Management chapter on page 83 describes suspend/resume signaling in detail, and presents an example which uses the Wakeup Interrupt.

### 4.4.2 USB Interrupts

Table 4-10 on page 65 shows the 27 USB requests that share the USB Interrupt. Figure 4-1 on page 66 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.



Exhibit 2033 - Page 64 of 346

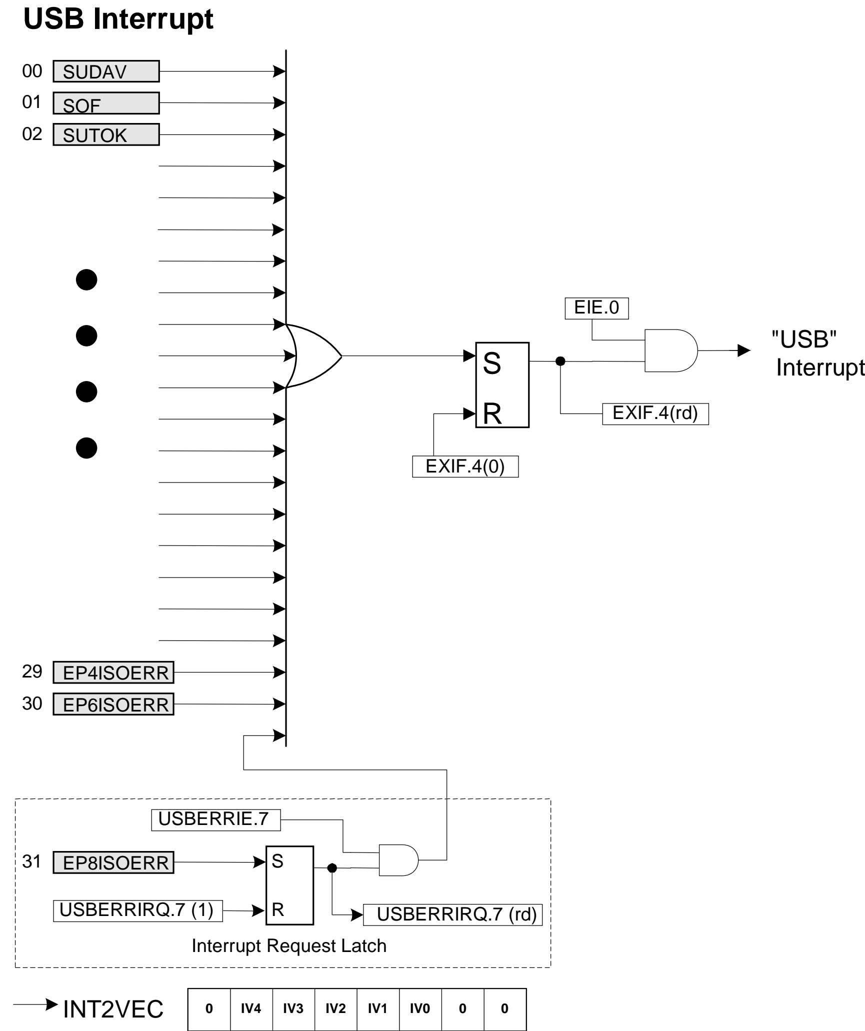
Table 4-10. Individual USB Interrupt Sources

| 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 | MoBL-USB FX2LP18 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 NAKed* |
| 20 | 4C | EP1PING | EP1 OUT was Pinged and it NAKed* |
| 21 | 50 | EP2PING | EP2 OUT was Pinged and it NAKed* |
| 22 | 54 | EP4PING | EP4 OUT was Pinged and it NAKed* |
| 23 | 58 | EP6PING | EP6 OUT was Pinged and it NAKed* |
| 24 | 5C | EP8PING | EP8 OUT was Pinged and it NAKed* |
| 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 |



Exhibit 2033 - Page 65 of 346

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 MoBL-USB FX2LP18; 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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18's program memory. This causes the MoBL-USB FX2LP18 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 on page 69.

Due to the OR gate in Figure 4-1, assertion of any of the individual USB interrupt sources sets the MoBL-USB FX2LP18'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, MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 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 Figure 4-1, starting with SUDAV (priority 00) as the highest priority, and ending with EP8ISOERR (priority 31) as the lowest.



Exhibit 2033 - Page 66 of 346

**Note** 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’**.

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.

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

```
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
```

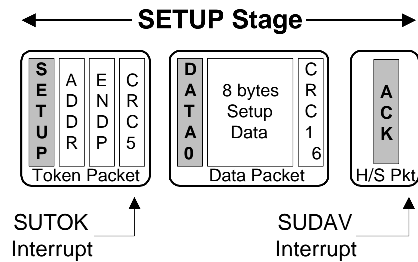
The registers associated with the individual USB interrupt sources are described in the Registers chapter on page 237 and section 8.6 CPU Control of MoBL-USB FX2LP18 Endpoints on page 96. Each interrupt source has an enable (IE) and a request (IRQ) bit. Firmware sets the IE bit to ‘1’ to enable the interrupt. The MoBL-USB FX2LP18 sets an IRQ bit to ‘1’ to request an interrupt, and the firmware clears an IRQ bit by writing a ‘1’ to it.



**Exhibit 2033 - Page 67 of 346**

#### 4.4.2.1 SUTOK, SUDAV Interrupts

Figure 4-3. SUTOK and SUDAV Interrupts



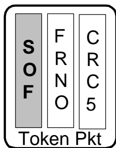

SUTOK and SUDAV are supplied to the MoBL-USB FX2LP18 by CONTROL endpoint zero. The first portion of a USB CONTROL transfer is the SETUP stage shown in Figure 4-3 (a full CONTROL transfer is shown in Figure 2-1 on page 34). When the MoBL-USB FX2LP18 decodes a SETUP packet, it asserts the SUTOK (SETUP Token) Interrupt Request. After the MoBL-USB FX2LP18 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. See chapter “Endpoint Zero” on page 33.

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 Mbps) mode, and once every 125 microseconds in high-speed (480 Mbps) mode.

When the MoBL-USB FX2LP18 receives an SOF packet, it copies the eleven-bit frame number (FRNO in Figure 4-4) into the USBFRAMEH:L registers and asserts the SOF Interrupt Request. Isochronous endpoint data may be serviced via the SOF Interrupt.

#### 4.4.2.3 Suspend Interrupt

If the MoBL-USB FX2LP18 detects a ‘suspend’ condition from the host, it asserts the SUSP (Suspend) Interrupt Request. A full description of Suspend-Resume signaling appears in the Power Management chapter on page 83.

#### 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 MoBL-USB FX2LP18 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 FX2LP18.

#### 4.4.2.6 EP0ACK Interrupt

This interrupt is asserted when the MoBL-USB FX2LP18 has acknowledged the STATUS stage of a CONTROL transfer on endpoint 0.



Exhibit 2033 - Page 68 of 346

#### 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 MoBL-USB FX2LP18, and is in the endpoint buffer memory.

For an IN endpoint, the interrupt request signifies that the data previously loaded by the MoBL-USB FX2LP18 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

| Interrupt Name | Description |
|---|---|
| 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)<br>OUT: Buffer has received USB data (Empty Flag 0-to-1) |
| EP4 | IN: Buffer available (Empty Flag 1-to-0)<br>OUT: Buffer has received USB data (Empty Flag 0-to-1) |
| EP6 | IN: Buffer available (Empty Flag 1-to-0)<br>OUT: Buffer has received USB data (Empty Flag 0-to-1) |
| EP8 | IN: Buffer available (Empty Flag 1-to-0)<br>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 MoBL-USB FX2LP18 automatically NAKs the IN token and asserts this interrupt.

#### 4.4.2.9 EPxPING Interrupt

These interrupts are active only during high-speed (480 Mbps) operation.

High-speed USB implements 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 (for example, if it has an empty buffer). If a buffer is not available, the FX2LP18 returns a NAK handshake. PING-NAK transactions continue to occur until an OUT buffer is available, at which time the FX2LP18 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 FX2LP18 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 USBERRIE, USBERRIRQ, ERRCNTLIM, CLRERRCNT on page 103 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).

## 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 MoBL-USB FX2LP18 provides a second level of interrupt vectoring, called 'Autovectoring.' When a USB interrupt is asserted, the MoBL-USB FX2LP18 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.



Exhibit 2033 - Page 69 of 346

The MoBL-USB FX2LP18 jump instruction is encoded as follows:

Table 4-12. MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 substitutes its INT2VEC byte (see Table 4-10 on page 65) for the byte at address 0x0045. Therefore, if the high byte ('page') of a jump-table address is pre-loaded 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 |



Exhibit 2033 - Page 70 of 346

### 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 two 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

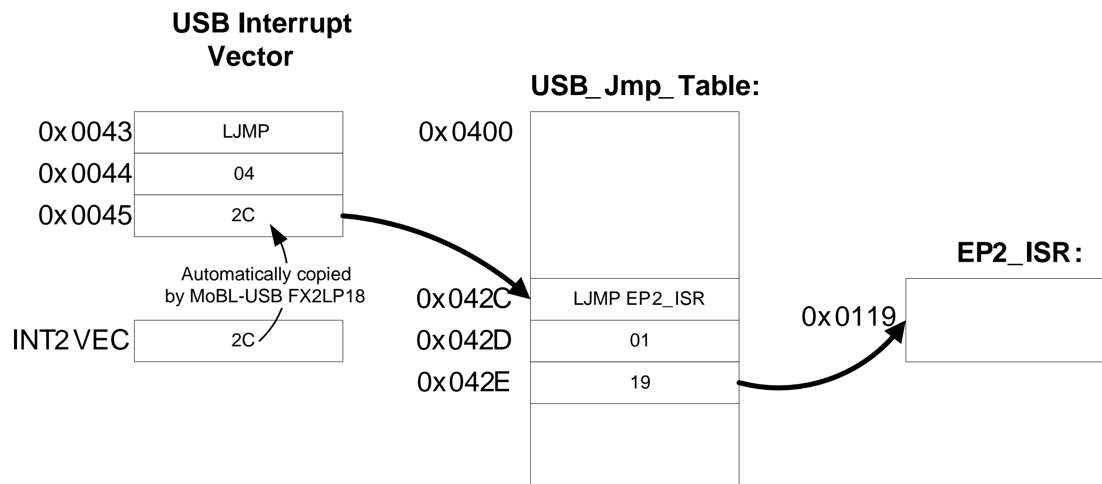



Figure 4-5 illustrates an ISR that services endpoint 2. When endpoint 2 requires service, the MoBL-USB FX2LP18 asserts the USB interrupt request, vectoring to location 0x0043.

The jump instruction at this location, which was originally coded as 'LJMP 04**00**,' becomes 'LJMP 04**2C**' because the MoBL-USB FX2LP18 automatically inserts **2C**, the INT2VEC value for EP2 (Table 4-13 on page 70).

The MoBL-USB FX2LP18 jumps to 0x042C, where it executes the jump instruction to the EP2 ISR, arbitrarily located for this example at address 0x0119.

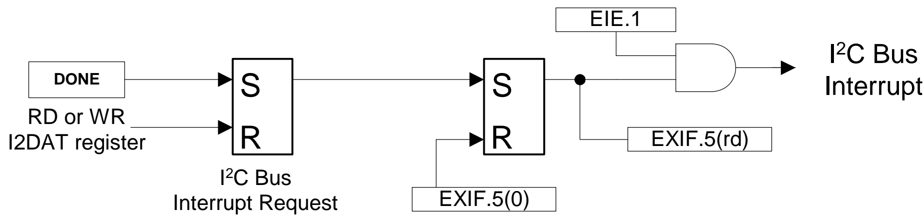
Once the MoBL-USB FX2LP18 vectors to 0x0043, initiation of the endpoint-specific ISR takes only eight instruction cycles.



Exhibit 2033 - Page 71 of 346

## 4.6 $I^2C$™ Bus Interrupt

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




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| I2CS 0xE678 | START | STOP | LASTRD | ID1 | ID0 | BERR | ACK | DONE |
| I2DAT 0xE679 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |

The Input/Output chapter on page 203 describes the interface to the MoBL-USB FX2LP18's I2C Bus controller. The MoBL-USB FX2LP18 uses two registers, I2CS (Control and Status) and I2DAT (Data), to transfer data over the bus.

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

- The DONE bit (I2CS.0) makes a zero-to-one transition, signaling 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.

**Note** Firmware must make sure the STOP bit is zero before writing to I2CS or I2DAT.



**Exhibit 2033 - Page 72 of 346**

## 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<br>(See General Programmable Interface, on page 135) |
| 14 | B4 | GPIFWF | GPIF Waveform<br>(See General Programmable Interface, on page 135) |

When FIFO/GPIF interrupt sources are asserted, the MoBL-USB FX2LP18 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 MoBL-USB FX2LP18’s program memory. This causes the MoBL-USB FX2LP18 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.

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 individual FIFO/GPIF interrupt will immediately try to generate another main INT4 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 the Registers chapter on page 237 and in section 8.6 CPU Control of MoBL-USB FX2LP18 Endpoints on page 96. Each interrupt source has an enable (IE) and a request (IRQ) bit. Firmware sets the IE bit to ‘1’ to enable the interrupt. The MoBL-USB FX2LP18 sets an IRQ bit to ‘1’ to request an interrupt, and the firmware clears an IRQ bit by setting it to ‘1’.

**Note** 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 MoBL-USB FX2LP18 provides a second level of interrupt vectoring, called *Autovectoring*. When a FIFO/GPIF interrupt is asserted, the MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 jump instruction is encoded as follows:

Table 4-15. MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 substitutes its INT4VEC byte (see Table 4-14 on page 73) 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 |



Exhibit 2033 - Page 74 of 346

### 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

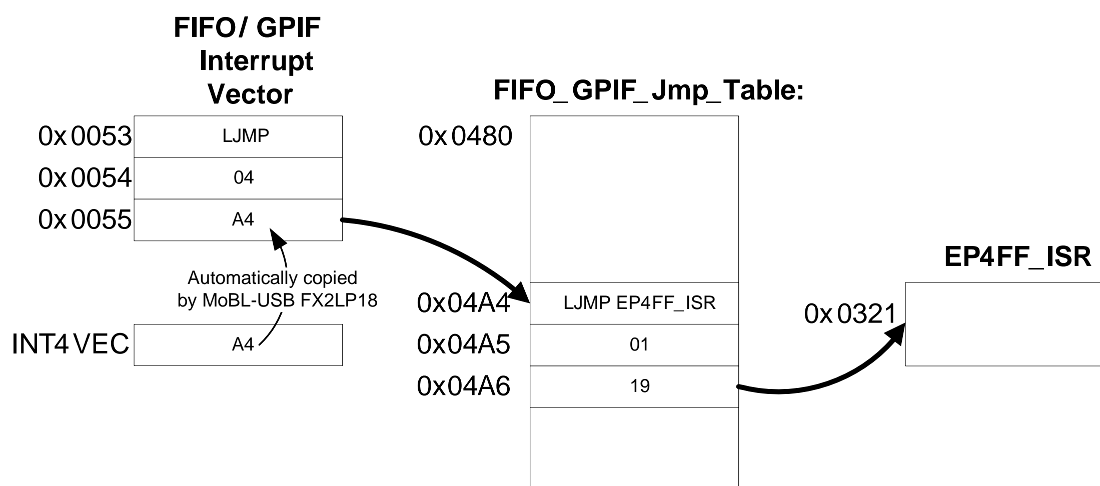



Figure 4-7 illustrates an ISR that services EP4's Full Flag. When EP4 goes full, the MoBL-USB FX2LP18 asserts the FIFO/GPIF interrupt request, vectoring to location 0x0053.

The jump instruction at this location, which was originally coded as 'LJMP 04**80**,' becomes 'LJMP 04**A4**' because the MoBL-USB FX2LP18 automatically inserts **A4**, the INT4VEC value for EP4FF (Table 4-13 on page 70).

The MoBL-USB FX2LP18 jumps to 0x04A4, where it executes the jump instruction to the EP4FF ISR, arbitrarily located for this example at address 0x0321.

Once the MoBL-USB FX2LP18 vectors to 0x0053, initiation of the endpoint-specific ISR takes only eight instruction cycles.



Exhibit 2033 - Page 75 of 346

Exhibit 2033 - Page 76 of 346

# 5. Memory

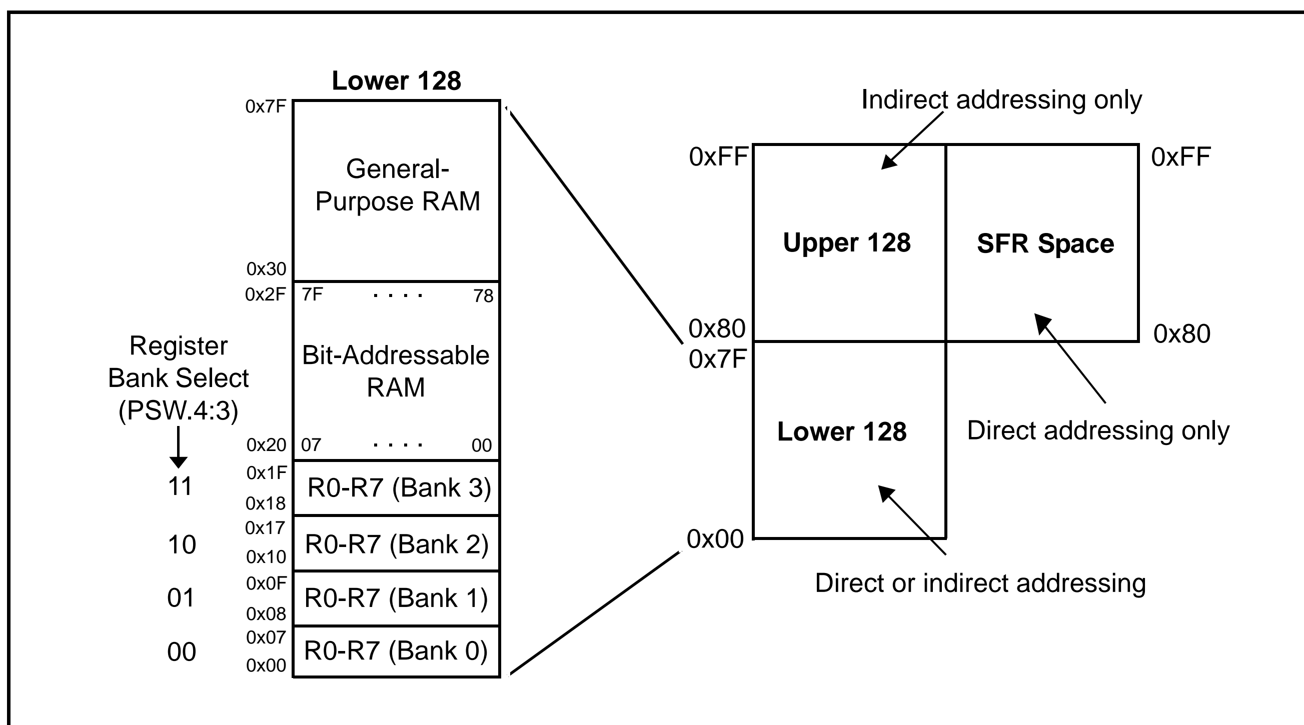
## 5.1 Introduction

Memory organization in the MoBL-USB FX2LP18 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 is explained below, *'External' memory is **not** necessarily external to the MoBL-USB FX2LP18 chip.*

## 5.2 Internal Data RAM

As shown in Figure 5-1, the MoBL-USB FX2LP18'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 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 addressing modes. See the Instruction Set chapter on page 197).

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 accesses 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 addressing modes. See the Instruction Set chapter on page 197).

Since the MoBL-USB FX2LP18'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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 control and status registers; an overview is in section 11.12 Special Function Registers on page 195, and a full description of all the SFRs is in the Registers chapter on page 237.

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 (for example, 0x80.0, 0xC8.7, and so on).

## 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 MoBL-USB FX2LP18 uses a modified version of this memory model; the 'on chip' program and data memories are unified in a Von Neumann architecture. This allows the MoBL-USB FX2LP18's on chip RAM to be loaded from an external source (USB or EEPROM, see Enumeration and ReNumeration™, on page 51), then used as program memory.

#### Standard 8051

The standard 8051 has separate address spaces for program and data memory; it can address 64 kB of read only program memory at addresses 0x0000-0xFFFF, and another 64 kB 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 64 kB ROM chip (enabled by the 8051's PSEN# signal) might be used for program memory and an external 64 kB 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.

#### MoBL-USB FX2LP18

The MoBL-USB FX2LP18 has 16 kB of on chip RAM (the 'Main RAM') at addresses 0x0000 – 0x3FFF, and 512 bytes of on chip RAM (the 'Scratch RAM') at addresses 0xE000 – 0xE1FF. Although this RAM is physically located inside the chip, it's addressed by firmware as 'External' memory, just as though it were in an external RAM chip.



Exhibit 2033 - Page 78 of 346

The RD# and PSEN# strobes are automatically combined for accesses to addresses below 0x4000, 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.

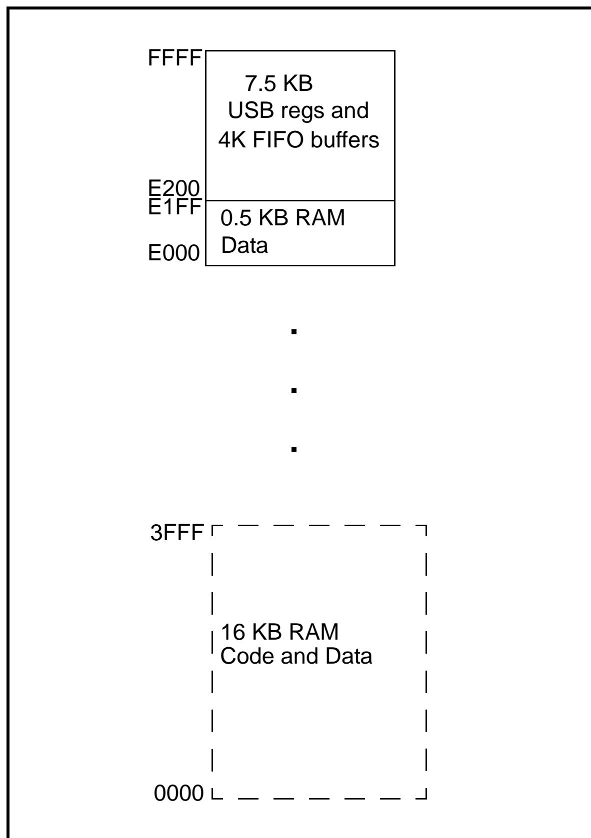
The Scratch RAM is never accessible as program memory.

The MoBL-USB FX2LP18 also reserves 7.5 kB (0xE200 – 0xFFFF) of the data memory address space for control/status registers and endpoint buffers (see section "On Chip Data Memory at 0xE000 – 0xFFFF" on page 81).

The MoBL-USB FX2LP18 chip has 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 MoBL-USB FX2LP18 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.

## 5.4 MoBL-USB FX2LP18 Memory Map

Figure 5-2. MoBL-USB FX2LP18 External Program/Data Memory Map



The on chip MoBL-USB FX2LP18 memory consists of three RAM regions:

- 0x0000 – 0x3FFF (Main RAM)
- 0xE000 – 0xE1FF (Scratch RAM)
- 0xE200 – 0xFFFF (Registers/Buffers)

The 16 kB 'Main RAM' occupies program memory (PSEN#) and data memory (RD#/WR#) addresses 0x0000 – 0x3FFF.

The 512 byte 'Scratch RAM' occupies data memory (RD#/WR#) addresses 0xE000 – 0xE1FF.

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

**Note** The asterisks in Figure 5-2 on page 79 indicate memory regions that may be accessed using three special MoBL-USB FX2LP18 resources:

- Setup Data Pointer (see section 8.7 The Setup Data Pointer on page 104)
- Upload or download via USB (see section 3.8 MoBL-USB FX2LP18 Vendor Request for Firmware Load on page 56)
- Code boot from an I2C EEPROM (see section 13.5 EEPROM Boot Loader on page 216 and section 3.4 'C2' EEPROM Boot-load Data Format on page 53)

**Note** MoBL-USB FX2LP18 code execution begins at address 0x0000, where the reset vector is located.

## 5.5 On Chip Data Memory at 0xE000 – 0xFFFF

Figure 5-3. On Chip Data Memory at 0xE000 – 0xFFFF

| Address | Contents |
|---|---|
| FFFF – F000 | EP2-EP8 (4 KB) Buffers |
| EFFF – E800 | Reserved (2KB) |
| E7FF – E7C0 | EP1IN (64) |
| E7BF – E780 | EP1OUT (64) |
| E77F – E740 | EP0 IN/OUT (64) |
| E73F – E700 | Reserved (64) |
| E6FF – E500 | MoBL-USB Control and Status Registers (512) |
| E4FF – E480 | Reserved (128) |
| E47F – E400 | GPIF Waveforms (128) |
| E3FF – E200 | Reserved (512) |
| E1FF – E000 | Scratch RAM (512) |

Figure 5-3 shows the memory map for on chip data RAM at 0xE000 – 0xFFFF.

512 bytes of Scratch RAM are available at 0xE000 – 0xE1FF. This is data RAM only; code cannot be executed from it. The 128 bytes at 0xE400 – 0xE47F hold the four waveform descriptors for the GPIF, described in the General Programmable Interface chapter on page 135. The area from 0xE500 – 0xE6FF contains control and status registers.

Memory blocks 0xE200 – 0xE3FF, 0xE480 – 0xE4FF, 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 FIFOs (via the Autopointer, described in section 8.8 Autopointers on page 105).



Exhibit 2033 - Page 81 of 346

Exhibit 2033 - Page 82 of 346

# 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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18's oscillator. Once suspended, the MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 also provides a register called SUSPEND; writing any value to it will allow the MoBL-USB FX2LP18 to enter the suspended state even when a SUSPEND condition does not exist on the bus.
- **RESUME** is a signal initiated by the device or host driving a 'K' state on the USB bus, requesting that the host or device be taken out of its low-power 'suspended' mode. A USB device can only signal a RESUME if it has reported (via its Configuration Descriptor) that it is 'remote wakeup capable', and only if the host has enabled remote wakeup from that device.
- **Idle** is a MoBL-USB FX2LP18 low-power state. Firmware initiates this mode by setting bit zero of the PCON (Power Control) register. To meet the stringent USB suspend current specification, the MoBL-USB FX2LP18'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) all three WAKEUP sources are either disabled or false (WAKEUP, WU2, USB Resume). The MoBL-USB FX2LP18 exits the **Idle** state when it receives a Wakeup Interrupt.
- **Wakeup** is the mechanism which restarts the oscillator and asserts an interrupt to force the MoBL-USB FX2LP18 to exit the Idle state and resume code execution. The MoBL-USB FX2LP18 recognizes three wakeup sources: one from the USB itself (when bus activity resumes) and two from device pins (WAKEUP and WU2).

The MoBL-USB FX2LP18 enters and exits its **Idle** state independent of USB activity; in other words, the chip 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. A suspend interrupt is automatically generated when the USB goes inactive for 3 milliseconds; firmware may respond to that interrupt by entering the **Idle** state to reduce power. If the MoBL-USB FX2LP18 is in the **Idle** state, a Wakeup Interrupt is generated when one of the three Wakeup sources is asserted; the MoBL-USB FX2LP18 responds to that interrupt by exiting the **Idle** state and resuming code execution.

Once the MoBL-USB FX2LP18 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 must wait 5 milliseconds after the wakeup interrupt, set the SIGRSUME bit, wait 10-15 milliseconds, then clear it.



Exhibit 2033 - Page 83 of 346

Figure 6-1 illustrates the MoBL-USB FX2LP18 logic that implements USB suspend and resume. These operations are explained in the next sections.

Figure 6-1. Suspend-Resume Control

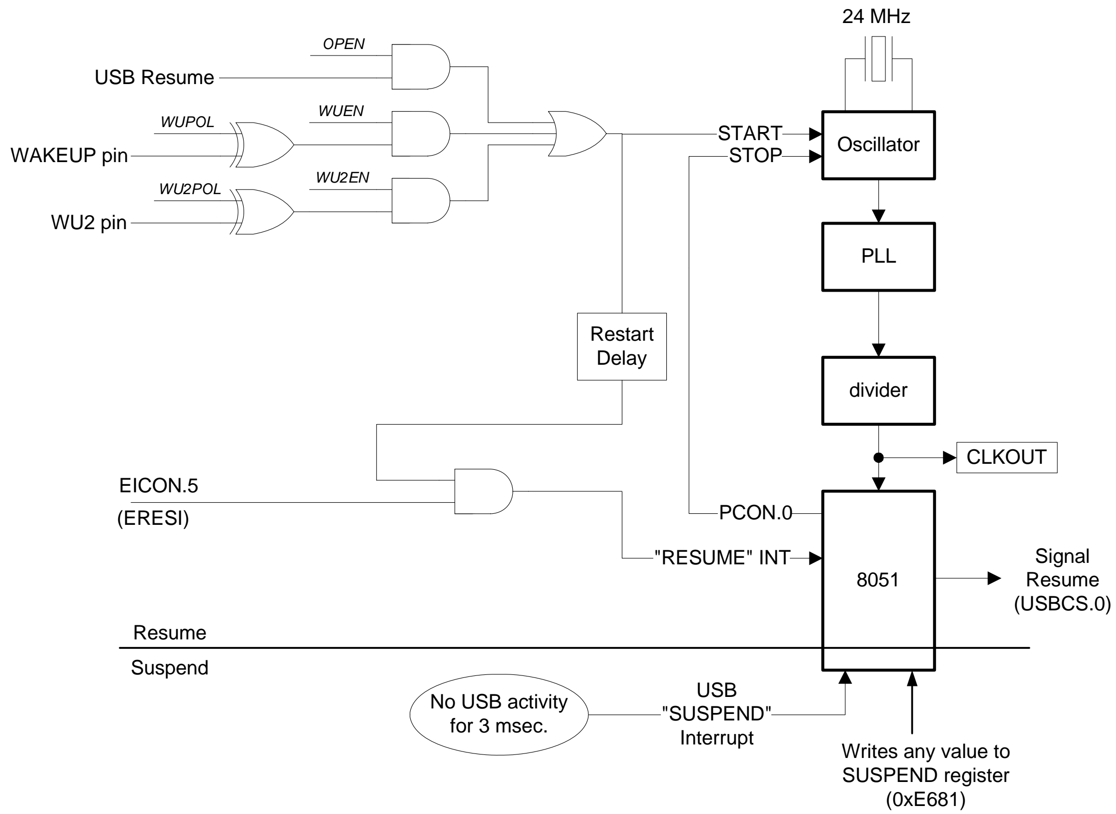
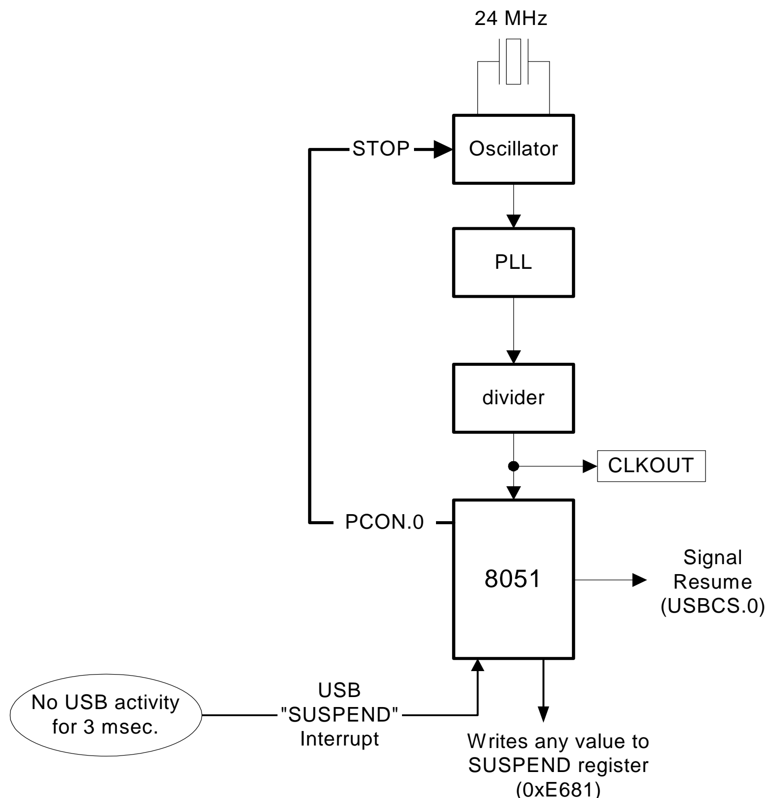





Exhibit 2033 - Page 84 of 346

## 6.2 USB Suspend

Figure 6-2. USB Suspend sequence




A USB device recognizes a SUSPEND request as three milliseconds of the bus-idle state. When the MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 automatically turns its oscillators (and keeps the CPU in reset) until an enabled wakeup source is asserted.

**Note** The bus-idle state is **not** equivalent to the disconnected-from-USB state; for full-speed, bus-idle is a 'J' state which means that the voltage on D+ is higher than that on D-.

MoBL-USB FX2LP18 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 zero of the PCON register.

These actions put the MoBL-USB FX2LP18 into a low power 'suspend' state, as required by the USB Specification.

### 6.2.1 SUSPEND Register

MoBL-USB FX2LP18 firmware can force the chip into its low-power mode at any time, even without detecting a 3-millisecond period of inactivity on the USB bus. This 'unconditional suspend' functionality is useful in applications which require the MoBL-USB FX2LP18 to enter its low-power mode even while disconnected from the USB bus.

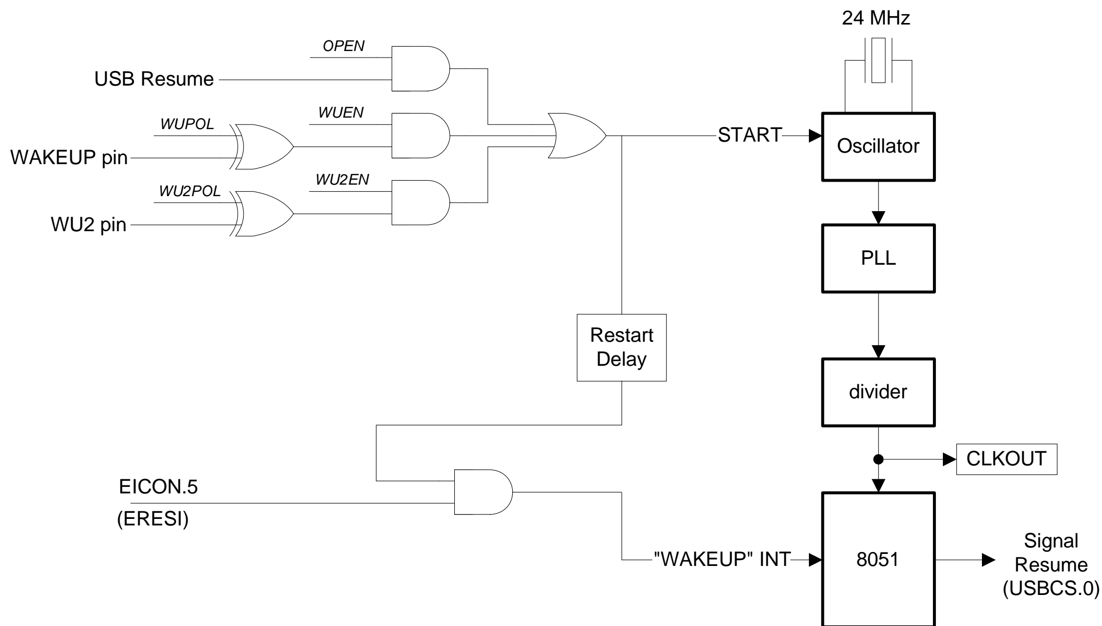
To force the MoBL-USB FX2LP18 unconditionally to enter its low-power mode, firmware simply writes any value to the SUSPEND register (at 0xE681) before setting the PCON.0 bit.



Exhibit 2033 - Page 85 of 346

## 6.3 Wakeup/Resume

Figure 6-3. MoBL-USB FX2LP18 Wakeup/Resume sequence




Once in the low-power mode, there are three ways to wake up the MoBL-USB FX2LP18:

- USB activity on the MoBL-USB FX2LP18'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 MoBL-USB FX2LP18 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.



**Exhibit 2033 - Page 86 of 346**

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. Firmware clears the WU and WU2 flags by writing '1' to them.

**Note** Holding either WAKEUP pin in its active state (as determined by the programmed polarity) inhibits the MoBL-USB FX2LP18 chip from turning off its oscillator in order to enter the 'suspend' state.

**Note** While disconnected from the USB bus, the DPLUS and DMINUS lines may float. Noise on these lines may indicate activity to the MoBL-USB FX2LP18 and initiate a wakeup event. Firmware must set DPEN to '0' if this is not desired.

Some designs also use the WAKEUP# pin as a general purpose input pin. Due to the built-in latch on this pin, it must be cleared before it will show the current state of the pin. For example, to detect a '1' on the WAKEUP# pin use the following code:

```
WAKEUPCS = bmWU | bmWUPOL | bmWUEN;  // Write one to bmWU to clear it, set
                                     // active high, enable

WAKEUPCS = bmWU | bmWUPOL | bmWUEN;  // This line is required only if WUPOL
                                     // is changing.

                                     // A WUPOL change can trigger a WAKEUP event

if (WAKEUPCS & bmWU)
  {
  // WAKEUP# is a one
  }
else
  {
  // WAKEUP# is 0
  }
```

**Note** If the polarity is changed, an additional WAKEUP event may be triggered. Always clear the WAKEUP event after changing the polarity.

### 6.3.1 Wakeup Interrupt

When a wakeup event occurs, the MoBL-USB FX2LP18 restarts its oscillator and, after the PLL stabilizes, it generates an interrupt request. This applies whether or not the MoBL-USB FX2LP18 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 ERISI (EICON.5), and its IRQ flag is at EICON.4 (EICON is SFR 0xD8). **Note** If the MoBL-USB FX2LP18 is suspended with ERISI (EICON.5) low, it never 'wakes up'.

The Wakeup Interrupt Service Routine clears the interrupt request flag RESI (using the 'bit clear' instruction, for example, 'clr EICON.4'), and then executes a 'reti' (return from interrupt) instruction. This causes the MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 enters its idle state when it sets PCON.0 to '1'. Although a standard 8051 exits the idle state when *any* interrupt occurs, the MoBL-USB FX2LP18 supports only the Wakeup Interrupt to exit the idle state.

**Note** If PCON.0 is set when no Suspend condition exists (for example, the USB is not signaling 'Suspend', and firmware has not written to the SUSPEND register), the Wakeup Interrupt will fire immediately.



Exhibit 2033 - Page 87 of 346

## 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 |

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

**Note** Before setting the SIGRSUME bit to ‘1’, MoBL-USB FX2LP18 firmware must check that the source of the wakeup event was one of the WAKEUP pins. If neither WAKEUP pin was the source, the wakeup event was the resumption of USB DPLUS activity, and in this case, the device must not signal a remote-wakeup by setting the SIGRSUME bit to ‘1’.

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 IO pin PA3. Unlike other multi-purpose IO pins that use configuration registers (PORTACFG, PORTCCFG, and PORTECFG) 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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 will unconditionally stop the oscillator when the firmware sets PCON.0.
7. Set PCON.0 to ‘1’. This commands the MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 chip can operate at extremely low average power, since the on/off (active/suspend) duty-cycle is very short.



Exhibit 2033 - Page 88 of 346

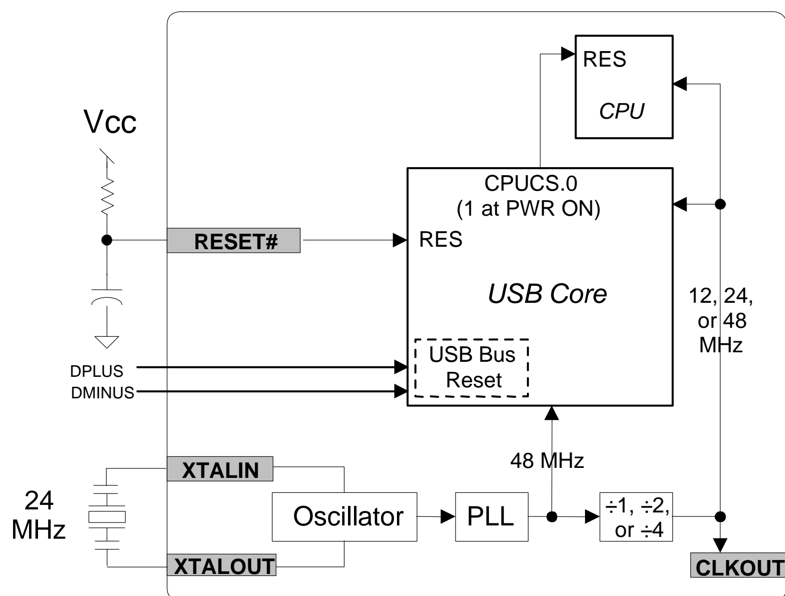
# 7. Resets

## 7.1 Introduction

There are three different reset functions on the MoBL-USB FX2LP18. This chapter describes their effects.

- *Hard Reset.* An active low reset pin (RESET#) is provided in order to reset the MoBL-USB FX2LP18 to a known state at power-on or any other application-specific reset event.
- *CPU Reset*, controlled by the MoBL-USB FX2LP18's USB Core logic. The CPU Reset is always asserted (for example, the CPU is always held in reset) while the MoBL-USB FX2LP18's RESET# pin is asserted.
- *USB Bus Reset,* which is a condition on the USB bus initiated by the USB host in order to put every device's USB functions in a known state.

Figure 7-1. MoBL-USB FX2LP18 Resets




## 7.2 Hard Reset

The RESET# pin can be connected to an external R-C network or other external reset source in order to ensure that, when power is first applied, the MoBL-USB FX2LP18 is held in reset until the operating parameters (VCC voltage, crystal frequency, PLL frequency, and so on.) stabilize. The 24 MHz oscillator and PLL stabilize 5 ms after VCC reaches 3.0 V. An R-C network can satisfy the power-on reset requirements of the MoBL-USB FX2LP18. See Figure 7-1 for a sample connection scheme (for example, R = 27K ohm, C = 1 µF).

The RESET# pin can also be asserted at any time after the MoBL-USB FX2LP18 is running. If the MoBL-USB FX2LP18's XTALIN pin is driven by an external clock source that continues to run while the chip is in reset, RESET# need only be asserted for 200 us. Otherwise, it must be asserted for at least 5 ms.



Exhibit 2033 - Page 89 of 346

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 MoBL-USB FX2LP18 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.

Power-on default values for all MoBL-USB FX2LP18 register bits are shown in the Registers chapter on page 237. 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 MoBL-USB FX2LP18 will NAK IN and OUT tokens while the CPU is reset.
- Endpoint data 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.
- D+ pull up resistor is disconnected from the data line during a hard reset.

## 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).

**Note** MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 RAM using the 'Firmware Load' vendor request (Endpoint Zero chapter on page 33). The last packet loaded writes 0x00 to the CPUCS register, which releases the CPU from reset. Note that only CPUCS.0 can be written in this way.

### 7.3.2 EEPROM Load

The Enumeration and ReNumeration™ chapter on page 51 describes the EEPROM boot loads in detail. At power-on, the MoBL-USB FX2LP18 checks for the presence of an EEPROM on its I2C bus. If found, it reads the first EEPROM byte. If it reads 0xC2 as the first byte, the MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 sets the RENUM bit to '1', so the firmware will be responsible for responding to USB device requests.



**Exhibit 2033 - Page 90 of 346**

## 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 MoBL-USB FX2LP18 RAM. For such applications, it is important to know the state of the MoBL-USB FX2LP18 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 resets described in section 7.2 Hard Reset on page 89 This discussion applies only to the condition in which the MoBL-USB FX2LP18 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 IO ports retain their configurations and values. Stalled endpoints remain stalled, data toggles do not 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 MoBL-USB FX2LP18 enters 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 MoBL-USB FX2LP18 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:

- Data 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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 Disconnect

Although not strictly a 'reset,' the disconnect-reconnect sequence used for ReNumeration™ affects the MoBL-USB FX2LP18 in ways similar to the other resets. When the MoBL-USB FX2LP18 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.



Exhibit 2033 - Page 91 of 346

## 7.7 Reset Summary

Table 7-1. Effects of Various Resets on MoBL-USB FX2LP18 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 | — | — | — |



Exhibit 2033 - Page 92 of 346

# 8. Access to Endpoint Buffers

## 8.1 Introduction

USB data enters and exits the MoBL-USB FX2LP18 via endpoint buffers. *External logic usually reads and writes this data by direct connection to the endpoint FIFOs without any participation by the MoBL-USB FX2LP18's CPU*. This is especially necessary for the MoBL-USB FX2LP18, which can operate at the high-speed 480 Mbps transfer rate. However, this feature is also available when attached to a full-speed host.

**Note** The chapters Slave FIFOs, on page 107 and General Programmable Interface, on page 135 give details about how external logic directly connects to the large endpoint FIFOs. Direct connection is available only on endpoints 2, 4, 6, and 8.

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, IO pins, and others.

## 8.2 MoBL-USB FX2LP18 Large and Small Endpoints

MoBL-USB FX2LP18 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 cannot 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 Slave FIFOs, on page 107 and General Programmable Interface, on page 135, the CPU can access the large endpoints if necessary.

## 8.3 High-Speed and Full-Speed Differences

The data payload size and transfer speed requirements differ between full-speed (12 Mbps) and high-speed (480 Mbps). The MoBL-USB FX2LP18 architecture is optimized for high-speed transfers, but does not limit full-speed transfers:

- Instead of many small endpoint buffers, MoBL-USB FX2LP18 provides a reduced number of large buffers.
- MoBL-USB FX2LP18 provides double, triple or quad buffering on its large endpoints (EP2, 4, 6, and 8).
- *The CPU need not participate in data transfers.* Instead, dedicated logic and unified endpoint/interface FIFOs move data on and off the chip without any CPU intervention.

In the MoBL-USB FX2LP18, endpoint buffers appear to have different sizes depending on whether it is operating at full- or high-speed. This is due to the difference in maximum data payload sizes allowed by the USB specification for the two modes, as illustrated by Table 8-1 on page 94.



Exhibit 2033 - Page 93 of 346

Table 8-1. Maximum Data Payload Sizes for Full-speed and High-speed

| Transfer Type | Max Data Payload Size | |
|---|---|---|
| | Full-speed | High-speed |
| 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 for the non-isochronous types when the MoBL-USB FX2LP18 is operating at full-speed. This is to account for the smaller maximum data payload sizes.

When operating at high-speed, firmware can configure the large endpoints' size, type, and buffering depth; when operating at full-speed, type and buffering are configurable but the buffer 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 bytes, double-buffered in high-speed mode, 64 bytes double-buffered in full-speed mode for non-iso) |
| 0xE614 | **EP6CFG** | valid, direction, type, size, buffering |
| 0xE615 | **EP8CFG** | valid, direction, type (always 512 bytes double-buffered in high-speed mode, 64 bytes double-buffered in full-speed mode for non-iso) |
| **Note 1**: For EP1, 'type' may be set to Interrupt or Bulk only. Even though these buffers are 64 bytes in size, they are reported as 512 for USB 2.0 compliance. The user must never transfer packets larger than 64 bytes to EP1. | | |

**Note** The Registers chapter on page 237 gives full bit-level details for all endpoint configuration 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 EP2 and EP6 are the most flexible endpoints, as they are configurable for size (512 or 1024 bytes in high-speed mode, 64 bytes in full-speed mode for the non-isochronous types) and depth of buffering (double, triple, or quad). Endpoints EP4 and EP8 are fixed at 512 bytes, double-buffered in high-speed mode. They are fixed at 64 bytes, double-buffered in full-speed mode for the non-isochronous types.

The bits in the EPxCFG 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.



Exhibit 2033 - Page 94 of 346

- **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.

**Note** The Valid bit is ignored when buffer space is allocated by the EZ-USB (for example, BUF[1:0] takes precedence over the Valid bit).

When you are not using all of the endpoints in the endpoint configuration, disable the unused endpoints by writing a zero into the "valid" bit of the corresponding EPxCFG register without disturbing the default state of the other bits in the register.

For example, if the endpoint configuration 11 (see 1.18 MoBL-USB FX2LP18 Endpoint Buffers on page 28), which utilizes only endpoints 2 and 8, must be used, configure the endpoints as follows.

```
EP2CFG = 0xDB;
SYNCDELAY;
EP8CFG = 0x92;
SYNCDELAY;
EP4CFG &= 0x7F;
SYNCDELAY;
EP6CFG &=0x7F;
SYNCDELAY;
```

## 8.5 CPU Access to MoBL-USB FX2LP18 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 Table 8-3, 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 (that is, from 0xF000 to 0xF1FF). Similarly, if the MoBL-USB FX2LP18 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 (for example, 0xF000-0xF03F for EP2) will be used for Bulk data.

Table 8-3. Endpoint Buffers in RAM Space

| Name | Address | Size (bytes) |
|---|---|---|
| **EPOBUF** | 0xE740-0xE77F | 64 |
| **EP1OUTBUF** | 0xE780-0xE7BF | 64 |
| **EP1INBUF** | 0xE7C0-0xE7FF | 64 |
| **EP2FIFOBUF** | 0xF000-0xF3FF | 1024 |
| **EP4FIFOBUF** | 0xF400-0xF7FF | 1024 |
| **EP6FIFOBUF** | 0xF800-0xFBFF | 1024 |
| **EP8FIFOBUF** | 0xFC00-0xFFFF | 1024 |

**Note** EP0BUF is for the (optional) data stage of a CONTROL transfer. The eight bytes of data from the CONTROL packet appear in a separate MoBL-USB FX2LP18 RAM buffer called SETUPDAT, at 0xE6B8-0xE6BF.



Exhibit 2033 - Page 95 of 346

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 do not exist (the RAM space which they would normally occupy is used to implement the EP2 triple-buffering).

## 8.6 CPU Control of MoBL-USB FX2LP18 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<br>0xE68B | **EP0BCH**<br>**EP0BCL** | EP0 Byte Count (MSB)<br>EP0 Byte Count (LSB) |
| 0xE65C<br>0xE65D | **USBIE**<br>**USBIRQ** | EP0 Interrupt Enables<br>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 MoBL-USB FX2LP18 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.

**Note** 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.



Exhibit 2033 - Page 96 of 346

For EP0 IN transfers, MoBL-USB FX2LP18 logic NAKs all IN tokens to EP0 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 must 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 EP0 OUT transfers, MoBL-USB FX2LP18 logic NAKs all OUT tokens to EP0 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 MoBL-USB FX2LP18 also generates an EP0OUT interrupt request.

**Note** The MoBL-USB FX2LP18's autovectored interrupt system automatically transfers control to the appropriate ISR (Interrupt Service Routine) for the endpoint requiring service. The Interrupts chapter on page 59 describes this mechanism.

#### STALL

Set STALL=1 to instruct the MoBL-USB FX2LP18 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 word-aligned 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 MoBL-USB FX2LP18 automatically transfers the entire block, partitioning the data into MaxPacketSize packets as necessary.

**Note** The Setup Data Pointer is the subject of section 8.7 The Setup Data Pointer on page 104.

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.

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

### *8.6.1.3 USBIE and 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 MoBL-USB FX2LP18 receives the SETUP token.
- **SUDAV** (Setup Data Available) asserts when MoBL-USB FX2LP18 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 eight setup bytes are available for examination in the SETUPDAT buffer.



**Exhibit 2033 - Page 97 of 346**

#### 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 MoBL-USB FX2LP18 verifies the OUT data for accuracy and ACKs the transfer. If a transmission error occurs, the MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 NAKs all EP1OUT transfers until the firmware arms EP1OUT by writing any value to EP1OUTBC.

**STALL**

Firmware sets STALL=1 to instruct the MoBL-USB FX2LP18 to return the STALL PID (instead of ACK or NAK) in response to an EP1OUT transfer. The MoBL-USB FX2LP18 continues 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 MoBL-USB FX2LP18 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.



Exhibit 2033 - Page 98 of 346

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 MoBL-USB FX2LP18 NAKs 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 MoBL-USB FX2LP18 to return the STALL PID (instead of ACK or NAK) in response to an EP1IN transfer. The MoBL-USB FX2LP18 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 rates required by USB 2.0's high-speed mode, and to maximize full-speed transfer rates, the MoBL-USB FX2LP18's CPU should not participate in transfers to and from the 'large' endpoints.** Instead, those endpoints are usually connected directly to external logic (see chapters Slave FIFOs, on page 107 and General Programmable Interface, on page 135 for details). Although especially suited for high-speed (480 Mbps) transfers, the functionality of these endpoints is identical at full-speed, except for packet size.

Some applications, however, may require the firmware to have at least some small amount of control over the large endpoints. For those applications, the MoBL-USB FX2LP18 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 |
| 0xE649 | **OUTPKTEND** | skip or commit an OUT 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 Section 8.6.2.3) are replicated here in order to allow faster access by the firmware.



Exhibit 2033 - Page 99 of 346

#### *8.6.2.2 EP2ISOINPKTS, EP4ISOINPKTS, EP6ISOINPKTS, EP8ISOINPKTS*

These registers only apply to ISOCHRONOUS IN endpoints. Refer to the EPxISOINPKTS register descriptions in the *Registers chapter on page 237* for details.

MoBL-USB FX2LP18 has the capability of sending a zero-length packet (ZLP) when the host issues an IN token to an isochronous IN endpoint and the SIE does not have any data available.

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

#### *8.6.2.3 EP2CS, EP4CS, EP6CS, EP8CS*

Because the four large 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.

**Note** Multiple-buffered endpoint data must be read or written **only** at the buffer addresses given in Table 8-3 on page 95. The MoBL-USB FX2LP18 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 (that is, 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 (that is, 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 MoBL-USB FX2LP18 to return the STALL PID (instead of ACK or NAK) in response to an IN or OUT transfer. The MoBL-USB FX2LP18 continues to respond to IN or OUT transfers with the STALL PID until the firmware clears this bit.



**Exhibit 2033 - Page 100 of 346**

### *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 the Slave FIFOs chapter on page 107

When outside logic is connected to the interface FIFOs, the normal data flow is for the MoBL-USB FX2LP18 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 the high-speed mode.

In some cases, it may be desirable to insert a 'hook' into this data flow, so that -- rather than the MoBL-USB FX2LP18 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 either to commit the packet to the outside interface (the output FIFO), or to 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.

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



**Exhibit 2033 - Page 101 of 346**

## 8.6.3 Registers That Control All Endpoints

Table 8-6. Registers That Control All Endpoints

| Address | Name | Description |
|---|---|---|
| 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** | Endpoint data toggles |

### *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 a 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 MoBL-USB FX2LP18 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 are not 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.

**Note** The MoBL-USB FX2LP18 interrupt system is described in detail in the Interrupts chapter on page 59

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, and so on), 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).

**Note** 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 Mbps) BULK OUT transfers.



**Exhibit 2033 - Page 102 of 346**

When operating at full-speed, 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, 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 MoBL-USB FX2LP18 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 a BULK OUT endpoint returned a NAK in response to a PING.

**Note** PING only applies at high-speed (480 Mbps).

Unlike the IBN bits, which are combined into a single IBN interrupt for all endpoints, each BULK OUT endpoint has a separate PING 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 MoBL-USB FX2LP18 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 endpoint BUSY bit making a 1-to-0 transition (busy to not-busy). As with all interrupts, this one is enabled by writing a '1' to its enable bit, and the interrupt flag is cleared 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 (for example, 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' (for example, 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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 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

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, and others) have occurred, the 'USB Error Limit' interrupt flag will be set. At power-on-reset, the error limit defaults to 4 (0100 binary).



**Exhibit 2033 - Page 103 of 346**

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 the Introducing MoBL-USB™ FX2LP18 chapter on page 15 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 (for example, after the host issues a *Set Configuration* request).
- After an interface's alternate setting changes (that is, 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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 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 (for example, 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, and the address of SUDPTRH:L must be word-aligned (for example, the LSB of SUDPTRL must be '0').



Exhibit 2033 - Page 104 of 346

Table 8-7 lists the registers which configure the Setup Data Pointer.

Table 8-7. 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 word-aligned 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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 then automatically loads the EP0 data buffer with the required number of packets and transfers them to the host.

To command the MoBL-USB FX2LP18 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 (for example, 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 (that is, 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 MoBL-USB FX2LP18 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 MoBL-USB FX2LP18 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 (16 KB at 0x0000-0x3FFF)
- On-chip Scratch RAM (512 bytes at 0xE000-0xE1FF)

## 8.8 Autopointers

Endpoint data is available to the CPU in RAM buffers (see Table 8-3 on page 95). In some cases, it is faster for the firmware to access endpoint data as though it were in a FIFO register. The MoBL-USB FX2LP18 provides two special data pointers, called 'Autopointers', that automatically increment after each byte transfer. Using the Autopointers, firmware can access contiguous blocks of on-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.



Exhibit 2033 - Page 105 of 346

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 8-8. Registers that Control the Autopointers

| Address | Register Name | Function |
|---|---|---|
| SFR 0xAF | **AUTOPTRSETUP** | Increment/freeze, memory 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.

The Autopointers must not be used to read or write registers in the 0xE600-0xE6FF range; Autopointer accesses within that range produce undefined results.



**Exhibit 2033 - Page 106 of 346**

# 9. Slave FIFOs

## 9.1 Introduction

Although some MoBL-USB FX2LP18-based devices may use the MoBL-USB FX2LP18's CPU to process USB data directly (See chapter "Access to Endpoint Buffers" on page 93), most will use the chip simply as a conduit between the USB and external data-processing logic (for example, 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 MoBL-USB FX2LP18's CPU* — through the 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, and others.

These FIFO signals must, of course, be controlled by a FIFO 'master'. The MoBL-USB FX2LP18's General Programmable Interface (GPIF) can act as an *internal* master when the MoBL-USB FX2LP18 is connected to external logic which does not include a standard FIFO interface (General Programmable Interface, on page 135 discusses the internal-master GPIF), or the FIFOs can be controlled by an external master. While its FIFOs are controlled by an external master, the MoBL-USB FX2LP18 is said to be in 'Slave FIFO' mode.

This chapter provides details on the interface — both hardware and software — between the MoBL-USB FX2LP18'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 MoBL-USB FX2LP18 System

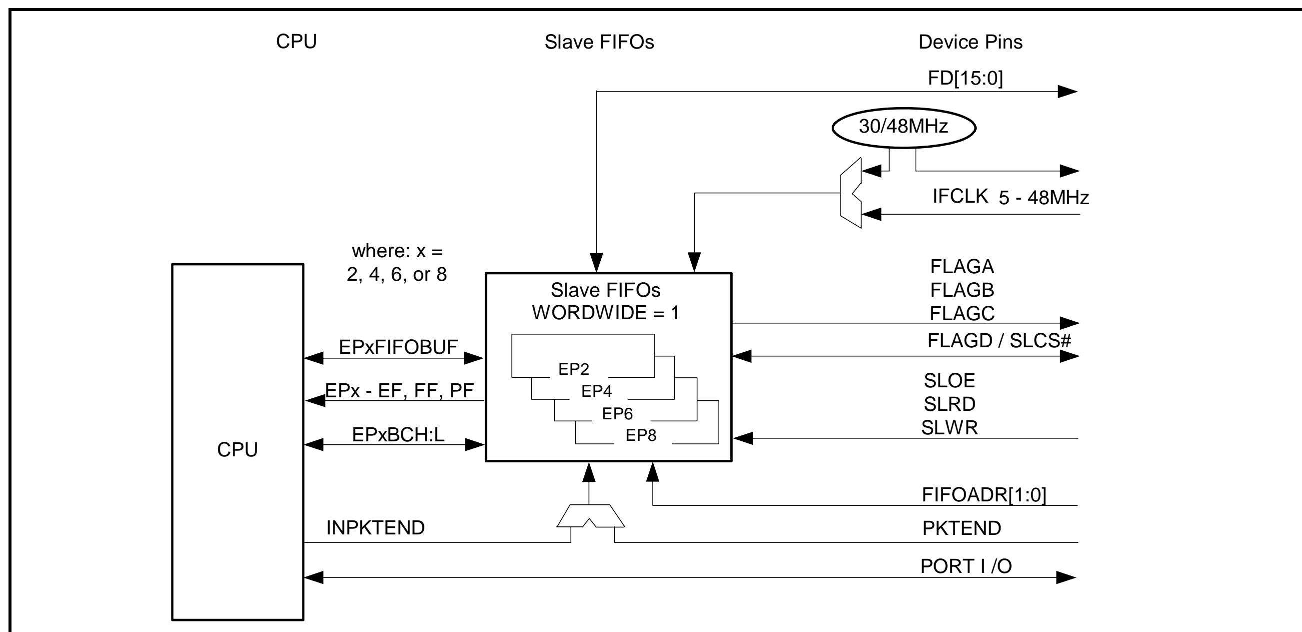


Table 9-1 lists the registers associated with the slave FIFO hardware. The registers are fully described in the Registers chapter on page 237

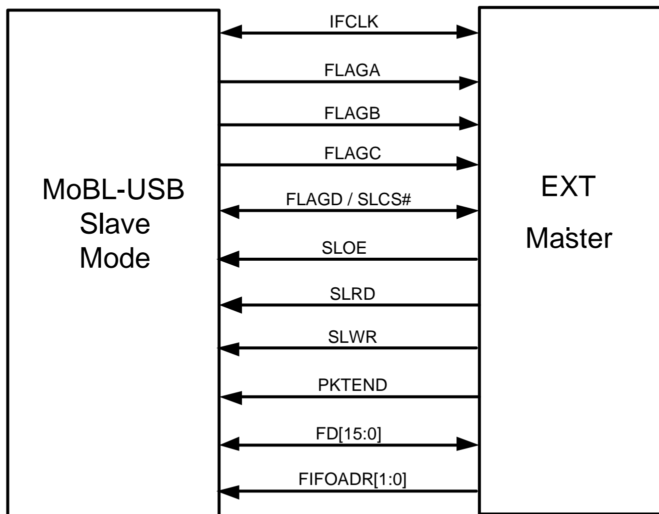
Table 9-1. Registers Associated with Slave FIFO Hardware

| | |
|---|---|
| IFCONFIG | EPxFIFOPFH/L |
| PINFLAGSAB | PORTACFG |
| PINFLAGSCD | INPKTEND |
| FIFORESET | EPxFIFOIE |
| FIFOPINPOLAR | EPxFIFOIRQ |
| EPxCFG | EPxFIFOBCH:L |
| EPxFIFOCFG | EPxFLAGS |
| EPxAUTOINLENH:L | EPxFIFOBUF |

### 9.2.1 Slave FIFO Pins

The MoBL-USB FX2LP18 comes out of reset with its IO pins configured in 'Ports' mode, not 'Slave FIFO' mode. To configure the pins for Slave FIFO mode, the IFCFG[1:0] bits in the IFCONFIG register must be set to '11' (see Table 13-10, "IFCFG Selection of Port IO Pin Functions," on page 211 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. MoBL-USB FX2LP18 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 and is being controlled by the external master.

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

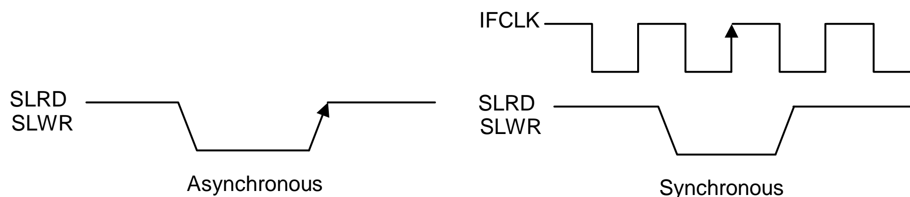





Exhibit 2033 - Page 108 of 346

## 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 on page 110. FD[7:0] is the LSB of the word, and FD[15:8] is the MSB of the word.

On a hard 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 IO. If **any** FIFO is configured for 16-bit mode, Port D is unavailable for use as general-purpose IO regardless of which FIFO is currently selected via the FIFOADR[1:0] pins.

**Note** In 16-bit mode, the MoBL-USB FX2LP18 only transfers even-sized packets of data across the FD bus. This should be considered when the MoBL-USB FX2LP18 interfaces to host software that sends or receives odd-sized packets.

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

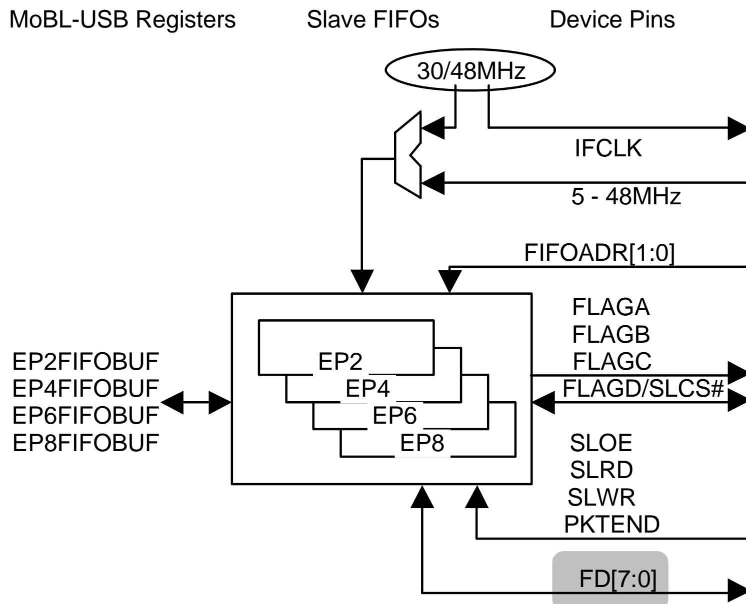
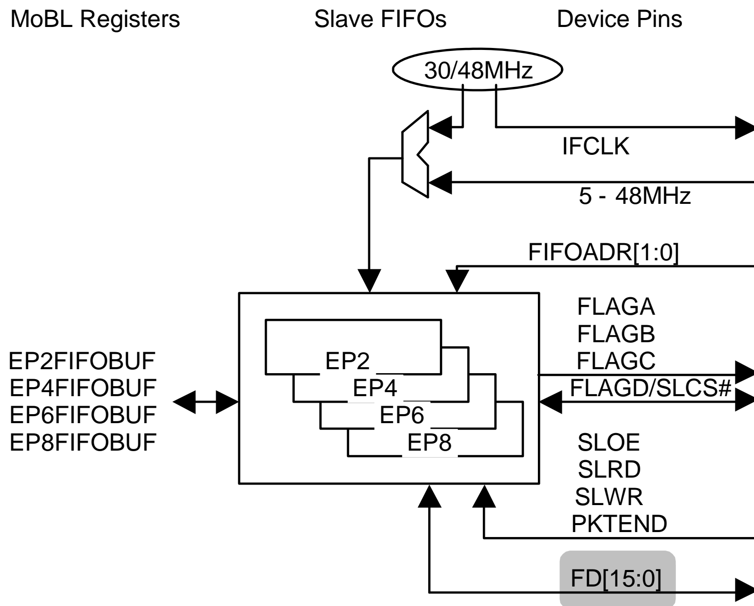





Exhibit 2033 - Page 109 of 346

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 MoBL-USB FX2LP18'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 MoBL-USB FX2LP18 is configured to use an external clock source, the IFCLK pin can be driven at any frequency between 5 MHz and 48 MHz. On a hard reset, it defaults to the internal source at 48 MHz, normal polarity, with the IFCLK output disabled. See Figure 9-6 on page 111.

IFCONFIG.7 selects between internal and external sources: 0 = external, 1 = internal. If an external IFCLK is chosen, it must be free-running at a minimum frequency of 5 MHz. In addition, in order to provide synchronization for the internal endpoint FIFO logic, the external IFCLK source must be present before the firmware sets IFCONFIG.7 = 0.

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 must not be set to '1' when IFCONFIG.7 = 0.

IFCONFIG.4 inverts the polarity of the interface clock (either internal or external): 0 = normal, 1 = inverted. IFCLK inversion can make it easier to interface the MoBL-USB FX2LP18 with certain external circuitry. When an internal IFCLK is used (IFCONFIG.7 = 1), IFCONFIG.4 only affects the IFCLK output polarity if IFCONFIG.5 = 1. Figure 9-7 on page 111 demonstrates the use of IFCLK output inversion in order to ensure a long enough setup time ($t_S$) for reading the FIFO flags.

When IFCLK is configured as an input, the minimum external frequency that can be applied to it is 5 MHz. This clock must be applied prior to initialization of the GPIF; only interruptions of it will lower the overall frequency, causing violations of the minimum frequency requirement.



Exhibit 2033 - Page 110 of 346

Figure 9-6. IFCLK Configuration

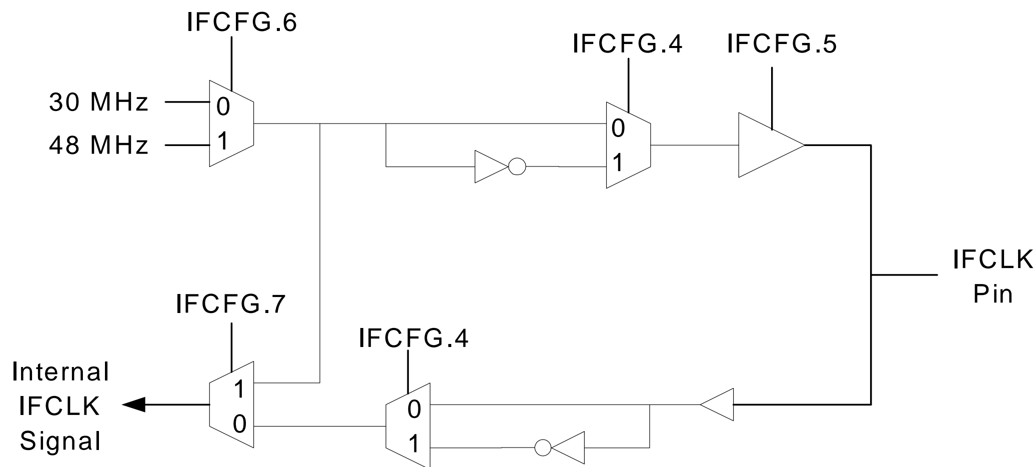
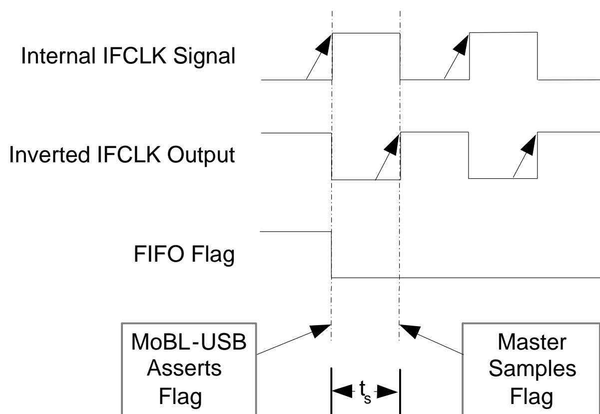



Figure 9-7. Satisfying Setup Timing by Inverting the IFCLK Output




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

Four pins — FLAGA, FLAGB, FLAGC, and FLAGD (see Figure 9-7) — report the status of the MoBL-USB FX2LP18'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 (EF) of OUT endpoints and the 'full' (FF) flag of IN endpoints; the 'programmable-level' flag (PF) 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 PIN-FLAGSAB and PINFLAGSCD registers. The FLAGD pin operates in Fixed mode only. FLAGA-FLAGC pins can be configured independently; some pins can be in Fixed mode while others are in Indexed mode. See the PINFLAGSAB and PINFLAGSCD register descriptions in the Registers chapter on page 237 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.

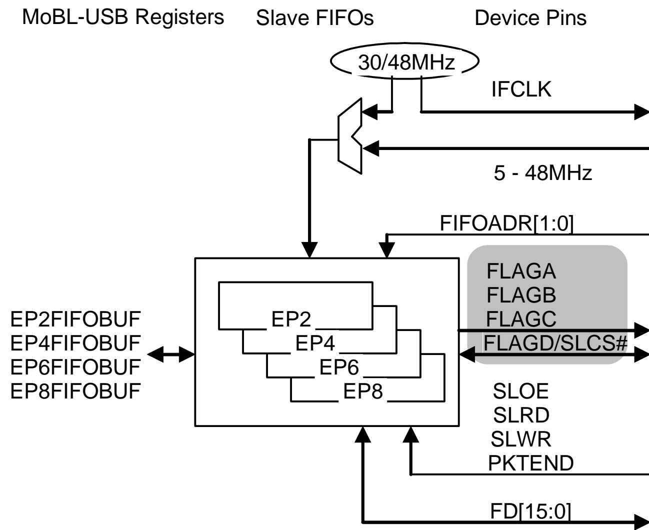
The polarity of the 'empty' and 'full' flag pins defaults to active-low but may be inverted via the FIFOPINPOLAR register.



Exhibit 2033 - Page 111 of 346

On a hard reset, the FIFO flags are configured for Indexed operation.

Figure 9-8. FLAGx Pins




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

The Slave FIFO 'control' pins are SLOE (Slave Output Enable), SLRD (Slave Read), SLWR (Slave 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 on page 113.

### Slave Output Enable and Slave 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. In synchronous mode, when SLOE is asserted, this causes the FD bus to be driven with the data that the FIFO pointer is currently pointing to. The data is pre-fetched and is output only when SLOE is asserted. In asynchronous mode, the data is not pre-fetched, and SLRD must be asserted when SLOE is asserted for the FD bus to be driven with the data that the FIFO pointer is currently pointing to. SLOE has no other function besides enabling the FD bus to be in a driven state.

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

### Slave 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.



Exhibit 2033 - Page 112 of 346

### 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 (for example, 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 MoBL-USB FX2LP18 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 pulse the PKTEND pin.

If the FIFO is configured to allow zero-length packets (EPxFIFOCFG.2 = 1), pulsing the PKTEND pin when a FIFO buffer is available commits 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.

**Note** In synchronous mode, there is no specific timing requirement for PKTEND assertion with respect to SLWR assertion. PKTEND can be asserted anytime. In asynchronous mode, SLWR and PKTEND should not be pulsed at the same time. PKTEND should be asserted after SLWR has been de-asserted for the minimum de-asserted pulse width. In both modes, FIFOADR[1:0] should be held constant during the PKTEND pin assertion.

Figure 9-9. Slave FIFO Control Pins

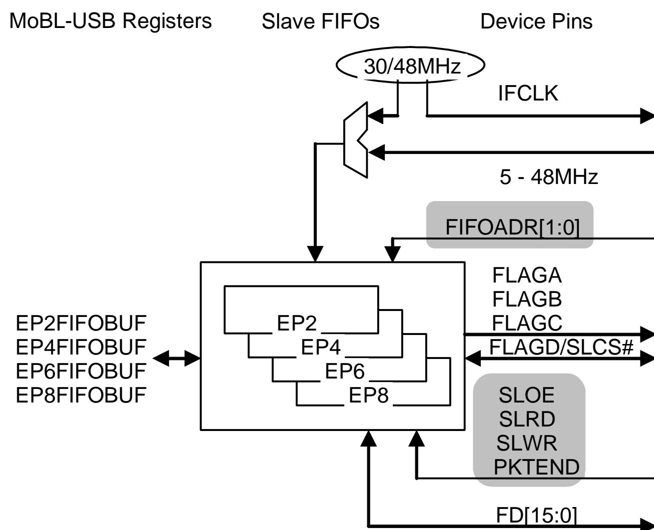





Exhibit 2033 - Page 113 of 346

### 9.2.6 Slave FIFO Chip Select

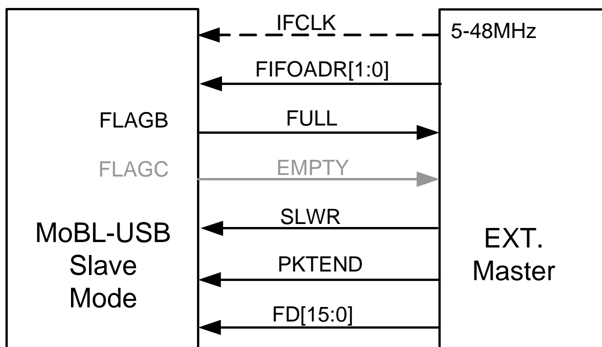
The Slave FIFO Chip Select (SLCS#) pin is an alternate function of pin PA7; it's enabled via the PORTACFG.6 bit (see section 13.3.1 Port A Alternate Functions on page 207).

The SLCS# pin allows external logic to effectively remove the MoBL-USB FX2LP18 from the FIFO Data bus, in order to, for example, share that bus among multiple slave devices. For applications that do not need to share the FD bus among multiple slave devices, the SLCS# pin can be tied to GND to permanently select the slave FIFO interface. This configuration is assumed for the interface and timing examples that follow.

While the SLCS# pin is pulled high by external logic, the MoBL-USB FX2LP18 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




In order to implement synchronous FIFO writes, a typical 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] (setup time must be met with respect to the rising edge of IFCLK), 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 (setup and hold times must be met with respect to the rising edge of IFCLK), de-assert SLWR. 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

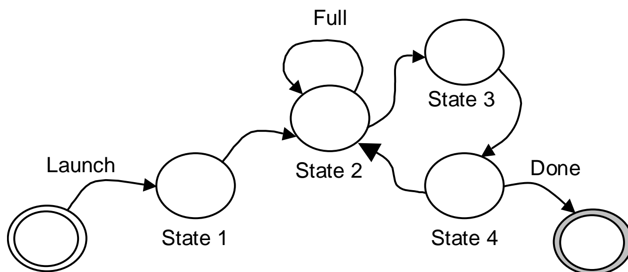
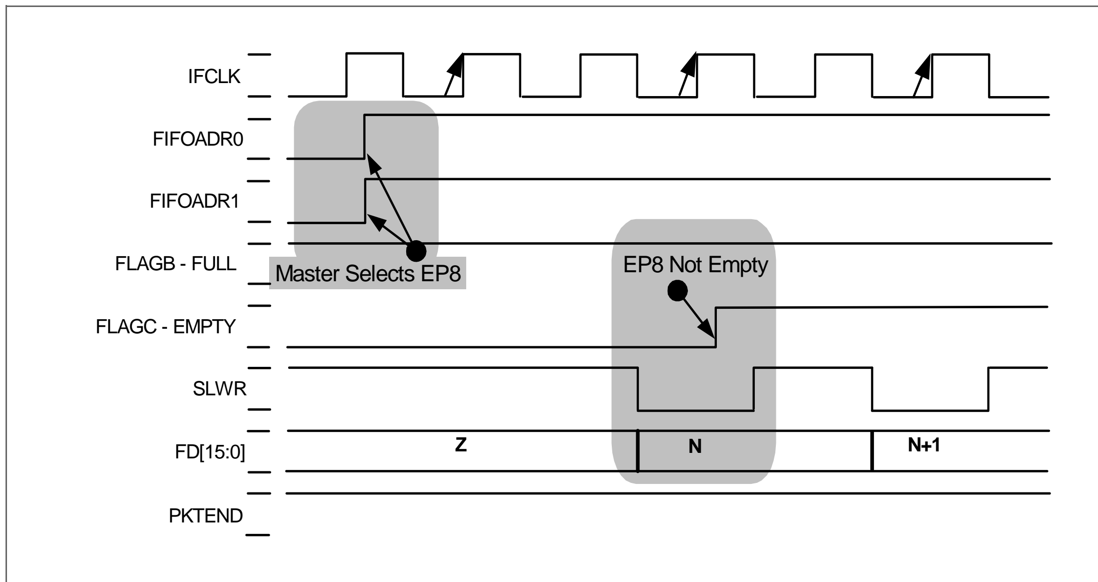





Exhibit 2033 - Page 114 of 346

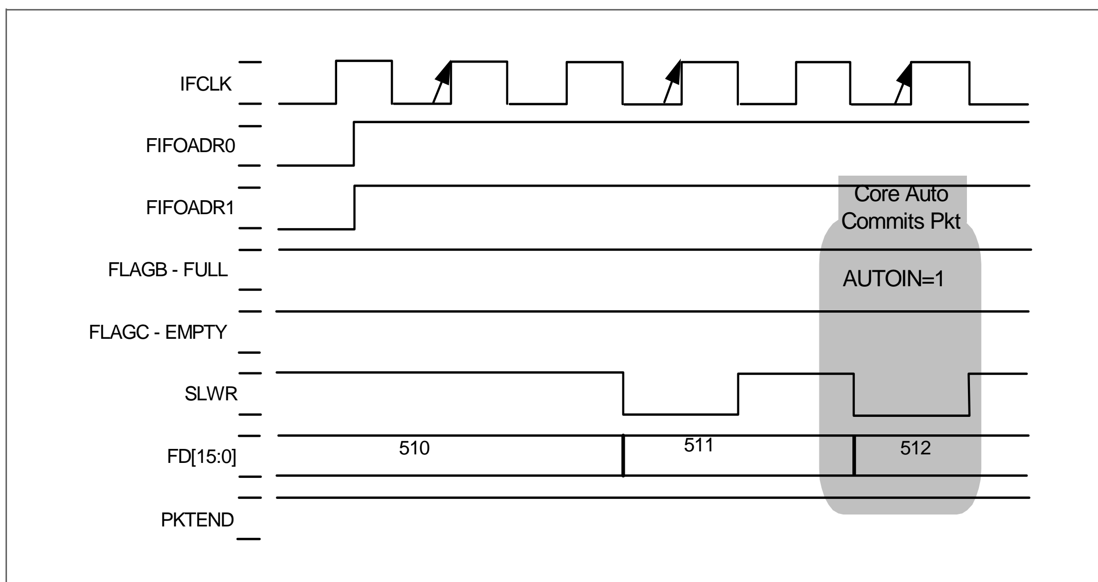
Figure 9-12. Timing Example: Synchronous FIFO Writes, Waveform 1




Figures 9-12 to 9-14 show timing examples of an external master performing synchronous FIFO writes to EP8. These examples assume that EP8 is configured as IN, Bulk, 512 bytes buffer size, 4x buffered, WORDWIDE = 1, AUTOIN = 1, EP8AUTOINLENH:L = 512. With AUTOIN = 1, and EP8AUTOINLENH:L = 512, this causes data packets to be automatically committed to USB whenever the master fills the FIFO with 512 bytes (or 256 words since WORDWIDE = 1).

In Figure 9-12, the external master selects EP8 by setting FIFOADR[1:0] to ‘11’ and once it writes the first data value over the FD bus, FLAGC - EMPTY exhibits a ‘not-empty’ condition.

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




In Figure 9-13, once the external master writes the 512th word into the EP8 FIFO, the second 512-byte packet is automatically committed to USB. The first 512-byte packet was automatically committed to USB when the external master wrote the 256th word into the EP8 FIFO.



**Exhibit 2033 - Page 115 of 346**

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

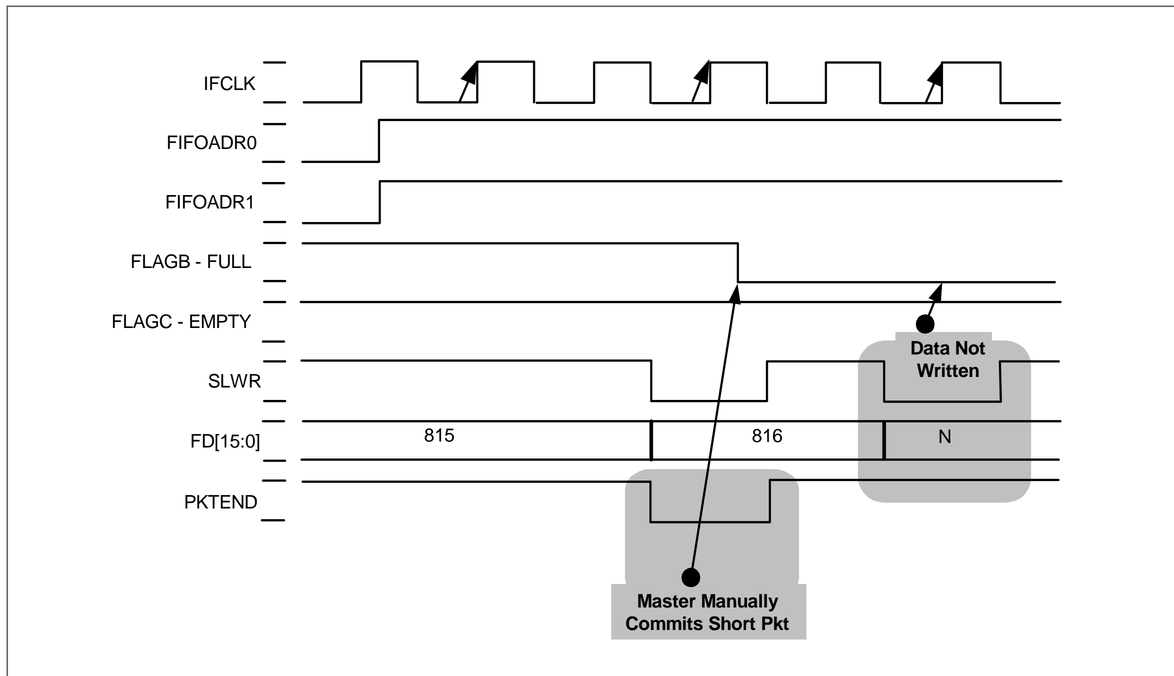



Figure 9-14 shows the 4th packet in the EP8 FIFO being manually committed by pulsing PKTEND. There is no specific timing requirement for PKTEND assertion with respect to SLWR assertion. Hence, PKTEND is asserted the same time the 816th word is written into EP8. This causes the short packet to be committed, which contains 48 words (or 96 bytes). The 4th packet would have been automatically committed if the external master finished writing the 1024th word.

Once the 4th packet has been committed, FLAGB - FULL is asserted, indicating that no more FIFO buffers are available for the external master to write into. A buffer will become available once the host has read an entire packet.

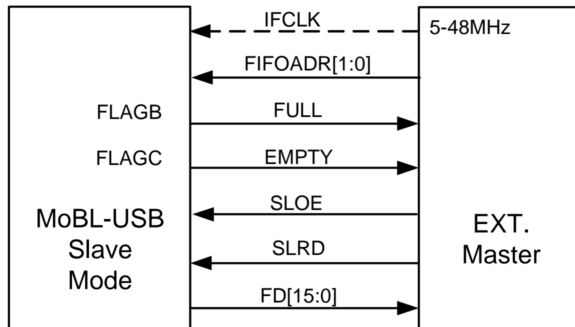
**Note** FIFOADR[1:0] must be held constant during the PKTEND assertion.



**Exhibit 2033 - Page 116 of 346**

### 9.2.8 Implementing Synchronous Slave FIFO Reads

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




In order to implement synchronous FIFO reads, a typical 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] (setup time must be met with respect to the rising edge of IFCLK), 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, assert SLRD (setup and hold times must be met with respect to the rising edge of IFCLK), de-assert SLRD. De-assert SLOE, transition to State 4.

**Note** Since SLOE has no other function than to enable the FD outputs, it is also correct to tie the SLRD and SLOE signals together.

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

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

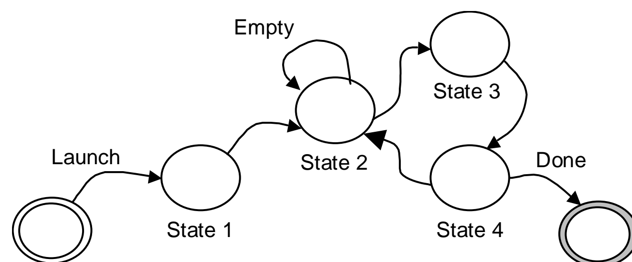
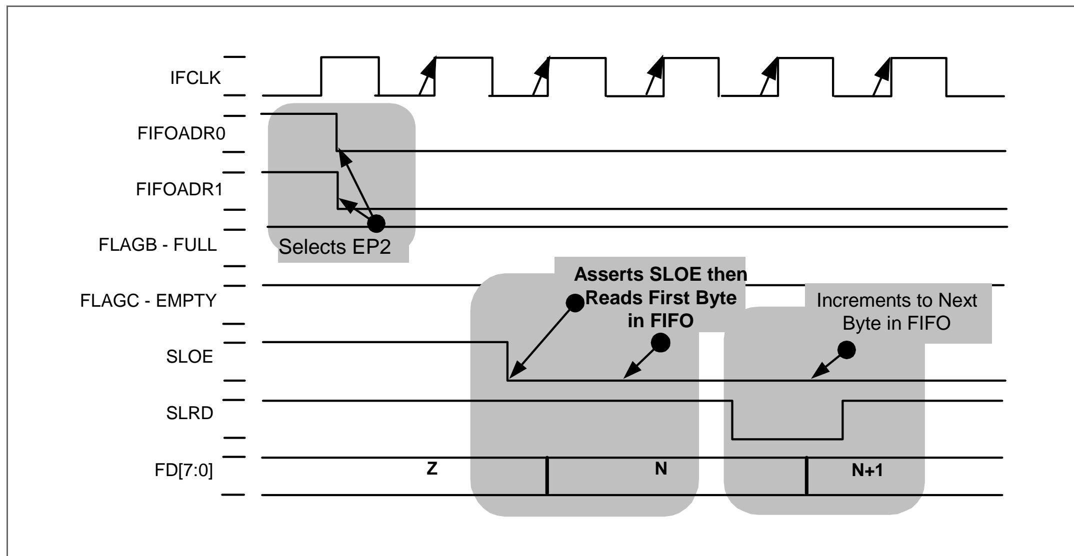





Exhibit 2033 - Page 117 of 346

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




Figures 9-17 and 9-18 show timing examples of an external master performing synchronous FIFO reads from EP2. These examples assume that EP2 is configured as OUT, Bulk, 512 bytes buffer size, 2x buffered, WORDWIDE = 0, AUTOOUT = 1.

In Figure 9-17, the external master selects EP2 by setting FIFOADR[1:0] to 00. It asserts SLOE to turn on the FD output drivers, samples the first byte in the FIFO, and then pulses SLRD to increment the FIFO pointer.

Figure 9-18. Timing Example: Synchronous FIFO Reads, Waveform 2, EMPTY Flag Illustrated

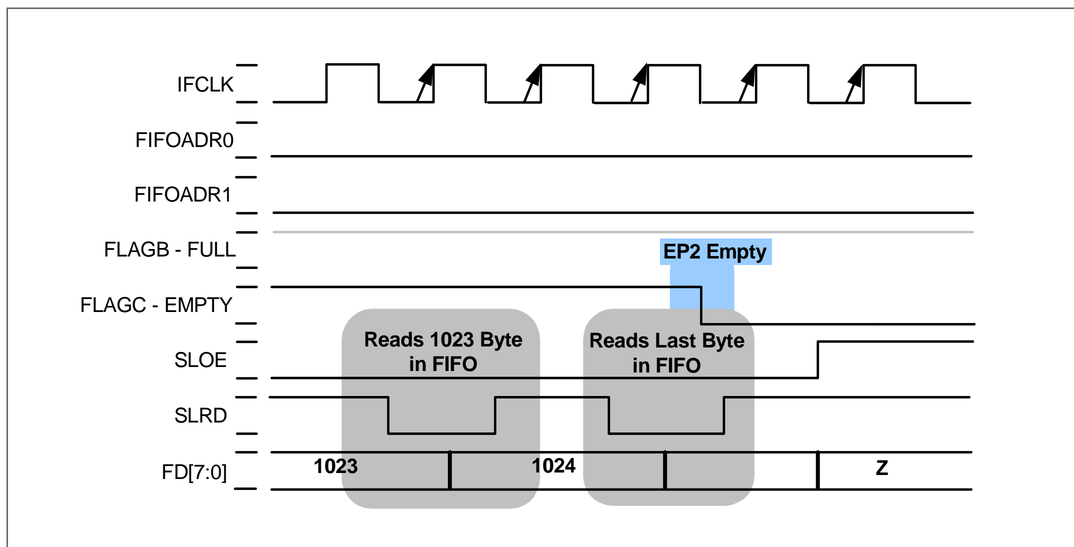


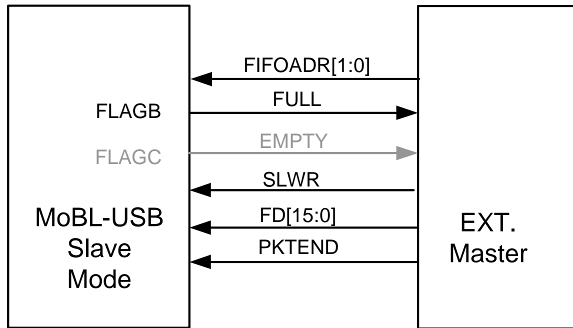

Figure 9-18 shows FLAGC - EMPTY assert after the master reads the 1024th (last) byte in the FIFO. This assumes that the host has only sent 1024 bytes to EP2.



Exhibit 2033 - Page 118 of 346

### 9.2.9 Implementing Asynchronous Slave FIFO Writes

Figure 9-19. Interface Pins Example: Asynchronous FIFO Writes




In order to implement asynchronous FIFO writes, a typical 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] (setup time must be met with respect to the asserting edge of SLWR), 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 (setup time must be met with respect to the de-asserting edge of SLWR), write data to the FIFO and increment the FIFO 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

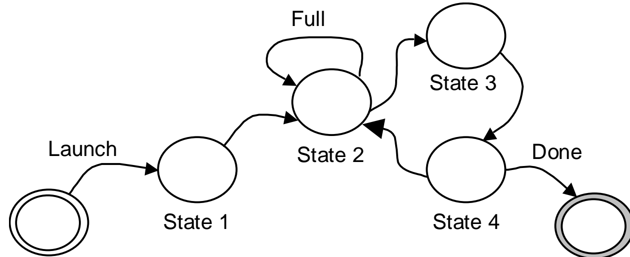





**Exhibit 2033 - Page 119 of 346**

Figure 9-21. Timing Example: Asynchronous FIFO Writes

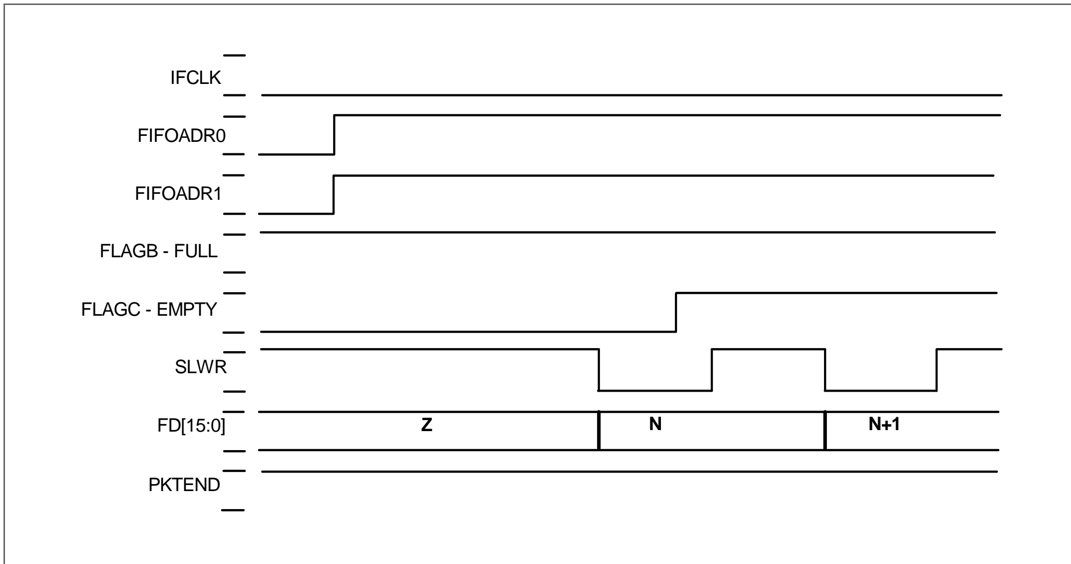


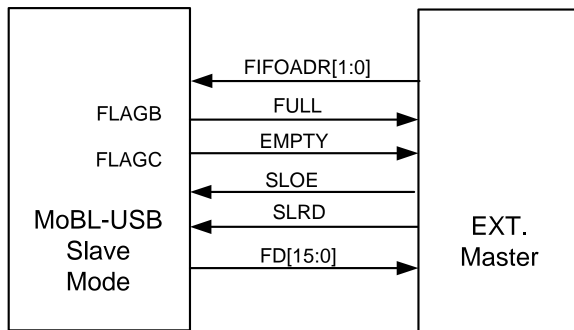

Figure 9-21 shows a timing example of asynchronous FIFO writes to EP8. The external master selects EP8 by setting FIFOADR[1:0] to 11. Once it writes the first data value over the FD bus, FLAGC - EMPTY exhibits a 'not-empty' condition.



Exhibit 2033 - Page 120 of 346

### 9.2.10 Implementing Asynchronous Slave FIFO Reads

Figure 9-22. Interface Pins Example: Asynchronous FIFO Reads




In order to implement asynchronous FIFO reads, a typical 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] (setup time must be met with respect to the asserting edge of SLRD), 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 FIFO pointer), de-assert SLOE, transition to State 4.

**Note** Since SLOE has no other function than to enable the FD outputs, it is also correct to tie the SLRD and SLOE signals together.

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

Figure 9-23. State Machine Example: Asynchronous FIFO Reads

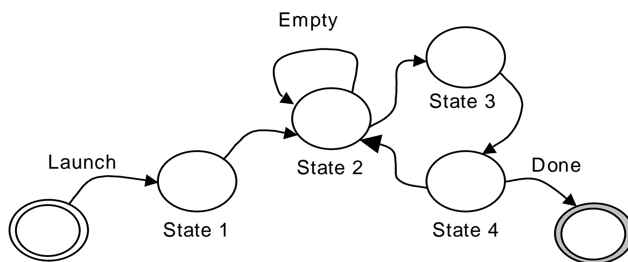


Figure 9-24. Timing Example: Asynchronous FIFO Reads

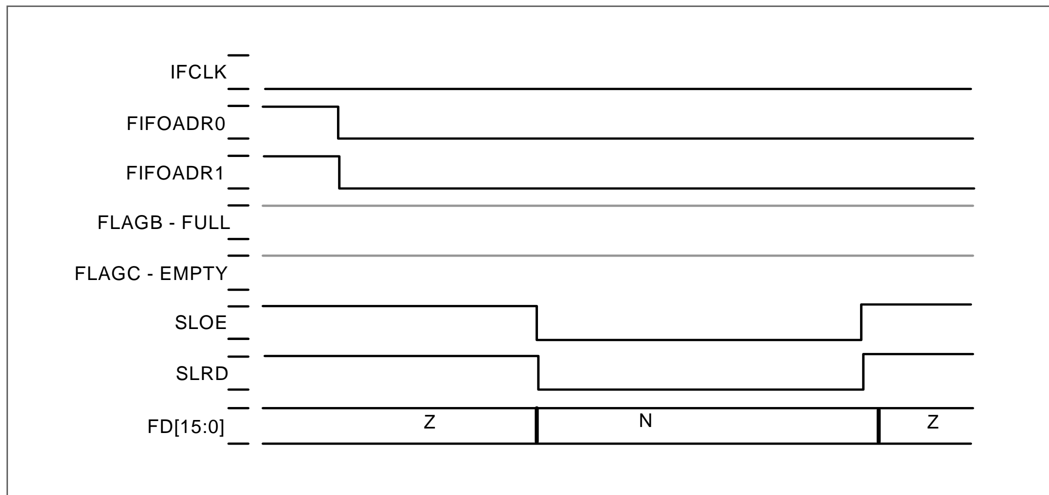



Figure 9-24 shows a timing example of asynchronous FIFO reads from EP2. The external master selects EP2 by setting FIFOADR[1:0] to 00, and strobes SLOE/SLRD to sample data on the FD bus.

## 9.3 Firmware

This section describes the interface between firmware and the FIFOs. More information is available in the Access to Endpoint Buffers chapter on page 93.

Table 9-3. Registers Associated with Slave FIFO Firmware

| Register Name | |
|---|---|
| EPxCFG | INPKTEND/OUTPKTEND |
| 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

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 Autopointer mechanism built into the MoBL-USB FX2LP18 (see section "Autopointers" on page 105).

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, and others.

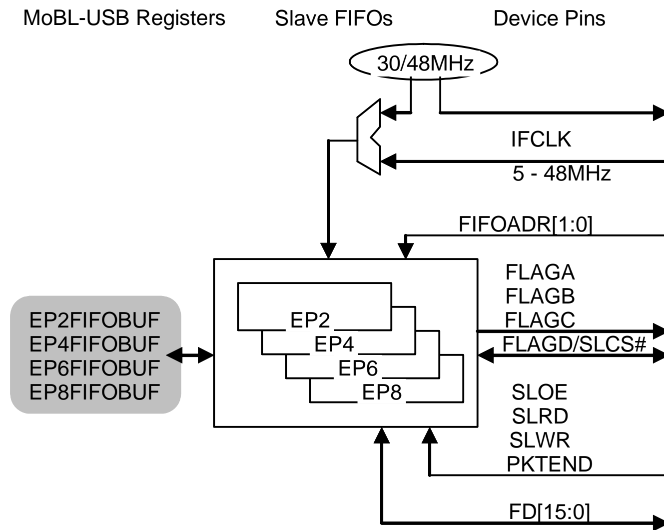


**Exhibit 2033 - Page 122 of 346**

This chapter focuses on the registers and bits which are specific to slave-FIFO operation; for a more detailed description of all the FIFO registers, see the chapters Access to Endpoint Buffers, on page 93 and Registers, on page 237.

Setting the REVCTL bits enables features that are not required by every application. So although not necessary, for proper operation as described in this chapter, 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 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. MoBL-USB FX2LP18 can automatically commit endpoint data to and from the slave FIFO interface (AUTOIN = 1, AUTOOUT = 1), or manually commit endpoint data to and from the slave FIFO interface (AUTOIN = 0, AUTOOUT = 1).

On a hard reset, these endpoint memories are configured as follows:

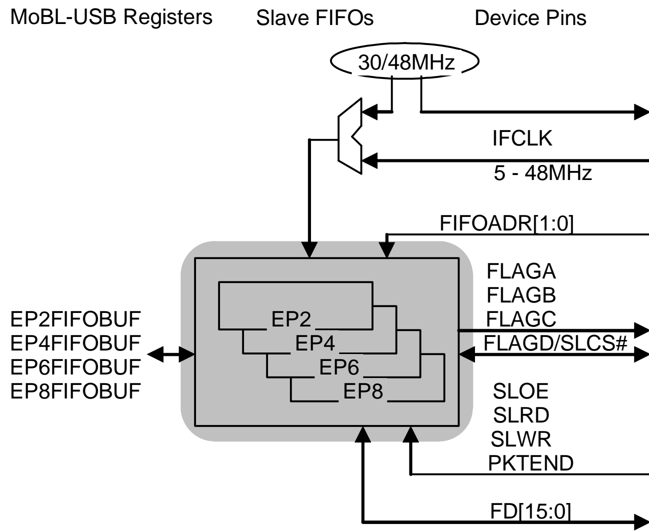
1. EP2 - Bulk OUT, 512 bytes/packet, 2x buffered.
2. EP4 - Bulk OUT, 512 bytes/packet, 2x buffered.
3. EP6 - Bulk IN, 512 bytes/packet, 2x buffered.
4. EP8 - Bulk IN, 512 bytes/packet, 2x buffered.

**Note** In full-speed mode, buffer sizes scale down to 64 bytes for the non-isochronous types.



Exhibit 2033 - Page 123 of 346

Figure 9-26. EPx Memories




## 9.3.3 Slave FIFO Programmable-Level Flag

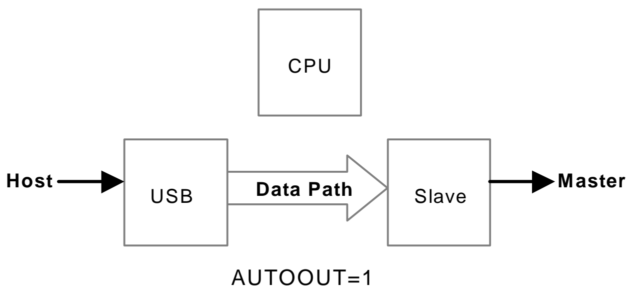
Each FIFO's Programmable-level Flag (PF) asserts when the FIFO reaches a user-defined fullness threshold.

See the discussion of the EPxFIFOPFH:L registers in the Registers chapter on page 237 for full details.

## 9.3.4 Auto-In / Auto-Out Modes

The 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 firmware to packetize the data or explicitly signal the MoBL-USB FX2LP18 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 firmware to handshake each incoming packet, arm the endpoint buffers, and so on. 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 Figure 9-28 and Figure 9-29 on page 125.

On a hard reset, all FIFOs default to Manual Mode (for example, AUTOIN = 0 and AUTOOUT = 0).



Exhibit 2033 - Page 124 of 346

Figure 9-28.  TD_Init Example: Configuring AUTOOUT = 1

```
TD_Init():
… … … … …
REVCTL = 0x03;         // REVCTL.0 and REVCTL.1 to set 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;
OUTPKTEND = 0x82;      // Arm both EP2 buffers to “prime the pump”
SYNCDELAY;
OUTPKTEND = 0x82;
SYNCDELAY;
EP2FIFOCFG = 0x10;     // EP2 is AUTOOUT=1, AUTOIN=0, ZEROLEN=0, WORDWIDE=0
… … … … …
```

Figure 9-29.  TD_Init Example: Configuring AUTOIN = 1

```
TD_Init():
… … … … …
REVCTL = 0x03;         // REVCTL.0 and REVCTL.1 set to 1
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;
… … … … …
```



Exhibit 2033 - Page 125 of 346

### 9.3.5 CPU Access to OUT Packets, AUTOOUT = 1

The MoBL-USB FX2LP18's CPU is not in the host-to-master data path when AUTOOUT = 1. To achieve the maximum 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.

Figure 9-30. TD_Poll Example: No Code Necessary for OUT Packets When AUTOOUT=1

```
TD_Poll():
… … … … …
// no code necessary to xfr data from host to master!
// AUTOOUT=1 auto-commits packets
… … … … …
```

**Note** If AUTOOUT = 1, an OUT FIFO buffer is automatically committed, and could contain 0-1024 bytes, depending on the size of the OUT packet transmitted by the host. The buffer size must be set appropriately (512 or 1024) to accommodate the USB data payload size.

### 9.3.6 CPU Access to OUT Packets, AUTOOUT = 0

In some systems, it may be desirable to allow the MoBL-USB FX2LP18'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).

Figure 9-31. TD_Init Example, Configuring AUTOOUT=0

```
TD_Init():
… … … … …
REVCTL = 0x03;        // REVCTL.0 and REVCTL.1 set 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;
… … … … …
```

As Illustrated in Figure 9-32 on page 127, firmware can do one of three things when the MoBL-USB FX2LP18 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 (Figure 9-33 on page 127).
2. It can *skip* (discard) the packet by writing OUTPKTEND with SKIP=1 (Figure 9-34 on page 127).
3. It can *edit* the packet (or *source* an entire OUT packet) by writing to the FIFO buffer directly, then write the length of the packet to EPxBCH:L. The write to EPxBCL commits the edited packet, so EPxBCL should be written *after* writing EPxBCH (Figure 9-35 on page 128).

In all cases, the OUT buffer automatically re-arms so it can receive the next packet, once the external master has finished reading all data in the OUT buffer.

See section 8.6.2.4 EP2BCH:L, EP4BCH:L, EP6BCH:L, EP8BCH:L on page 101 for a detailed description of the SKIP bit.



Exhibit 2033 - Page 126 of 346

Figure 9-32. Skip, Commit, or Source (AUTOOUT=0)

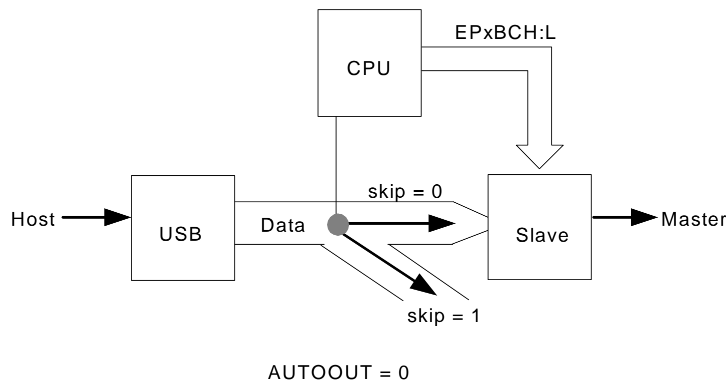



Figure 9-33. TD_Poll Example, AUTOOUT=0, Commit Packet

```
TD_Poll():
… … … … …
if( !( EP2468STAT & 0x01 ) )
{ // EP2EF=0 when FIFO NOT empty, host sent packet
  OUTPKTEND = 0x02;  // SKIP=0, pass buffer on to master
}
… … … … …
```

Figure 9-34. TD_Poll Example, AUTOOUT=0, Skip 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
}
… … … … …
```



Exhibit 2033 - Page 127 of 346

Figure 9-35. TD_Poll Example, AUTOOUT=0, Source

```
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"
}
… … … … …
```

**Note** If an uncommitted packet is in an OUT endpoint buffer when the MoBL-USB FX2LP18 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 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.

Figure 9-36. TD_Init Example, OUT Endpoint Initialization

```
TD_Init():
… … … … …
REVCTL = 0x03;       // REVCTL.0 and REVCTL.1 set to 1
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
… … … … …
```



Exhibit 2033 - Page 128 of 346

### 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).

Figure 9-37. TD_Poll Example, AUTOIN = 1

```
TD_Poll():
… … … … …
// no code necessary to xfr data from master to host!
// AUTOIN=1 and EP8AUTOINLEN=512 auto commits packets
// in 512 byte chunks.
… … … … …
```

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, firmware can temporarily halt the flow of data from the external master (via a signal on a general-purpose IO 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 on page 130.

Figure 9-38. Master Writes Directly to Host, AUTOIN = 1

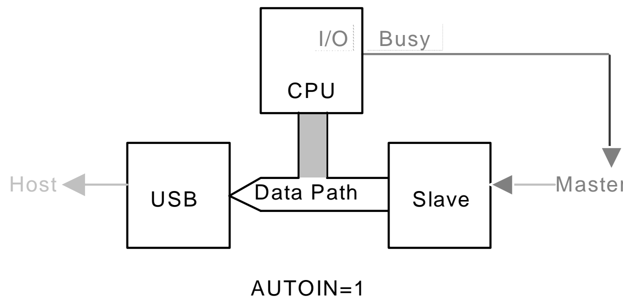



Figure 9-39. Firmware Intervention, AUTOIN = 0 or 1

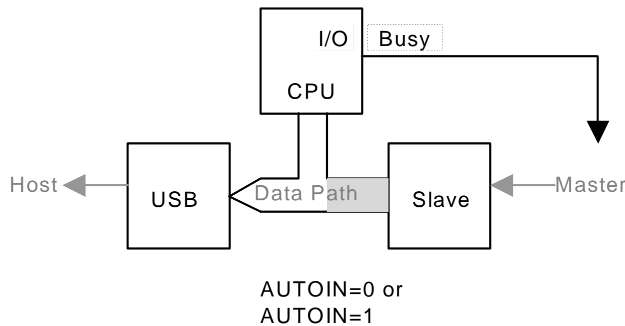





Exhibit 2033 - Page 129 of 346

Figure 9-40. TD_Poll Example: Sourcing an IN Packet

```
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 );
  }
}
… … … … …
```



Exhibit 2033 - Page 130 of 346

### 9.3.8 Access to IN Packets, AUTOIN=0

In some systems, it may be desirable to allow the MoBL-USB FX2LP18'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 EPxFIFO-CFG register must be cleared to '0'.

In Manual-In mode, 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 commit length to EPxBCH and EPxBCL (in that order).

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 = 0x08;  // firmware commits EP8 packet
                      // by writing 8 to INPKTEND
    release_master( EP8 );
  }
}
… … … … …
```

Figure 9-42. TD_Poll Example, AUTOIN=0, Skipping 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 9-43. TD_Poll Example, AUTOIN=0, Editing a Packet via EPxBCH:L

```
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, packet size is 8
}
… … … … …
```



Exhibit 2033 - Page 131 of 346

### 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. Set REVCTL.0 and REVCTL.1 to ‘1’.
4. Configure EPxCFG.
5. Reset the FIFOs.
6. Set bit EPxFIFOCFG.3=1.
7. 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. Set REVCTL.0 and REVCTL.1 to ‘1’.
4. Configure EPxCFG.
5. Reset the FIFOs.
6. Arm OUT buffers by writing to OUTPKTEND N times with skip = 1, where N is buffering depth.
7. Set bit EPxFIFOCFG.4=1.



Exhibit 2033 - Page 132 of 346

### 9.3.10 Auto-Mode Example: Synchronous FIFO IN Data Transfers

Figure 9-44. Code Example, Synchronous Slave FIFO IN Data Transfer

```
TD_Init():
IFCONFIG = 0x03;      // use IFCLK pin driven by external logic (5MHz to 48MHz)
                      // use slave FIFO interface pins driven sync by external master
SYNCDELAY;
REVCTL = 0x03;        // REVCTL.0 and REVCTL.1 set to 1
SYNCDELAY;
EP8CFG = 0xE0;        // sets EP8 valid for IN's
                      // and defines the endpoint for 512 byte packets, 2x buffered
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:
EP8FIFOCFG = 0x0C;    // this lets the MoBL-USB FX2LP18 auto commit IN packets, gives the
                      // ability to send zero length packets,
                      // and sets the slave FIFO data interface to 8-bits

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
SYNCDELAY;
FIFOPINPOLAR = 0x00; // set all slave FIFO interface pins as active low
SYNCDELAY;
EP8AUTOINLENH = 0x02; // MoBL-USB FX2LP18 automatically commits data in 512-byte chunks
SYNCDELAY;
EP8AUTOINLENL = 0x00;

SYNCDELAY;
EP8FIFOPFH = 0x80;    // you can define the programmable flag (FLAGA)
SYNCDELAY;            // to be active at the level you wish
EP8FIFOPFL = 0x00;
```

```
TD_Poll():
// nothing! The MoBL-USB FX2LP18 is doing all the work of transferring packets
// from the external master sync interface to the endpoint buffer...
```



Exhibit 2033 - Page 133 of 346

### 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 "Auto-Mode Example: Synchronous FIFO IN Data Transfers" on page 133, 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.

Figure 9-45. TD_Init Example, Asynchronous Slave FIFO IN Data Transfers

```
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
```

Code to perform the transfers is, as before, unnecessary; as Figure 9-46 illustrates.

Figure 9-46. TD_Poll Example, Asynchronous Slave FIFO IN Data Transfers

```
TD_Poll( ):
// nothing! The MoBL-USB FX2LP18 is doing all the work of transferring packets
// from the external master async interface to the endpoint buffer…
```

## 9.4 Switching Between Manual-Out and Auto-Out

Because OUT endpoints are not automatically armed when the MoBL-USB FX2LP18 enters Auto-Out mode, the firmware can safely switch the MoBL-USB FX2LP18 between Manual-Out and Auto-Out modes without any need to flush or reset the FIFOs.

**Note** Switching between Manual-Out mode to Auto-Out mode is not required for every application. Most applications remain in either mode for each endpoint.



Exhibit 2033 - Page 134 of 346

# 10. General Programmable Interface

## 10.1 Introduction

The General Programmable Interface (GPIF) is an 'internal master' to the MoBL-USB FX2LP18's endpoint FIFOs. It replaces the external 'glue' logic which might otherwise be required to build an interface between the MoBL-USB FX2LP18 and the outside world.

At the GPIF's core is a programmable state machine which generates up to six 'control' and nine 'address' outputs, and accepts six external and two internal 'ready' inputs. Four user-defined *Waveform Descriptors* control the state machine; generally (but not necessarily), one is written for FIFO reads, one for FIFO writes, one for single-byte/word reads, and one for single-byte/word writes.

**'Read' and 'Write' are from the MoBL-USB FX2LP18's point of view.** 'Read' waveforms transfer data from the outside world to the MoBL-USB FX2LP18; 'Write' waveforms transfer data from the MoBL-USB FX2LP18 to the outside world.

Firmware can assign the FIFO-read and -write waveforms to any of the four FIFOs, and the GPIF will generate the proper strobes and handshake signals to the outside-world interface as data is transferred into or out of that FIFO.

As with external mastering (see Slave FIFOs, on page 107), the data bus between the FIFOs and the outside world can be either 8 or 16 bits wide.

The GPIF is not limited to simple handshaking interfaces between the MoBL-USB FX2LP18 and external ASICs or microprocessors; it is powerful enough to directly implement such protocols as ATAPI (PIO and UDMA), IEEE 1284 (EPP Parallel Port), Utopia, and others. A MoBL-USB FX2LP18 can, for instance, function as a single-chip interface between USB and an IDE hard disk drive or CompactFlash™ memory card.

This chapter provides an overview of GPIF, discusses external connections, and explains the operation of the GPIF engine. Figure 10-1 on page 136 presents a block diagram illustrating GPIF's place in the MoBL-USB FX2LP18 system.

GPIF waveforms are created with the Cypress GPIF Designer utility, a Windows™-based application which is distributed with the Cypress MoBL-USB FX2LP18 Development Kit. Although this chapter will describe the structure of the Waveform Descriptors in some detail, knowledge of that structure is usually **not** necessary. The GPIF Designer simply hides the complexity of the Waveform Descriptors; it does not compromise the programmer's control over the GPIF in any way.



**Exhibit 2033 - Page 135 of 346**

Figure 10-1. GPIF's Place in the MoBL-USB FX2LP18 System

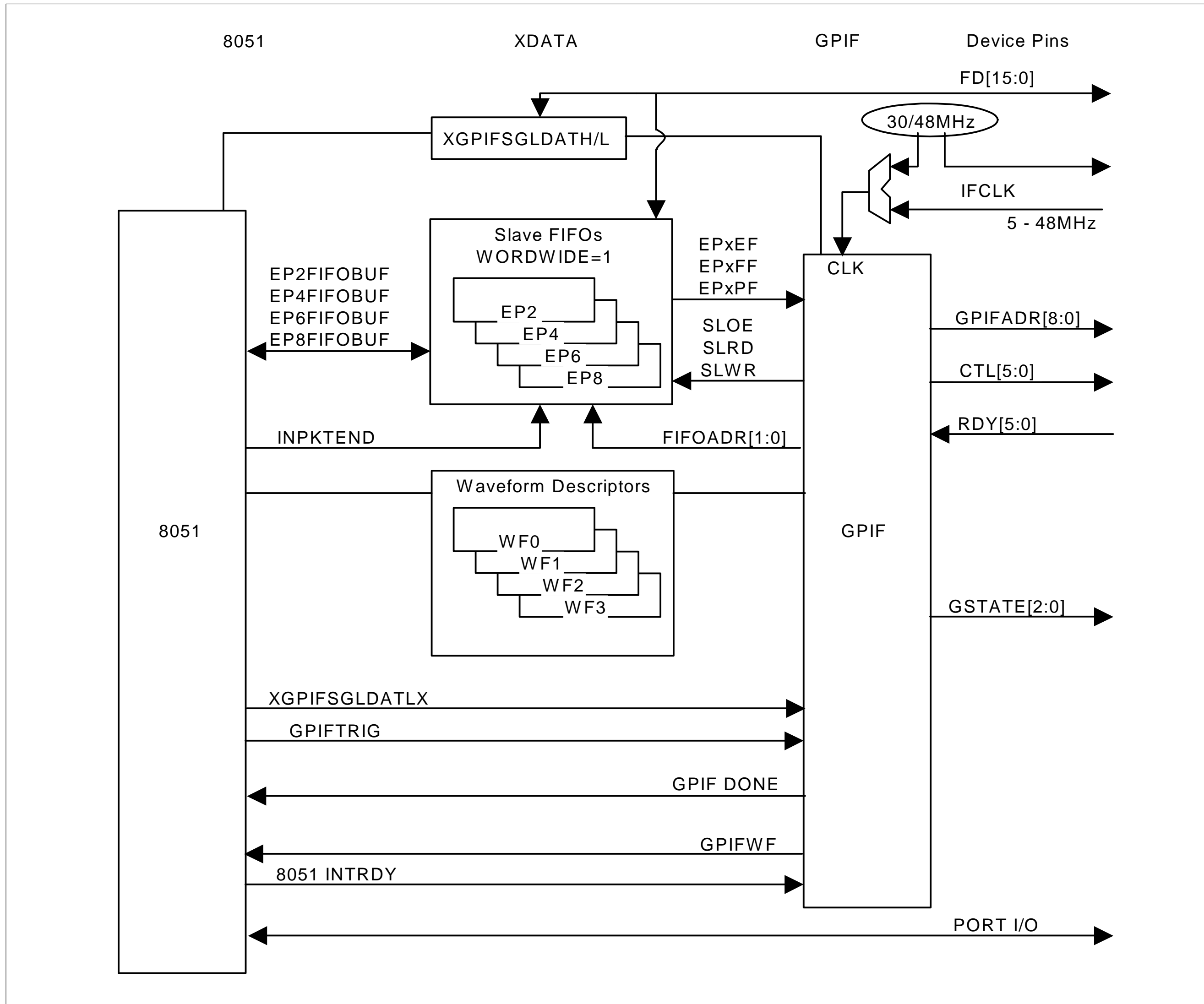
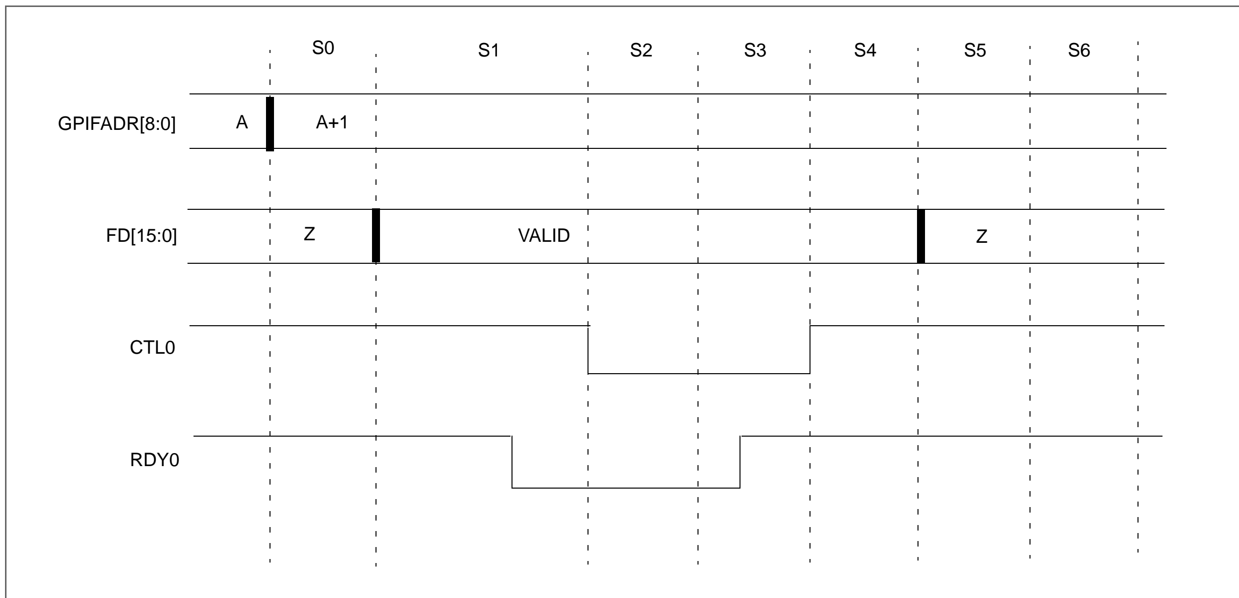



Figure 10-2 on page 137 shows an example of a simple GPIF transaction. For this transaction, the GPIF generates an address (GPIFADR[8:0]), drives the FIFO data bus (FD[15:0]), then waits for an externally-supplied handshake signal (RDY0) to go low, after which it pulls its CTL0 output low. When the RDY0 signal returns high, the GPIF brings its CTL0 output high, then floats the data bus.



**Exhibit 2033 - Page 136 of 346**

Figure 10-2. Example GPIF Waveform




## 10.1.1 Typical GPIF Interface

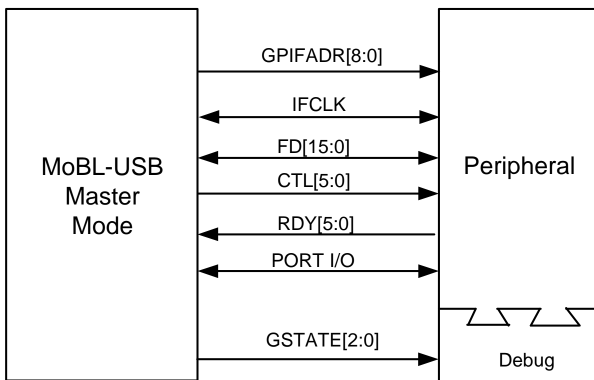
The GPIF allows the MoBL-USB FX2LP18 to connect directly to external peripherals such as ASICs, DSPs, or other digital logic that uses an 8- or 16-bit parallel interface.

The GPIF provides external pins that can operate as outputs (CTL[5:0]), inputs (RDY[5:0]), Data bus (FD[15:0]), and Address Lines (GPIFADR[8:0]).

A Waveform Descriptor in internal RAM describes the behavior of each of the GPIF signals. The Waveform Descriptor is loaded into the GPIF registers by the firmware during initialization, and it is then used throughout the execution of the code to perform transactions over the GPIF interface.

Figure 10-3 shows a block diagram of a typical interface between the MoBL-USB FX2LP18 and a peripheral function.

Figure 10-3. MoBL-USB FX2LP18 Interfacing to a Peripheral




The following sections detail the features available and steps needed to create an efficient GPIF design. This includes definition of the external GPIF connections and the internal register settings, along with firmware needed to execute data transactions over the interface.



Exhibit 2033 - Page 137 of 346

## 10.2 Hardware

Table 10-1 lists the registers associated with the GPIF hardware; a detailed description of each register may be found in the Registers, on page 237

Table 10-1. Registers Associated with GPIF Hardware

| Register Name | |
|---|---|
| GPIFIDLECS | IFCONFIG |
| GPIFIDLECTL | FIFORESET |
| GPIFCTLCFG | EP*x*CFG |
| PORTCCFG | EP*x*FIFOCFG |
| PORTECFG | EP*x*AUTOINLENH/L |
| GPIFADRH/L | EP*x*FIFOPFH/L |
| GPIFTCB3:0 | |
| GPIFWFSELECT | EPxGPIFTRIG |
| EP*x*GPIFFLGSEL | GPIFABORT |
| EP*x*GPIFPFSTOP | XGPIFSGLDATH/LX/LNOX |
| GPIFREADYCFG | GPIFSGLDATH/LX/LNOX |
| GPIFREADYSTAT | GPIFTRIG |
| **Note** The '*x*' in these register names represents 2, 4, 6, or 8; endpoints 0 and 1 are not associated with the GPIF. | |

### 10.2.1 The External GPIF Interface

The GPIF provides many general input and output signals with which external peripherals may be interfaced 'gluelessly' to the MoBL-USB FX2LP18.

The GPIF interface signals are shown in Table 10-2.

Table 10-2. GPIF Pin Descriptions

| PIN | IN/OUT | Description |
|---|---|---|
| CTL[5:0] | O / Hi-Z | Programmable control outputs |
| RDY[5:0] | I | Sampleable ready inputs |
| FD[15:0] | I / O / Hi-Z | Bidirectional FIFO data bus |
| GPIFADR[8:0] | O / Hi-Z | Address outputs |
| IFCLK | I / O | Interface clock |
| GSTATE[2:0] | O / Hi-Z | Current GPIF State number (for debug) |

The Control Output pins (CTL[5:0]) are usually used as strobes (enable lines), read/write lines, and others.

The Ready Input pins (RDY[5:0]) are sampled by the GPIF and can force a transaction to wait (inserting wait states), continue, or repeat until they're in a particular state.

The GPIF Data Bus is a collection of the FD[15:0] pins.

- An 8-bit wide GPIF interface uses pins FD[7:0].
- A 16 bit-wide GPIF interface uses pins FD[15:0].

The GPIF Address lines (GPIFADR[8:0]) can generate an incrementing address as data is transferred. If higher-order address lines are needed, other non-GPIF IO signals (for example, general-purpose IO pins) may be used.

The Interface Clock, IFCLK, can be configured to be either an input (default) or an output interface clock for synchronous interfaces to external logic.

The GSTATE[2:0] pins are outputs which show the current GPIF State number; they are used for debugging GPIF waveforms.



Exhibit 2033 - Page 138 of 346

### 10.2.2 Default GPIF Pins Configuration

The MoBL-USB FX2LP18 comes out of reset with its IO pins configured in 'Ports' mode, not 'GPIF Master' mode. To configure the pins for GPIF mode, the IFCFG1:0 bits in the IFCONFIG register must be set to 10 (see Table 13-10, "IFCFG Selection of Port IO Pin Functions," on page 211 for details).

### 10.2.3 Six Control OUT Signals

The 100- pin MoBL-USB FX2LP18 package bring out all six Control Output pins, CTL[5:0]. The 56-pin package brings out three of these signals, CTL[2:0]. CTLx waveform edges can be programmed to make transitions as often as once per IFCLK clock (once every 20.8 ns if IFCLK is running at 48MHz).

By default, these signals are driven high.

#### *10.2.3.1 Control Output Modes*

The GPIF Control pins (CTL[5:0]) have several output modes:

- CTL[3:0] can act as CMOS outputs (optionally tristatable) or open-drain outputs.
- CTL[5:4] can act as CMOS outputs or open-drain outputs.

  If CTL[3:0] are configured to be tristatable, CTL[5:4] are not available.

Table 10-3. CTL[5:0] Output Modes

| TRICTL (GPIFCTLCFG.7) | GPIFCTLCFG[6:0] | CTL[3:0] | CTL[5:4] |
|---|---|---|---|
| 0 | 0 | CMOS, Not Tristatable | CMOS, Not Tristatable |
| 0 | 1 | Open-Drain | Open-Drain |
| 1 | X | CMOS, Tristatable | Not Available |

### 10.2.4 Six Ready IN signals

The 100- pin MoBL-USB FX2LP18 packages bring out all six Ready inputs, RDY[5:0]. The 56-pin package brings out two of these signals, RDY[1:0].

The RDY inputs can be sampled synchronously or asynchronously. When the GPIF samples RDY inputs asynchronously (SAS=0), the RDY inputs are unavoidably delayed by a small amount (approximately 24 ns at 48 MHz IFCLK). In other words, when the GPIF "looks" at a RDY input, it actually "sees" the state of that input 24 ns ago.

### 10.2.5 Nine GPIF Address OUT Signals

Nine GPIF address lines, GPIFADR[8:0], are available. If the GPIF address lines are configured as outputs, writing to the GPIFADRH:L registers drives these pins immediately. The GPIF engine can then increment them under control of the Waveform Descriptors. The GPIF address lines can be tri-stated by clearing the associated PORTxCFG bits and OEx bits to 0 (see section 13.3.3 Port C Alternate Functions on page 209 and section 13.3.4 Port E Alternate Functions on page 210).

### 10.2.6 Three GSTATE OUT Signals

Three GPIF State lines, GSTATE[2:0], are available as an alternate configuration of PORTE[2:0]. These default to general-purpose inputs; setting GSTATE (IFCONFIG.2) to '1' selects the alternate configuration and overrides PORTECFG[2:0] bit settings.

The GSTATE[2:0] pins output the current GPIF State number; this feature is used for debugging GPIF waveforms, and is useful for correlating intended GPIF waveform behavior with actual observed GPIF signaling.

### 10.2.7 8/16-Bit Data Path, WORDWIDE = 1 (default) and WORDWIDE = 0

When the MoBL-USB FX2LP18 is configured for GPIF Master mode, PORTB is always configured as FD[7:0].

If any of the WORDWIDE bits (EPxFIFOCFG.0) are set to '1', PORTD is automatically configured as FD[15:8]. If all the WORDWIDE bits are cleared to 0, PORTD is available for general-purpose IO.



Exhibit 2033 - Page 139 of 346

### 10.2.8 Byte Order for 16-bit GPIF Transactions

Data is sent over USB in packets of 8-bit bytes, not 16-bit words. When the FIFO Data bus is 16 bits wide, the first byte in every pair sent over USB is transferred over FD[7:0] and the second byte is transferred over FD[15:8].

### 10.2.9 Interface Clock (IFCLK)

The GPIF interface can be clocked from either an internal or an external source. The MoBL-USB FX2LP18'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 MoBL-USB FX2LP18 is configured to use an external clock source, the IFCLK pin can be driven at any frequency between 5 MHz and 48 MHz. On a hard reset, the MoBL-USB FX2LP18 defaults to the internal source at 48 MHz, normal polarity, with the IFCLK output disabled. See Figure 10-4.

IFCONFIG.7 selects between internal and external sources: 0 = external, 1 = internal. If an external IFCLK is chosen, it must be free-running at a minimum frequency of 5 MHz. In addition, in order to provide synchronization for the internal endpoint FIFO logic, the external IFCLK source must be present before the firmware sets IFCONFIG.7 = 0.

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 MoBL-USB FX2LP18 with certain external circuitry. When an internal IFCLK is used (IFCONFIG.7 = 1), IFCONFIG.4 only affects the IFCLK output polarity (if IFCONFIG.5 = 1). When an external IFCLK is used (IFCONFIG.7 = 0), IFCONFIG.4 only affects the IFCLK input polarity. Figure 10-5 on page 141, for example, demonstrates the use of IFCLK output inversion in order to ensure a long enough setup time ($t_S$) for a control signal to the peripheral.

When IFCLK is configured as an input, the minimum external frequency that can be applied to it is 5 MHz. This clock must be applied prior to initialization of the GPIF and interruptions of it will lower the overall frequency, causing violations of the minimum frequency requirement.

Figure 10-4. IFCLK Configuration

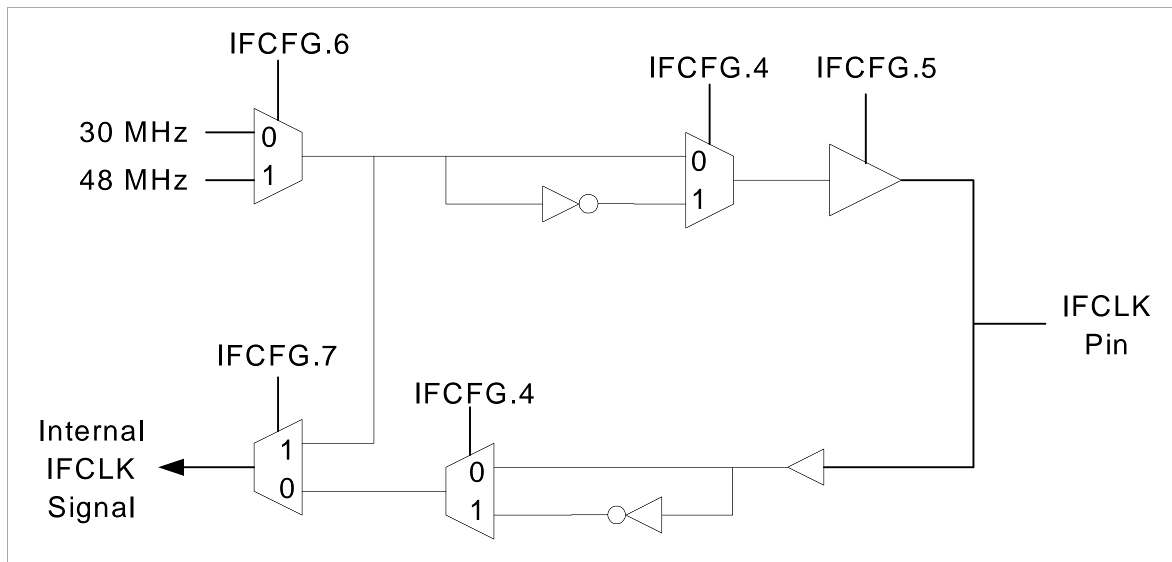
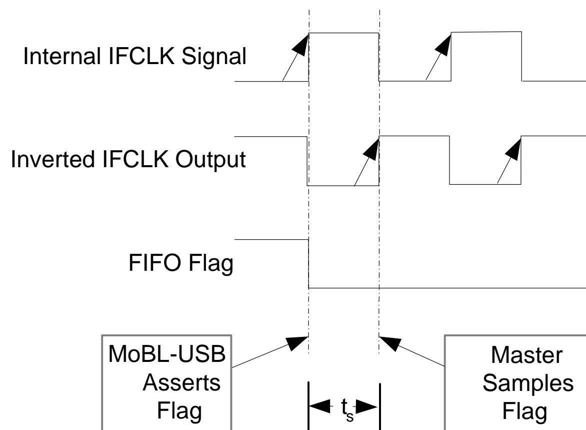





Exhibit 2033 - Page 140 of 346

Figure 10-5. Satisfying Setup Timing by Inverting the IFCLK Output




## 10.2.10 Connecting GPIF Signal Pins to Hardware

The first step in creating the interface between the MoBL-USB FX2LP18's GPIF and an external peripheral is to define the hardware interconnects.

1. **Choose IFCLK settings**. Choose either the internal or external interface clock. If internal, choose either 30 or 48 MHz; if external, ensure that the frequency of the external clock is in the range 5-48 MHz, and that it is free-running.
2. **Determine the proper FIFO Data Bus size.** If the data bus for the interface is 8 bits wide, use the FD[7:0] pins and set WORDWIDE=0. If the data bus for the interface is 16 bits wide, use FD[15:0] and set WORDWIDE=1.
3. **Assign the CTLx signals to the interface**. Make a list of all interface signals to be driven from the GPIF to the peripheral, and assign them to the CTL[5:0] inputs. If there are more output signals than available CTLx outputs, non-GPIF IO signals must be driven manually by MoBL-USB FX2LP18 firmware. In this case, the CTLx outputs should be assigned only to signals that must be driven as part of a data transaction.
4. **Assign the RDYn signals to the interface.** Make a list of all interface signals to be driven from the peripheral to the GPIF, and assign them to the RDY[5:0] inputs. If there are more input signals than available RDY inputs, non-GPIF IO signals must be sampled manually by firmware. In this case, the RDYn inputs should be used only for signals that must be sampled as part of a data transaction.
5. **Determine the proper GPIF Address connections**. If the interface uses an Address Bus, use the GPIFADR[8:0] signals for the least significant bits, and other non-GPIF IO signals for the most significant bits. If the address pins are not needed (as when, for instance, the peripheral is a FIFO) they may be left unconnected.

## 10.2.11 Example GPIF Hardware Interconnect

The following example illustrates the hardware connections that can be made for a standard interface to a 27C256 EPROM.

Table 10-4. Example GPIF Hardware Interconnect

| Step | Result | Connection Made |
|---|---|---|
| 1. Choose IFCLK settings. | Internal IFCLK, 48MHz, Async RDY sampling, GPIF. | No connection. |
| 2. Determine proper FIFO Data Bus size. | 8 bits from the EPROM. | FD[7:0] to D[7:0]. Firmware writes WORDWIDE=0. |
| 3. Assign CTLx signals to the interface. | $\overline{CS}$ and $\overline{OE}$ are inputs to the EPROM. | CTL0 to $\overline{CS}$. CTL1 to $\overline{OE}$. |
| 4. Assign RDYn signals to the interface. | 27C256 EPROM has no output ready/wait signals. | No connection. |
| 5. Determine the proper GPIFADR connections. | 16 bits of address. | GPIFADR[8:0] to A[8:0] and other IO pins to A[15:9]. |

The process is the same for larger, more-complicated interfaces.
**Note** Two other GPIF hardware interconnect examples are also available in the GPIF Designer utility. These examples illustrate a connection between the GPIF and the asynchronous FIFO as well as a connection between the GPIF and a DSP from Texas Instrument.



Exhibit 2033 - Page 141 of 346

## 10.3 Programming the GPIF Waveforms

Each GPIF Waveform Descriptor can define up to 7 States. In each State, the GPIF can be programmed to:

- Drive (high or low) or float the CTL outputs
- Sample or drive the FIFO Data bus
- Increment the value on the GPIF Address bus
- Increment the pointer into the current FIFO
- Trigger a GPIFWF (GPIF Waveform) interrupt

Additionally, each State may either sample any two of the following:

- The RDYx input pins
- A FIFO flag
- The INTRDY (internal RDY) flag
- The Transaction-Count-Expired flag

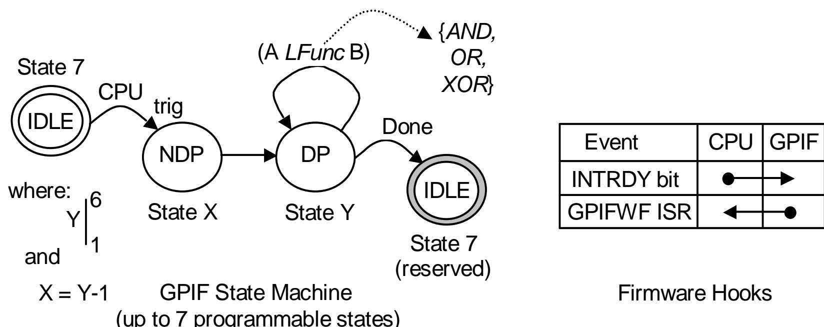
then AND, OR, or XOR the two terms and branch on the result to any State

or:

- Delay a specified number [1-256] of IFCLK cycles

States which sample and branch are called 'Decision Points' (DPs); States which do not are called 'Non-Decision Points' (NDPs).

Figure 10-6. GPIF State Machine Overview




### 10.3.1 The GPIF Registers

Two blocks of registers control the GPIF state machine:

- **GPIF Configuration Registers** — These registers configure the general settings and report the status of the interface. Refer to the Registers chapter on page 237 and the remainder of this chapter for details.
- **Waveform Registers** — These registers are loaded with the Waveform Descriptors that configure the GPIF state machine; there are a total of 128 bytes located at addresses 0xE400 to 0xE47F. *The GPIF Designer utility must be used to create Waveform Descriptors.*

GPIF transactions cannot be initiated until the Configuration Registers and Waveform Registers are loaded by firmware.

Access to the waveform registers is only allowed while the MoBL-USB FX2LP18 is in GPIF mode (that is, IFCFG1:0 = 10). The waveform registers may only be written while the GPIF engine is halted (that is, DONE = 1).

If it's desired to dynamically reconfigure Waveform Descriptors, this may be accomplished by writing just the bytes which change; it's not necessary to reload the entire set of Waveform Descriptors in order to modify only a few bytes.



Exhibit 2033 - Page 142 of 346

## 10.3.2 Programming GPIF Waveforms

The ‘programs’ for GPIF waveforms are the *Waveform Descriptors*, which are stored in the Waveform Registers by firmware.

The MoBL-USB FX2LP18 can hold up to four Waveform Descriptors, each of which can be used for one of four types of transfers: Single Write, Single Read, FIFO Write, or FIFO Read. By default, one Waveform Descriptor is assigned to each transfer type, but it’s not necessary to retain that configuration; all four Waveform Descriptors could, for instance, be configured for FIFO Write usage (see the GPIFWFSELECT register in the Registers chapter on page 237).

Each Waveform Descriptor consists of up to seven 32-bit *State Instructions* that program key transition points for GPIF interface signals. There’s a one-to-one correspondence between the State Instructions and the GPIF state-machine States. Among other things, each State Instruction defines the state of the CTLx outputs, the state of FD[15:0], the use of the RDYn inputs, and the behavior of GPIFADR[8:0].

Transitions from one State to another always happen on a rising edge of the IFCLK, but the GPIF may remain in one State for many IFCLK cycles.

### *10.3.2.1 The GPIF IDLE State*

A Waveform consists of *up to* seven programmable States, numbered S0 to S6, and one special *Idle State,* S7*.* **A Waveform terminates when the GPIF program branches to its Idle State.**

To complete a GPIF transaction, the GPIF program must branch to the IDLE State, *regardless of the State that the GPIF program is currently executing*. For example, a GPIF Waveform might be defined by a program which contains only 2 programmed States, S0 and S1. The GPIF program would branch from S1 (or S0) to S7 when it wished to terminate.

The state of the GPIF signals during the Idle State is determined by the contents of the
GPIFIDLECS and GPIFIDLECTL registers.

Once a waveform is triggered, another waveform may not be started until the first one terminates. Termination of a waveform is signaled through the DONE bit (GPIFIDLECS.7 or GPIFTRIG.7) or, optionally, through the GPIFDONE interrupt.

- If DONE = 0, the GPIF is *busy* generating a Waveform.
- If DONE = 1, the GPIF is *done* (GPIF is in the Idle State) and ready for firmware to start the next GPIF transaction.

**Important** With one exception (writing to the GPIFABORT register in order to force the current waveform to terminate) it is illegal to write to any of the GPIF-related registers (including the Waveform Registers) while the GPIF is busy. Doing so will cause indeterminate behavior likely to result in data corruption.

#### GPIF Data Bus During IDLE

During the Idle State, the GPIF Data Bus (FD[15:0]) can be either driven or tri-stated, depending on the setting of the IDLEDRV bit (GPIFIDLECS.0):

- If IDLEDRV = 0, the GPIF Data Bus is tri-stated during the Idle State.
- If IDLEDRV = 1, the GPIF Data Bus is actively driven during the Idle State, to the value last placed on the bus by a GPIF Waveform.

#### CTL Outputs During IDLE

During the IDLE State, the state of CTL[5:0] depends on the following register bits:

- TRICTL (GPIFCTLCFG.7), as described in section 10.2.3.1 Control Output Modes on page 139.
- GPIFCTLCFG[5:0]
- GPIFIDLECTL[5:0].

The combination of these bits defines CTL5:0 during IDLE as follows:

- If TRICTL is 0, GPIFIDLECTL[5:0] directly represent the output states of CTL5:0 during the IDLE State. The GPIFCTLCFG[5:0] bits determine whether the CTL5:0 outputs are CMOS or open-drain: If GPIFCTLCFG.x = 0, CTLx is CMOS; if GPIFCTLCFG.x = 1, CTLx is open-drain.
- If TRICTL is 1, GPIFIDLECTL[7:4] are the output enables for the CTL[3:0] signals, and GPIFIDLECTL[3:0] are the output values for CTL[3:0]. CTL4 and CTL5 are unavailable in this mode.



Exhibit 2033 - Page 143 of 346

Table 10-5 illustrates this relationship.

Table 10-5. Control Outputs (CTLx) During the IDLE State

| TRICTL | Control Output | Output State | Output Enable |
|---|---|---|---|
| 0 | CTL0 | GPIFIDLECTL.0 | N/A (CTL Outputs are always enabled when TRICTL = 0) |
| | CTL1 | GPIFIDLECTL.1 | |
| | CTL2 | GPIFIDLECTL.2 | |
| | CTL3 | GPIFIDLECTL.3 | |
| | CTL4 | GPIFIDLECTL.4 | |
| | CTL5 | GPIFIDLECTL.5 | |
| 1 | CTL0 | GPIFIDLECTL.0 | GPIFIDLECTL.4 |
| | CTL1 | GPIFIDLECTL.1 | GPIFIDLECTL.5 |
| | CTL2 | GPIFIDLECTL.2 | GPIFIDLECTL.6 |
| | CTL3 | GPIFIDLECTL.3 | GPIFIDLECTL.7 |
| | CTL4 | N/A (CTL4 and CTL5 are not available when TRICTL = 1) | |
| | CTL5 | | |

### *10.3.2.2 Defining States*

Each Waveform is made up of a number of States, each of which is defined by a 32-bit State Instruction. Each State can be one of two basic types: a *Non-Decision Point* (NDP) or a *Decision Point* (DP).

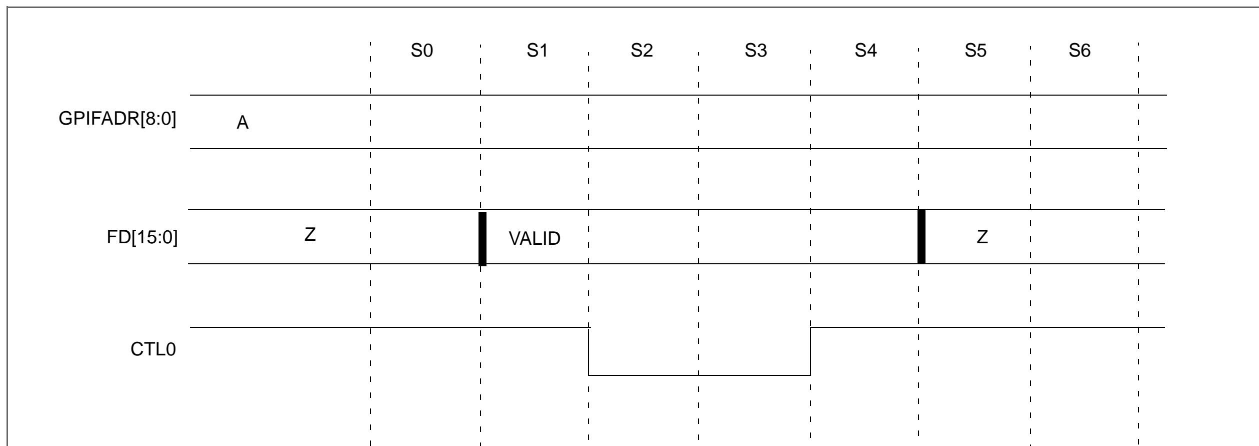
For 'write' waveforms, the data bus is either driven or tri-stated during each State. For 'read' waveforms, the data bus is either sampled/stored or not sampled during each State.

#### Non-Decision Point (NDP) States

For NDP States, the control outputs (CTLx) are defined by the GPIF instruction to be either '1', '0,' or tri-stated during the entire State. NDP States have a programmable fixed duration in units of IFCLK cycles.

Figure 10-7 illustrates the basic concept of NDP States. A write waveform is shown, and for simplicity all the States are shown with equal spacing. Although there are a total of six programmable CTL outputs, only one (CTL0) is shown in Figure 10-7.

Figure 10-7. Non-Decision Point (NDP) States




The following information refers to Figure 10-7.

**In State 0:**

- FD[7:0] is programmed to be tri-stated.
- CTL0 is programmed to be driven to a logic 1.



Exhibit 2033 - Page 144 of 346

**In State 1:**
- FD[7:0] is programmed to be driven.
- CTL0 is still programmed to be driven to a logic 1.

**In State 2:**
- FD[7:0] is programmed to be driven.
- CTL0 is programmed to be driven to a logic 0.

**In State 3:**
- FD[7:0] is programmed to be driven.
- CTL0 is still programmed to be driven to a logic 0.

**In State 4:**
- FD[7:0] is programmed to be driven.
- CTL0 is programmed to be driven to a logic 1.

**In State 5:**
- FD[7:0] is programmed to be tri-stated.
- CTL0 is still programmed to be driven to a logic 1.

**In State 6:**
- FD[7:0] is programmed to be tri-stated.
- CTL0 is still programmed to be driven to a logic 1.

Since all States in this example are coded as NDPs, the GPIF automatically branches from the last State (S6) to the Idle State (S7). This is the State in which the GPIF waits until the next GPIF waveform is triggered by the firmware.

States 2 and 3 in the example are identical, as are States 5 and 6. In a real application, these would probably be combined (there's no need to duplicate a State in order to 'stretch' it, since each NDP State can be assigned a duration in terms of IFCLK cycles). If fewer than 7 States were defined for this waveform, the Idle State would not automatically be entered after the last programmed State; that last programmed State's State Instruction would have to include an explicit unconditional branch to the Idle State.

### Decision Point (DP) States

Any State can be designated as a Decision Point (DP). A DP allows the GPIF engine to sample two signals — each of the 'two' can be the same signal, if desired — perform a boolean operation on the sampled values, then branch to other States (or loop back on itself, remaining in the current State) based on the result.

If a State Instruction includes a control task (advance the FIFO pointer, increment the GPIFADR address, and so on), that task is always executed once upon entering the State, regardless of whether the State is a DP or NDP. If the State is a DP that loops back on itself, however, it can be programmed to re-execute the control task on every loop.

With a Decision Point, the GPIF can perform simple tasks (wait until a RDY line is low before continuing to the next State, for instance). Decision point States can also perform more-complex tasks by branching to one State if the operation on the sampled signals results in a logic 1, or to a different State if it results in a logic 0.

In each State Instruction, the two signals to sample can be selected from any of the following:
- the six external RDY signals (RDY0-RDY5)
- one of the current FIFO's flags (PF, EF, FF)
- the INTRDY bit in the READY register
- a 'Transaction Count Expired' signal (which replaces RDY5)

The State Instruction also specifies a logic function (AND, OR, or XOR) to be applied to the two selected signals. If it's desired to act on the state of only one signal, the usual procedure is to select the same signal twice and specify the logic function as AND.

The State Instruction also specifies which State to branch to if the result of the logical expression is 0, and which State to branch to if the result of the logical expression is 1.

Below is an example waveform created using one Decision Point State (State 1); Non-Decision Point States are used for the rest of the waveform.

Figure 10-8. One Decision Point: Wait States Inserted Until RDY0 Goes Low

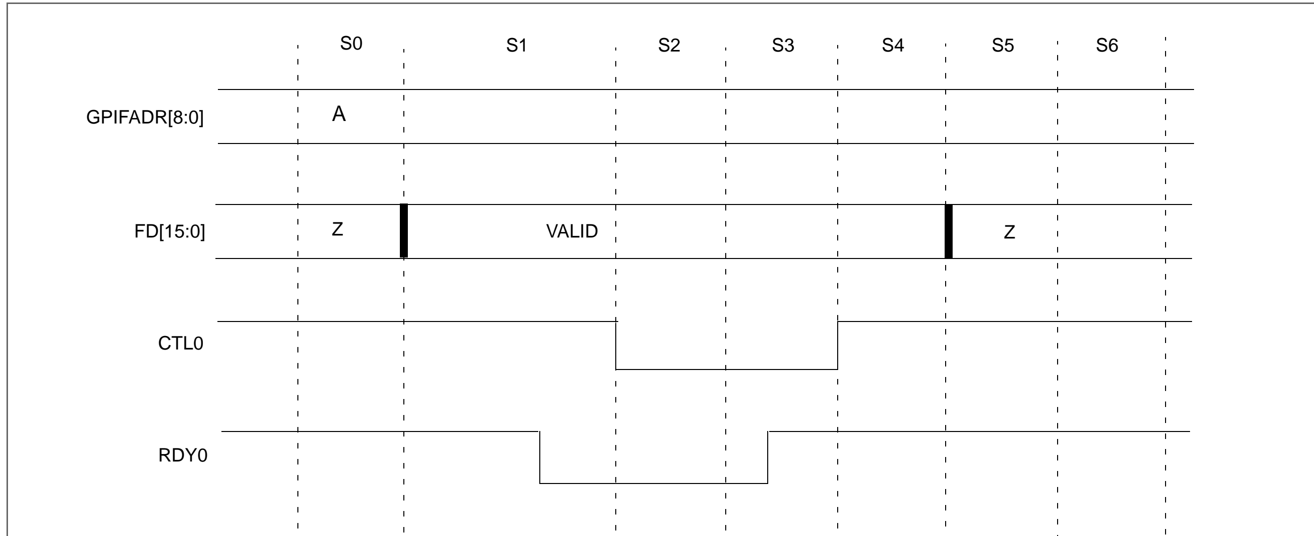
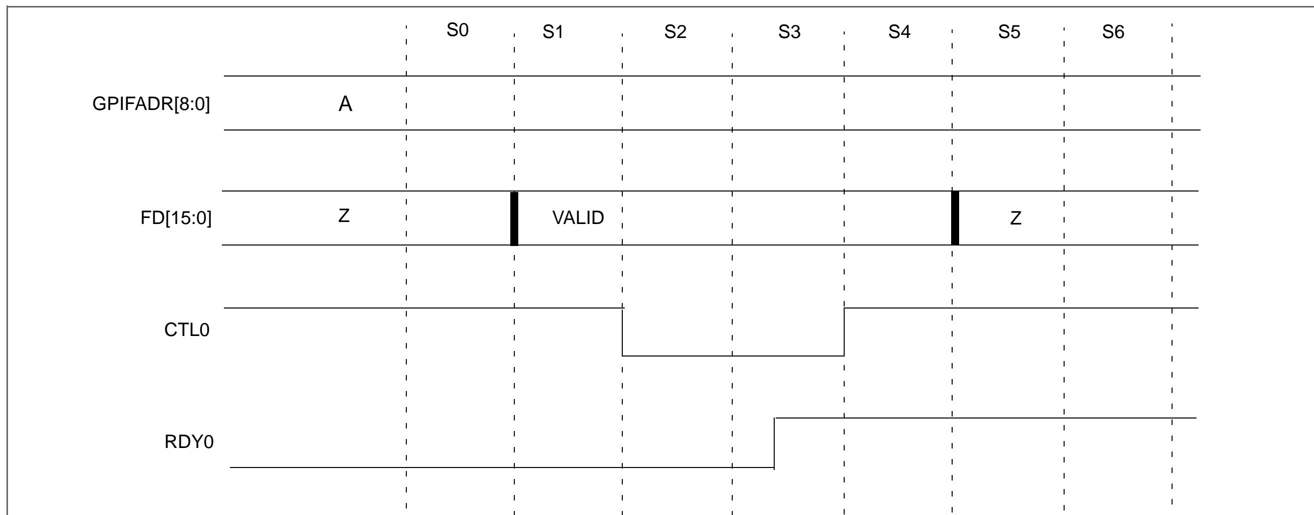



Figure 10-9. One Decision Point: No Wait States Inserted: RDY0 is Already Low at Decision Point I1




In Figure 10-8 and Figure 10-9, there is a single Decision Point defined as State 1. In this example, the input ready signal is assumed to be connected to RDY0, and the State Instruction for S1 is configured to branch to State 2 if RDY0 is a logic 0 or to branch to State 1 (for example, loop indefinitely) if RDY0 is a logic ‘1’.

In Figure 10-8, the GPIF remains in S1 until the RDY0 signal goes low, then branches to S2. Figure 10-9 illustrates the GPIF behavior when the RDY0 signal is *already* low when S1 is entered: The GPIF branches to S2.

**Note** Although it appears in Figure 10-9 that the GPIF branches **immediately** from State 1 to State 2, this is not exactly true. Even if RDY0 is already low before the GPIF enters State 1, the GPIF spends one IFCLK cycle in State 1 to evaluate the decision point. The logic function is applied on the rising edge of IFCLK entering State 1. If the logic function holds TRUE at this point, then the branch is effective on the next rising of IFCLK.



**Exhibit 2033 - Page 146 of 346**

### 10.3.3 Re-Executing a Task Within a DP State

In the simple DP examples shown earlier in this chapter, a control task (for example, output a word on FD[15:0] and increment GPIFADR[8:0]) executes only once at the start of a DP State, then the GPIF waits, sampling a RDYx input repeatedly until that input "informs" the GPIF to branch to the next State.

The GPIF also has the capability to re-execute the control task every time the RDYx input is sampled; this feature can be used to burst a large amount of data without passing through the Idle State (a waveform example is shown in Figure 10-10).

To re-execute a task within a decision point state, the 're-execute' bit for that decision point must be enabled. This is performed by checking the 'Loop (Re-Execute)' check-box within GPIF Designer (an example is shown in Figure 10-11). Figure 10-13 on page 148 shows an example of a GPIF waveform that uses a DP state which does not re-execute its control tasks.

Figure 10-10. Re-Executing a Task within a DP State

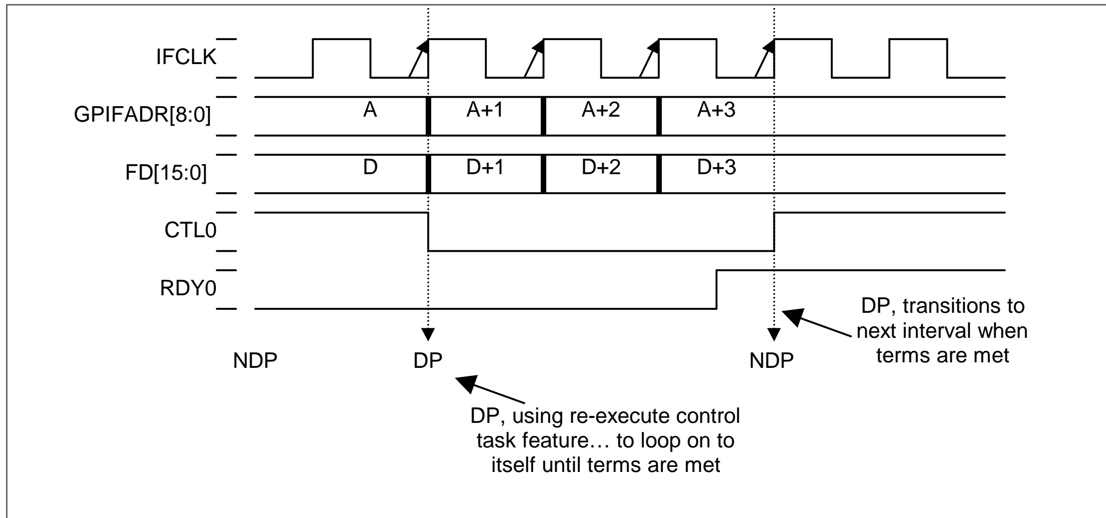



Figure 10-11. GPIF Designer Setup for the Waveform of Figure 10-10

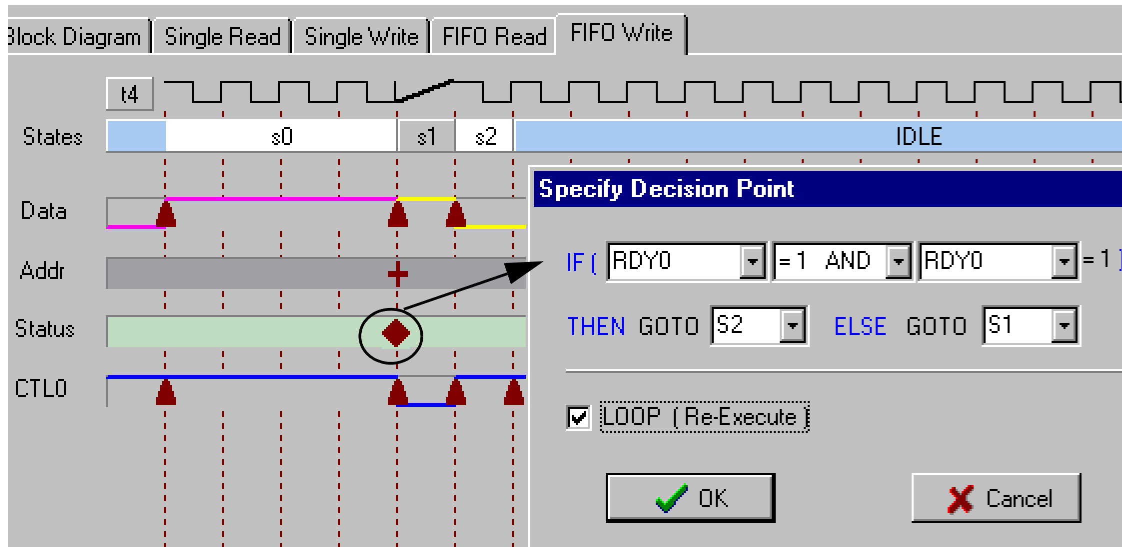





Exhibit 2033 - Page 147 of 346

Figure 10-12. GPIF Designer Output for the Waveform of Figure 10-10

| State | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| AddrMode | Same Val | Inc Val | Same Val | Same Val | Same Val | Same Val | Same Val | |
| DataMode | Activate | Activate | No Data | No Data | No Data | No Data | No Data | |
| NextData | SameData | NextData | SameData | SameData | SameData | SameData | SameData | |
| Int Trig | No Int | No Int | No Int | No Int | No Int | No Int | No Int | |
| IF/Wait | Wait 4 | IF | Wait 1 | Wait 1 | Wait 1 | Wait 1 | Wait 1 | |
| Term A | | RDY0 | | | | | | |
| LFUNC | | AND | | | | | | |
| Term B | | RDY0 | | | | | | |
| Branch1 | | Then 2 | | | | | | |
| Branch0 | | Else 1 | | | | | | |
| Re-execute | | Yes | | | | | | |
| CTL0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL5 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Figure 10-13. A DP State That Does NOT Re-Execute the Task

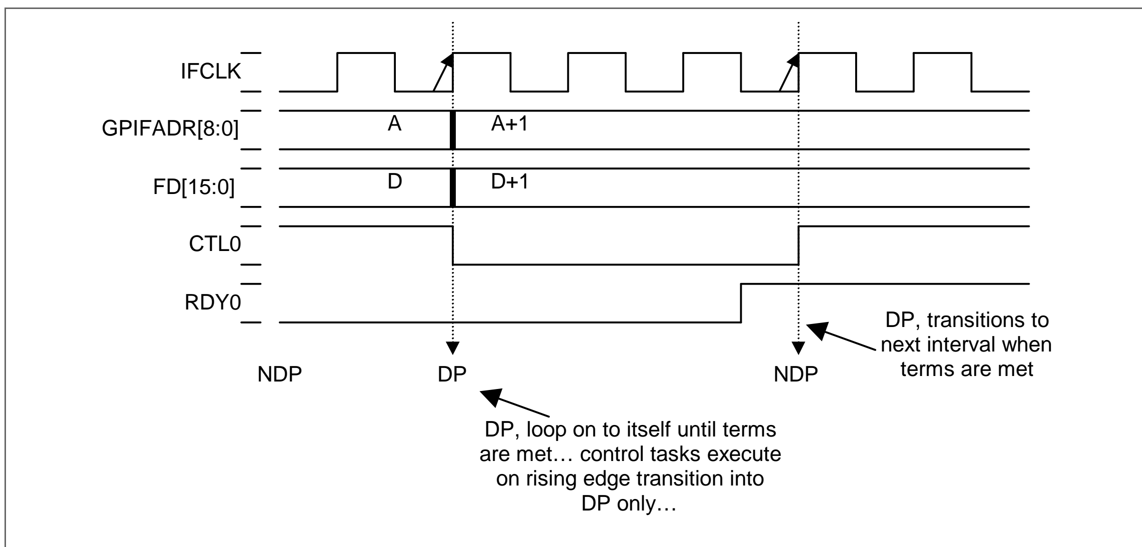





Exhibit 2033 - Page 148 of 346

Figure 10-14. GPIF Designer Setup for the Waveform of Figure 10-13

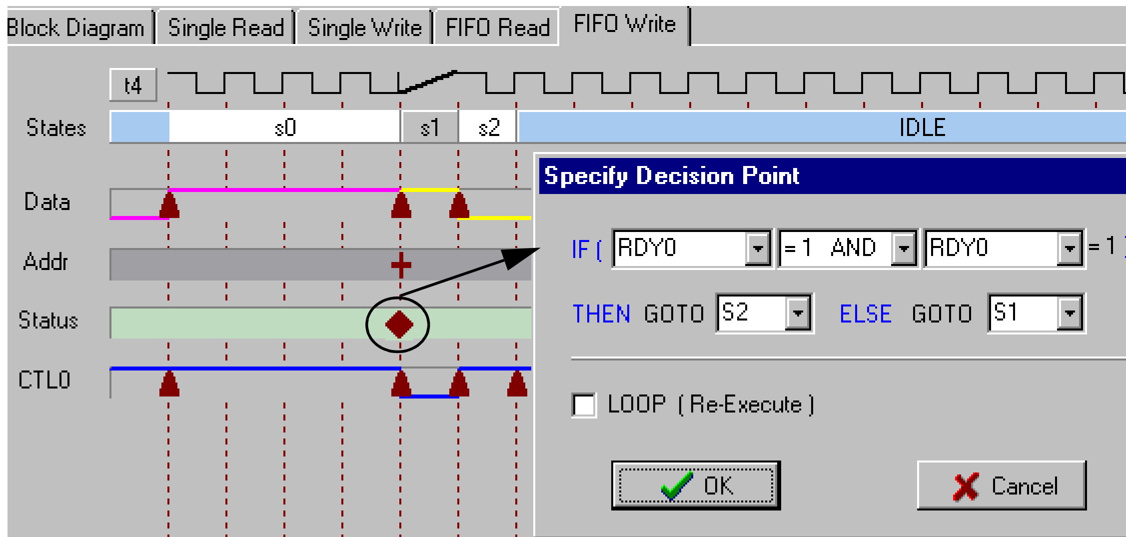



Figure 10-15. GPIF Designer Output for the Waveform of Figure 10-13

| State | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| AddrMode | Same Val | Inc Val | Same Val | Same Val | Same Val | Same Val | Same Val | |
| DataMode | Activate | Activate | No Data | No Data | No Data | No Data | No Data | |
| NextData | SameData | NextData | SameData | SameData | SameData | SameData | SameData | |
| Int Trig | No Int | No Int | No Int | No Int | No Int | No Int | No Int | |
| IF/Wait | Wait 4 | IF | Wait 1 | Wait 1 | Wait 1 | Wait 1 | Wait 1 | |
| Term A | | RDY0 | | | | | | |
| LFUNC | | AND | | | | | | |
| Term B | | RDY0 | | | | | | |
| Branch1 | | Then 2 | | | | | | |
| Branch0 | | Else 1 | | | | | | |
| Re-execute | | No | | | | | | |
| CTL0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL5 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |



Exhibit 2033 - Page 149 of 346

### 10.3.4 State Instructions

Each State's characteristics are defined by a 4-byte State Instruction. The four bytes are named *LENGTH / BRANCH*, *OPCODE*, *LOGIC FUNCTION*, and *OUTPUT*.

Note that the State Instructions are interpreted differently for Decision Points (DP = 1) and Non-Decision Points (DP = 0).

#### Non-Decision Point State Instruction (DP = 0)

**LENGTH / BRANCH**

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| Number of IFCLK cycles to stay in this State (0 = 256 cycles) | | | | | | | |

**OPCODE**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| x | x | SGL | GINT | INCAD | NEXT/ SGLCRC | DATA | DP = 0 |

**LOGIC FUNCTION**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| Not Used | | | | | | | |

**OUTPUT (if TRICTL Bit = 1)**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| OE3 | OE2 | OE1 | OE0 | CTL3 | CTL2 | CTL1 | CTL0 |

**OUTPUT (if TRICTL Bit = 0)**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| x | x | CTL5 | CTL4 | CTL3 | CTL2 | CTL1 | CTL0 |



Exhibit 2033 - Page 150 of 346

### Decision Point State Instruction (DP = 1)

**LENGTH / BRANCH**

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| Re-Execute | x | BRANCHON1 | | | BRANCHON0 | | |

**OPCODE**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| x | x | SGL | GINT | INCAD | NEXT/SGLCRC | DATA | DP = 1 |

**LOGIC FUNCTION**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| LFUNC | | TERMA | | | TERMB | | |

**OUTPUT (if TRICTL Bit = 1)**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| OE3 | OE2 | OE1 | OE0 | CTL3 | CTL2 | CTL1 | CTL0 |

**OUTPUT (if TRICTL Bit = 0)**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| x | x | CTL5 | CTL4 | CTL3 | CTL2 | CTL1 | CTL0 |

**LENGTH / BRANCH Register**: This register's interpretation depends on the DP bit:

- For DP = 0 (Non-Decision Point), this is a LENGTH field; it holds the fixed duration of this State in IFCLK cycles. A value of 0 is interpreted as 256 IFCLK cycles.
- For DP = 1 (Decision Point), this is a BRANCH field; it specifies the State to which the GPIF will branch:

  **BRANCHON1**: Specifies the State to which the GPIF will branch if the logic expression evaluates to '1'.

  **BRANCHON0**: Specifies the State to which the GPIF will branch if the logic expression evaluates to '0'.

  **Re-Execute**: Setting this bit allows the DP to re-execute its control tasks.

**OPCODE Register**: This register sets a number of State characteristics.

**SGL Bit**: has no effect in a Single-Read or Single-Write waveform. In a FIFO waveform, it specifies whether a single-data transaction should occur (from/to the SGLDATH:L or UDMACRCH:L registers), even in a FIFO-Write or FIFO-Read transaction. See also 'NEXT/SGLCRC', below.

1 = Use SGLDATH:L or UDMACRCH:L.

0 = Use the FIFO.

**GINT Bit**: specifies whether to generate a GPIFWF interrupt during this State.

1 = Generate GPIFWF interrupt (on INT4) when this State is reached.

0 = Do not generate interrupt.

**INCAD Bit**: specifies whether to increment the GPIF Address lines GPIFADR[8:0].

1 = Increment the GPIFADR[8:0] bus at the beginning of this State.

0 = Do not increment the GPIFADR[8:0] signals.

**NEXT/SGLCRC Bit**:

If SGL = 0, specifies whether the FIFO should be advanced at the start of this State.

1 = Move the next data in the OUT FIFO to the top.

0 = Do not advance the FIFO.

The NEXT bit has no effect when the waveform is applied to an IN FIFO.

If SGL = 1, specifies whether data should be transferred to/from SGLDATH:L or UDMACRCH:L. See also ‘SGL Bit’, above.

1 = Use UDMACRCH:L.

0 = Use SGLDATH:L.

**DATA Bit**: specifies whether the FIFO Data bus is to be driven, tri-stated, or sampled.

During a write:

1 = Drive the FIFO Data bus with the output data.

0 = Tri-state (do not drive the bus).

During a read:

1 = Sample the FIFO Data bus and store the data.

0 = Do not sample the data bus.

**DP Bit**: indicates whether the State is a DP or NDP:

1 = Decision Point.

0 = Non-Decision Point.

**LOGIC FUNCTION Register**: This register is used only in DP State Instructions. It specifies the inputs (TERMA and TERMB) and the Logic Function (LFUNC) to apply to those inputs. The result of the logic function determines the State to which the GPIF will branch (see also ‘LENGTH /BRANCH Register’, above).

**TERMA** and **TERMB bits**:

= 000: RDY0

= 001: RDY1

= 010: RDY2

= 011: RDY3

= 100: RDY4

= 101: RDY5 (or Transaction-Count Expiration, if GPIFREADYCFG.5 = 1)

= 110: FIFO flag (PF, EF, or FF), preselected via EPxGPIFFLGSEL

= 111: INTRDY (Bit 7 of the GPIFREADYCFG register)

**LFUNC bits**:

= 00: A AND B

= 01: A OR B

= 10: A XOR B

= 11: $\overline{A}$ AND B

The TERMA and TERMB inputs are sampled at each rising edge of IFCLK. The logic function is applied, then the branch is taken on the next rising edge.

This register is meaningful only for DP Instructions; when the DP bit of the OPCODE register is cleared to 0, the contents of this register are ignored.



**Exhibit 2033 - Page 152 of 346**

**OUTPUT Register**: This register controls the state of the six Control outputs (CTL5:0) during the entire State defined by this State Instruction.

**OEx Bit**: If TRICTL = 1, specifies whether the corresponding CTLx output signal is tri-stated.

1 = Drive CTLx

0 = Tri-state CTLx

**CTLx Bit**: specifies the state to set each CTLx signal to during this entire State.

1 = High level
If the CTLx bit in the GPIFCTLCFG register is set to '1', the output driver will be an open-drain.
If the CTLx bit in the GPIFCTLCFG register is set to '0', the output driver will be driven to CMOS levels.

0 = Low level

### *10.3.4.1 Structure of the Waveform Descriptors*

Up to four different waveforms can be defined. Each Waveform Descriptor comprises up to 7 State Instructions which are loaded into the Waveform Registers as defined in this section.

Table 10-6. Waveform Descriptor Addresses

| Waveform Descriptor | Base XDATA Address |
|---|---|
| 0 | 0xE400 |
| 1 | 0xE420 |
| 2 | 0xE440 |
| 3 | 0xE460 |

Within each Waveform Descriptor, the State Instructions are packed as described in Table 10-7. Waveform Descriptor 0 is shown as an example. The other Waveform Descriptors follow exactly the same structure but at higher XDATA addresses.

Table 10-7. Waveform Descriptor 0 Structure

| XDATA Address | Contents |
|---|---|
| 0xE400 | LENGTH / BRANCH [0] (LENGTH / BRANCH field of State 0 of Waveform Program 0) |
| 0xE401 | LENGTH / BRANCH [1] (LENGTH / BRANCH field of State 1 of Waveform Program 0) |
| 0xE402 | LENGTH / BRANCH [2] (LENGTH / BRANCH field of State 2 of Waveform Program 0) |
| 0xE403 | LENGTH / BRANCH [3] (LENGTH / BRANCH field of State 3 of Waveform Program 0) |
| 0xE404 | LENGTH / BRANCH [4] (LENGTH / BRANCH field of State 4 of Waveform Program 0) |
| 0xE405 | LENGTH / BRANCH [5] (LENGTH / BRANCH field of State 5 of Waveform Program 0) |
| 0xE406 | LENGTH / BRANCH [6] (LENGTH / BRANCH field of State 6 of Waveform Program 0) |
| 0xE407 | Reserved |
| 0xE408 | OPCODE[0] (OPCODE field of State 0 of Waveform Program 0) |
| 0xE409 | OPCODE[1] (OPCODE field of State 1 of Waveform Program 0) |
| 0xE40A | OPCODE[2] (OPCODE field of State 2 of Waveform Program 0) |
| 0xE40B | OPCODE[3] (OPCODE field of State 3 of Waveform Program 0) |
| 0xE40C | OPCODE[4] (OPCODE field of State 4 of Waveform Program 0) |
| 0xE40D | OPCODE[5] (OPCODE field of State 5 of Waveform Program 0) |
| 0xE40E | OPCODE[6] (OPCODE field of State 6 of Waveform Program 0) |
| 0xE40F | Reserved |
| 0xE410 | OUTPUT[0] (OUTPUT field of State 0 of Waveform Program 0) |
| 0xE411 | OUTPUT[1] (OUTPUT field of State 1 of Waveform Program 0) |
| 0xE412 | OUTPUT[2] (OUTPUT field of State 2 of Waveform Program 0) |
| 0xE413 | OUTPUT[3] (OUTPUT field of State 3 of Waveform Program 0) |
| 0xE414 | OUTPUT[4] (OUTPUT field of State 4 of Waveform Program 0) |
| 0xE415 | OUTPUT[5] (OUTPUT field of State 5 of Waveform Program 0) |
| 0xE416 | OUTPUT[6] (OUTPUT field of State 6 of Waveform Program 0) |



Exhibit 2033 - Page 153 of 346

Table 10-7. Waveform Descriptor 0 Structure *(continued)*

| XDATA Address | Contents |
|---|---|
| 0xE417 | Reserved |
| 0xE418 | LOGIC FUNCTION[0] (LOGIC FUNCTION field of State 0 of Waveform Program 0) |
| 0xE419 | LOGIC FUNCTION[1] (LOGIC FUNCTION field of State 1 of Waveform Program 0) |
| 0xE41A | LOGIC FUNCTION[2] (LOGIC FUNCTION field of State 2 of Waveform Program 0) |
| 0xE41B | LOGIC FUNCTION[3] (LOGIC FUNCTION field of State 3 of Waveform Program 0) |
| 0xE41C | LOGIC FUNCTION[4] (LOGIC FUNCTION field of State 4 of Waveform Program 0) |
| 0xE41D | LOGIC FUNCTION[5] (LOGIC FUNCTION field of State 5 of Waveform Program 0) |
| 0xE41E | LOGIC FUNCTION[6] (LOGIC FUNCTION field of State 6 of Waveform Program 0) |
| 0xE41F | Reserved |

#### *10.3.4.2 Terminating a GPIF Transfer*

Once a GPIF transfer is initiated, the ONLY way to terminate the transfer is to either:

- have it terminate naturally when the byte count expires or
- have the 8051 terminate and abort the transfer by writing to the GPIFABORT register.

Once a GPIF transfer is triggered, it will not terminate until the Transaction Count (TC) has expired. The GPIF engine checks the state of the TC only when in IDLE state. While designing a GPIF waveform, you must have the waveform pass through an IDLE state in order for the GPIF to check the TC and finally terminate when TC has expired.

GPIF does allow you to save time and avoid going through the IDLE state by using the 'Transaction Count Expired' (TCxpire) signal. This TCxpire replaces RDY5, if GPIFREADYCFG.5 = 1. Section 10.4.3.2 Reading the Transaction-Count Status in a DP State on page 170 provides further information on this.



Exhibit 2033 - Page 154 of 346

## 10.4 Firmware

Table 10-8. Registers Associated with GPIF Firmware

| | |
|---|---|
| GPIFTRIG (SFR) | EPxCFG |
| GPIFSGLDATH (SFR) | EPxFIFOCFG |
| GPIFSGLDATLX (SFR) | EPxAUTOINLENH/L |
| GPIFSGLDATLNOX (SFR) | EPxFIFOPFH/L |
| EPxGPIFTRIG | EP2468STAT(SFR) |
| XGPIFSGLDATH | EP24FIFOFLGS(SFR) |
| XGPIFSGLDATLX | EP68FIFOFLGS(SFR) |
| XGPIFSGLDATLNOX | EPxCS |
| GPIFABORT | EPxFIFOFLGS |
| GPIFIE | |
| GPIFIRQ | EPxFIFOIE |
| GPIFTCB3 | EPxFIFOIRQ |
| GPIFTCB2 | INT2IVEC |
| GPIFTCB1 | INT4IVEC |
| GPIFTCB0 | INTSETUP |
| | IE (SFR) |
| EPxBCH/L | IP (SFR) |
| EPxFIFOBCH/L | INT2CLR(SFR) |
| EPxFIFOBUF | INT4CLR(SFR) |
| INPKTEND/OUTPKTEND | EIE (SFR) |
| | EXIF (SFR) |

The '*x'* in these register names represents 2, 4, 6, or 8; endpoints 0 and 1 are not associated with the Slave FIFOs.

The GPIF Designer utility, distributed with the Cypress MoBL-USB FX2LP18 Development Kit, generates C code which may be linked with the rest of an application's source code. Except for GpifInit(), the *GPIF Designer* output source file does not include the following basic GPIF framework and functions:



**Exhibit 2033 - Page 155 of 346**

```
TD_Init():
    … … … … …
    GpifInit();  // Configures GPIF from GPIF Designer generated waveform data

    // TODO: configure other endpoints, etc. here

    // TODO: arm OUT buffer(s) here

    // setup INT4 as internal source for GPIF interrupts
    // using INT4CLR (SFR), automatically enabled
    // INTSETUP |= 0x03;  //Enable INT4 Autovectoring
    // SYNCDELAY;
    // GPIFIE = 0x03;  // Enable GPIFDONE and GPIFWF interrupt(s)
    // SYNCDELAY;
    // EIE |= 0x04;  // Enable INT4 ISR, EIE.2(EIEX4)=1

    // TODO: configure GPIF interrupt(s) to meet your needs here
       … … … … …

void GpifInit( void )
{
  BYTE i;

  // Registers which require a synchronization delay, see section 15.14
  // FIFORESET        FIFOPINPOLAR
  // INPKTEND         OUTPKTEND
  // EPxBCH:L         REVCTL
  // GPIFTCB3         GPIFTCB2
  // GPIFTCB1         GPIFTCB0
  // EPxFIFOPFH:L     EPxAUTOINLENH:L
  // EPxFIFOCFG       EPxGPIFFLGSEL
  // PINFLAGSxx       EPxFIFOIRQ
  // EPxFIFOIE        GPIFIRQ
  // GPIFIE           GPIFADRH:L
  // UDMACRCH:L       EPxGPIFTRIG
  // GPIFTRIG


  // 8051 doesn't have access to waveform memories 'til
  // the part is in GPIF mode.

  IFCONFIG = 0xCE;
  // IFCLKSRC=1   , FIFO's executes on internal clk source
  // xMHz=1       , 48MHz internal clk rate
  // IFCLKOE=0    , Don't drive IFCLK pin signal at 48MHz
  // IFCLKPOL=0   , Don't invert IFCLK pin signal from internal clk
  // ASYNC=1      , master samples asynchronous
  // GSTATE=1     , Drive GPIF states out on PORTE[2:0], debug WF
  // IFCFG[1:0]=10, FX2LP18 in GPIF master mode

  GPIFABORT = 0xFF;  // abort any waveforms pending

  GPIFREADYCFG = InitData[ 0 ];
  GPIFCTLCFG = InitData[ 1 ];
  GPIFIDLECS = InitData[ 2 ];
  GPIFIDLECTL = InitData[ 3 ];
  GPIFWFSELECT = InitData[ 5 ];
  GPIFREADYSTAT = InitData[ 6 ];
```



**Exhibit 2033 - Page 156 of 346**

```
    // use dual autopointer feature...
    AUTOPTRSETUP = 0x07;          // inc both pointers

    // source
    APTR1H = MSB( &WaveData );
    APTR1L = LSB( &WaveData );

    // destination
    AUTOPTRH2 = 0xE4;
    AUTOPTRL2 = 0x00;

    // transfer
    for ( i = 0x00; i < 128; i++ )
    {
      EXTAUTODAT2 = EXTAUTODAT1;
    }

  // Configure GPIF Address pins, output initial value,
    PORTCCFG = 0xFF;    // [7:0] as alt. func. GPIFADR[7:0]
    OEC = 0xFF;         // and as outputs
    PORTECFG |= 0x80;   // [8] as alt. func. GPIFADR[8]
    OEE |= 0x80;        // and as output

  // ...OR... tri-state GPIFADR[8:0] pins
  //  PORTCCFG = 0x00;  // [7:0] as port I/O
  //  OEC = 0x00;       // and as inputs
  //  PORTECFG &= 0x7F; // [8] as port I/O
  //  OEE &= 0x7F;      // and as input

  // GPIF address pins update when GPIFADRH/L written
    SYNCDELAY;                    //
    GPIFADRH = 0x00;    // bits[7:1] always 0
    SYNCDELAY;                    //
    GPIFADRL = 0x00;    // point to PERIPHERAL address 0x0000

  // Configure GPIF FlowStates registers for Wave 0 of WaveData
    FLOWSTATE = FlowStates[ 0 ];
    FLOWLOGIC = FlowStates[ 1 ];
    FLOWEQ0CTL = FlowStates[ 2 ];
    FLOWEQ1CTL = FlowStates[ 3 ];
    FLOWHOLDOFF = FlowStates[ 4 ];
    FLOWSTB = FlowStates[ 5 ];
    FLOWSTBEDGE = FlowStates[ 6 ];
    FLOWSTBHPERIOD = FlowStates[ 7 ];
  }

  // Set Address GPIFADR[8:0] to PERIPHERAL
  void Peripheral_SetAddress( WORD gaddr )
  {
    SYNCDELAY;                    //
    GPIFADRH = gaddr >> 8;
    SYNCDELAY;                    //
    GPIFADRL = ( BYTE )gaddr; // setup GPIF address
  }
```

```
// Set GPIF Transaction Count
void Peripheral_SetGPIFTC( WORD xfrcnt )
{
  SYNCDELAY;                    //
  GPIFTCB1 = xfrcnt >> 8;  // setup transaction count
  SYNCDELAY;                    //
  GPIFTCB0 = ( BYTE )xfrcnt;
}

#define GPIF_FLGSELPF 0
#define GPIF_FLGSELEF 1
#define GPIF_FLGSELFF 2

// Set EP2GPIF Decision Point FIFO Flag Select (PF, EF, FF)
void SetEP2GPIFFLGSEL( WORD DP_FIFOFlag )
{
  EP2GPIFFLGSEL = DP_FIFOFlag;
}

// Set EP4GPIF Decision Point FIFO Flag Select (PF, EF, FF)
void SetEP4GPIFFLGSEL( WORD DP_FIFOFlag )
{
  EP4GPIFFLGSEL = DP_FIFOFlag;
}

// Set EP6GPIF Decision Point FIFO Flag Select (PF, EF, FF)
void SetEP6GPIFFLGSEL( WORD DP_FIFOFlag )
{
  EP6GPIFFLGSEL = DP_FIFOFlag;
}

// Set EP8GPIF Decision Point FIFO Flag Select (PF, EF, FF)
void SetEP8GPIFFLGSEL( WORD DP_FIFOFlag )
{
  EP8GPIFFLGSEL = DP_FIFOFlag;
}

// Set EP2GPIF Programmable Flag STOP, overrides Transaction Count
void SetEP2GPIFPFSTOP( void )
{
  EP2GPIFPFSTOP = 0x01;
}

// Set EP4GPIF Programmable Flag STOP, overrides Transaction Count
void SetEP4GPIFPFSTOP( void )
{
  EP4GPIFPFSTOP = 0x01;
}

// Set EP6GPIF Programmable Flag STOP, overrides Transaction Count
void SetEP6GPIFPFSTOP( void )
{
  EP6GPIFPFSTOP = 0x01;
}

// Set EP8GPIF Programmable Flag STOP, overrides Transaction Count
void SetEP8GPIFPFSTOP( void )
{
  EP8GPIFPFSTOP = 0x01;
}
```



Exhibit 2033 - Page 158 of 346

```
// write single byte to PERIPHERAL, using GPIF
void Peripheral_SingleByteWrite( BYTE gdata )
{
  while( !( GPIFTRIG & 0x80 ) ) // poll GPIFTRIG.7 Done bit
  {
     ;
  }

  XGPIFSGLDATLX = gdata;        // trigger GPIF
                                // ...single byte write transaction
}

// write single word to PERIPHERAL, using GPIF
void Peripheral_SingleWordWrite( WORD gdata )
{
  while( !( GPIFTRIG & 0x80 ) )  // poll GPIFTRIG.7 Done bit
  {
    ;
  }

// using register(s) in XDATA space
  XGPIFSGLDATH = gdata >> 8;
  XGPIFSGLDATLX = gdata;        // trigger GPIF
                                // ...single word write transaction
}

// read single byte from PERIPHERAL, using GPIF
void Peripheral_SingleByteRead( BYTE xdata *gdata )
{
  static BYTE g_data = 0x00;

  while( !( GPIFTRIG & 0x80 ) )  // poll GPIFTRIG.7 Done bit
  {
     ;
  }

// using register(s) in XDATA space, dummy read
  g_data = XGPIFSGLDATLX;       // trigger GPIF
                                // ...single byte read transaction
  while( !( GPIFTRIG & 0x80 ) ) // poll GPIFTRIG.7 Done bit
  {
     ;
  }

// using register(s) in XDATA space,
  *gdata = XGPIFSGLDATLNOX;     // ...GPIF reads byte from PERIPHERAL
}

// read single word from PERIPHERAL, using GPIF
void Peripheral_SingleWordRead( WORD xdata *gdata )
{
  BYTE g_data = 0x00;

  while( !( GPIFTRIG & 0x80 ) ) // poll GPIFTRIG.7 Done bit
  {
     ;
  }

// using register(s) in XDATA space, dummy read
  g_data = XGPIFSGLDATLX;       // trigger GPIF
                                // ...single word read transaction

  while( !( GPIFTRIG & 0x80 ) ) // poll GPIFTRIG.7 Done bit
  {
     ;
  }
```



Exhibit 2033 - Page 159 of 346

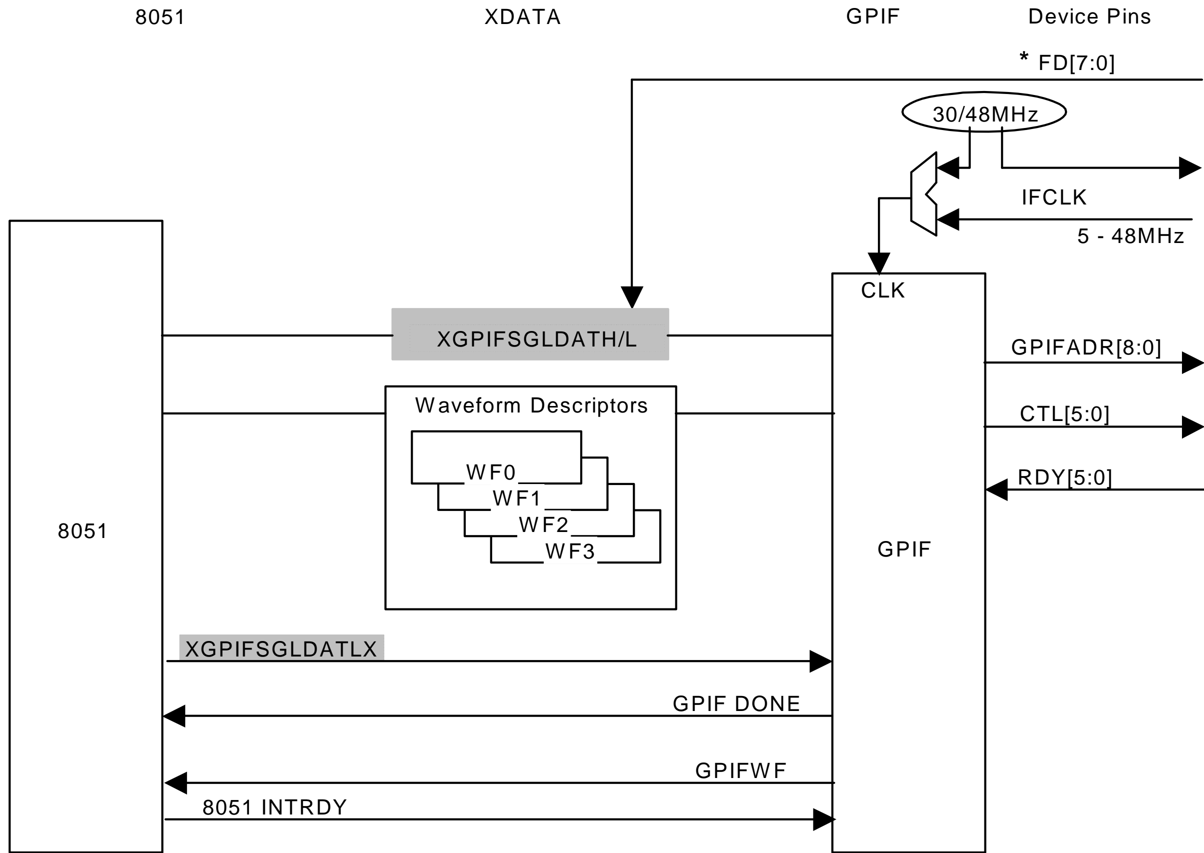
```
// using register(s) in XDATA space, GPIF reads word from PERIPHERAL
  *gdata = ( ( WORD )XGPIFSGLDATH << 8 ) | ( WORD )XGPIFSGLDATLNOX;
}

#define GPIFTRIGWR 0
#define GPIFTRIGRD 4

#define GPIF_EP2 0
#define GPIF_EP4 1
#define GPIF_EP6 2
#define GPIF_EP8 3

// write byte(s)/word(s) to PERIPHERAL, using GPIF and EPxFIFO
// if EPx WORDWIDE=0 then write byte(s)
// if EPx WORDWIDE=1 then write word(s)
void Peripheral_FIFOWrite( BYTE FIFO_EpNum )
{
  while( !( GPIFTRIG & 0x80 ) ) // poll GPIFTRIG.7 Done bit
  {
     ;
  }

  // trigger FIFO write transaction(s), using SFR
  GPIFTRIG = FIFO_EpNum;  // R/W=0, EP[1:0]=FIFO_EpNum for EPx write(s)
}

// read byte(s)/word(s) from PERIPHERAL, using GPIF and EPxFIFO
// if EPx WORDWIDE=0 then read byte(s)
// if EPx WORDWIDE=1 then read word(s)

void Peripheral_FIFORead( BYTE FIFO_EpNum )
{
  while( !( GPIFTRIG & 0x80 ) ) // poll GPIFTRIG.7 GPIF Done bit
  {
     ;
  }

  // trigger FIFO read transaction(s), using SFR
  GPIFTRIG = GPIFTRIGRD | FIFO_EpNum; // R/W=1, EP[1:0]=FIFO_EpNum for EPx read(s)
}
```



Exhibit 2033 - Page 160 of 346

### 10.4.1 Single-Read Transactions

Figure 10-16. Firmware Launches a Single-Read Waveform, WORDWIDE=0




* All EPx WORDWIDE bits must be cleared to '0' for 8-bit single transactions. If any of the EPx WORDWIDE bits are set to '1', then single transactions will be 16 bits wide.



Exhibit 2033 - Page 161 of 346

Figure 10-17. Single-Read Transaction Waveform

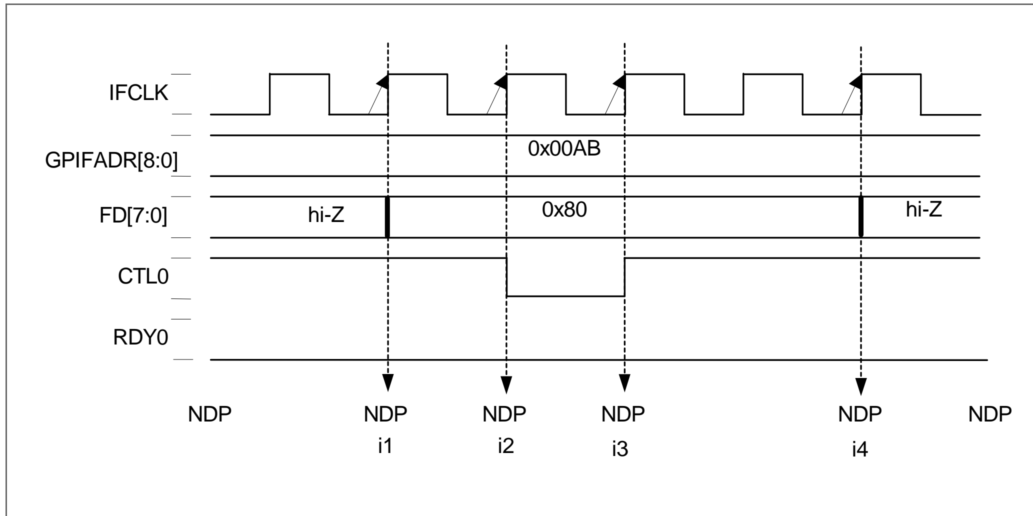



Figure 10-18. GPIF Designer Setup for the Waveform of Figure 10-17

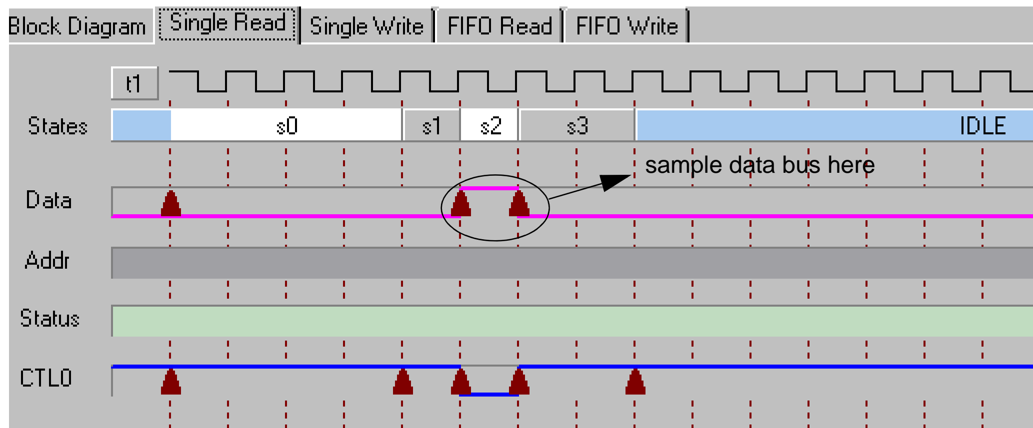


Figure 10-19. GPIF Designer Output for the Waveform of Figure 10-17

| State | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| AddrMode | Same Val | Same Val | Same Val | Same Val | Same Val | Same Val | Same Val | |
| DataMode | No Data | No Data | Activate | No Data | No Data | No Data | No Data | |
| NextData | SameData | SameData | SameData | SameData | SameData | SameData | SameData | |
| Int Trig | No Int | No Int | No Int | No Int | No Int | No Int | No Int | |
| IF/Wait | Wait 4 | Wait 1 | Wait 1 | Wait 2 | Wait 1 | Wait 1 | Wait 1 | |
| Term A | | | | | | | | |
| LFUNC | | | | | | | | |
| Term B | | | | | | | | |
| Branch1 | | | | | | | | |
| Branch0 | | | | | | | | |
| Re-execute | | | | | | | | |
| CTL0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| CTL1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL5 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

To perform a Single-Read transaction:

1. Initialize the GPIF Configuration Registers and Waveform Descriptors.
2. Check that the GPIF is IDLE by checking if the DONE bit (GPIFIDLECS.7 or GPIFTRIG.7) is set.
3. Perform a dummy **read** of the XGPIFSGLDATLX register to start a single transaction.
4. Wait for the GPIF to indicate that the transaction is complete. When the transaction is complete, the DONE bit (GPI-FIDLECS.7 or GPIFTRIG.7) will be set to '1'. If enabled, a GPIFDONE interrupt will also be generated.
5. Depending on the bus width and the desire to start another transaction, the data read by the GPIF can be retrieved from the XGPIFSGLDATH, XGPIFSGLDATLX, and/or the XGPIFSGLDATLNOX register (or from the SFR-space copies of these registers):

   In 16-bit mode **only**, the most significant byte, FD[15:8], of data is read from the XGPIFSGLDATH register.

   In 8- and 16-bit modes, the least significant byte of data is read by either:

   - reading XGPIFSGLDATLX, which reads the least significant byte and starts another Single-Read transaction.
   - reading XGPIFSGLDATLNOX, which reads the least significant byte but does **not** start another Single-Read transaction.

   The following C program fragments (Figure 10-20 on page 164 and Figure 10-21 on page 165) illustrate how to perform a Single-Read transaction in 8-bit mode (WORDWIDE=0):



Exhibit 2033 - Page 163 of 346

Figure 10-20. Single-Read Transaction Functions

```
#define PERIPHCS 0x00AB
#define AOKAY 0x80
#define BURSTMODE 0x0000
#define TRISTATE 0xFFFF
#define EVER ;;

// prototypes
void GpifInit( void );

// Set Address GPIFADR[8:0] to PERIPHERAL
void Peripheral_SetAddress( WORD gaddr )
{
  if( gaddr < 512 )
  { // drive GPIF address bus w/gaddr
    GPIFADRH = gaddr >> 8;
    SYNCDELAY;
    GPIFADRL = ( BYTE )gaddr; // setup GPIF address
  }
  else
  { // tri-state GPIFADR[8:0] pins
    PORTCCFG = 0x00;  // [7:0] as port I/O
    OEC = 0x00;  // and as inputs
    PORTECFG &= 0x7F;  // [8] as port I/O
     OEE &= 0x7F;  // and as input
  }
}

// read single byte from PERIPHERAL, using GPIF
void Peripheral_SingleByteRead( BYTE xdata *gdata )
{
  static BYTE g_data = 0x00;

  while( !( GPIFTRIG & 0x80 ) )  // poll GPIFTRIG.7 Done bit
  {
     ;
  }

  // using register(s) in XDATA space, dummy read
  g_data = XGPIFSGLDATLX;  // to trigger GPIF single byte read transaction

  while( !( GPIFTRIG & 0x80 ) )  // poll GPIFTRIG.7 Done bit
  {
     ;
  }

  // using register(s) in XDATA space, GPIF read byte from PERIPHERAL here
  *gdata = XGPIFSGLDATLNOX;
}
```



Exhibit 2033 - Page 164 of 346

Figure 10-21. Initialization Code for Single-Read Transactions

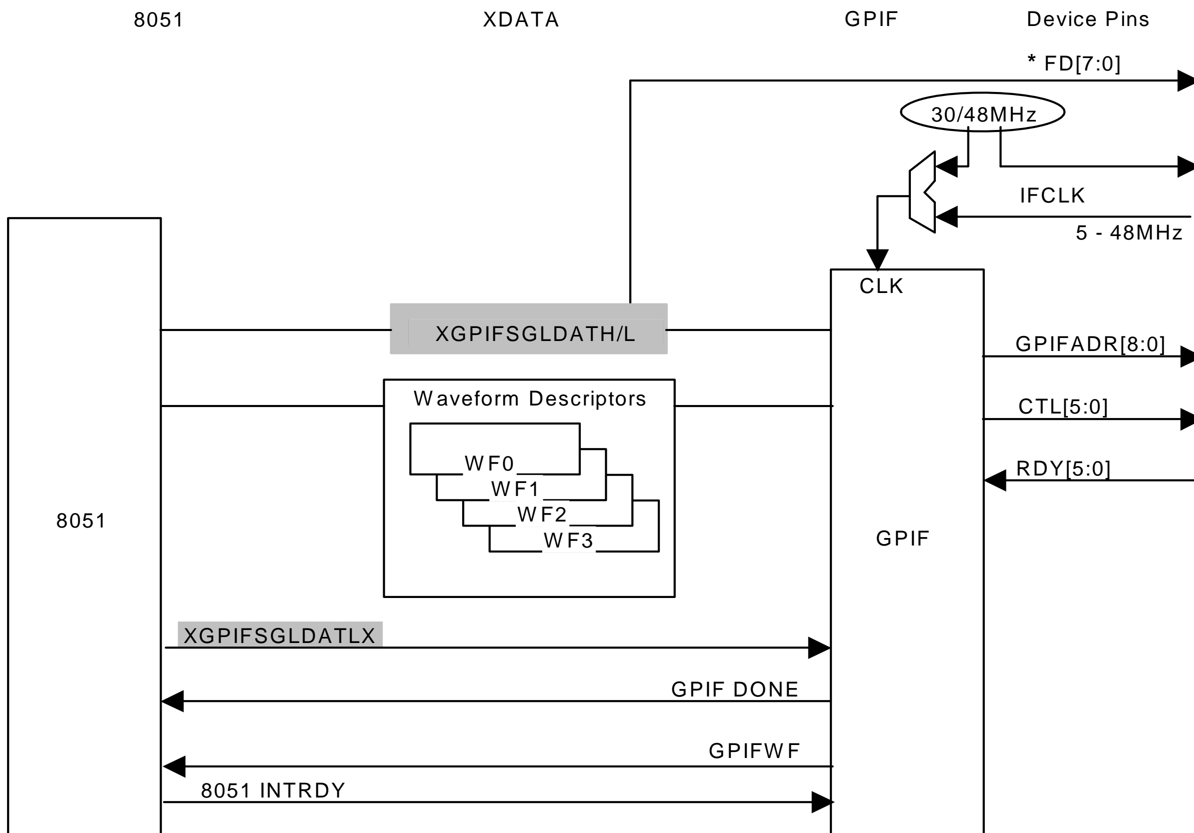
```
void TD_Init( void )
{
  BYTE xdata periph_status;

  … … … … …
  GpifInit();  // Configures GPIF from GPIF Designer generated waveform data

  // TODO: configure other endpoints, etc. here

  // TODO: arm OUT buffer(s) here

  // setup INT4 as internal source for GPIF interrupts
  // using INT4CLR (SFR), automatically enabled
  // INTSETUP |= 0x03;  //Enable INT4 Autovectoring
  // SYNCDELAY;
  // GPIFIE = 0x03;  // Enable GPIFDONE and GPIFWF interrupt(s)
  // SYNCDELAY;
  // EIE |= 0x04;  // Enable INT4 ISR, EIE.2(EIEX4)=1

  // TODO: configure GPIF interrupt(s) to meet your needs here
     … … … … …

  // get status of peripheral function
  Peripheral_SetAddress( PERIPHCS );
  Peripheral_SingleByteRead( &periph_status );

  if( periph_status == AOKAY )
  { // set it and forget it
    Peripheral_SetAddress( BURSTMODE );
  }
  else
  {
    Peripheral_SetAddress( TRISTATE );
  }
}
```



Exhibit 2033 - Page 165 of 346

## 10.4.2 Single-Write Transactions

Figure 10-22. Firmware Launches a Single-Write Waveform, WORDWIDE=0




* All EPx WORDWIDE bits must be cleared to zero for 8-bit single transactions. If any of the EPx WORDWIDE bits are set to ‘1’, then single transactions will be 16 bits wide.



Exhibit 2033 - Page 166 of 346

Figure 10-23. Single-Write Transaction Waveform

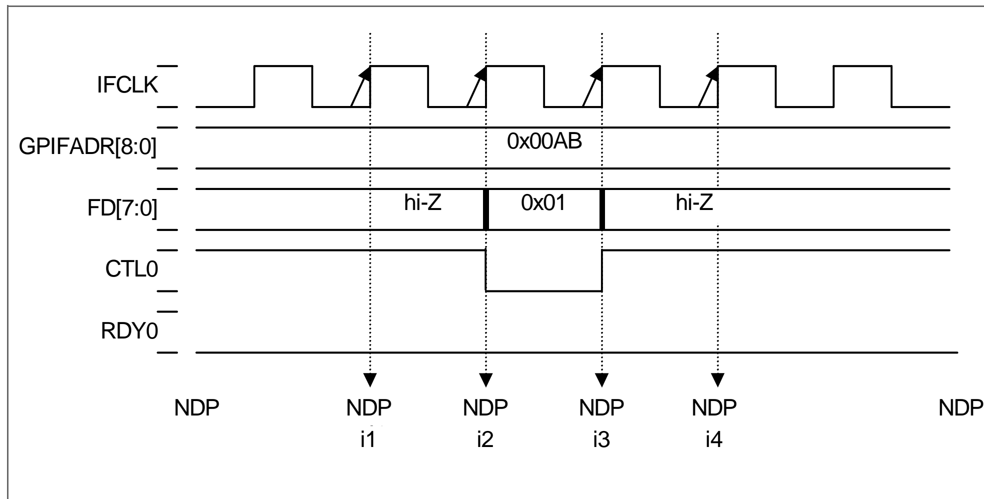



Figure 10-24. GPIF Designer Setup for the Waveform of Figure 10-23

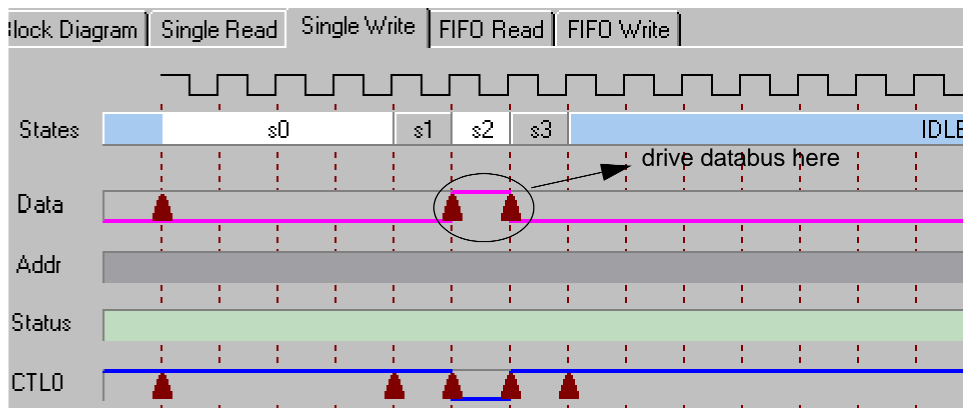





Exhibit 2033 - Page 167 of 346

Figure 10-25. GPIF Designer Output for the Waveform of Figure 10-23

| State | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| AddrMode | Same Val | Same Val | Same Val | Same Val | Same Val | Same Val | Same Val | |
| DataMode | No Data | No Data | Activate | No Data | No Data | No Data | No Data | |
| NextData | SameData | SameData | SameData | SameData | SameData | SameData | SameData | |
| Int Trig | No Int | No Int | No Int | No Int | No Int | No Int | No Int | |
| IF/Wait | Wait 4 | Wait 1 | Wait 1 | Wait 1 | Wait 1 | Wait 1 | Wait 1 | |
| Term A | | | | | | | | |
| LFUNC | | | | | | | | |
| Term B | | | | | | | | |
| Branch1 | | | | | | | | |
| Branch0 | | | | | | | | |
| Re-execute | | | | | | | | |
| CTL0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| CTL1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL5 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Single-Write transactions are simpler than Single-Read transactions because no dummy-read operation is required. To execute a Single-Write transaction:

1. Initialize the GPIF Configuration Registers and Waveform Descriptors.
2. Check that the GPIF is IDLE by checking if the DONE bit (GPIFIDLECS.7 or GPIFTRIG.7) is set.
3. If in 16-bit mode (WORDWIDE = 1), write the most-significant byte of the data to the XGPIFSGLDATH register, then **write** the least-significant byte to the XGPIFSGLDATLX register to start a Single-Write transaction.

   In 8-bit mode, simply **write** the data to the XGPIFSGLDATLX register to start a Single-Write transaction.
4. Wait for the GPIF to indicate that the transaction is complete. When the transaction is complete, the DONE bit (GPIFIDLECS.7 or GPIFTRIG.7) will be set to ‘1’. If enabled, a GPIFDONE interrupt will also be generated.

   The following C program fragments (Figure 10-26 and Figure 10-27 on page 169) illustrate how to perform a Single-Write transaction in 8-bit mode (WORDWIDE=0):



Exhibit 2033 - Page 168 of 346

Figure 10-26. Single-Write Transaction Functions

```
#define PERIPHCS 0x00AB
#define P_HSMODE 0x01

// prototypes
void GpifInit( void );

// Set Address GPIFADR[8:0] to PERIPHERAL
void Peripheral_SetAddress( WORD gaddr )
{
  GPIFADRH = gaddr >> 8;
  SYNCDELAY;
  GPIFADRL = ( BYTE )gaddr; // setup GPIF address
}

// write single byte to PERIPHERAL, using GPIF
void Peripheral_SingleByteWrite( BYTE gdata )
{
  while( !( GPIFTRIG & 0x80 ) )  // poll GPIFTRIG.7 Done bit
  {
     ;
  }

  XGPIFSGLDATLX = gdata;  // trigger GPIF single byte write transaction
}
```

Figure 10-27. Initialization Code for Single-Write Transactions

```
void TD_Init( void )
{
  … … … … …
  GpifInit();  // Configures GPIF from GPIF Designer generated waveform data

  // TODO: configure other endpoints, etc. here

  // TODO: arm OUT buffer(s) here

  // setup INT4 as internal source for GPIF interrupts
  // using INT4CLR (SFR), automatically enabled
  // INTSETUP |= 0x03;  //Enable INT4 Autovectoring
  // SYNCDELAY;
  // GPIFIE = 0x03;  // Enable GPIFDONE and GPIFWF interrupt(s)
  // SYNCDELAY;
  // EIE |= 0x04;  // Enable INT4 ISR, EIE.2(EIEX4)=1

  // TODO: configure GPIF interrupt(s) to meet your needs here
     … … … … …

  // tell peripheral we're going into high-speed xfr mode
  Peripheral_SetAddress( PERIPHCS );
  Peripheral_SingleByteWrite( P_HSMODE );
}
```



Exhibit 2033 - Page 169 of 346

### 10.4.3 FIFO-Read and FIFO-Write (Burst) Transactions

FIFO-Read and FIFO-Write waveforms transfer data to and from the MoBL-USB FX2LP18's Slave FIFOs (See chapter "Slave FIFOs" on page 107). The waveform is started by writing to EPxGPIFTRIG, where 'x' represents the FIFO (2, 4, 6, or 8) to/from which data should be transferred, or to GPIFTRIG.

A FIFO-Read or FIFO-Write waveform will generally transfer a long stream of data rather than a single byte or word. Usually, the waveform is programmed to terminate after a specified number of *transactions* or when a FIFO flag asserts (for example, when an IN FIFO is full or an OUT FIFO is empty). A 'transaction' is a transfer of a single byte (if WORDWIDE = 0) or word (if WORDWIDE = 1) to or from a FIFO. Using the *GPIF Designer*'s terminology, a transaction is either an 'Activate Data' for a FIFO-Read or a 'Next FIFO Data' for a FIFO-Write.

#### *10.4.3.1 Transaction Counter*

To use the Transaction Counter for FIFO 'x', load GPIFTCB3:0 with the desired number of transactions (1 to 4,294,967,295). When a FIFO-Read or -Write waveform is triggered on that FIFO, the GPIF will transfer the specified number of bytes (or words, if WORDWIDE = 1) automatically.

This mode of operation is called *Long Transfer Mode*; when the Transaction Counter is used in this way, the Waveform Descriptor should branch to the Idle State after each transaction.

Each time through the Idle State, the GPIF checks the Transaction Count; when it expires, the waveform terminates and the DONE bit is set. Otherwise, the GPIF re-executes the entire Waveform Descriptor. **Note** In Long Transfer Mode, the DONE bit is not set until the Transaction Count expires.

While the Transaction Count is active, the GPIF checks the Full Flag (for IN FIFOs) or the Empty Flag (for OUT FIFOs) on every pass through the Idle State. If the flag is asserted, the GPIF pauses until the over/underflow threat is removed, then it automatically resumes. In this way, the GPIF automatically throttles data flow in Long Transfer Mode.

The GPIFTCB3:0 registers are readable and they update as transactions occur, so the CPU can read the Transaction Count value at any time.

#### *10.4.3.2 Reading the Transaction-Count Status in a DP State*

To sample the transaction-count status in a DP State, set GPIFREADYCFG.5 to '1' (which instructs the MoBL-USB FX2LP18 to replace the RDY5 input with the transaction-count expiration flag), then launch a FIFO transaction which uses a transaction count. The MoBL-USB FX2LP18 will set the transaction-count expiration flag to '1' when the transaction count expires. This feature allows the Transaction Counter to be used without passing through the Idle State after each transaction.

### 10.4.4 GPIF Flag Selection

The GPIF can examine the PF, EF, or FF (of the current FIFO) during a waveform. One of the three flags is selected by the FS[1:0] bits in the EPxGPIFFLGSEL register; that selected flag is called the GPIF Flag.

### 10.4.5 GPIF Flag Stop

When EPxGPIFPFSTOP.0 is set to '1', FIFO-Read and -Write transactions are terminated by the assertion of the GPIF Flag. When this feature is used, it overrides the Transaction Counter; the GPIF waveform terminates (sets DONE to '1') *only* when the GPIF Flag asserts. If the GPIF Flag is already asserted at the time the waveform is launched, a GPIF DONE interrupt is not generated.

No special programming of the Waveform Descriptors is necessary, and FIFO Waveform Descriptors that transition through the Idle State on each transaction (for example, waveforms that do not use the Transaction Counter) are unaffected. Automatic throttling of the FIFOs in IDLE still occurs, so there's no danger that the GPIF will write to a full FIFO or read from an empty FIFO.

Unless the firmware aborts the GPIF transfer by writing to the GPIFABORT register, **only** the GPIF Flag assertion will terminate the waveform and set the DONE bit.

A waveform can potentially execute forever if the GPIF Flag never asserts.

**Important** The GPIF Flag is only automatically tested by the MoBL-USB FX2LP18 core while transitioning through the IDLE State, and the assertion of the GPIF Flag is not latched. Since the assertion of the GPIF Flag is not latched, if it is asserted and de-asserted during the waveform (due to the dynamic relationship between USB host activity and status of the MoBL-USB FX2LP18 FIFOs), the core would not see the GPIF Flag asserted in the IDLE state.



Exhibit 2033 - Page 170 of 346

### 10.4.5.1 Performing a FIFO-Read Transaction

Figure 10-28. Firmware Launches a FIFO-Read Waveform

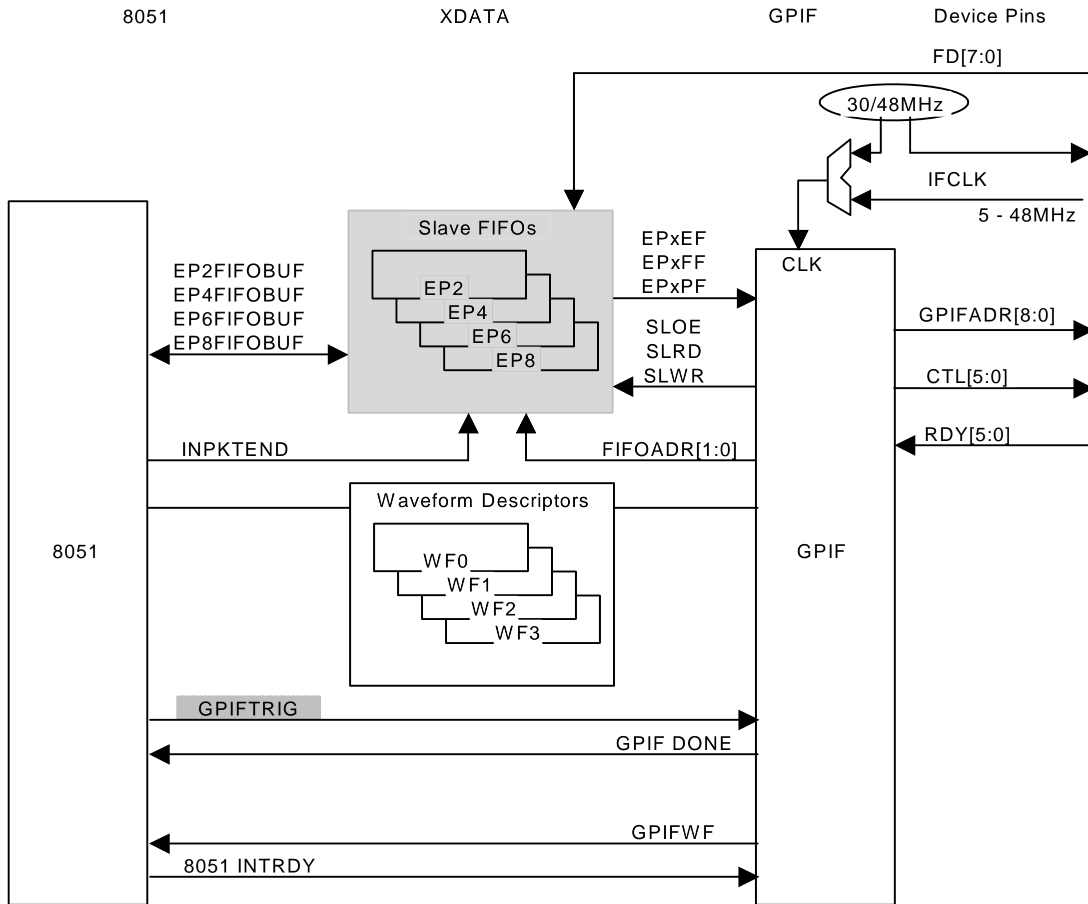



Figure 10-29. Example FIFO-Read Transaction

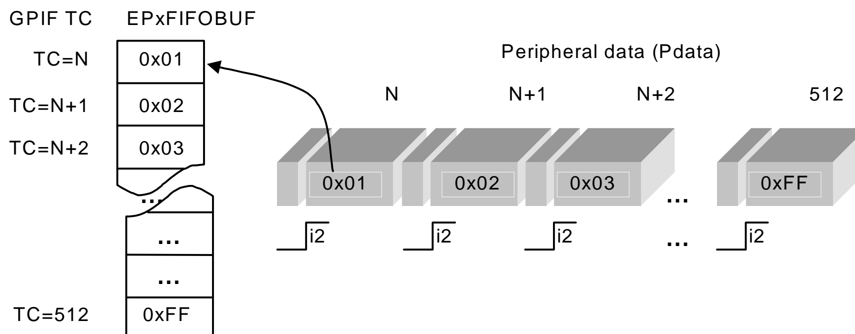
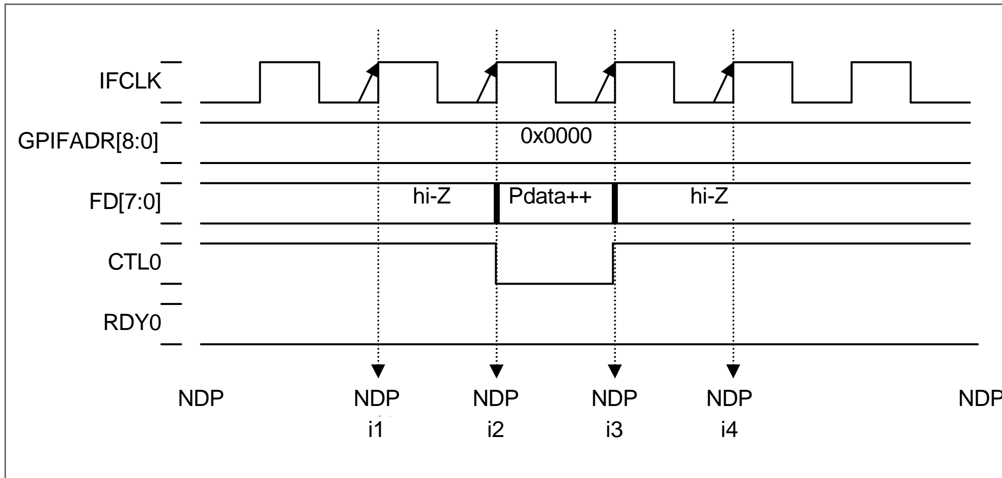





Exhibit 2033 - Page 171 of 346

Figure 10-30. FIFO-Read Transaction Waveform




The above waveform executes until the Transaction Counter expires (until it counts to 512, in this example). The Transaction Counter is decremented and sampled on each pass through the IDLE state. When the Transaction Counter is used without passing through the IDLE state, the Transaction Counter is decremented on each 'Activate' (which samples the data bus).

Each iteration of the waveform reads a data value from the FIFO Data bus into the FIFO, then decrements and checks the Transaction Counter. When it expires, the DONE bit is set to '1' and the GPIFDONE interrupt request is asserted.

Figure 10-31. GPIF Designer Setup for the Waveform of Figure 10-30

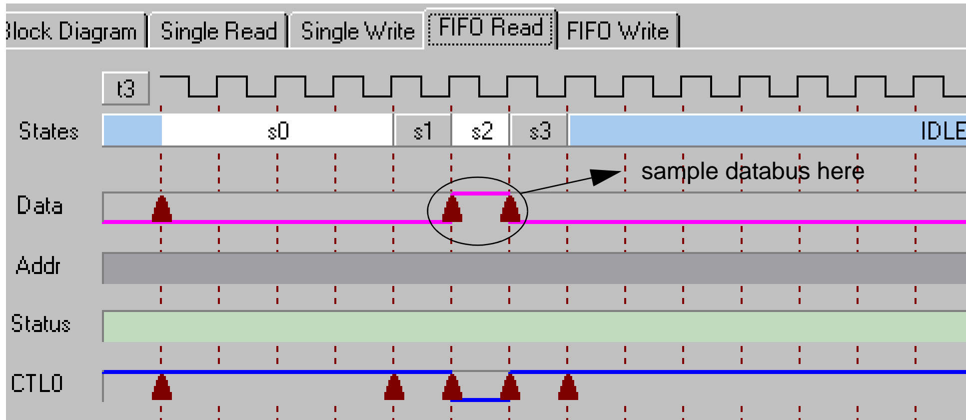





Exhibit 2033 - Page 172 of 346

Figure 10-32. GPIF Designer Output for the Waveform of Figure 10-30

| State | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| AddrMode | Same Val | Same Val | Same Val | Same Val | Same Val | Same Val | Same Val | |
| DataMode | No Data | No Data | Activate | No Data | No Data | No Data | No Data | |
| NextData | SameData | SameData | SameData | SameData | SameData | SameData | SameData | |
| Int Trig | No Int | No Int | No Int | No Int | No Int | No Int | No Int | |
| IF/Wait | Wait 4 | Wait 1 | Wait 1 | Wait 1 | Wait 1 | Wait 1 | Wait 1 | |
| Term A | | | | | | | | |
| LFUNC | | | | | | | | |
| Term B | | | | | | | | |
| Branch1 | | | | | | | | |
| Branch0 | | | | | | | | |
| Re-execute | | | | | | | | |
| CTL0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| CTL1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL5 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Typically, when performing a FIFO Read, only one 'Activate' is needed in the waveform, since each execution of 'Activate' increments the internal FIFO pointer (and EPxBCH:L) automatically.

To perform a FIFO-Read Transaction:

1. Program the MoBL-USB FX2LP18 to detect completion of the transaction. As with all GPIF Transactions, bit 7 of the GPIFTRIG register (the DONE bit) signals when the Transaction is complete.
2. In the GPIFTRIG register, set the RW bit to '1' and load EP[1:0] with the appropriate value for the FIFO which is to receive the data.
3. Program the MoBL-USB FX2LP18 to commit ('pass-on') the data from the FIFO to the endpoint. The data can be transferred from the FIFO to the endpoint by either of the following methods:
   - AUTOIN=1: CPU is not in the data path; the MoBL-USB FX2LP18 automatically commits data from the FIFO Data bus to the USB.
   - AUTOIN=0: Firmware must manually commit data to the USB by writing either EPxBCL or INPKTEND (with SKIP=0).

The following C program fragments (Figure 10-33 on page 174 through Figure 10-36 on page 176) illustrate how to perform a FIFO-Read transaction in 8-bit mode (WORDWIDE = 0) with AUTOIN = 0:



Exhibit 2033 - Page 173 of 346

Figure 10-33. FIFO-Read Transaction Functions

```
#define GPIFTRIGRD 4

#define GPIF_EP2 0
#define GPIF_EP4 1
#define GPIF_EP6 2
#define GPIF_EP8 3

#define BURSTMODE 0x0000
#define HSPKTSIZE 512

… … … … …

// read(s) from PERIPHERAL, using GPIF and EPxFIFO
void Peripheral_FIFORead( BYTE FIFO_EpNum )
{
  while( !( GPIFTRIG & 0x80 ) ) // poll GPIFTRIG.7 GPIF Done bit
  {
     ;
  }

  // trigger FIFO read transaction(s), using SFR
  GPIFTRIG = GPIFTRIGRD | FIFO_EpNum;    // R/W=1, EP[1:0]=FIFO_EpNum
                                         // for EPx read(s)
}

// Set GPIF Transaction Count
void Peripheral_SetGPIFTC( WORD xfrcnt)
{
  GPIFTCB1 = xfrcnt >> 8;  // setup transaction count
  SYNCDELAY;
  GPIFTCB0 = ( BYTE )xfrcnt;
}

… … … … …
```



Exhibit 2033 - Page 174 of 346

Figure 10-34. Initialization Code for FIFO-Read Transactions

```
void TD_Init( void )
{
  … … … … …
  GpifInit();  // Configures GPIF from GPIF Designer generated waveform data

  // TODO: configure other endpoints, etc. here
  EP8CFG = 0xE0;  // EP8 is DIR=IN, TYPE=BULK
  SYNCDELAY;
  EP8FIFOCFG = 0x04;  // EP8 is AUTOOUT=0, AUTOIN=0, ZEROLEN=1, WORDWIDE=0

  // TODO: arm OUT buffer(s) here

  // setup INT4 as internal source for GPIF interrupts
  // using INT4CLR (SFR), automatically enabled
  // INTSETUP |= 0x03;  //Enable INT4 Autovectoring
  // SYNCDELAY;
  // GPIFIE = 0x03;  // Enable GPIFDONE and GPIFWF interrupt(s)
  // SYNCDELAY;
  // EIE |= 0x04;  // Enable INT4 ISR, EIE.2(EIEX4)=1

  // TODO: configure GPIF interrupt(s) to meet your needs here
     … … … … …

  // tell peripheral we're going into high-speed xfr mode
  Peripheral_SetAddress( PERIPHCS );
  Peripheral_SingleByteWrite( P_HSMODE );

  // configure some GPIF registers
  Peripheral_SetAddress( BURSTMODE );
  Peripheral_SetGPIFTC( HSPKTSIZE );
}
```

Figure 10-35. FIFO-Read w/ AUTOIN = 0, Committing Packets via INPKTEND w/SKIP=0

```
void TD_Poll( void )
{
  … … … … …
  if( !( EP68FIFOFLGS & 0x10 ) )
  { // EP8FF=0 when buffer available
    // host is taking EP8 data fast enough
    Peripheral_FIFORead( GPIF_EP8 );
  }

  if( gpifdone_event_flag )
  { // GPIF currently pointing to EP8, last FIFO accessed
    if( !( EP2468STAT & 0x80 ) )
    { // EP8F=0 when buffer available
      INPKTEND = 0x08;     // Firmware commits pkt by writing 8 to INPKTEND
      gpifdone_event_flag = 0;
    }
  }
  … … … … …
}
```



Exhibit 2033 - Page 175 of 346

Figure 10-36. FIFO-Read w/ AUTOIN = 0, Committing Packets via EPxBCL

```
void TD_Poll( void )
{
  … … … … …
  if( !( EP68FIFOFLGS & 0x10 ) )
  { // EP8FF=0 when buffer available
    // host is taking EP8 data fast enough
    Peripheral_FIFORead( GPIF_EP8 );
  }

  if( gpifdone_event_flag )
  { // GPIF currently pointing to EP8, last FIFO accessed
    if( !( EP2468STAT & 0x80 ) )
    { // EP8F=0 when buffer available
      // 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 8-byte packet on to host
    }
  }
  … … … … …
}
```

## 10.4.6 Firmware Access to IN Packets, (AUTOIN=1)

The only difference between auto (AUTOIN=1) and manual (AUTOIN=0) modes for IN packets is the packet length feature (EPxAUTOINLENH/L) in AUTOIN=1.

Figure 10-37. AUTOIN=1, GPIF FIFO Read Transactions, AUTOIN = 1

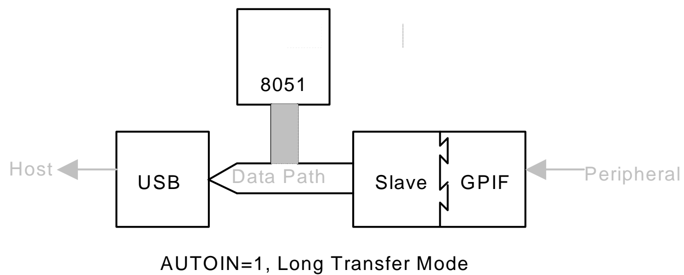


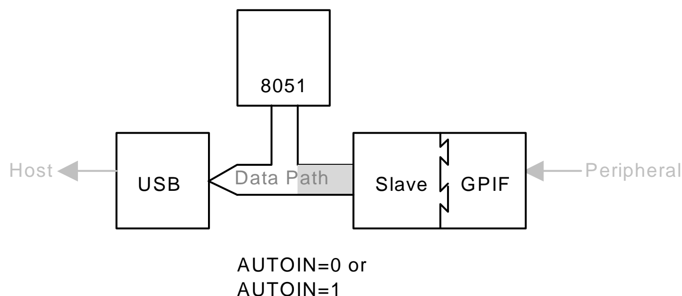



Exhibit 2033 - Page 176 of 346

Figure 10-38. FIFO-Read Transaction Code, AUTOIN = 1

```
void TD_Init( void )
{
    EP8CFG = 0xE0;  // EP8 is DIR=IN, TYPE=BULK
    SYNCDELAY;
    EP8FIFOCFG = 0x0C;  // EP8 is AUTOOUT=0, AUTOIN=1, ZEROLEN=1, WORDWIDE=0
    SYNCDELAY;
    EP8AUTOINLENH = 0x02;  // if AUTOIN=1, auto commit 512 byte packets
    SYNCDELAY;
    EP8AUTOINLENL = 0x00;
}

void TD_Poll( void )
{
    // no code necessary to xfr data from master to host!
    // AUTOIN=1 and EP8AUTOINLENH:L=512 auto commits IN packets,
    // in 512 byte chunks.
}
```

Figure 10-39. Firmware intervention, AUTOIN = 0/1




## 10.4.7 Firmware Access to IN Packets, (AUTOIN = 0)

In manual IN mode (AUTOIN=0), the firmware has the following options:

1. It can commit ('pass-on') packets sent from the master to the host when a buffer is available, by writing the INPKTEND register with the corresponding EPx number and SKIP=0 (see Figure 10-40).
2. It can skip a packet by writing to INPKTEND with SKIP=1. See Figure 10-41 on page 178.
3. It can source or edit a packet (for example, write directly to EPxFIFOBUF) then write the EPxBCL. See Figure 10-42 on page 178.

Figure 10-40. Committing a Packet by Writing INPKTEND with EPx Number (w/SKIP=0)

```
TD_Poll():
… … … … …
if( master_finished_longxfr( ) )
{  // master currently points to EP8, last FIFO accessed
  if( !( EP68FIFOFLGS & 0x10 ) )
  { // EP8FF=0 when buffer available
    INPKTEND = 0x08;  // Firmware commits pkt
                      // by writing 0x08 to INPKTEND
    release_master( EP8 );
  }
}
… … … … …
```



Exhibit 2033 - Page 177 of 346

Figure 10-41.  Skipping a Packet by Writing to INPKTEND w/SKIP=1

```
TD_Poll():
… … … … …
if( master_finished_longxfr( ) )
{  // master currently points to EP8, last FIFO accessed
  if( !( EP68FIFOFLGS & 0x10 ) )
  { // EP8FF=0 when buffer available
    INPKTEND = 0x88;  // Firmware commits pkt
                      // by writing 0x88 to INPKTEND
    release_master( EP8 );
  }
}
… … … … …
```

Figure 10-42.  Sourcing an IN Packet by writing to EPxBCH:L

```
TD_Poll():
… … … … …
if( source_pkt_event )
{  // 100msec 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
    }

    // Reset FIFO 8.

    FIFORESET = 0x80;  // Activate NAK-All to avoid race conditions.
    SYNCDELAY;
    FIFORESET = 0x08;  // Reset FIFO 8.
    SYNCDELAY;
    FIFORESET = 0x00;  // Deactivate NAK-All.

    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 src'd buffer on to host
  }
  else
  {
    history_record( EP8, BAD_MASTER );
  }
}
… … … … …
```



Exhibit 2033 - Page 178 of 346

#### 10.4.7.1 Performing a FIFO-Write Transaction

Figure 10-43. Firmware Launches a FIFO-Write Waveform

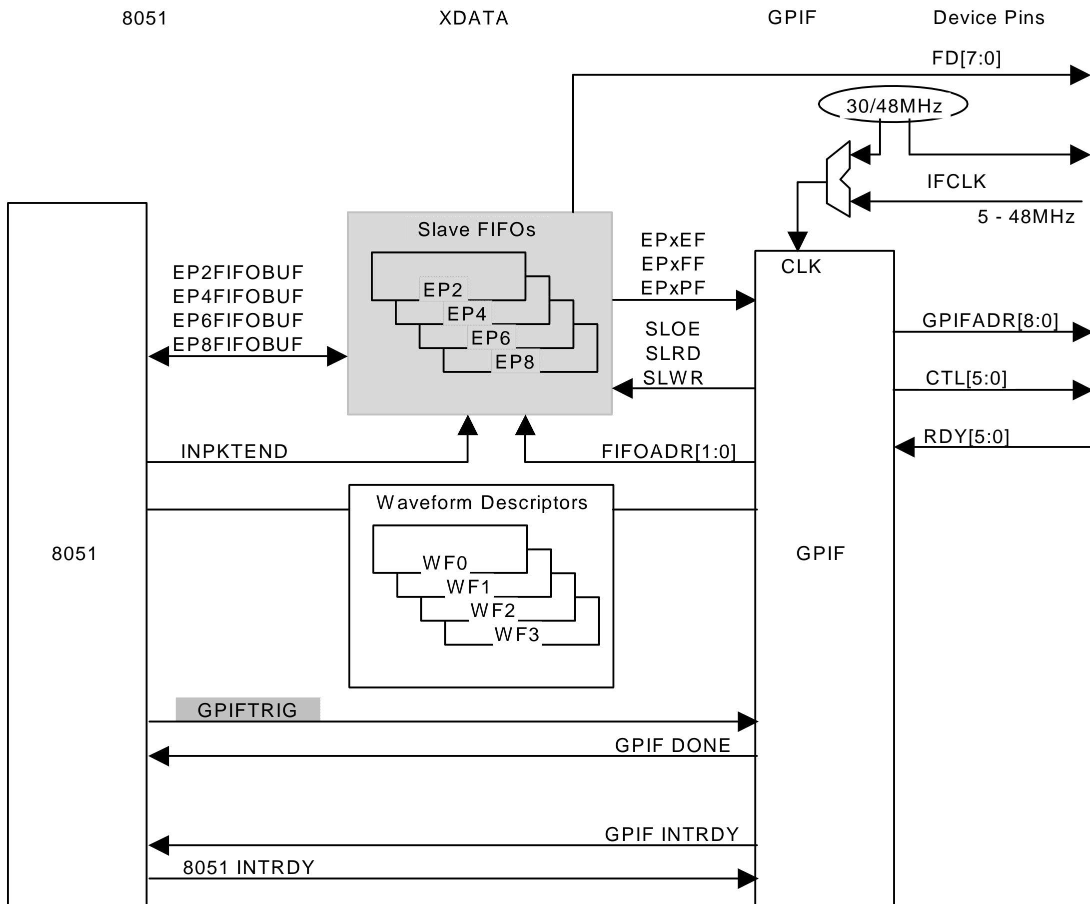



Figure 10-44. Example FIFO-Write Transaction

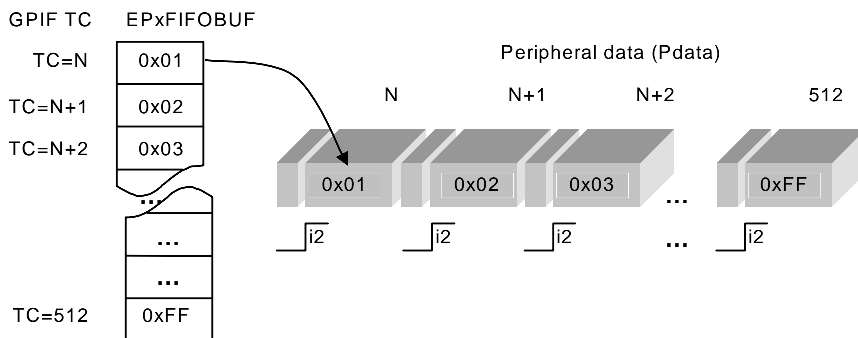
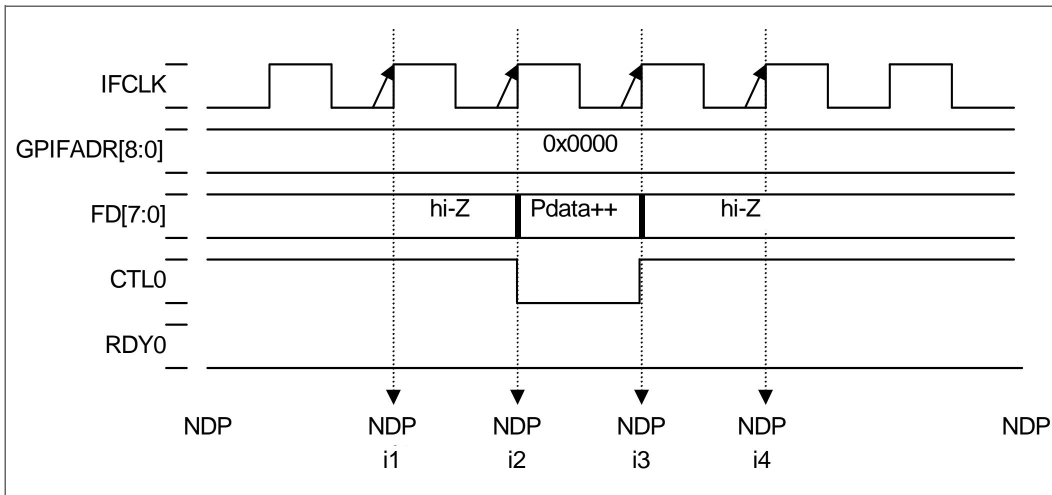





Exhibit 2033 - Page 179 of 346

Figure 10-45. FIFO-Write Transaction Waveform




The above waveform executes until the Transaction Counter expires (until it counts to 512, in this example). The Transaction Counter is decremented and sampled on each pass through the Idle State. When the Transaction Counter is used without passing through the IDLE state, the Transaction Counter is decremented on each ‘Nextdata’ (which increments the FIFO pointer).

Each iteration of the waveform writes a data value from the FIFO to the FIFO Data bus, then decrements and checks the Transaction Counter. When it expires, the DONE bit is set to ‘1’ and the GPIFDONE interrupt request is asserted.

Figure 10-46. GPIF Designer Setup for the Waveform of Figure 10-45

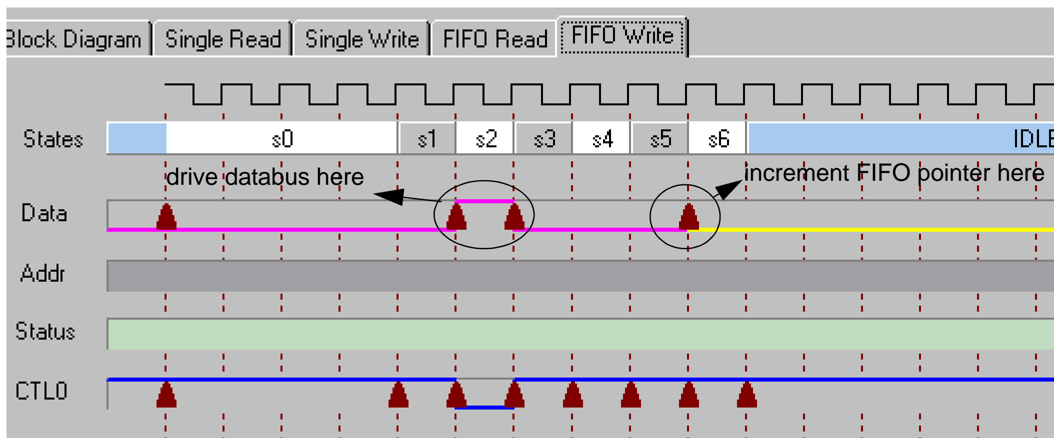





Exhibit 2033 - Page 180 of 346

Figure 10-47. GPIF Designer Output for the Waveform of Figure 10-45

| State | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| AddrMode | Same Val | Same Val | Same Val | Same Val | Same Val | Same Val | Same Val | |
| DataMode | No Data | No Data | Activate | No Data | No Data | No Data | No Data | |
| NextData | SameData | SameData | SameData | SameData | SameData | SameData | NextData | |
| Int Trig | No Int | No Int | No Int | No Int | No Int | No Int | No Int | |
| IF/Wait | Wait 4 | Wait 1 | Wait 1 | Wait 1 | Wait 1 | Wait 1 | Wait 1 | |
| Term A | | | | | | | | |
| LFUNC | | | | | | | | |
| Term B | | | | | | | | |
| Branch1 | | | | | | | | |
| Branch0 | | | | | | | | |
| Re-execute | | | | | | | | |
| CTL0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| CTL1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CTL5 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Typically, when performing a FIFO-Write, only one ‘NextData’ is needed in the waveform, since each execution of ‘NextData’ increments the FIFO pointer.

To perform a FIFO-Write Transaction:

1. Program the MoBL-USB FX2LP18 to detect completion of the transaction. As with all GPIF Transactions, bit 7 of the GPIFTRIG register (the DONE bit) signals when the Transaction is complete.
2. In the GPIFTRIG register, set the RW bit to 0 and load EP[1:0] with the appropriate value for the FIFO which is to source the data.
3. Program the MoBL-USB FX2LP18 to commit (‘pass-on’) the data from the endpoint to the FIFO. The data can be transferred by either of the following methods:
   - AUTOOUT=1: CPU is not in the data path; the MoBL-USB FX2LP18 automatically commits data from the USB to the FIFO Data bus.
   - AUTOOUT=0: Firmware must manually commit data to the FIFO Data bus by writing EPxBCL.7=0 (firmware can choose to skip the current packet by writing EPxBCL.7=1).



Exhibit 2033 - Page 181 of 346

The following C program fragments (Figures 10-48 through 10-50) illustrate how to perform a FIFO-Read transaction in 8-bit mode (WORDWIDE = 0) with AUTOOUT = 0:

Figure 10-48. FIFO-Write Transaction Functions

```
#define GPIFTRIGWR 0

#define GPIF_EP2 0
#define GPIF_EP4 1
#define GPIF_EP6 2
#define GPIF_EP8 3

#define BURSTMODE 0x0000
#define HSPKTSIZE 512

… … … … …

// write byte(s) to PERIPHERAL, using GPIF and EPxFIFO
void Peripheral_FIFOWrite( BYTE FIFO_EpNum )
{
  while( !( GPIFTRIG & 0x80 ) )  // poll GPIFTRIG.7 Done bit
  {
     ;
  }

  // trigger FIFO write transaction(s), using SFR
  GPIFTRIG = FIFO_EpNum;  // R/W=0, EP[1:0]=FIFO_EpNum for EPx write(s)
}

// Set GPIF Transaction Count
void Peripheral_SetGPIFTC( WORD xfrcnt)
{
  GPIFTCB1 = xfrcnt >> 8;  // setup transaction count
  SYNCDELAY;
  GPIFTCB0 = ( BYTE )xfrcnt;
}
… … … … …
```



Exhibit 2033 - Page 182 of 346

Figure 10-49. Initialization Code for FIFO-Write Transactions

```
void TD_Init( void )
{
  … … … … …
  GpifInit();  // Configures GPIF from GPIF Designer generated waveform data

  // TODO: configure other endpoints, etc. here
  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
  SYNCDELAY;
  // "all" EP2 buffers automatically arm when AUTOOUT=1

  // TODO: arm OUT buffer(s) here
  OUTPKTEND = 0x82;    // Arm both EP2 buffers to "prime the pump"
  SYNCDELAY;
  OUTPKTEND = 0x82;
  SYNCDELAY;

  // setup INT4 as internal source for GPIF interrupts
  // using INT4CLR (SFR), automatically enabled
  // INTSETUP |= 0x03;  //Enable INT4 Autovectoring
  // GPIFIE = 0x03;  // Enable GPIFDONE and GPIFWF interrupt(s)
  // EIE |= 0x04;  // Enable INT4 ISR, EIE.2(EIEX4)=1

  // TODO: configure GPIF interrupt(s) to meet your needs here
    … … … … …

  // tell peripheral we're going into high-speed xfr mode
  Peripheral_SetAddress( PERIPHCS );
  Peripheral_SingleByteWrite( P_HSMODE );

  // configure some GPIF control registers
  Peripheral_SetAddress( BURSTMODE );
}
```

Figure 10-50. FIFO-Write w/ AUTOOUT = 0, Committing Packets via OUTPKTEND

```
void TD_Poll( void )
{
  … … … … …
  if( !( EP2468STAT & 0x01 ) )
  { // EP2EF=0 when FIFO "not" empty, host sent pkt.
    OUTPKTEND = 0x02;  // SKIP=0, pass buffer on to master

    if( gpifdone_event_flag )
    {
      Peripheral_SetGPIFTC( HSPKTSIZE );
      Peripheral_FIFOWrite( GPIF_EP2 );
      gpifdone_event_flag = 0;
    }
  }
  … … … … …
}
```



Exhibit 2033 - Page 183 of 346

### 10.4.8 Firmware Access to OUT Packets, (AUTOOUT=1)

To achieve the maximum USB 2.0 bandwidth, the host and master are directly connected when AOUTOOUT=1; the CPU is bypassed and the OUT FIFO is automatically committed to the host:

Figure 10-51. CPU not in data path, AUTOOUT=1

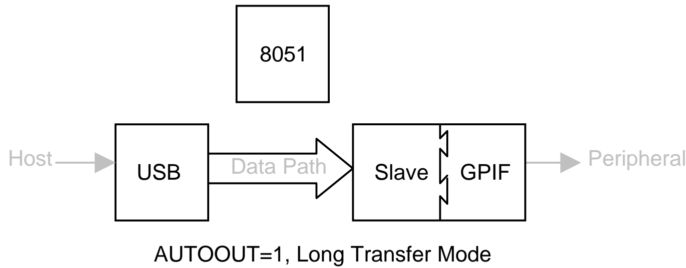



Figure 10-52. TD_Init Example: Configuring AUTOOUT = 1

```
TD_Init():
… … … … …
REVCTL = 0x03;       // REVCTL.0 and REVCTL.1 set 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;
OUTPKTEND = 0x82;    // Arm both EP2 buffers to "prime the pump"
SYNCDELAY;
OUTPKTEND = 0x82;
SYNCDELAY;
EP2FIFOCFG = 0x10;  // EP2 is AUTOOUT=1, AUTOIN=0, ZEROLEN=0, WORDWIDE=0

… … … … …
```

Figure 10-53. FIFO-Write Transaction Code, AUTOOUT = 1

```
TD_Poll():
… … … … …
// no code necessary to xfr data from host to master!
// AUTOOUT=1 auto-commits packets
… … … … …
```



Exhibit 2033 - Page 184 of 346

### 10.4.9 Firmware access to OUT packets, (AUTOOUT = 0)

Figure 10-54. Firmware can Skip or Commit, AUTOOUT = 0

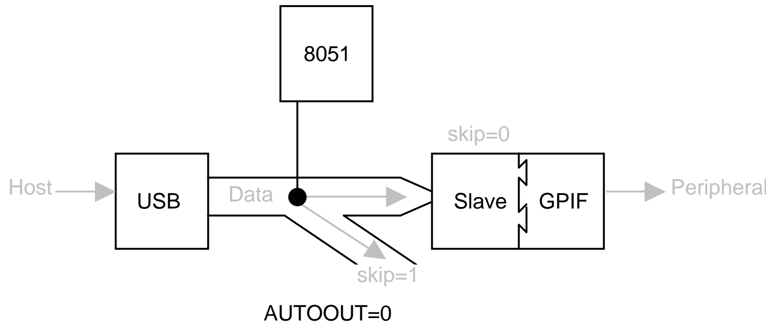



Figure 10-55. Initialization Code for AUTOOUT = 0

```
TD_Init():
… … … … …
REVCTL = 0x03;  // REVCTL.0 and REVCTL.1 set 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;

// OUT endpoints do NOT come up armed
OUTPKTEND = 0x82;  // arm first buffer by writing OUTPKTEND w/skip=1
SYNCDELAY;
OUTPKTEND = 0x82;  // arm second buffer by writing OUTPKTEND w/skip=1
… … … … …
```

In manual OUT mode (AUTOOUT = 0), the firmware has the following options:

1. It can commit ('pass-on') packet(s) sent from the host to the master when a buffer is available, by writing the OUTPKTEND register with the SKIP bit (OUTPKTEND.7) cleared to 0 (see Figure 10-56) and the endpoint number in EP[3:0].

Figure 10-56. Committing an OUT Packet by Writing OUTPKTEND w/SKIP=0

```
TD_Poll():
… … … … …
if( !( EP24FIFOFLGS & 0x02 ) )
{ // EP2EF=0 when FIFO "not" empty, host sent pkt.
  OUTPKTEND = 0x02;  // SKIP=0, pass buffer on to master
}
… … … … …
```

2. It can skip packet(s) sent from the host to the master by writing the OUTPKTEND register with the SKIP bit (OUTPKTEND.7) set to ‘1’ (see Figure 10-57) and the endpoint number in EP[3:0].

Figure 10-57. Skipping an OUT Packet by Writing OUTPKTEND w/SKIP=1

```
TD_Poll():
… … … … …
if( !( EP24FIFOFLGS & 0x02 ) )
{ // EP2EF=0 when FIFO “not” empty, host sent pkt.
  OUTPKTEND = 0x82;  // SKIP=1, do NOT pass buffer on to master
}
… … … … …
```

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 (Figure 10-58).

   In all cases, the OUT buffer automatically re-arms so it can receive the next packet, after the GPIF has transmitted all data in the OUT buffer.

   See section “EP2BCH:L, EP4BCH:L, EP6BCH:L, EP8BCH:L” on page 101 for a detailed description of the SKIP bit.

Figure 10-58. Sourcing an OUT Packet (AUTOOUT = 0)

```
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"
}
… … … … …
```

The master is not notified when a packet has been skipped by the firmware.

The OUT FIFO is not committed to the host after a hard reset. This means that it is not available to initially accept any OUT packets. In its initialization routine, therefore, the firmware should skip *n* packets (where *n* = 2, 3, or 4 depending on the buffering depth) in order to ensure that the entire FIFO is committed to the host. See Figure 10-59 on page 187.



Exhibit 2033 - Page 186 of 346

Figure 10-59. Ensuring that the FIFO is Clear after a Hard Reset

```
TD_Init():
    … … … … …
    REVCTL = 0x03;  // REVCTL.0 and REVCTL.1 set 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;
    // OUT endpoints do NOT come up armed
    OUTPKTEND = 0x82;  // arm first buffer by writing OUTPKTEND w/skip=1
    SYNCDELAY;
    OUTPKTEND = 0x82;  // arm second buffer by writing OUTPKTEND w/skip=1
    … … … … …
```

## 10.5 UDMA Interface

The MoBL-USB FX2LP18 has additional GPIF registers specifically for implementing a UDMA (Ultra-ATA) interface. For more information, refer to the Registers chapter on page 237.

## 10.6 ECC Generation

The MoBL-USB FX2LP18 has additional registers specifically for implementing ECC based on the SmartMedia™ standard. For more information, refer to the Registers chapter on page 237.



Exhibit 2033 - Page 187 of 346

Exhibit 2033 - Page 188 of 346

# 11. CPU Introduction

## 11.1 Introduction

The MoBL-USB FX2LP18's CPU, an enhanced 8051, is fully described in chapters Instruction Set, on page 197, Input/Output, on page 203, and Timers/Counters and Serial Interface, on page 217. This chapter introduces the processor, its interface to the MoBL-USB FX2LP18 logic, and describes architectural differences from a standard 8051. Figure 11-1 is a block diagram of the MoBL-USB FX2LP18's 8051-based CPU.

Figure 11-1. MoBL-USB FX2LP18 CPU Features

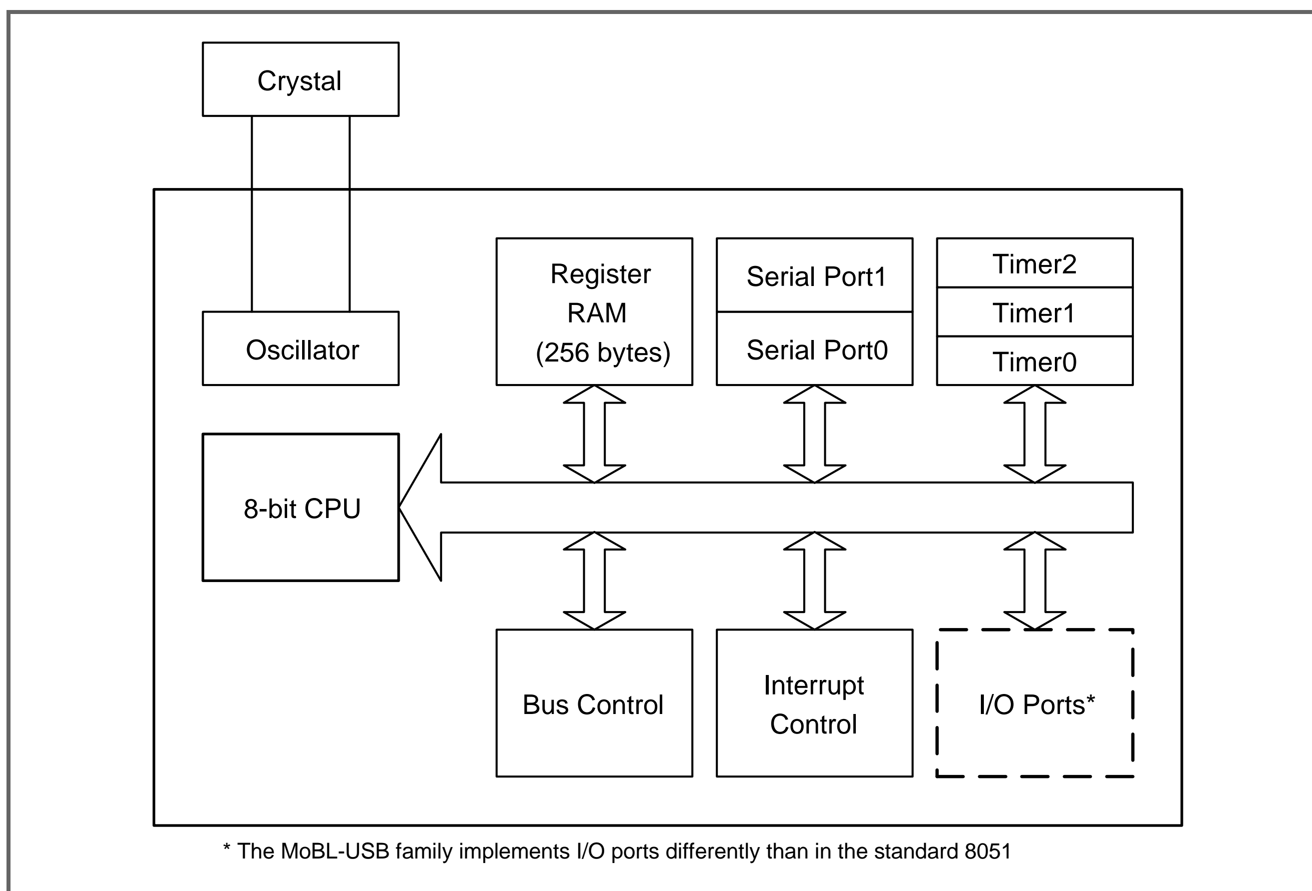





Exhibit 2033 - Page 189 of 346

## 11.2 8051 Enhancements

The MoBL-USB FX2LP18 uses the standard 8051 instruction set, so it's supported by industry-standard 8051 compilers and assemblers. Instructions execute faster on the MoBL-USB FX2LP18 than on the standard 8051:

- Wasted bus cycles are eliminated; an instruction cycle uses only four clocks, rather than the standard 8051's 12 clocks.
- The MoBL-USB FX2LP18's CPU clock runs at 12 MHz, 24 MHz, or 48 MHz — up to four times the clock speed of the standard 8051.

In addition to speed improvements, the MoBL-USB FX2LP18 includes the following architectural enhancements to the CPU:

- A second data pointer
- A second USART
- A third, 16-bit timer (TIMER2)
- Eight additional interrupts (INT2-INT6, WAKEUP, T2, and USART1)
- Variable MOVX timing to accommodate fast and slow RAM peripherals
- Two Autopointers (auto-incrementing data pointers)
- Vectored USB and FIFO/GPIF interrupts
- Baud rate timer for 115K/230K baud USART operation
- Sleep mode with three wakeup sources
- An $I^2C$™ bus controller that runs at 100 or 400 kHz
- MoBL-USB FX2LP18-specific SFRs
- Separate buffers for the SETUP and DATA portions of a USB CONTROL transfer
- A hardware pointer for SETUP data, plus logic to process entire CONTROL transfers automatically
- CPU clock-rate selection of 12, 24 or 48 MHz
- Breakpoint facility
- IO Port C read and write strobes

## 11.3 Performance Overview

The MoBL-USB FX2LP18 has been designed to offer increased performance by executing instructions in a 4-clock bus cycle, as opposed to the 12-clock bus cycle in the standard 8051 (see Figure 11-2 on page 191). This shortened bus timing improves the instruction execution rate for most instructions by a factor of three over the standard 8051 architectures.

Some instructions require a different number of instruction cycles on the MoBL-USB FX2LP18 than they do on the standard 8051. In the standard 8051, all instructions except for MUL and DIV take one or two instruction cycles to complete. In the MoBL-USB FX2LP18, instructions can take between one and five instruction cycles to complete. However, due to the shortened bus timing of the MoBL-USB FX2LP18, every instruction executes faster than on a standard 8051, and the average speed improvement over the entire instruction set is approximately 2.5×. Table 11-1 catalogs the speed improvements.

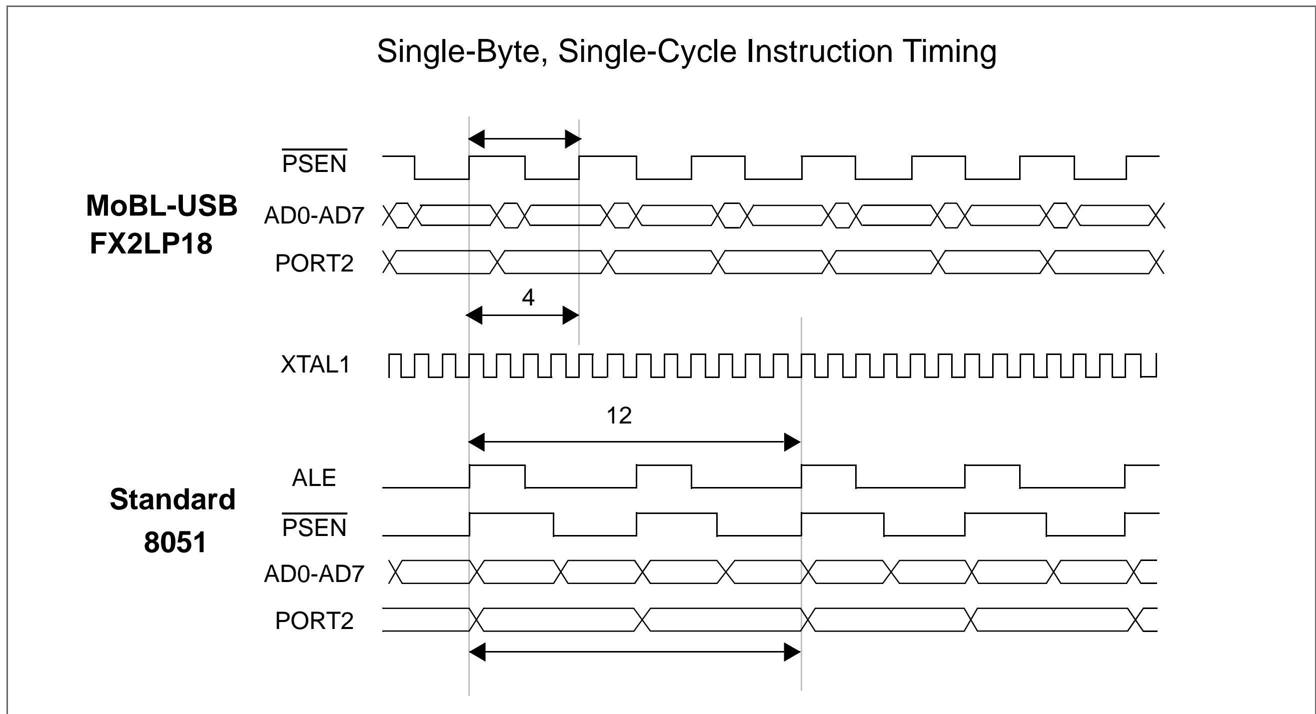
Table 11-1. MoBL-USB FX2LP18 Speed Compared to Standard 8051

| **Of the 246 MoBL-USB FX2LP18 opcodes...** | |
|---|---|
| 150 execute at | 3.0× standard speed |
| 51 execute at | 1.5× standard speed |
| 43 execute at | 2.0× standard speed |
| 2 execute at | 2.4× standard speed |
| **Average Improvement:** | **2.5×** |
| **Note** Comparison is between MoBL-USB FX2LP18 and standard 8051 running at the same clock frequency. | |



Exhibit 2033 - Page 190 of 346

Figure 11-2. MoBL-USB FX2LP18 to Standard 8051 Timing Comparison




## 11.4 Software Compatibility

The MoBL-USB FX2LP18 is object-code-compatible with the industry-standard 8051 microcontroller. That is, object code compiled with an industry-standard 8051 compiler or assembler executes on the MoBL-USB FX2LP18 and is functionally equivalent. However, because the MoBL-USB FX2LP18 uses a different instruction timing than the standard 8051, existing code with timing loops may require modification.

The MoBL-USB FX2LP18 instruction timing is identical to that of the Dallas Semiconductor DS80C320.

## 11.5 803x/805x Feature Comparison

Table 11-2 provides a feature-by-feature comparison between the MoBL-USB FX2LP18 and several common 803x/805x devices.

Table 11-2. Comparison Between MoBL-USB FX2LP18 and Other 803x/805x Devices

| Feature | Intel | | | | Dallas DS80C320 | Cypress MoBL-USB FX2LP18 |
|---|---|---|---|---|---|---|
| | 8031 | 8051 | 80C32 | 80C52 | | |
| Clocks per instruction cycle | 12 | 12 | 12 | 12 | 4 | 4 |
| Program / Data Memory | - | 4 kB ROM | - | 8 kB ROM | - | 16 kB RAM |
| Internal RAM | 128 bytes | 128 bytes | 256 bytes | 256 bytes | 256 bytes | 256 bytes |
| Data Pointers | 1 | 1 | 1 | 1 | 2 | 2 |
| Serial Ports | 1 | 1 | 1 | 1 | 2 | 2 |
| 16-bit Timers | 2 | 2 | 3 | 3 | 3 | 3 |
| Interrupt sources (internal and external) | 5 | 5 | 6 | 6 | 13 | 13 |
| Stretch data-memory cycles | no | no | no | no | yes | yes |



Exhibit 2033 - Page 191 of 346

## 11.6 MoBL-USB FX2LP18/DS80C320 Differences

Although the MoBL-USB FX2LP18 is similar to the DS80C320 in terms of hardware features and instruction cycle timing, there are some important differences between the MoBL-USB FX2LP18 and the DS80C320.

### 11.6.1 Serial Ports

The MoBL-USB FX2LP18 does not implement serial port framing-error detection and does not implement slave address comparison for multiprocessor communications. Therefore, the MoBL-USB FX2LP18 also does not implement the following SFRs: SADDR0, SADDR1, SADEN0, and SADEN1.

### 11.6.2 Timer 2

The MoBL-USB FX2LP18 does not implement Timer 2 downcounting mode or the downcount enable bit (TMOD2, Bit 0). Also, the MoBL-USB FX2LP18 does not implement Timer 2 output enable (T2OE) bit (TMOD2, Bit 1). Therefore, the TMOD2 SFR is also not implemented in the MoBL-USB FX2LP18.

The MoBL-USB FX2LP18 Timer 2 overflow output is active for one clock cycle. In the DS80C320, the Timer 2 overflow output is a square wave with a 50% duty cycle.

Although the T2OE bit is not present in the MoBL-USB FX2LP18, Timer 2 output can still be enabled or disabled via the PORTECFG.2 bit, since the T2OUT pin is multiplexed with PORTE.2.

PORTECFG.2=0 configures the pin as a general-purpose IO pin and disabled Timer 2 output;
PORTECFG.2=1 configures the pin as the T2OUT pin and enables Timer 2 output.

### 11.6.3 Timed Access Protection

The MoBL-USB FX2LP18 does not implement timed access protection and, therefore, does not implement the TA SFR.

### 11.6.4 Watchdog Timer

The MoBL-USB FX2LP18 does not implement a watchdog timer.

### 11.6.5 Power Fail Detection

The MoBL-USB FX2LP18 does not implement a power fail detection circuit.

### 11.6.6 Port IO

The MoBL-USB FX2LP18's port IO implementation is significantly different from that of the DS80C320, mainly because of the alternate functions shared with most of the IO pins. See Input/Output, on page 203.

### 11.6.7 Interrupts

Although the basic interrupt structure of the MoBL-USB FX2LP18 is similar to that of the DS80C320, five of the interrupt sources are different:

Table 11-3. Differences between MoBL-USB FX2LP18 and DS80C320 Interrupts

| Interrupt Priority | Dallas DS80C320 | Cypress MoBL-USB FX2LP18 |
|---|---|---|
| 0 | Power Fail | RESUME (USB Wakeup) |
| 8 | External Interrupt 2 | USB |
| 9 | External Interrupt 3 | I²C Bus |
| 10 | External Interrupt 4 | GPIF/FIFOs |
| 12 | Watchdog Timer | External Interrupt 6 |

For more information, refer to the Timers/Counters and Serial Interface chapter on page 217.



Exhibit 2033 - Page 192 of 346

## 11.7 MoBL-USB FX2LP18 Register Interface

The MoBL-USB FX2LP18 peripheral logic (USB, GPIF, FIFOs, and so on) is controlled via a set of memory mapped registers and buffers at addresses 0xE400 through 0xFFFF. These registers and buffers are grouped as follows:

- GPIF Waveform Descriptor Tables
- General configuration
- Endpoint configuration
- Interrupts
- Input/Output
- USB Control
- Endpoint operation
- GPIF/FIFOs
- Endpoint buffers

These registers and their functions are described throughout this manual. A full description of every MoBL-USB FX2LP18 register appears in the Registers chapter on page 237.

## 11.8 MoBL-USB FX2LP18 Internal RAM

Figure 11-3. MoBL-USB FX2LP18 Internal Data RAM

| Address | | |
|---|---|---|
| 0xFF – 0x80 | **Upper 128**<br>Indirect Addr | **SFR Space**<br>Direct Addr |
| 0x7F – 0x00 | **Lower 128**<br>Direct Addr | |

Like the standard 8051, the MoBL-USB FX2LP18 contains 128 bytes of Internal Data RAM at addresses 0x00-0x7F and a partially populated SFR space at addresses 0x80-0xFF. An additional 128 indirectly-addressed bytes of Internal Data RAM (sometimes called 'IDATA') are also available at addresses 0x80-0xFF.

All other on-chip MoBL-USB FX2LP18 RAM (program/data memory, endpoint buffer memory, and the MoBL-USB FX2LP18 control registers) is addressed as though it were off-chip 8051 memory. Firmware reads or writes these bytes as data using the MOVX ('move external') instruction, even though the RAM and register set is actually inside the MoBL-USB FX2LP18 chip.

## 11.9 IO Ports

The MoBL-USB FX2LP18 implements IO ports differently than a standard 8051, as described in Input/Output, on page 203.

The MoBL-USB FX2LP18 has up to five 8-bit wide, bidirectional IO ports. Each port is associated with a pair of registers.

- An 'OEx' register. It sets the input/output direction of each of the 8 port pins (0 = input, 1 = output).
- An 'IOx' register. Values written to IOx appear on the pins configured as outputs; values read from IOx indicate the states of the 8 pins, regardless of input/output configuration.

Most IO pins have alternate functions which are selected using configuration registers. When an alternate configuration is selected for an IO pin, the corresponding OEx bit is ignored (see section 13.2 IO Ports on page 203). The default (power-on reset) state of all IO ports is: alternate configurations 'off', all IO pins configured as 'inputs'.



Exhibit 2033 - Page 193 of 346

## 11.10 Interrupts

All standard 8051 interrupts, plus additional interrupts, are supported by the MoBL-USB FX2LP18. Table 11-4 lists the MoBL-USB FX2LP18 interrupts.

Table 11-4. MoBL-USB FX2LP18 Interrupts

| Standard 8051 Interrupts | Additional MoBL-USB FX2LP18 Interrupts | Source |
|---|---|---|
| INT0 | | Pin PA0 / INT0# |
| INT1 | | Pin PA1 / INT1# |
| Timer 0 | | Internal, Timer 0 |
| Timer 1 | | Internal, Timer 1 |
| Tx0 & Rx0 | | Internal, USART0 |
| | INT2 | Internal, USB |
| | INT3 | Internal, I²C Bus Controller |
| | INT4 | Pin INT4 (100-pin only) **OR** Internal, GPIF/FIFOs |
| | INT5 | Pin INT5# (100-pin only) |
| | INT6 | Pin INT6 (100-pin only) |
| | WAKEUP | Pin WAKEUP or Pin RA3/WU2 |
| | Tx1 & Rx1 | Internal, USART1 |
| | Timer 2 | Internal, Timer 2 |

The MoBL-USB FX2LP18 uses INT2 for 27 different USB interrupts. To help determine which interrupt is active, it provides a feature called Autovectoring, which dynamically changes the address pointed to by the 'jump' instruction at the INT2 vector address. This second level of vectoring automatically transfers control to the appropriate USB interrupt service routine (ISR). The MoBL-USB FX2LP18 interrupt system, including a full description of the Autovector mechanism, is the subject of the Interrupts chapter on page 59.

## 11.11 Power Control

The MoBL-USB FX2LP18 implements a low-power mode that allows it to be used in USB bus-powered devices (which are required by the USB specification to draw no more than 500 µA when suspended) and other low-power applications. The mechanism by which the MoBL-USB FX2LP18 enters and exits this low-power mode is described in detail in the Power Management chapter on page 83.



Exhibit 2033 - Page 194 of 346

## 11.12 Special Function Registers

The MoBL-USB FX2LP18 was designed to keep coding as standard as possible, to allow easy integration of existing 8051 software development tools. The MoBL-USB FX2LP18 Special Function Registers (SFR) are summarized in Table 11-5. Standard 8051 SFRs are shown in normal type and MoBL-USB FX2LP18-added SFRs are shown in **bold** type. Full details of the SFRs can be found in the Registers chapter on page 237.

Table 11-5. MoBL-USB FX2LP18 Special Function Registers (SFR)

| x | 8x | 9x | Ax | Bx | Cx | Dx | Ex | Fx |
|---|---|---|---|---|---|---|---|---|
| 0 | **IOA** | **IOB** | **IOC** | **IOD** | **SCON1** | PSW | ACC | B |
| 1 | SP | **EXIF** | **INT2CLR** | **IOE** | **SBUF1** | | | |
| 2 | DPL0 | **MPAGE** | **INT4CLR** | **OEA** | | | | |
| 3 | DPH0 | | | **OEB** | | | | |
| 4 | **DPL1** | | | **OEC** | | | | |
| 5 | **DPH1** | | | **OED** | | | | |
| 6 | **DPS** | | | **OEE** | | | | |
| 7 | PCON | | | | | | | |
| 8 | TCON | SCON0 | **IE** | **IP** | **T2CON** | **EICON** | **EIE** | **EIP** |
| 9 | TMOD | SBUF0 | | | | | | |
| A | TL0 | **AUTOPTRH1** | **EP2468STAT** | **EP01STAT** | **RCAP2L** | | | |
| B | TL1 | **AUTOPTRL1** | **EP24FIFOFLGS** | **GPIFTRIG** | **RCAP2H** | | | |
| C | TH0 | | **EP68FIFOFLGS** | | **TL2** | | | |
| D | TH1 | **AUTOPTRH2** | | **GPIFSGLDATH** | **TH2** | | | |
| E | **CKCON** | **AUTOPTRL2** | | **GPIFSGLDATLX** | | | | |
| F | | | **AUTOPTRSETUP** | **GPIFSGLDATLNOX** | | | | |

All un-labeled SFRs are reserved.

## 11.13 Reset

The various MoBL-USB FX2LP18 resets and their effects are described in the Resets chapter on page 89.

Exhibit 2033 - Page 196 of 346

# 12. Instruction Set

## 12.1 Introduction

This chapter provides a technical overview and description of the MoBL-USB FX2LP18's assembly-language instruction set.

All MoBL-USB FX2LP18 instructions are binary-code-compatible with the standard 8051. The MoBL-USB FX2LP18 instructions affect bits, flags, and other status functions just as the 8051 instructions do. Instruction timing, however, is different both in terms of the number of clock cycles per instruction cycle and the number of instruction cycles used by each instruction.

Table 12-2 on page 198 lists the MoBL-USB FX2LP18 instruction set and the number of instruction cycles required to complete each instruction. Table 12-1 defines the symbols and mnemonics used in Table 12-2.

Table 12-1. Legend for Instruction Set Table

| Symbol | Function |
|---|---|
| A | Accumulator |
| Rn | Register (R0–R7, in the bank selected by RS1:RS0) |
| direct | Internal RAM location (0x00 - 0x7F in the 'Lower 128', or 0x80 - 0xFF in 'SFR' space) |
| @Ri | Internal RAM location (0x00 - 0x7F in the 'Lower 128', or 0x80 - 0xFF in the 'Upper 128') pointed to by R0 or R1 |
| rel | Program-memory offset (-128 to +127 bytes relative to the first byte of the following instruction). Used by conditional jumps and SJMP. |
| bit | Bit address (0x20 - x2F in the 'Lower 128,' and SFRs 0x80, 0x88, …, 0xF0, 0xF8) |
| #data | 8-bit constant (0 - 255) |
| #data16 | 16-bit constant (0 - 65535) |
| addr16 | 16-bit destination address; used by LCALL and LJMP, which branch anywhere in program memory |
| addr11 | 11-bit destination address; used by ACALL and AJMP, which branch only within the current 2K page of program memory (that is, the upper 5 address bits are copied from the PC) |
| PC | Program Counter; holds the address of the currently-executing instruction. For the purposes of 'ACALL', 'AJMP', and 'MOVC A,@A+PC' instructions, the PC holds the address of the first byte of the instruction *following* the currently-executing instruction. |



**Exhibit 2033 - Page 197 of 346**

Table 12-2. MoBL-USB FX2LP18 Instruction Set

| Mnemonic | Description | Bytes | Cycles | PSW Flags Affected | Opcode (Hex) |
|---|---|---|---|---|---|
| **Arithmetic** | | | | | |
| ADD A, Rn | Add register to A | 1 | 1 | CY OV AC | 28-2F |
| ADD A, direct | Add direct byte to A | 2 | 2 | CY OV AC | 25 |
| ADD A, @Ri | Add data memory to A | 1 | 1 | CY OV AC | 26-27 |
| ADD A, #data | Add immediate to A | 2 | 2 | CY OV AC | 24 |
| ADDC A, Rn | Add register to A with carry | 1 | 1 | CY OV AC | 38-3F |
| ADDC A, direct | Add direct byte to A with carry | 2 | 2 | CY OV AC | 35 |
| ADDC A, @Ri | Add data memory to A with carry | 1 | 1 | CY OV AC | 36-37 |
| ADDC A, #data | Add immediate to A with carry | 2 | 2 | CY OV AC | 34 |
| SUBB A, Rn | Subtract register from A with borrow | 1 | 1 | CY OV AC | 98-9F |
| SUBB A, direct | Subtract direct byte from A with borrow | 2 | 2 | CY OV AC | 95 |
| SUBB A, @Ri | Subtract data memory from A with borrow | 1 | 1 | CY OV AC | 96-97 |
| SUBB A, #data | Subtract immediate from A with borrow | 2 | 2 | CY OV AC | 94 |
| INC A | Increment A | 1 | 1 | | 04 |
| INC Rn | Increment register | 1 | 1 | | 08-0F |
| INC direct | Increment direct byte | 2 | 2 | | 05 |
| INC @ Ri | Increment data memory | 1 | 1 | | 06-07 |
| DEC A | Decrement A | 1 | 1 | | 14 |
| DEC Rn | Decrement Register | 1 | 1 | | 18-1F |
| DEC direct | Decrement direct byte | 2 | 2 | | 15 |
| DEC @Ri | Decrement data memory | 1 | 1 | | 16-17 |
| INC DPTR | Increment data pointer | 1 | 3 | | A3 |
| MUL AB | Multiply A and B (unsigned; product in B:A) | 1 | 5 | CY=0 OV | A4 |
| DIV AB | Divide A by B<br>(unsigned; quotient in A, remainder in B) | 1 | 5 | CY=0 OV | 84 |
| DA A | Decimal adjust A | 1 | 1 | CY | D4 |
| **Logical** | | | | | |
| ANL, Rn | AND register to A | 1 | 1 | | 58-5F |
| ANL A, direct | AND direct byte to A | 2 | 2 | | 55 |
| ANL A, @Ri | AND data memory to A | 1 | 1 | | 56-57 |
| ANL A, #data | AND immediate to A | 2 | 2 | | 54 |
| ANL direct, A | AND A to direct byte | 2 | 2 | | 52 |
| ANL direct, #data | AND immediate data to direct byte | 3 | 3 | | 53 |
| ORL A, Rn | OR register to A | 1 | 1 | | 48-4F |
| ORL A, direct | OR direct byte to A | 2 | 2 | | 45 |
| ORL A, @Ri | OR data memory to A | 1 | 1 | | 46-47 |
| ORL A, #data | OR immediate to A | 2 | 2 | | 44 |
| ORL direct, A | OR A to direct byte | 2 | 2 | | 42 |
| ORL direct, #data | OR immediate data to direct byte | 3 | 3 | | 43 |
| XRL A, Rn | Exclusive-OR register to A | 1 | 1 | | 68-6F |
| XRL A, direct | Exclusive-OR direct byte to A | 2 | 2 | | 65 |
| XRL A, @Ri | Exclusive-OR data memory to A | 1 | 1 | | 66-67 |
| XRL A, #data | Exclusive-OR immediate to A | 2 | 2 | | 64 |
| XRL direct, A | Exclusive-OR A to direct byte | 2 | 2 | | 62 |
| XRL direct, #data | Exclusive-OR immediate to direct byte | 3 | 3 | | 63 |
| CLR A | Clear A | 1 | 1 | | E4 |
| CPL A | Complement A | 1 | 1 | | F4 |



Exhibit 2033 - Page 198 of 346

Table 12-2. MoBL-USB FX2LP18 Instruction Set *(continued)*

| Mnemonic | Description | Bytes | Cycles | PSW Flags Affected | Opcode (Hex) |
|---|---|---|---|---|---|
| SWAP A | Swap nibbles of a | 1 | 1 | | C4 |
| RL A | Rotate A left | 1 | 1 | | 23 |
| RLC A | Rotate A left through carry | 1 | 1 | CY | 33 |
| RR A | Rotate A right | 1 | 1 | | 03 |
| RRC A | Rotate A right through carry | 1 | 1 | CY | 13 |
| **Data Transfer** | | | | | |
| MOV A, Rn | Move register to A | 1 | 1 | | E8-EF |
| MOV A, direct | Move direct byte to A | 2 | 2 | | E5 |
| MOV A, @Ri | Move data byte at Ri to A | 1 | 1 | | E6-E7 |
| MOV A, #data | Move immediate to A | 2 | 2 | | 74 |
| MOV Rn, A | Move A to register | 1 | 1 | | F8-FF |
| MOV Rn, direct | Move direct byte to register | 2 | 2 | | A8-AF |
| MOV Rn, #data | Move immediate to register | 2 | 2 | | 78-7F |
| MOV direct, A | Move A to direct byte | 2 | 2 | | F5 |
| MOV direct, Rn | Move register to direct byte | 2 | 2 | | 88-8F |
| MOV direct, direct | Move direct byte to direct byte | 3 | 3 | | 85 |
| MOV direct, @Ri | Move data byte at Ri to direct byte | 2 | 2 | | 86-87 |
| MOV direct, #data | Move immediate to direct byte | 3 | 3 | | 75 |
| MOV @Ri, A | MOV A to data memory at address Ri | 1 | 1 | | F6-F7 |
| MOV @Ri, direct | Move direct byte to data memory at address Ri | 2 | 2 | | A6-A7 |
| MOV @Ri, #data | Move immediate to data memory at address Ri | 2 | 2 | | 76-77 |
| MOV DPTR, #data16 | Move 16-bit immediate to data pointer | 3 | 3 | | 90 |
| MOVC A, @A+DPTR | Move code byte at address DPTR+A to A | 1 | 3 | | 93 |
| MOVC A, @A+PC | Move code byte at address PC+A to A | 1 | 3 | | 83 |
| MOVX A, @Ri | Move external data at address Ri to A | 1 | 2-9* | | E2-E3 |
| MOVX A, @DPTR | Move external data at address DPTR to A | 1 | 2-9* | | E0 |
| MOVX @Ri, A | Move A to external data at address Ri | 1 | 2-9* | | F2-F3 |
| MOVX @DPTR, A | Move A to external data at address DPTR | 1 | 2-9* | | F0 |
| PUSH direct | Push direct byte onto stack | 2 | 2 | | C0 |
| POP direct | Pop direct byte from stack | 2 | 2 | | D0 |
| XCH A, Rn | Exchange A and register | 1 | 1 | | C8-CF |
| XCH A, direct | Exchange A and direct byte | 2 | 2 | | C5 |
| XCH A, @Ri | Exchange A and data memory at address Ri | 1 | 1 | | C6-C7 |
| XCHD A, @Ri | Exchange the low-order nibbles of A and data memory at address Ri | 1 | 1 | | D6-D7 |
| * Number of cycles is user-selectable. See “Stretch Memory Cycles (Wait States)” on page 200. | | | | | |
| **Boolean** | | | | | |
| CLR C | Clear carry | 1 | 1 | CY=0 | C3 |
| CLR bit | Clear direct bit | 2 | 2 | | C2 |
| SETB C | Set carry | 1 | 1 | CY=1 | D3 |
| SETB bit | Set direct bit | 2 | 2 | | D2 |
| CPL C | Complement carry | 1 | 1 | CY | B3 |
| CPL bit | Complement direct bit | 2 | 2 | | B2 |
| ANL C, bit | AND direct bit to carry | 2 | 2 | CY | 82 |
| ANL C, /bit | AND inverse of direct bit to carry | 2 | 2 | CY | B0 |
| ORL C, bit | OR direct bit to carry | 2 | 2 | CY | 72 |
| ORL C, /bit | OR inverse of direct bit to carry | 2 | 2 | CY | A0 |



**Exhibit 2033 - Page 199 of 346**

Table 12-2. MoBL-USB FX2LP18 Instruction Set *(continued)*

| Mnemonic | Description | Bytes | Cycles | PSW Flags Affected | Opcode (Hex) |
|---|---|---|---|---|---|
| MOV C, bit | Move direct bit to carry | 2 | 2 | CY | A2 |
| MOV bit, C | Move carry to direct bit | 2 | 2 | | 92 |
| **Branching** | | | | | |
| ACALL addr11 | Absolute call to subroutine | 2 | 3 | | 11-F1 |
| LCALL addr16 | Long call to subroutine | 3 | 4 | | 12 |
| RET | Return from subroutine | 1 | 4 | | 22 |
| RETI | Return from interrupt | 1 | 4 | | 32 |
| AJMP addr11 | Absolute jump unconditional | 2 | 3 | | 01-E1 |
| LJMP addr16 | Long jump unconditional | 3 | 4 | | 02 |
| SJMP rel | Short jump (relative address) | 2 | 3 | | 80 |
| JC rel | Jump if carry = 1 | 2 | 3 | | 40 |
| JNC rel | Jump if carry = 0 | 2 | 3 | | 50 |
| JB bit, rel | Jump if direct bit = 1 | 3 | 4 | | 20 |
| JNB bit, rel | Jump if direct bit = 0 | 3 | 4 | | 30 |
| JBC bit, rel | Jump if direct bit = 1, then clear the bit | 3 | 4 | | 10 |
| JMP @ A+DPTR | Jump indirect to address DPTR+A | 1 | 3 | | 73 |
| JZ rel | Jump if accumulator = 0 | 2 | 3 | | 60 |
| JNZ rel | Jump if accumulator is non-zero | 2 | 3 | | 70 |
| CJNE A, direct, rel | Compare A to direct byte; jump if not equal | 3 | 4 | CY | B5 |
| CJNE A, #d, rel | Compare A to immediate; jump if not equal | 3 | 4 | CY | B4 |
| CJNE Rn, #d, rel | Compare register to immediate; jump if not equal | 3 | 4 | CY | B8-BF |
| CJNE @ Ri, #d, rel | Compare data memory to immediate; jump if not equal | 3 | 4 | CY | B6-B7 |
| DJNZ Rn, rel | Decrement register; jump if not zero | 2 | 3 | | D8-DF |
| DJNZ direct, rel | Decrement direct byte; jump if not zero | 3 | 4 | | D5 |
| **Miscellaneous** | | | | | |
| NOP | No operation | 1 | 1 | | 00 |
| There is an additional reserved opcode (A5) that performs the same function as NOP. All mnemonics are copyright 1980, Intel Corporation. | | | | | |

### 12.1.1 Instruction Timing

Instruction cycles in the MoBL-USB FX2LP18 are 4 clock cycles in length, as opposed to the 12 clock cycles per instruction cycle in the standard 8051. For full details of the instruction-cycle timing differences between the MoBL-USB FX2LP18 and the standard 8051, see section 11.3 Performance Overview on page 190.

In the standard 8051, all instructions except for MUL and DIV take one or two instruction cycles to complete. In the MoBL-USB FX2LP18, instructions can take between one and five instruction cycles to complete. For calculating the timing of software loops, use the 'Cycles' column from Table 12-2. The 'Bytes' column indicates the number of bytes occupied by each instruction.

By default, the MoBL-USB FX2LP18's timer/counters run at 12 clock cycles per increment so that timer-based events have the same timing as with the standard 8051. The timers can also be configured to run at 4 clock cycles per increment to take advantage of the higher speed of the MoBL-USB FX2LP18's CPU.

### 12.1.2 Stretch Memory Cycles (Wait States)

The MoBL-USB FX2LP18 can execute a MOVX instruction in 2 instruction cycles. The three LSBs of the Clock Control Register (CKCON, at SFR location 0x8E) control the stretch value; the stretch value should be set to zero. A stretch value of zero adds zero instruction cycles, resulting in MOVX instructions which execute in two instruction cycles. At power-on-reset, the stretch value defaults to one (three-cycle MOVX); for the fastest data memory access, MoBL-USB FX2LP18 software must explicitly set the stretch value to zero. The stretch value affects the width of the read/write strobe and all related timing. Table 12-3 lists the data memory access speeds for stretch value zero. MD2-0 are the three LSBs of the Clock Control Register (CKCON.2-0). The strobe width timing shown is typical.



**Exhibit 2033 - Page 200 of 346**

CPUCS.4:3 sets the basic clock reference for the MoBL-USB FX2LP18. These bits can be modified by firmware at any time. At power-on-reset, CPUCS.4:3 is set to ‘00’ (12 Mhz).

Table 12-3. Data Memory Stretch Value

| MD2 | MD1 | MD0 | MOVX Instruction Cycles | Read/Write Strobe Width (Clocks) | Strobe Width @ 12 MHz CPUCS.4:3 = 00 | Strobe Width @ 24 MHz CPUCS.4:3 = 01 | Strobe Width @ 48 MHz CPUCS.4:3 = 10 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 2 | 2 | 167 ns | 83.3 ns | 41.7 ns |

### 12.1.3 Dual Data Pointers

The MoBL-USB FX2LP18 employs dual data pointers to accelerate data memory block moves. The standard 8051 data pointer (DPTR) is a 16 bit pointer used to address external data RAM or peripherals. The MoBL-USB FX2LP18 maintains the standard data pointer as DPTR0 at the standard SFR locations 0x82 (DPL0) and 0x83 (DPH0); it is not necessary to modify existing code to use DPTR0.

The MoBL-USB FX2LP18 adds a second data pointer (DPTR1) at SFR locations 0x84 (DPL1) and 0x85 (DPH1). The SEL bit (bit 0 of the DPTR Select Register, DPS, at SFR 0x86), selects the active pointer. When SEL = 0, instructions that use the DPTR will use DPL0:DPH0. When SEL = 1, instructions that use the DPTR will use DPL1:DPH1. No other bits of the DPS SFR are used.

All DPTR-related instructions use the data pointer selected by the SEL Bit. Switching between the two data pointers by toggling the SEL bit relieves firmware from the burden of saving source and destination addresses when doing a block move; therefore, using dual data pointers provides significantly increased efficiency when moving large blocks of data.

The fastest way to toggle the SEL bit between the two data pointers is via the ‘INC DPS’ instruction, which toggles bit 0 of DPS between ‘0’ and ‘1’.

The SFR locations related to the dual data pointers are:

| | | |
|---|---|---|
| 0x82 | DPL0 | DPTR0 low byte |
| 0x83 | DPH0 | DPTR0 high byte |
| 0x84 | DPL1 | DPTR1 low byte |
| 0x85 | DPH1 | DPTR1 high byte |
| 0x86 | DPS | DPTR Select (Bit 0) |

### 12.1.4 Special Function Registers

The four SFRs listed below are related to CPU operation and program execution. Except for the Stack Pointer (SP), each of the registers is bit addressable.

| | | |
|---|---|---|
| 0x81 | SP | Stack Pointer |
| 0xD0 | PSW | Program Status Word |
| 0xE0 | ACC | Accumulator Register |
| 0xF0 | B | B Register |

Table 12-4 on page 202 lists the functions of the PSW bits.



Exhibit 2033 - Page 201 of 346

Table 12-4. PSW Register - SFR 0xD0

| Bit | Function |
| --- | --- |
| PSW.7 | **CY** - Carry flag. This is the **unsigned** carry bit. The CY flag is set when an arithmetic operation results in a carry from bit 7 to bit 8, and cleared otherwise. In other words, it acts as a virtual bit 8. The CY flag is cleared on multiplication and division. See the 'PSW Flags Affected' column in Table 12-2 on page 198. |
| PSW.6 | **AC** - Auxiliary carry flag. Set to 1 when the last arithmetic operation resulted in a carry into (during addition) or borrow from (during subtraction) the high order nibble, otherwise cleared to 0 by all arithmetic operations. See the 'PSW Flags Affected' column in Table 12-2 on page 198. |
| PSW.5 | **F0** - User flag 0. Available to firmware for general purpose. |
| PSW.4<br>PSW.3 | **RS1** - Register bank select bit 1.<br>**RS0** - Register bank select bit 0.<br>RS1:RS0 select a register bank in internal RAM:<br>RS1 RS0 Bank Selected<br>0 0 Register bank 0, addresses 0x00-0x07<br>0 1 Register bank 1, addresses 0x08-0x0F<br>1 0 Register bank 2, addresses 0x10-0x17<br>1 1 Register bank 3, addresses 0x18-0x1F |
| PSW.2 | **OV** - Overflow flag. This is the **signed** carry bit. The OV flag is set when a positive sum exceeds 0x7F or a negative sum (in two's complement notation) exceeds 0x80. After a multiply, OV = 1 if the result of the multiply is greater than 0xFF. After a divide, OV = 1 if a divide-by-0 occurred. See the 'PSW Flags Affected' column in Table 12-2 on page 198. |
| PSW.1 | **F1** - User flag 1. Available to firmware for general purpose. |
| PSW.0 | **P** - Parity flag. Contains the modulo-2 sum of the 8 bits in the accumulator (for example, set to '1' when the accumulator contains an odd number of '1' bits, set to '0' when the accumulator contains an even number of '1' bits). |



Exhibit 2033 - Page 202 of 346

# 13. Input/Output

## 13.1 Introduction

The 56-pin MoBL-USB FX2LP18 provides two input output systems:

- A set of programmable IO pins
- A programmable I²C bus controller

The 100 -pin package additionally provides two programmable USARTs, which are described in the Timers/Counters and Serial Interface chapter on page 217

The IO pins may be configured either for general-purpose IO or for alternate functions (GPIF address and data; FIFO data; USART, timer, and interrupt signals; and others). This chapter describes the usage of the pins in the general-purpose configuration, and the methods by which the pins may be configured for alternate functions.

This chapter also provides both the programming information for the I²C interface and the operating details of the EEPROM boot loader. The role of the boot loader is described in the Enumeration and ReNumeration™ chapter on page 51.

## 13.2 IO Ports

The MoBL-USB FX2LP18's IO ports are implemented differently than those of a standard 8051.

The MoBL-USB FX2LP18 has up to five eight-pin bidirectional IO ports, labeled A, B, C, D, and E. Individual IO pins are labeled P*x.n*, where *x* is the port (A, B, C, D, or E) and *n* is the pin number (0 to 7).

The 100-pin MoBL-USB FX2LP18 package provide all five ports; the 56-pin package provides only ports A, B, and D.

Each port is associated with a pair of registers:

- An OE*x* register (where *x* is A, B, C, D, or E), which sets the input/output direction of each of the 8 pins (0 = input, 1 = output). See the OEx register on page 204.
- An IO*x* register (where *x* is A, B, C, D, or E). Values written to IO*x* appear on the pins which are configured as outputs; values read from IO*x* indicate the states of the 8 pins, regardless of input/output configuration. See IOx register on page 205.

Most IO pins have alternate functions which may be selected using configuration registers (see Tables Table 13-1 on page 207 through Table 13-9 on page 210). Each alternate function is unidirectional; the MoBL-USB FX2LP18 'knows' whether the function is an input or an output, so when an alternate configuration is selected for an IO pin, the corresponding OE*x* bit is ignored (see Figures Figure 13-2 on page 206 and Figure 13-3 on page 206).

The default (power-on reset) state of all IO ports is:

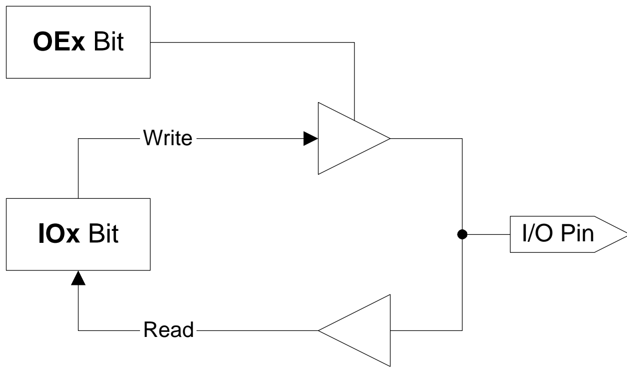
- Alternate configurations *off*
- All IO pins configured as *inputs*

Figure 13-1 on page 204 shows the basic structure of a MoBL-USB FX2LP18 IO pin.



**Exhibit 2033 - Page 203 of 346**

Figure 13-1. MoBL-USB FX2LP18 IO Pin




| OEA | | | Port A Output Enable | | | | SFR 0xB2 |
|---|---|---|---|---|---|---|---|
| b7 | b6 | b5 | v4 | v3 | b2 | b1 | b0 |
| **D7** | **D6** | **D5** | **D4** | **D3** | **D2** | **D1** | **D0** |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| OEB | | | Port B Output Enable | | | | SFR 0xB3 |
|---|---|---|---|---|---|---|---|
| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
| **D7** | **D6** | **D5** | **D4** | **D3** | **D2** | **D1** | **D0** |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| OEC | | | Port C Output Enable | | | | SFR 0xB4 |
|---|---|---|---|---|---|---|---|
| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
| **D7** | **D6** | **D5** | **D4** | **D3** | **D2** | **D1** | **D0** |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| OED | | | Port D Output Enable | | | | SFR 0xB5 |
|---|---|---|---|---|---|---|---|
| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
| **D7** | **D6** | **D5** | **D4** | **D3** | **D2** | **D1** | **D0** |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| OEE | | | Port E Output Enable | | | | SFR 0xB6 |
|---|---|---|---|---|---|---|---|
| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
| **D7** | **D6** | **D5** | **D4** | **D3** | **D2** | **D1** | **D0** |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |



**Exhibit 2033 - Page 204 of 346**

| IOA | | | Port A (Bit-Addressable) | | | | SFR 0x80 |
|---|---|---|---|---|---|---|---|
| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
| **D7** | **D6** | **D5** | **D4** | **D3** | **D2** | **D1** | **D0** |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| x | x | x | x | x | x | x | x |

| IOB | | | Port B (Bit-Addressable) | | | | SFR 0x90 |
|---|---|---|---|---|---|---|---|
| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
| **D7** | **D6** | **D5** | **D4** | **D3** | **D2** | **D1** | **D0** |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| x | x | x | x | x | x | x | x |

| IOC | | | Port C (Bit-Addressable) | | | | SFR 0xA0 |
|---|---|---|---|---|---|---|---|
| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
| **D7** | **D6** | **D5** | **D4** | **D3** | **D2** | **D1** | **D0** |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| x | x | x | x | x | x | x | x |

| IOD | | | Port D (Bit-Addressable) | | | | SFR 0xB0 |
|---|---|---|---|---|---|---|---|
| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
| **D7** | **D6** | **D5** | **D4** | **D3** | **D2** | **D1** | **D0** |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| x | x | x | x | x | x | x | x |

| IOE | | | Port E | | | | SFR 0xB1 |
|---|---|---|---|---|---|---|---|
| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
| **D7** | **D6** | **D5** | **D4** | **D3** | **D2** | **D1** | **D0** |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| x | x | x | x | x | x | x | x |



**Exhibit 2033 - Page 205 of 346**

## 13.3 IO Port Alternate Functions

Each IO pin may be configured for an *alternate* (for example, non-general-purpose IO) function. These alternate functions are selected through various configuration registers, as described in the following sections.

The IO-pin logic for alternate-function outputs is slightly different than for alternate-function inputs, as shown in Figures 13-2 (output) and 13-3 (input).

Figure 13-2. IO-Pin Logic when Alternate Function is an OUTPUT

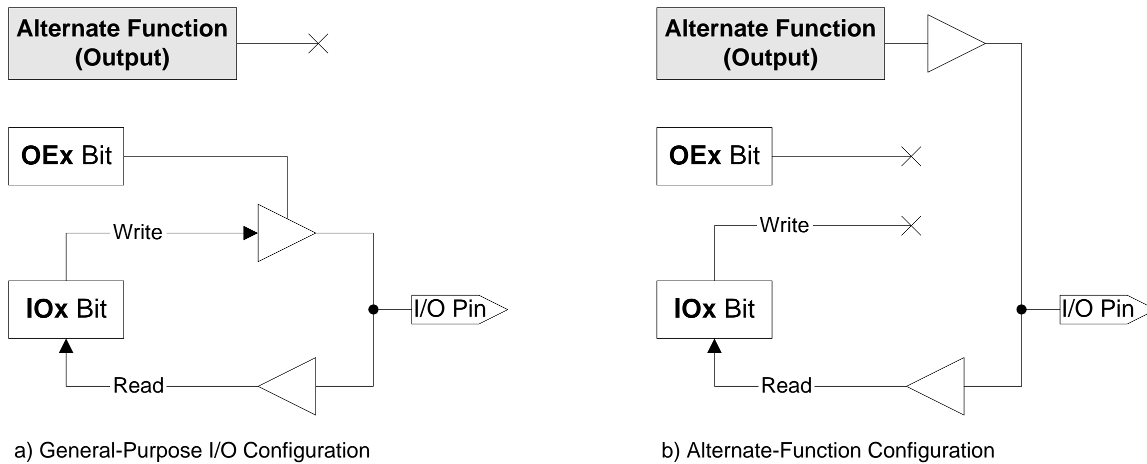



Figure 13-2 shows an IO pin whose alternate function is always an *output*.

In Figure 13-2a, the IO pin is configured for general-purpose IO. In this configuration, the alternate function is disconnected and the pin functions normally.

In Figure 13-2b, the IO pin is configured as an alternate-function output. In this configuration, the IOx/OEx output buffer is disconnected from the IO pin, so writes to IOx and OEx have no effect on the IO pin. *Reads* from IOx, however, continue to work normally; the state of the IO pin (and, therefore, the state of the alternate function) is always available.

Figure 13-3. IO-Pin Logic when Alternate Function is an INPUT

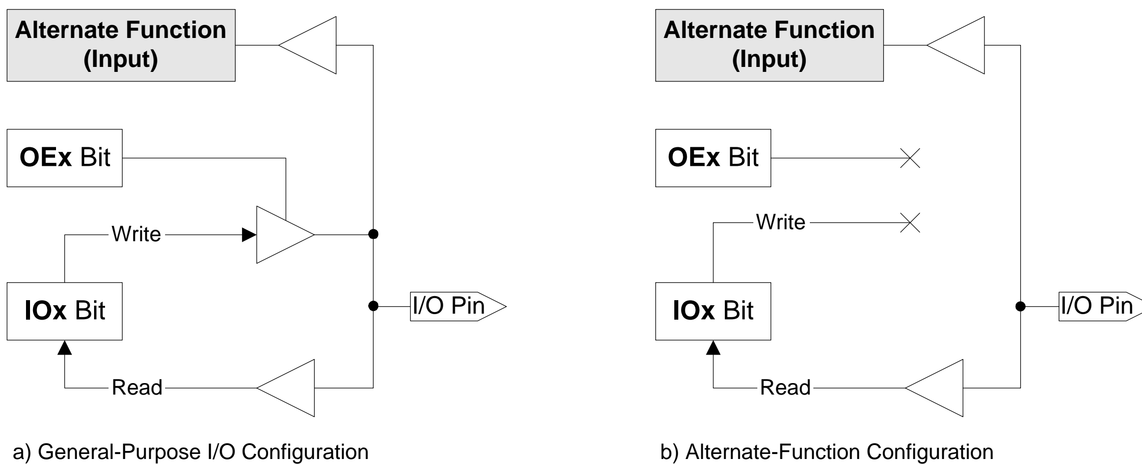



Figure 13-3 shows an IO pin whose alternate function is always an *input*.



**Exhibit 2033 - Page 206 of 346**

In Figure 13-3a, the IO pin is configured for general-purpose IO. There's an important difference between alternate-function *inputs* and the alternate-function *outputs* shown earlier in Figure 13-2 on page 206: *Alternate-function inputs are never disconnected; they're always listening.*

If the alternate function's interrupt is enabled, signals on the IO pin may trigger that interrupt. If the pin is to be used only for general purpose IO, the alternate function's interrupt must be disabled.

For example, suppose the PE5/INT6 pin is configured for general-purpose IO. Since the INT6 function is an input, the pin signal is also routed to the internal INT6 logic. If the INT6 interrupt is enabled, traffic on the PE5 pin will trigger an INT6 interrupt. If this is undesirable, the INT6 interrupt should be disabled.

Of course, this side-effect can be useful in certain situations. In the case of PE5/INT6, for example, PE5 can trigger an INT6 interrupt even if the IO pin is configured as an output (for example, OEE.5 = 1), so the firmware can directly generate 'external' interrupts.

In Figure 13-3b, the IO pin is configured as an alternate-function input. Just as with alternate-function outputs, the IOx/OEx output buffer is disconnected from the IO pin, so writes to IOx and OEx have no effect on the IO pin. *Reads* from IOx, however, continue to work normally; the state of the IO pin (and, therefore, the input to the alternate function) is always available.

## 13.3.1 Port A Alternate Functions

Alternate functions for the Port A pins are selected by bits in three registers, as shown in Tables 13-1 and 13-2.

Table 13-1. Register Bits that Select Port A Alternate Functions

| | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
|---|---|---|---|---|---|---|---|---|
| **PORTACFG** (0xE670) | **FLAGD** | **SLCS**[1] | 0 | 0 | 0 | 0 | **INT1** | **INT0** |
| **IFCONFIG** (0xE601) | IFCLKSRC | 3048MHZ | IFCLKOE | IFCLKPOL | ASYNC | GSTATE | **IFCFG1** | **IFCFG0** |
| **WAKEUPCS** (0xE682) | WU2 | WU | WU2POL | WUPOL | 0 | DPEN | **WU2EN** | WUEN |

**Note 1**:Although the SLCS alternate function is selected by bit 6 of PORTACFG, that function does not appear on pin PA6. Instead, the SLCS function appears on pin PA7 (see Table 13-2).

Table 13-2. Port A Alternate Function Configuration

| Port A Pin | Alternate Function | Alternate Function is Selected By... | Alternate Function is Described in... |
|---|---|---|---|
| PA.0 | INT0 | PORTACFG.0 = 1 | See chapter "Interrupts" on page 59 |
| PA.1 | INT1 | PORTACFG.1 = 1 | See chapter "Interrupts" on page 59 |
| PA.2 | SLOE | IFCFG1:0 = 11 | See chapter "Slave FIFOs" on page 107 |
| PA.3 | WU2[1] | WU2EN = 1 | See chapter "Power Management" on page 83 |
| PA.4 | FIFOADR0 | IFCFG1:0 = 11 | See chapter "Slave FIFOs" on page 107 |
| PA.5 | FIFOADR1 | IFCFG1:0 = 11 | See chapter "Slave FIFOs" on page 107 |
| PA.6 | PKTEND | IFCFG1:0 = 11 | See chapter "Slave FIFOs" on page 107 |
| PA.7 | FLAGD[2]<br>SLCS[3] | PORTACFG.7 = 1<br>PORTACFG.6 = 1 and IFCFG1:0 = 11 | See chapter "Slave FIFOs" on page 107<br>See chapter "Slave FIFOs" on page 107 |

**Note 1**:When PA.3 is configured for alternate function WU2, it continues to function as a general-purpose input pin as well. See section 6.4.1 WU2 Pin on page 88 for more information.
**Note 2**:Although PA.7's alternate function FLAGD is selected via the PORTACFG register, the state of the FLAGD output is undefined unless IFCFG1:0 = 11.
**Note 3**:FLAGD takes priority over SLCS if PORTACFG.6 and PORTACFG.7 are both set to '1'.



Exhibit 2033 - Page 207 of 346

## 13.3.2 Port B and Port D Alternate Functions

When IFCFG1 = 1, all eight Port B pins are configured for an alternate configuration (FIFO Data 7:0).

If *any* of the FIFOs are set to 16-bit mode (via the WORDWIDE bits in the EPxFIFOCFG registers), all eight Port D pins are also configured for an alternate configuration (FIFO Data 15:8). See Tables 13-3, 13-4, and 13-5.

If **all** WORDWIDE bits are cleared to 0 (for example, if all four FIFOs are operating in 8-bit mode), the eight Port D pins may be used as general-purpose IO pins even if IFCFG1 = 1.

Table 13-3. Register Bits Which Select Port B and Port D Alternate Functions

| | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
|---|---|---|---|---|---|---|---|---|
| **IFCONFIG** (0xE601) | IFCLKSRC | 3048MHZ | IFCLKOE | IFCLKPOL | ASYNC | GSTATE | **IFCFG1** | IFCFG0 |
| **EP2FIFOCFG** (0xE618) | 0 | INFM2 | OEP2 | AUTOOUT | AUTOIN | ZEROLENIN | 0 | **WORDWIDE** |
| **EP4FIFOCFG** (0xE619) | 0 | INFM4 | OEP4 | AUTOOUT | AUTOIN | ZEROLENIN | 0 | **WORDWIDE** |
| **EP6FIFOCFG** (0xE61A) | 0 | INFM6 | OEP6 | AUTOOUT | AUTOIN | ZEROLENIN | 0 | **WORDWIDE** |
| **EP8FIFOCFG** (0xE61B) | 0 | INFM8 | OEP8 | AUTOOUT | AUTOIN | ZEROLENIN | 0 | **WORDWIDE** |

Table 13-4. Port B Alternate-Function Configuration

| Port B Pin | Alternate Function | Alternate Function is Selected By... | Alternate Function is Described in... |
|---|---|---|---|
| PB.7:0 | FD[7:0] | IFCFG1 = 1 | See chapter “Slave FIFOs” on page 107 |

Table 13-5. Port D Alternate-Function Configuration

| Port D Pin | Alternate Function | Alternate Function is Selected By... | Alternate Function is Described in... |
|---|---|---|---|
| PD.7:0 | FD[15:8] | IFCFG1 = 1 and any WORDWIDE bit = 1 | See chapter “Slave FIFOs” on page 107 |



Exhibit 2033 - Page 208 of 346

### 13.3.3 Port C Alternate Functions

Each Port C pin may be individually configured for an alternate function by setting a bit in the PORTCCFG register, as shown in Tables 13-6 and 13-7.

Table 13-6. Register Bits That Select Port C Alternate Functions

| | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
|---|---|---|---|---|---|---|---|---|
| **PORTCCFG (0xE671)** | **GPIFA7** | **GPIFA6** | **GPIFA5** | **GPIFA4** | **GPIFA3** | **GPIFA2** | **GPIFA1** | **GPIFA0** |

Table 13-7. Port C Alternate-Function Configuration

| Port C Pin | Alternate Function | Alternate Function is Selected By... | Alternate Function is Described in... |
|---|---|---|---|
| PC.0 | GPIFA0[1] | PORTCCFG.0 = 1 | See chapter “General Programmable Interface” on page 135. |
| PC.1 | GPIFA1[1] | PORTCCFG.1 = 1 | See chapter “General Programmable Interface” on page 135. |
| PC.2 | GPIFA2[1] | PORTCCFG.2 = 1 | See chapter “General Programmable Interface” on page 135. |
| PC.3 | GPIFA3[1] | PORTCCFG.3 = 1 | See chapter “General Programmable Interface” on page 135. |
| PC.4 | GPIFA4[1] | PORTCCFG.4 = 1 | See chapter “General Programmable Interface” on page 135. |
| PC.5 | GPIFA5[1] | PORTCCFG.5 = 1 | See chapter “General Programmable Interface” on page 135. |
| PC.6 | GPIFA6[1] | PORTCCFG.6 = 1 | See chapter “General Programmable Interface” on page 135. |
| PC.7 | GPIFA7[1] | PORTCCFG.7 = 1 | See chapter “General Programmable Interface” on page 135. |

**Note 1**:Although the Port C alternate functions GPIFA0:7 are selected via the PORTCCFG register, the states of the GPIFA0:7 outputs are undefined unless IFCFG1:0 = 10.

### 13.3.4 Port E Alternate Functions

Each Port E pin may be individually configured for an alternate function by setting a bit in the PORTECFG register.

If the GSTATE bit in the IFCONFIG register is set to '1', the PE.2:0 pins are automatically configured as GPIF Status pins GSTATE[2:0], regardless of the PORTECFG.2:0 settings. In other words, GSTATE overrides PORTECFG.2:0. See Tables 13-8 and 13-9.

Table 13-8. Register Bits That Select Port E Alternate Functions

| | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
|---|---|---|---|---|---|---|---|---|
| **PORTECFG (0xE671)** | **GPIFA8** | **T2EX** | **INT6** | **RXD1OUT** | **RXD0OUT** | **T2OUT** | **T1OUT** | **T0OUT** |
| **IFCONFIG (0xE601)** | IFCLKSRC | 3048MHZ | IFCLKOE | IFCLKPOL | ASYNC | **GSTATE** | IFCFG1 | IFCFG0 |

Table 13-9. Port E Alternate-Function Configuration

| Port E Pin | Alternate Function | Alternate Function is Selected By... | Alternate Function is Described in... |
|---|---|---|---|
| PE.0 | T0OUT[1] | PORTECFG.0 = 1 **and** GSTATE = 0 | See chapter "Timers/Counters and Serial Interface" on page 217 |
| PE.1 | T1OUT[1] | PORTECFG.1 = 1 **and** GSTATE = 0 | See chapter "Timers/Counters and Serial Interface" on page 217 |
| PE.2 | T2OUT[1] | PORTECFG.2 = 1 **and** GSTATE = 0 | See chapter "Timers/Counters and Serial Interface" on page 217 |
| PE.3 | RXD0OUT | PORTECFG.3 = 1 | See chapter "Timers/Counters and Serial Interface" on page 217 |
| PE.4 | RXD1OUT | PORTECFG.4 = 1 | See chapter "Timers/Counters and Serial Interface" on page 217 |
| PE.5 | INT6 | PORTECFG.5 = 1 | See chapter "Interrupts" on page 59 |
| PE.6 | T2EX | PORTECFG.6 = 1 | See chapter "Timers/Counters and Serial Interface" on page 217 |
| PE.7 | GPIFA8[2] | PORTECFG.7 = 1 | See chapter "General Programmable Interface" on page 135 |

**Note 1**:If GSTATE is set to '1', these settings are overridden and PE.2:0 are all automatically configured as GPIF Status pins (see Chapter 10).
**Note 2**:Although the PE.7 alternate function GPIFA8 is selected via the PORTECFG register, the state of the GPIFA8 output is undefined unless IFCFG1:0 = 10.



Exhibit 2033 - Page 210 of 346

Table 13-10. IFCFG Selection of Port IO Pin Functions

| IFCFG1:0 = 00 (Ports) | IFCFG1:0 = 10 (GPIF Master) | IFCFG1:0 = 11 (Slave FIFO) |
|---|---|---|
| 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] |
| **INT0# / PA0** | **INT0# / PA0** | **INT0# / 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 / SLCS# |
| | | |
| **PC7:0** | **PC7:0** | **PC7:0** |
| **PE7:0** | **PE7:0** | **PE7:0** |
| **Note**: Signals shown in bold type do not change with IFCFG; they are shown for completeness. | | |



**Exhibit 2033 - Page 211 of 346**

## 13.4 I²C Bus Controller

The I2C bus controller uses the SCL (Serial Clock) and SDA (Serial Data) pins, and performs two functions:

- General-purpose interfacing to I²C peripherals
- Boot loading from a serial EEPROM

The I2C pins SCL and SDA must have external 2.2K ohm pull up resistors **even if no EEPROM is connected to the MoBL-USB FX2LP18**. The value of the pull up resistors required may vary, depending on the combination of VCC_IO and the supply used for the EEPROM. The pull up resistors used must be such that when the EEPROM pulls SDA low, the voltage level meets the VIL specification of the MoBL-USB FX2LP18. For example, if the EEPROM runs off a 3.3V supply and VCC_IO is 1.8V, the pull up resistors recommended are 10K ohm. This requirement may also vary depending on the devices being run on the I2C pins. Refer to the I2C specifications for details.

The bus frequency defaults to approximately 100 kHz for compatibility, and it can be configured to run at 400 kHz for devices that support the higher speed.

### 13.4.1 Interfacing to I²C Peripherals

Figure 13-4. General I²C Transfer

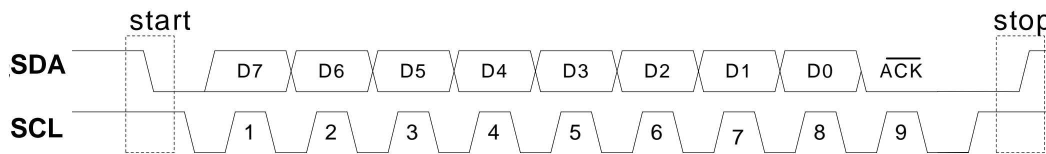
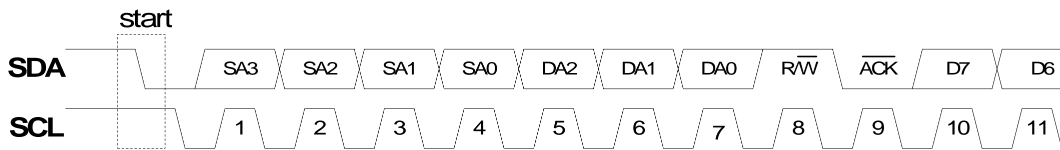



Figure 13-4 illustrates the waveforms for an I²C transfer. SCL and SDA are open-drain pins which must be pulled up to Vcc with external resistors. The MoBL-USB FX2LP18 is a bus master only, meaning that it generates clock pulses on SCL. Once the master drives SCL low, external slave devices can hold SCL low to extend clock-cycle times.

Serial data (SDA) is permitted to change state only while SCL is low, and must be valid while SCL is high. Two exceptions to this rule are the 'start' and 'stop' conditions. A 'start' condition is defined as a high-to-low transition on SDA while SCL is high, and a 'stop' condition is defined as a low-to-high transition on SDA while SCL is high. Data is sent MSB first. During the last bit time (clock #9 in Figure 13-4), the transmitting device floats the SDA line to allow the receiving device to acknowledge the transfer by pulling SDA low.

| Multiple Bus Masters | The MoBL-USB FX2LP18 acts only as a bus master, never as a slave. Conflicts with a second master can be detected, however, by checking for BERR=1 (see section 13.4.2.2 Status Bits on page 214). |
|---|---|

Figure 13-5. Addressing an I²C Peripheral




Each peripheral (slave) device on the I²C bus has a unique address. The first byte of an I²C transaction contains the address of the desired peripheral. Figure 13-5 shows the format for this first byte, which is sometimes called a *control* byte.

The MoBL-USB FX2LP18 sends the bit sequence shown in Figure 13-5 to select the peripheral at a particular address, to establish the transfer direction (using R/$\overline{W}$), and to determine if the peripheral is present by testing for $\overline{ACK}$.

The four most significant bits (SA3:0) are the peripheral chip's slave address. I²C devices are internally hard wired to pre-assigned slave addresses by device type. EEPROMs, for instance, are assigned slave address 1010. The next three bits (DA2:0) usually reflect the states of the device's address pins. A device with three address pins can be strapped to one of eight distinct addresses, which allows, for example, up to eight serial EEPROMs to be individually addressed on one I²C bus.

The eighth bit (R/$\overline{W}$) sets the direction for the data transfer to follow (1 = master read, 0 = master write). Most address transfers are followed by one or more data transfers, with the 'stop' condition generated after the last data byte is transferred.

In Figure 13-5, the master clocks the 7-bit slave/device address out on SDA, then sets the R/$\overline{W}$ bit high at clock 8, indicating that a read transfer will follow this address byte. At clock 9, the master releases SDA and treats it as an input; the peripheral device responds to its address by asserting $\overline{ACK}$. At clock 10, the master begins to clock in data from the slave on the SDA pin.

## 13.4.2 Registers

The three registers shown in this section are used to conduct transfers over the I²C bus.

I2CTL configures the bus. Data is transferred to and from the bus through the I2DAT register. The I2CS register controls the transfers and reports various status conditions.

Writing to I2DAT initiates a write transfer on the I²C bus; the value written to I2DAT will be transferred. Reading from I2DAT will retrieve the data that was transferred in the previous read transfer and (with one exception) initiate another read transfer. To retrieve data from the previous read transfer without initiating another transfer, I2DAT must be read *during* the generation of the 'stop' condition. See Step 13 of section 13.4.4 Receiving Data on page 215 for details.

| **I2CS** | | | **I²C Bus Control and Status** | | | | **E678** |
|---|---|---|---|---|---|---|---|
| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
| **START** | **STOP** | **LASTRD** | **ID1** | **ID0** | **BERR** | **ACK** | **DONE** |
| R/W | R/W | R/W | R | R | R | R | R |
| 0 | 0 | 0 | x | x | 0 | 0 | 0 |

| **I2DAT** | | | **I²C Bus Data** | | | | **E679** |
|---|---|---|---|---|---|---|---|
| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
| **D7** | **D6** | **D5** | **D4** | **D3** | **D2** | **D1** | **D0** |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| x | x | x | x | x | x | x | x |

| **I2CTL** | | | **I²C Bus Mode** | | | | **E67A** |
|---|---|---|---|---|---|---|---|
| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
| **0** | **0** | **0** | **0** | **0** | **0** | **STOPIE** | **400KHZ** |
| R | R | R | R | R | R | R/W | R/W |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |



Exhibit 2033 - Page 213 of 346

### 13.4.2.1 Control Bits

#### START

When START = 1, the next write to I2DAT will generate a 'start' condition followed by the byte of data written to I2DAT. The START bit is automatically cleared to 0 after the ACK interval (clock 9 in Figure 13-4 on page 212).

#### STOP

When STOP = 1 after the ACK interval (clock 9 in Figure 13-4 on page 212), a 'stop' condition is generated. STOP may be set by firmware at any time before, during, or after the 9-bit data transaction. STOP is automatically cleared after the 'stop' condition is generated.

An interrupt request is available to signal that the stop condition is complete; see "STOPIE", below.

#### LASTRD

An I²C master reads data by floating the SDA line and issuing clock pulses on the SCL line. After every eight bits, the master drives SDA low for one clock to indicate ACK. To signal the last byte of a multi-byte transfer, the master *floats* SDA at ACK time to instruct the slave to stop sending.

When LASTRD = 1 at ACK time, the MoBL-USB FX2LP18 will float the SDA line. The LASTRD bit may be set at any time before or during the data transfer; it's automatically cleared after the ACK interval.

**Note** Setting LASTRD does not automatically generate a 'stop' condition. At the end of a read transfer, the STOP bit should also be set.

#### STOPIE

Setting this bit to '1' enables the STOP bit Interrupt Request, which is activated when the STOP bit makes a 1-to-0 transition (for example, when generation of a 'stop' condition is complete).

#### 400KHZ

When this bit is at its default value of 0, SCL will be driven at approximately 100 kHz. Setting this bit to '1' causes the MoBL-USB FX2LP18 to drive SCL at approximately 400 kHz.

### 13.4.2.2 Status Bits

After each transaction's ACK interval, the MoBL-USB FX2LP18 updates the three status bits DONE, ACK, and BERR.

#### DONE

The MoBL-USB FX2LP18 sets this bit whenever it completes a byte transfer. The MoBL-USB FX2LP18 also generates an interrupt request when it sets the DONE bit. The DONE bit is automatically cleared when the I2DAT register is read or written, and the interrupt request bit is automatically cleared by reading or writing the I2CS or I2DAT registers (or by clearing EXIF.5 to 0).

#### ACK

During the ninth clock of a write transfer, the slave indicates reception of the byte by driving SDA low to acknowledge the byte it just received. The MoBL-USB FX2LP18 floats SDA during this time, samples the SDA line, and updates the ACK bit with the complement of the detected value: ACK=1 indicates that the slave acknowledged the transfer, and ACK=0 indicates the slave did not.

The ACK bit is only meaningful after a write transfer. After a read transfer, its state should be ignored.

#### BERR

This bit indicates a bus error. BERR=1 indicates that there was bus contention during the preceding transfer, which results when an outside device drives the bus when it should not, or when another bus master wins arbitration and takes control of the bus.



Exhibit 2033 - Page 214 of 346

When a bus error is detected, the MoBL-USB FX2LP18 immediately cancels its current transfer, floats the SCL and SDA lines, and sets DONE and BERR. BERR will remain set until it's updated at the next ACK interval. The I2C master will not drive SCL until BERR is reset. If the bus error causes the master to detect bus contention and the slave to be stuck in the middle of a transfer, then there is no built-in contention resolution method to work around this deadlock. If there is a possibility of this condition then the design must implement a method of resetting the slave or clocking the slave until the next ACK.

DONE is set with BERR only when bus contention occurs **during** a transfer. If BERR is set and the bus is still busy when firmware attempts to restart a transfer, that transfer request will be ignored and the DONE flag will **not** be set. If contention is expected, therefore, firmware should incorporate a time-out to test for this condition. See Steps 1 and 3 of sections Section 13.4.3 and Section 13.4.4.

**ID1, ID0**

These bits are automatically set by the boot loader to indicate the Boot EEPROM addressing mode. They are normally used only for debug purposes; for full details, see section 13.5 EEPROM Boot Loader on page 216.

### 13.4.3 Sending Data

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

1. Set START=1. If BERR=1, start timer*.
2. Write the 7-bit peripheral address and the direction bit (0 for a write) to I2DAT.
3. Wait for DONE=1 or for timer to expire*. If BERR=1, go to step 1.
4. If ACK=0, go to step 9.
5. Load I2DAT with a data byte.
6. Wait for DONE=1*. If BERR=1, go to step 1.
7. If ACK=0, go to step 9.
8. Repeat steps 5-7 for each byte until all bytes have been transferred.
9. Set STOP=1. Wait for STOP = 0 before initiating another transfer.

* The time-out should be at least as long as the longest expected Start-to-Stop interval on the bus.

### 13.4.4 Receiving Data

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

1. Set START=1. If BERR = 1, start timer*.
2. Write the 7-bit peripheral address and the direction bit (1 for a read) to I2DAT.
3. Wait for DONE=1 or for timer to expire*. If BERR=1, go to step 1.
4. If ACK=0, set STOP=1 and go to step 15.
5. Read I2DAT to initiate the first burst of nine SCL pulses to clock in the first byte from the slave. Discard the value that was read from I2DAT.
6. Wait for DONE=1. If BERR=1, go to step 1.
7. Read the just-received byte of data from I2DAT. This read also initiates the next read transfer.
8. Repeat steps 6 and 7 for each byte until ready to read the second-to-last byte.
9. Wait for DONE=1. If BERR=1, go to step 1.
10. Before reading the second-to-last I2DAT byte, set LASTRD=1.
11. Read the second-to-last byte from I2DAT. With LASTRD=1, this initiates the final byte read on the bus.
12. Wait for DONE=1. If BERR=1, go to step 1.
13. Set STOP=1.
14. Read the final byte from I2DAT immediately (the next instruction) after setting the STOP bit. By reading I2DAT *while* the 'stop' condition is being generated, the just-received data byte will be retrieved *without* initiating an extra read transaction (nine more SCL pulses) on the I²C bus.
15. Wait for STOP = 0 before initiating another transfer.

* The time-out should be at least as long as the longest expected Start-to-Stop interval on the bus.



Exhibit 2033 - Page 215 of 346

## 13.5 EEPROM Boot Loader

Whenever the MoBL-USB FX2LP18 is taken out of reset via the reset pin, its boot loader checks for the presence of an EEPROM on the I²C bus. If an EEPROM is detected, the loader reads the first EEPROM byte to determine how to enumerate. The various enumeration modes are described in the Enumeration and ReNumeration™ chapter on page 51.

The MoBL-USB FX2LP18 boot loader supports two I²C EEPROM types:

- EEPROMs with slave address 1010 that use an 8-bit internal address (for example, 24LC00, 24LC01/B, 24LC02/B).
- EEPROMs with slave address 1010 that use a 16-bit internal address (for example, 24AA64, 24LC128, 24AA256).

EEPROMs with densities up to 256 bytes require only a single address byte; larger EEPROMs require two address bytes. The MoBL-USB FX2LP18 must determine which EEPROM type is connected — one or two address bytes — so that it can properly read the EEPROM.

The MoBL-USB FX2LP18 uses the EEPROM device-address pins A2, A1, and A0 to determine whether to send out one or two bytes of address. As shown in Table 13-11, single-byte-address EEPROMs must be strapped to address 000, while double-byte-address EEPROMs must be strapped to address 001.

Table 13-11. Strap Boot EEPROM Address Lines to These Values

| Bytes | Example EEPROM | A2 | A1 | A0 |
|---|---|---|---|---|
| 16 | 24AA00* | N/A | N/A | N/A |
| 128 | 24AA01 | 0 | 0 | 0 |
| 256 | 24AA02 | 0 | 0 | 0 |
| 4K | 24AA32 | 0 | 0 | 1 |
| 8K | 24AA64 | 0 | 0 | 1 |
| 16K | 24AA128 | 0 | 0 | 1 |

* This EEPROM does not have device-address pins.

Additional EEPROM devices can be attached to the I²C bus for general-purpose use, as long as they are strapped for device addresses other than 000 or 001.

Many single-byte Address EEPROMs are special cases, because the EEPROM responds to all eight device addresses. If one of these EEPROMs used for boot loading, no other EEPROMS may share the bus. Consult your EEPROM data sheet for details

After determining whether a one- or two-byte-address EEPROM is attached, the MoBL-USB FX2LP18 reports its results in the ID1 and ID0 bits, as shown in Table 13-12.

Table 13-12. Results of Power-On_Reset EEPROM Test

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

The MoBL-USB FX2LP18 does not check for bus contention during the boot loading process; if another I²C master is sharing the bus, that master must not attempt to initiate any transfers while the boot loader is running.



Exhibit 2033 - Page 216 of 346

# 14. Timers/Counters and Serial Interface

## 14.1 Introduction

The MoBL-USB FX2LP18's timer/counters and serial interface are very similar to the standard 8051, with some differences and enhancements. This chapter provides technical information on configuring and using the timer/counters and serial interface.

## 14.2 Timers/Counters

The MoBL-USB FX2LP18 includes three timer/counters (Timer 0, Timer 1, and Timer 2). Each timer/counter can operate either as a timer with a clock rate based on the MoBL-USB FX2LP18's internal clock (CLKOUT) or as an event counter clocked by the T0 pin (Timer 0), T1 pin (Timer 1), or the T2 pin (Timer 2). Timers 1 and 2 may be used for baud clock generation for the serial interface (see section 14.3 Serial Interface on page 225 for details of the serial interface).

**Note** The MoBL-USB FX2LP18 can be configured to operate at 12, 24, or 48 MHz. In 'timer' mode, the timer/counters run at the same speed as the MoBL-USB FX2LP18, and they are not affected by the CLKOE and CLKINV configuration bits (CPUCS.1 and CPUCS.2).

Each timer/counter consists of a 16-bit register that is accessible to software as two SFRs:

- Timer 0 TH0 and TL0
- Timer 1 TH1 and TL1
- Timer 2 TH2 and TL2

### 14.2.1 803x/805x Compatibility

The implementation of the timers/counters is similar to that of the Dallas Semiconductor DS80C320. Table 14-1 summarizes the differences in timer/counter implementation between the Intel 8051, the Dallas Semiconductor DS80C320, and the MoBL-USB FX2LP18.

Table 14-1. Timer/Counter Implementation Comparison

| Feature | Intel 8051 | Dallas DS80C320 | MoBL-USB FX2LP18 |
|---|---|---|---|
| Number of timers | 2 | 3 | 3 |
| Timer 0/1 overflow available as output signals | No | No | Yes; T0OUT, T1OUT (one CLKOUT pulse) |
| Timer 2 output enable | n/a | Yes | Yes |
| Timer 2 down-count enable | n/a | Yes | No |
| Timer 2 overflow available as output signal | n/a | Yes | Yes; T2OUT (one CLKOUT pulse) |



Exhibit 2033 - Page 217 of 346

## 14.2.2 Timers 0 and 1

Timers 0 and 1 operate in four modes, as controlled through the TMOD SFR (Table 14-2 on page 219) and the TCON SFR (Table 14-3 on page 219). The four modes are:

- 13-bit timer/counter (mode 0)
- 16-bit timer/counter (mode 1)
- 8-bit counter with auto-reload (mode 2)
- Two 8-bit counters (mode 3, Timer 0 only)

### *14.2.2.1 Mode 0, 13-Bit Timer/Counter — Timer 0 and Timer 1*

Mode 0 operation is illustrated in Figure 14-1.

In mode 0, the timer is configured as a 13-bit counter that uses bits 0-4 of TL0 (or TL1) and all 8 bits of TH0 (or TH1). The timer enable bit (TR0/TR1) in the TCON SFR starts the timer. The C/$\overline{T}$ Bit selects the timer/counter clock source: either CLK-OUT or the T0/T1 pins.

The timer counts transitions from the selected source as long as the GATE Bit is 0, or the GATE Bit is 1 and the corresponding interrupt pin ($\overline{INT0}$ or $\overline{INT1}$) is 1.

When the 13-bit count increments from 0x1FFF (all ones), the counter rolls over to all zeros, the TF0 (or TF1) Bit is set in the TCON SFR, and the T0OUT (or T1OUT) pin goes high for one clock cycle.

Ignore the upper 3 bits of TL0 (or TL1) because they are indeterminate in mode 0.

Figure 14-1. Timer 0/1 - Modes 0 and 1

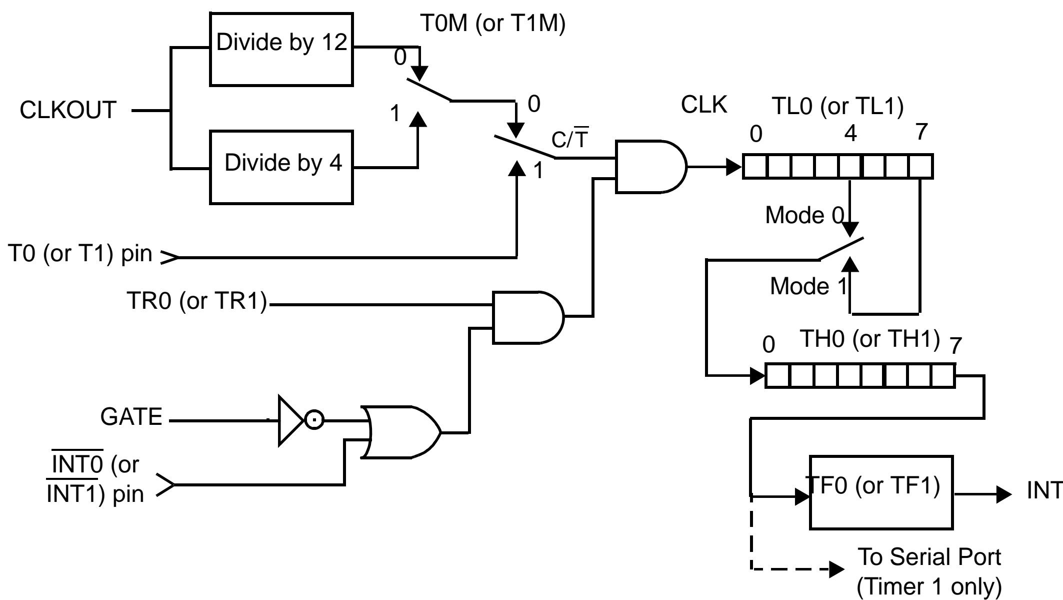





Exhibit 2033 - Page 218 of 346

### 14.2.2.2 Mode 1, 16-Bit Timer/Counter — Timer 0 and Timer 1

In mode 1, the timer is configured as a 16-bit counter. As illustrated in Figure 14-1 on page 218, all 8 bits of the LSB Register (TL0 or TL1) are used. The counter rolls over to all zeros when the count increments from 0xFFFF. Otherwise, mode 1 operation is the same as mode 0.

Table 14-2. TMOD Register — SFR 0x89

| Bit | Function |
|---|---|
| TMOD.7 | **GATE1** - Timer 1 gate control. When GATE1 = 1, Timer 1 will clock only when $\overline{INT1}$ = 1 and TR1 (TCON.6) = 1. When GATE1 = 0, Timer 1 will clock only when TR1 = 1, regardless of the state of $\overline{INT1}$. |
| TMOD.6 | **C/$\overline{T}$1** - Counter/Timer select. When C/$\overline{T}$1 = 0, Timer 1 is clocked by CLKOUT/4 or CLKOUT/12, depending on the state of T1M (CKCON.4). When C/$\overline{T}$1 = 1, Timer 1 is clocked by high-to-low transitions on the T1 pin. |
| TMOD.5<br>TMOD.4 | **M1** - Timer 1 mode select bit 1.<br>**M0** - Timer 1 mode select bit 0.<br>M1 M0 Mode<br>0 0 Mode 0: 13-bit counter<br>0 1 Mode 1: 16-bit counter<br>1 0 Mode 2: 8-bit counter with auto-reload<br>1 1 Mode 3: Timer 1 stopped |
| TMOD.3 | **GATE0** - Timer 0 gate control, When GATE0 = 1, Timer 0 will clock only when $\overline{INT0}$ = 1 and TR0 (TCON.4) = 1. When GATE0 = 0, Timer 0 will clock only when TR0 = 1, regardless of the state of $\overline{INT0}$. |
| TMOD.2 | **C/$\overline{T}$0** - Counter/Timer select. When C/$\overline{T}$0 = 0, Timer 0 is clocked by CLKOUT/4 or CLKOUT/12, depending on the state of T0M (CKCON.3). When C/$\overline{T}$0 = 1, Timer 0 is clocked by high-to-low transitions on the T0 pin. |
| TMOD.1<br>TMOD.0 | **M1** - Timer 0 mode select bit 1.<br>**M0** - Timer 0 mode select bit 0.<br>M1 M0 Mode<br>0 0 Mode 0: 13-bit counter<br>0 1 Mode 1: 16-bit counter<br>1 0 Mode 2: 8-bit counter with auto-reload<br>1 1 Mode 3: Two 8-bit counters |

Table 14-3. TCON Register — SRF 0x88

| Bit | Function |
|---|---|
| TCON.7 | **TF1** - Timer 1 overflow flag. Set to 1 when the Timer 1 count overflows; automatically cleared when the MoBL-USB FX2LP18 vectors to the interrupt service routine. |
| TCON.6 | **TR1** - Timer 1 run control. 1 = Enable counting on Timer 1. |
| TCON.5 | **TF0** - Timer 0 overflow flag. Set to 1 when the Timer 0 count overflows; automatically cleared when the MoBL-USB FX2LP18 vectors to the interrupt service routine. |
| TCON.4 | **TR0** - Timer 0 run control. 1 = Enable counting on Timer 0. |
| TCON.3 | **IE1** - Interrupt 1 edge detect. If external interrupt 1 is configured to be edge-sensitive (IT1 = 1), IE1 is set when a negative edge is detected on the $\overline{INT1}$ pin and is automatically cleared when the MoBL-USB FX2LP18 vectors to the corresponding interrupt service routine. In this case, IE1 can also be cleared by software. If external interrupt 1 is configured to be level-sensitive (IT1 = 0), IE1 is set when the $\overline{INT1}$ pin is 0 and automatically cleared when the $\overline{INT1}$ pin is 1. In level-sensitive mode, software cannot write to IE1. |
| TCON.2 | **IT1** - Interrupt 1 type select. $\overline{INT1}$ is detected on falling edge when IT1 = 1; $\overline{INT1}$ is detected as a low level when IT1 = 0. |
| TCON.1 | **IE0** - Interrupt 0 edge detect. If external interrupt 0 is configured to be edge-sensitive (IT0 = 1), IE0 is set when a negative edge is detected on the $\overline{INT0}$ pin and is automatically cleared when the MoBL-USB FX2LP18 vectors to the corresponding interrupt service routine. In this case, IE0 can also be cleared by software. If external interrupt 0 is configured to be level-sensitive (IT0 = 0), IE0 is set when the $\overline{INT0}$ pin is 0 and automatically cleared when the $\overline{INT0}$ pin is 1. In level-sensitive mode, software cannot write to IE0. |
| TCON.0 | **IT0** - Interrupt 0 type select. $\overline{INT0}$ is detected on falling edge when IT0 = 1; $\overline{INT0}$ is detected as a low level when IT0 = 0. |



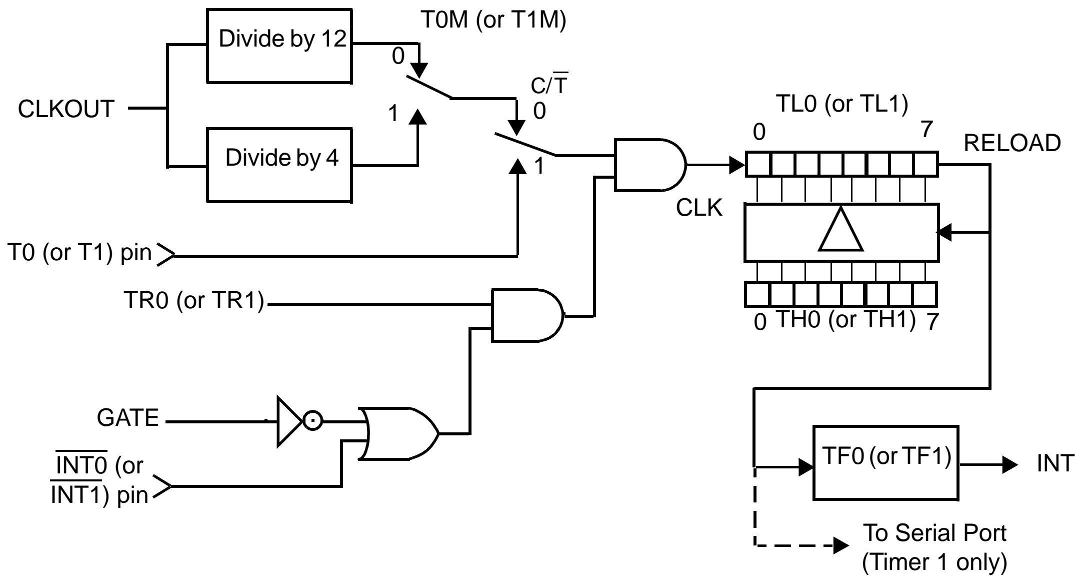
Exhibit 2033 - Page 219 of 346

### 14.2.2.3 Mode 2, 8-Bit Counter with Auto-Reload — Timer 0 and Timer 1

In mode 2, the timer is configured as an 8-bit counter, with automatic reload of the start value on overflow. TL0 (or TL1) is the counter, and TH0 (or TH1) stores the reload value.

As illustrated in Figure 14-2, mode 2 counter control is the same as for mode 0 and mode 1. When TL0/1 increments from 0xFF, the value stored in TH0/1 is reloaded into TL0/1.

Figure 14-2. Timer 0/1 - Mode 2




### 14.2.2.4 Mode 3, Two 8-Bit Counters — Timer 0 Only

In mode 3, Timer 0 operates as two 8-bit counters. Selecting mode 3 for Timer 1 simply stops Timer 1.

As shown in Figure 14-3 on page 221, TL0 is configured as an 8-bit counter controlled by the normal Timer 0 control bits. TL0 can either count CLKOUT cycles (divided by 4 or by 12) or high-to-low transitions on the T0 pin, as determined by the C/T Bit. The GATE function can be used to give counter enable control to the INT0 pin.

TH0 functions as an independent 8-bit counter. However, TH0 can only count CLKOUT cycles (divided by 4 or by 12). The Timer 1 control and flag bits (TR1 and TF1) are used as the control and flag bits for TH0.

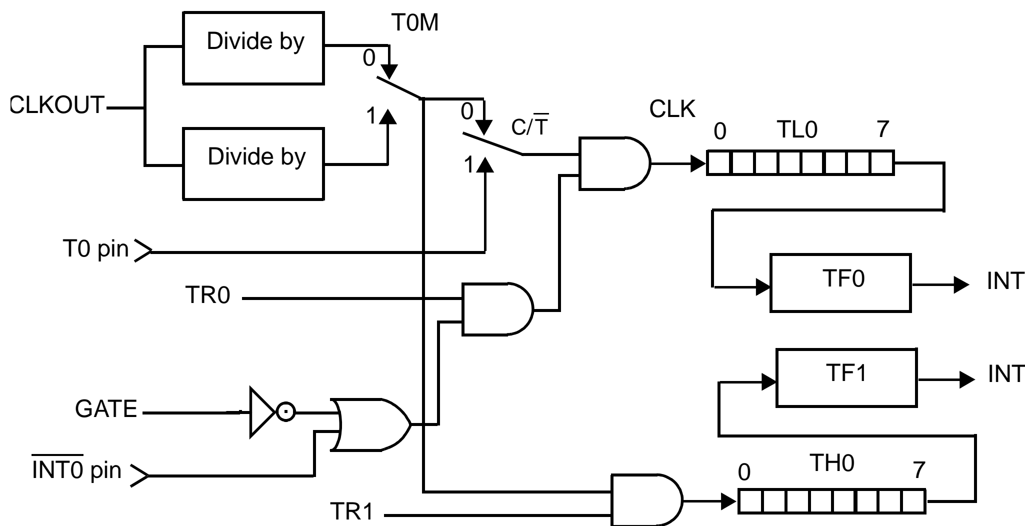
When Timer 0 is in mode 3, Timer 1 has limited usage because Timer 0 uses the Timer 1 control bit (TR1) and interrupt flag (TF1). Timer 1 can still be used for baud rate generation and the Timer 1 count values are still available in the TL1 and TH1 Registers.

Control of Timer 1 when Timer 0 is in mode 3 is through the Timer 1 mode bits. To turn Timer 1 on, set Timer 1 to mode 0, 1, or 2. To turn Timer 1 off, set it to mode 3. The Timer 1 C/T Bit and T1M Bit are still available to Timer 1. Therefore, Timer 1 can count CLKOUT/4, CLKOUT/12, or high-to-low transitions on the T1 pin. The Timer 1 GATE function is also available when Timer 0 is in mode 3.



Exhibit 2033 - Page 220 of 346

Figure 14-3. Timer 0 - Mode 3




## 14.2.3 Timer Rate Control

By default, the MoBL-USB FX2LP18 timers increment every 12 CLKOUT cycles, just as in the standard 8051. Using this default rate allows existing application code with real-time dependencies, such as baud rate, to operate properly.

Applications that require fast timing can set the timers to increment every 4 CLKOUT cycles instead, by setting bits in the Clock Control Register (CKCON) at SFR location 0x8E. (See Table 14-4).

Each timer's rate can be set independently. These settings have no effect in counter mode.

Table 14-4. CKCON (SFR 0x8E) Timer Rate Control Bits

| Bit | Function |
|---|---|
| CKCON.5 | **T2M** - Timer 2 clock select. When T2M = 0, Timer 2 uses CLKOUT/12 (for compatibility with standard 8051); when T2M = 1, Timer 2 uses CLKOUT/4. This bit has no effect when Timer 2 is configured for baud rate generation. |
| CKCON.4 | **T1M** - Timer 1 clock select. When T1M = 0, Timer 1 uses CLKOUT/12 (for compatibility with standard 8051); when T1M = 1, Timer 1 uses CLKOUT/4. |
| CKCON.3 | **T0M** - Timer 0 clock select. When T0M = 0, Timer 0 uses CLKOUT/12 (for compatibility with standard 8051); when T0M = 1, Timer 0 uses CLKOUT/4. |



Exhibit 2033 - Page 221 of 346

### 14.2.4 Timer 2

Timer 2 runs only in 16-bit mode and offers several capabilities not available with Timers 0 and 1. The modes available for Timer 2 are:

- 16-bit timer/counter
- 16-bit timer with capture
- 16-bit timer/counter with auto-reload
- Baud rate generator

The SFRs associated with Timer 2 are:

- T2CON (SFR 0xC8) Timer/Counter 2 Control register, (see Table 14-5).
- RCAP2L (SFR 0xCA) Captures the TL2 value when Timer 2 is configured for capture mode, or as the LSB of the 16-bit reload value when Timer 2 is configured for auto-reload mode.
- RCAP2H (SFR 0xCB) Captures the TH2 value when Timer 2 is configured for capture mode, or as the MSB of the 16-bit reload value when Timer 2 is configured for auto-reload mode.
- TL2 (SFR 0xCC) The lower 8 bits of the 16-bit count.
- TH2 (SFR 0xCD) The upper 8 bits of the 16-bit count.

Table 14-5. T2CON Register — SFR 0xC8

| Bit | Function |
| --- | --- |
| T2CON.7 | **TF2** - Timer 2 overflow flag. Hardware will set TF2 when the Timer 2 overflows from 0xFFFF. TF2 must be cleared to 0 by the software. TF2 will only be set to a 1 if RCLK and TCLK are both cleared to 0. Writing a 1 to TF2 forces a Timer 2 interrupt if enabled. |
| T2CON.6 | **EXF2** - Timer 2 external flag. Hardware will set EXF2 when a reload or capture is caused by a high-to-low transition on the T2EX pin, and EXEN2 is set. EXF2 must be cleared to 0 by software. Writing a 1 to EXF2 forces a Timer 2 interrupt if enabled. |
| T2CON.5 | **RCLK** - Receive clock flag. Determines whether Timer 1 or Timer 2 is used for Serial Port 0 timing of received data in serial mode 1 or 3. RCLK=1 selects Timer 2 overflow as the receive clock; RCLK=0 selects Timer 1 overflow as the receive clock. |
| T2CON.4 | **TCLK** - Transmit clock flag. Determines whether Timer 1 or Timer 2 is used for Serial Port 0 timing of transmit data in serial mode 1 or 3. TCLK=1 selects Timer 2 overflow as the transmit clock; TCLK=0 selects Timer 1 overflow as the transmit clock. |
| T2CON.3 | **EXEN2** - Timer 2 external enable. EXEN2=1 enables capture or reload to occur as a result of a high-to-low transition on the T2EX pin, if Timer 2 is not generating baud rates for the serial port. EXEN2=0 causes Timer 2 to ignore all external events on the T2EX pin. |
| T2CON.2 | **TR2** - Timer 2 run control flag. TR2=1 starts Timer 2; TR2=0 stops Timer 2. |
| T2CON.1 | **C/$\overline{T}$2** - Counter/Timer select. When C/$\overline{T}$2 = 1, Timer 2 is clocked by high-to-low transitions on the T2 pin.When C/$\overline{T}$2 = 0 in modes 0, 1, or 2, Timer 2 is clocked by CLKOUT/4 or CLKOUT/12, depending on the state of T2M (CKCON.5). When C/$\overline{T}$2 = 0 in mode 3, Timer 2 is clocked by CLKOUT/2, regardless of the state of CKCON.5. |
| T2CON.0 | **CP/$\overline{RL}$2** - Capture/reload flag. When CP/$\overline{RL}$2=1, Timer 2 captures occur on high-to-low transitions of the T2EX pin, if EXEN2 = 1. When CP/$\overline{RL}$2 = 0, auto-reloads occur when Timer 2 overflows or when high-to-low transitions occur on the T2EX pin, if EXEN2 = 1. If either RCLK or TCLK is set to 1, CP/$\overline{RL}$2 will not function and Timer 2 will operate in auto-reload mode following each overflow. |

#### *14.2.4.1 Timer 2 Mode Control*

Table 14-6 summarizes how the T2CON bits determine the Timer 2 mode.

Table 14-6. Timer 2 Mode Control Summary

| TR2 | TCLK | RCLK | CP/$\overline{RL}$2 | Mode |
| --- | --- | --- | --- | --- |
| 0 | X | X | X | Timer 2 stopped |
| 1 | 1 | X | X | Baud rate generator |
| 1 | X | 1 | X | Baud rate generator |
| 1 | 0 | 0 | 0 | 16-bit timer/counter with auto-reload |
| 1 | 0 | 0 | 1 | 16-bit timer/counter with capture |
| X = Don't care | | | | |



Exhibit 2033 - Page 222 of 346

## 14.2.5 Timer 2 The 6-Bit Timer/Counter Mode

Figure 14-4 illustrates how Timer 2 operates in timer/counter mode with the optional capture feature. The C/$\overline{\text{T}}$2 Bit determines whether the 16-bit counter counts CLKOUT cycles (divided by 4 or 12), or high-to-low transitions on the T2 pin. The TR2 Bit enables the counter. When the count increments from 0xFFFF, the TF2 flag is set and the T2OUT pin goes high for one CLK-OUT cycle.

### *14.2.5.1 Timer 2 The 16-Bit Timer/Counter Mode with Capture*

The Timer 2 capture mode (Figure 14-4) is the same as the 16-bit timer/counter mode, with the addition of the capture registers and control signals.

The CP/$\overline{\text{RL}}$2 Bit in the T2CON SFR enables the capture feature. When CP/$\overline{\text{RL}}$2 = 1, a high-to-low transition on the T2EX pin when EXEN2 = 1 causes the Timer 2 value to be loaded into the capture registers RCAP2L and RCAP2H.

Figure 14-4. Timer 2 - Timer/Counter with Capture

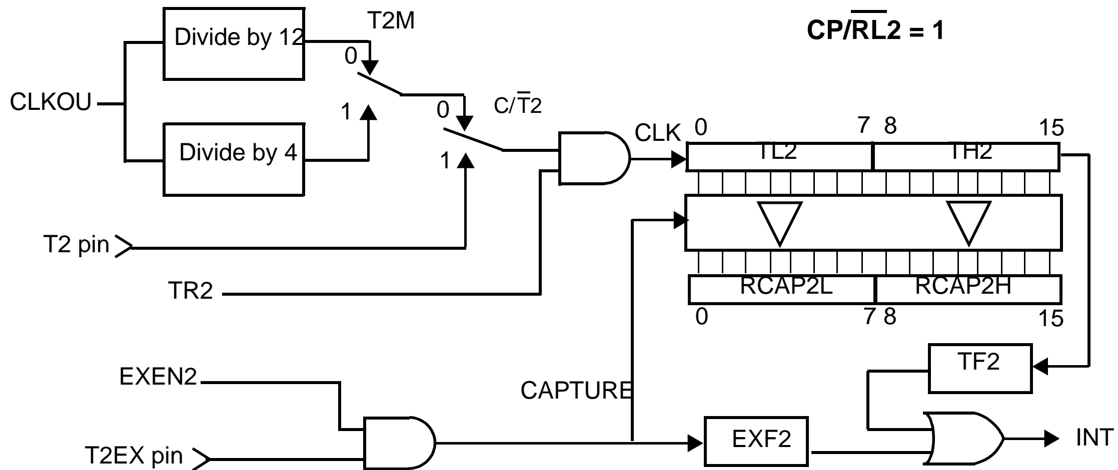




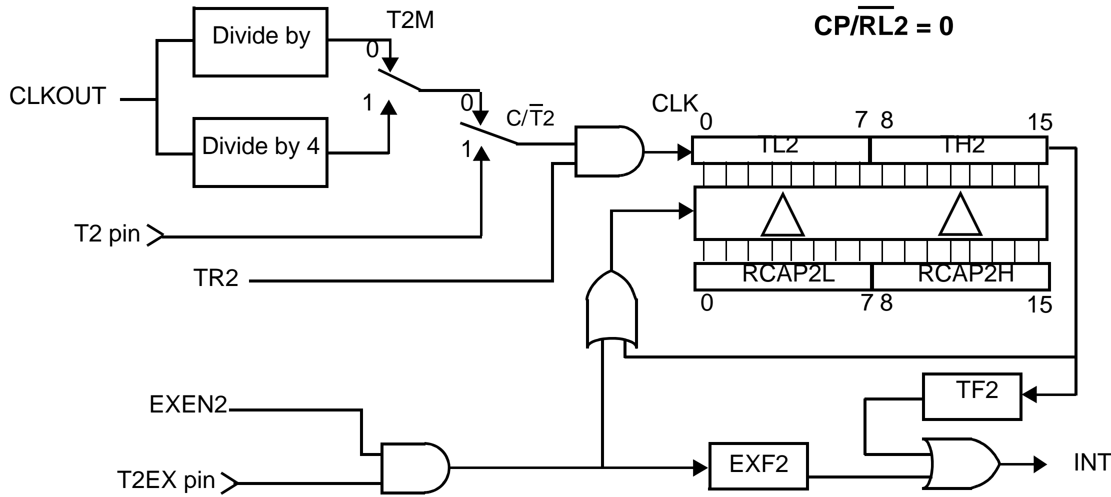

Exhibit 2033 - Page 223 of 346

### 14.2.6 Timer 2 16-Bit Timer/Counter Mode with Auto-Reload

When CP/$\overline{\text{RL}}$2 = 0, Timer 2 is configured for the auto-reload mode illustrated in Figure 14-5. Control of counter input is the same as for the other 16-bit counter modes. When the count increments from 0xFFFF, Timer 2 sets the TF2 flag and the starting value is reloaded into TL2 and TH2. Software must preload the starting value into the RCAP2L and RCAP2H registers.

When Timer 2 is in auto-reload mode, a reload can be forced by a high-to-low transition on the T2EX pin, if enabled by EXEN2 = 1.

Figure 14-5. Timer 2 - Timer/Counter with Auto Reload




### 14.2.7 Timer 2 Baud Rate Generator Mode

Set either RCLK or TCLK to ‘1’ to configure Timer 2 to generate baud rates for Serial Port 0 in serial mode 1 or 3. Figure 14-6 on page 225 is the functional diagram for the Timer 2 baud rate generator mode. In baud rate generator mode, Timer 2 functions in auto-reload mode. However, instead of setting the TF2 flag, the counter overflow is used to generate a shift clock for the serial port function. As in normal auto-reload mode, the overflow also causes the pre-loaded start value in the RCAP2L and RCAP2H Registers to be reloaded into the TL2 and TH2 Registers.

When either TCLK = 1 or RCLK = 1, Timer 2 is forced into auto-reload operation, regardless of the state of the CP/$\overline{\text{RL}}$2 Bit. Timer 2 is used as the receive baud clock source when RCLK=1, and as the transmit baud clock source when TCLK=1.

When operating as a baud rate generator, Timer 2 does not set the TF2 Bit. In this mode, a Timer 2 interrupt can only be generated by a high-to-low transition on the T2EX pin setting the EXF2 Bit, and only if enabled by EXEN2 = 1.

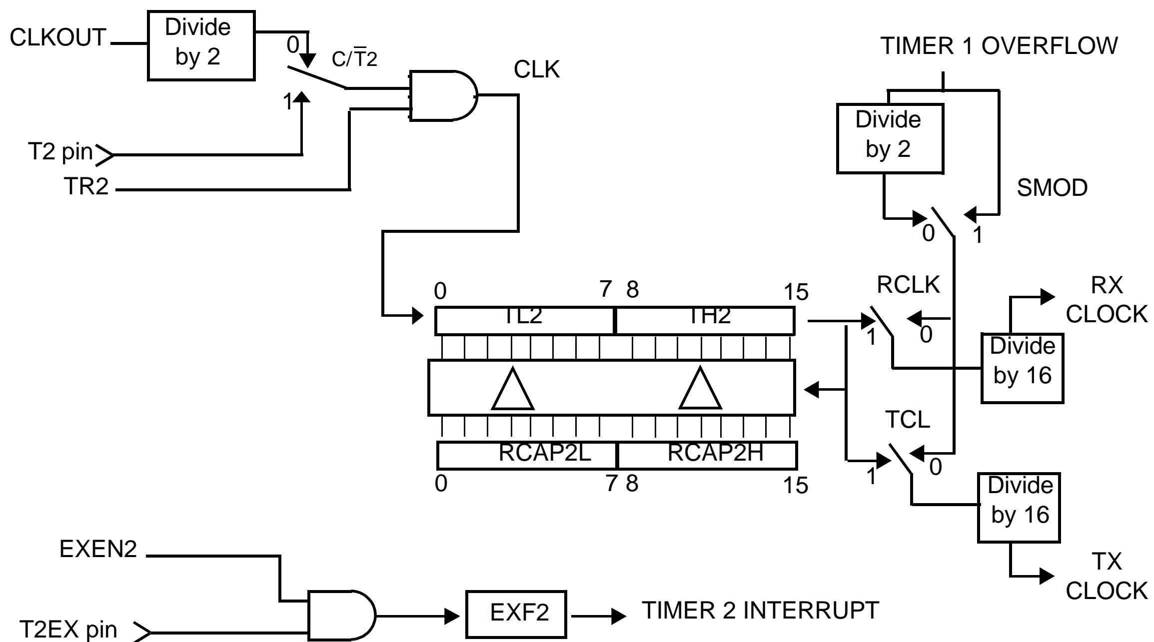
The counter time base in baud rate generator mode is CLKOUT/2. To use an external clock source, set C/$\overline{\text{T}}$2 to ‘1’ and apply the desired clock source to the T2 pin.

The maximum frequency for an external clock source on the T2 pin is 6 MHz.



Exhibit 2033 - Page 224 of 346

Figure 14-6. Timer 2 - Baud Rate Generator Mode




## 14.3 Serial Interface

The MoBL-USB FX2LP18 provides two serial ports. Serial Port 0 operates almost exactly as a standard 8051 serial port; depending on the configured mode (see Table 14-7), its baud-clock source can be CLKOUT/4 or CLKOUT/12, Timer 1, Timer 2, or the High-Speed Baud Rate Generator (see section 14.3.2 High-Speed Baud Rate Generator on page 226). Serial Port 1 is identical to Serial Port 0, except that it cannot use Timer 2 as its baud rate generator.

Each serial port can operate in synchronous or asynchronous mode. In synchronous mode, the MoBL-USB FX2LP18 generates the serial clock and the serial port operates in half-duplex mode. In asynchronous mode, the serial port operates in full-duplex mode. In all modes, the MoBL-USB FX2LP18 double-buffers the incoming data so that a byte of incoming data can be received while firmware is reading the previously-received byte.

Each serial port can operate in one of four modes, as outlined in Table 14-7.

Table 14-7. Serial Port Modes

| Mode | Sync/ Async | Baud-Clock Source | Data Bits | Start/ Stop | 9th Bit Function |
|---|---|---|---|---|---|
| 0 | Sync | CLKOUT/4 or CLKOUT/12 | 8 | None | None |
| 1 | Async | Timer 1 (Ports 0 and 1),<br>Timer 2 (Port 0 only), or<br>High-Speed Baud Rate Generator (Ports 0 and 1) | 8 | 1 start, 1 stop | None |
| 2 | Async | CLKOUT/32 or CLKOUT/64 | 9 | 1 start, 1 stop | 0, 1, or parity |
| 3 | Async | Timer 1 (Ports 0 and 1),<br>Timer 2 (Port 0 only), or<br>High-Speed Baud Rate Generator (Ports 0 and 1) | 9 | 1 start, 1 stop | 0, 1, or parity |

**Note:** The High-Speed Baud Rate Generator provides 115.2K or 230.4K baud rates (see section 14.3.2 High-Speed Baud Rate Generator on page 226).



**Exhibit 2033 - Page 225 of 346**

The registers associated with the serial ports are as follows. (Registers PCON and EICON also include some functionality which is not part of the Serial Interface).

- PCON (SFR 0x87) Bit 7, Serial Port 0 rate control SMOD0 (Table 14-13 on page 227).
- SCON0 (SFR 0x98) Serial Port 0 control (Table 14-11 on page 227).
- SBUF0 (SFR 0x99) Serial Port 0 transmit/receive buffer.
- EICON (SFR 0xD8) Bit 7, Serial Port 1 rate control SMOD1 (Table 14-12 on page 227).
- SCON1 (SFR 0xC0) Serial Port 1 control (Table 14-14 on page 228).
- SBUF1 (SFR 0xC1) Serial Port 1 transmit/receive buffer.
- T2CON (SFR 0xC8) Baud clock source for modes 1 and 3 (RCLK and TCLK in Table 14-5 on page 222).
- UART230 (0xE608) High-Speed Baud Rate Generator enable (see section 14.3.2 High-Speed Baud Rate Generator).

## 14.3.1 803x/805x Compatibility

The implementation of the serial interface is similar to that of the Dallas Semiconductor, DS80C320. Table 14-8 summarizes the differences in serial interface implementation between the Intel 8051, the Dallas Semiconductor DS80C320, and the MoBL-USB FX2LP18.

Table 14-8. Serial Interface Implementation Comparison

| Feature | Intel 8051 | Dallas DS80C320 | MoBL-USB FX2LP18 |
|---|---|---|---|
| Number of serial ports | 1 | 2 | 2 |
| Framing error detection | not implemented | implemented | not implemented |
| Slave address comparison for multiprocessor communication | not implemented | implemented | not implemented |

## 14.3.2 High-Speed Baud Rate Generator

The MoBL-USB FX2LP18 incorporates a high-speed baud rate generator which can provide 115.2K and 230.4K baud rates for either or both serial ports, regardless of the MoBL-USB FX2LP18's internal clock frequency (12, 24, or 48 MHz).

The high-speed baud rate generator is enabled for Serial Port 0 by setting UART230.0 to 1; it's enabled for Serial Port 1 by setting UART230.1 to 1.

When enabled, the high-speed baud rate generator defaults to 115.2K baud. To select 230.4K baud for Serial Port 0, set SMOD0 (PCON.7) to 1; for Serial Port 1, set SMOD1 (EICON.7) to 1.

Table 14-9. UART230 Register — Address 0xE608

| Bit | Function |
|---|---|
| UART230.7:2 | **Reserved** |
| UART230.1 | **230UART1** - Enable high-speed baud rate generator for serial port 1. When 230UART1 = 1, a 115.2K baud (if SMOD1 = 0) or 230.4K baud (if SMOD1 = 1) clock is provided to serial port 1. When 230UART1 = 0, serial port 1's baud clock is provided by one of the sources shown in Table 14-7 on page 225. |
| UART230.0 | **230UART0** - Enable high-speed baud rate generator for serial port 0. When 230UART0 = 1, a 115.2K baud (if SMOD0 = 0) or 230.4K baud (if SMOD0 = 1) clock is provided to serial port 0. When 230UART1 = 0, serial port 0's baud clock is provided by one of the sources shown in Table 14-7 on page 225. |

**Note** When the High-Speed Baud Rate Generator is enabled for either serial port, **neither** port may use Timer 1 as its baud-clock source. Therefore, the allowable combinations of baud-clock sources for Modes 1 and 3 are listed below.

Table 14-10. Allowable Baud-Clock Combinations for Modes 1 and 3

| Port 0 | Port 1 |
|---|---|
| Timer 1 | Timer 1 |
| Timer 2 | Timer 1 |
| Timer 2 | High-Speed Baud Rate Generator |
| High-Speed Baud Rate Generator | High-Speed Baud Rate Generator |



Exhibit 2033 - Page 226 of 346

### 14.3.3 Mode 0

Serial mode 0 provides synchronous, half-duplex serial communication. For Serial Port 0, serial data output occurs on the RXD0OUT pin, serial data is received on the RXD0 pin, and the TXD0 pin provides the shift clock for both transmit and receive. For Serial Port 1, the corresponding pins are RXD1OUT, RXD1, and TXD1.

The serial mode 0 baud rate is either CLKOUT/12 or CLKOUT/4, depending on the state of the SM2_0 bit (or SM2_1 for Serial Port 1). When SM2_0 = 0, the baud rate is CLKOUT/12, when SM2_0 = 1, the baud rate is CLKOUT/4.

Mode 0 operation is identical to the standard 8051. Data transmission begins when an instruction writes to the SBUF0 (or SBUF1) SFR. The USART shifts the data, LSB first, at the selected baud rate, until the 8-bit value has been shifted out.

Mode 0 data reception begins when the REN_0 (or REN_1) bit is set and the RI_0 (or RI_1) bit is cleared in the corresponding SCON SFR. The shift clock is activated and the USART shifts data, LSB first, in on each rising edge of the shift clock until 8 bits have been received. One CLKOUT cycle after the 8th bit is shifted in, the RI_0 (or RI_1) bit is set and reception stops until the software clears the RI bit.

Figure 14-7 on page 228 through Figure 14-10 on page 230 illustrate Serial Port Mode 0 transmit and receive timing for both low-speed (CLKOUT/12) and high-speed (CLKOUT/4) operation. The figures show Port 0 signal names, RXD0, RXD0OUT, and TXD0. The timing is the same for Port 1 signals RXD1, RXD1OUT, and TXD1, respectively.

Table 14-11. SCON0 Register — SFR 98h

| Bit | Function |
|---|---|
| SCON0.7 | **SM0_0** - Serial Port 0 mode bit 0. |
| SCON0.6 | **SM1_0** - Serial Port 0 mode bit 1, decoded as:<br>SM0_0 SM1_0 Mode<br>0 0 0<br>0 1 1<br>1 0 2<br>1 1 3 |
| SCON0.5 | **SM2_0** - Multiprocessor communication enable. In modes 2 and 3, this bit enables the multiprocessor communication feature. If SM2_0 = 1 in mode 2 or 3, then RI_0 will not be activated if the received 9th bit is 0.<br>If SM2_0=1 in mode 1, then RI_0 will only be activated if a valid stop is received. In mode 0, SM2_0 establishes the baud rate: when SM2_0=0, the baud rate is CLKOUT/12; when SM2_0=1, the baud rate is CLKOUT/4. |
| SCON0.4 | **REN_0** - Receive enable. When REN_0=1, reception is enabled. |
| SCON0.3 | **TB8_0** - Defines the state of the 9th data bit transmitted in modes 2 and 3. |
| SCON0.2 | **RB8_0** - In modes 2 and 3, RB8_0 indicates the state of the 9th bit received. In mode 1, RB8_0 indicates the state of the received stop bit. In mode 0, RB8_0 is not used. |
| SCON0.1 | **TI_0** - Transmit interrupt flag. Indicates that the transmit data word has been shifted out. In mode 0, TI_0 is set at the end of the 8th data bit. In all other modes, TI_0 is set when the stop bit is placed on the TXD0 pin. TI_0 must be cleared by firmware. |
| SCON0.0 | **RI_0** - Receive interrupt flag. Indicates that serial data word has been received. In mode 0, RI_0 is set at the end of the 8th data bit. In mode 1, RI_0 is set after the last sample of the incoming stop bit, subject to the state of SM2_0. In modes 2 and 3, RI_0 is set at the end of the last sample of RB8_0. RI_0 must be cleared by firmware. |

Table 14-12. EICON (SFR 0xD8) SMOD1 Bit

| Bit | Function |
|---|---|
| EICON.7 | **SMOD1** - Serial Port 1 baud rate doubler enable. When SMOD1 = 1 the baud rate for Serial Port is doubled. |

Table 14-13. PCON (SFR 0x87) SMOD0 Bit

| Bit | Function |
|---|---|
| PCON.7 | **SMOD0** - Serial Port 0 baud rate double enable. When SMOD0 = 1, the baud rate for Serial Port 0 is doubled. |



Exhibit 2033 - Page 227 of 346

Table 14-14. SCON1 Register — SFR C0h

| Bit | Function |
|---|---|
| SCON1.7 | **SM0_1** - Serial Port 1 mode bit 0. |
| SCON1.6 | **SM1_1** - Serial Port 1 mode bit 1, decoded as:<br>SM0_1 SM1_1 Mode<br>0 0 0<br>0 1 1<br>1 0 2<br>1 1 3 |
| SCON1.5 | **SM2_1** - Multiprocessor communication enable. In modes 2 and 3, this bit enables the multiprocessor communication feature. If SM2_1 = 1 in mode 2 or 3, then RI_1 will not be activated if the received 9th bit is 0.<br>If SM2_1=1 in mode 1, then RI_1 will only be activated if a valid stop is received. In mode 0, SM2_1 establishes the baud rate: when SM2_1=0, the baud rate is CLKOUT/12; when SM2_1=1, the baud rate is CLKOUT/4. |
| SCON1.4 | **REN_1** - Receive enable. When REN_1=1, reception is enabled. |
| SCON1.3 | **TB8_1** - Defines the state of the 9th data bit transmitted in modes 2 and 3. |
| SCON1.2 | **RB8_1** - In modes 2 and 3, RB8_1 indicates the state of the 9th bit received. In mode 1, RB8_1 indicates the state of the received stop bit. In mode 0, RB8_1 is not used. |
| SCON1.1 | **TI_1** - Transmit interrupt flag. Indicates that the transmit data word has been shifted out. In mode 0, TI_1 is set at the end of the 8th data bit. In all other modes, TI_1 is set when the stop bit is placed on the TXD1 pin. TI_1 must be cleared by the software. |
| SCON1.0 | **RI_1** - Receive interrupt flag. Indicates that serial data word has been received. In mode 0, RI_1 is set at the end of the 8th data bit. In mode 1, RI_1 is set after the last sample of the incoming stop bit, subject to the state of SM2_1. In modes 2 and 3, RI_1 is set at the end of the last sample of RB8_1. RI_1 must be cleared by the software. |

Figure 14-7. Serial Port Mode 0 Receive Timing - Low Speed Operation

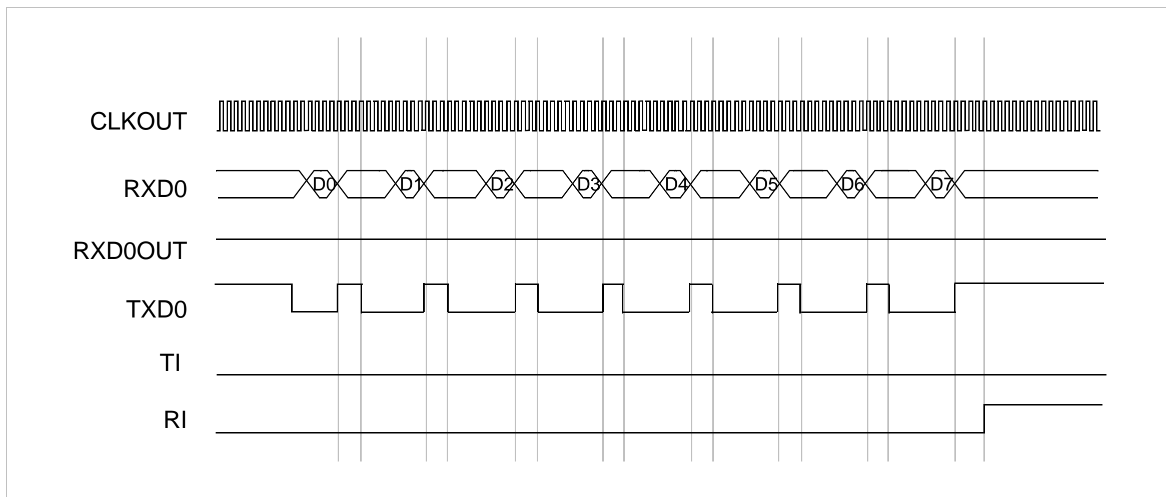
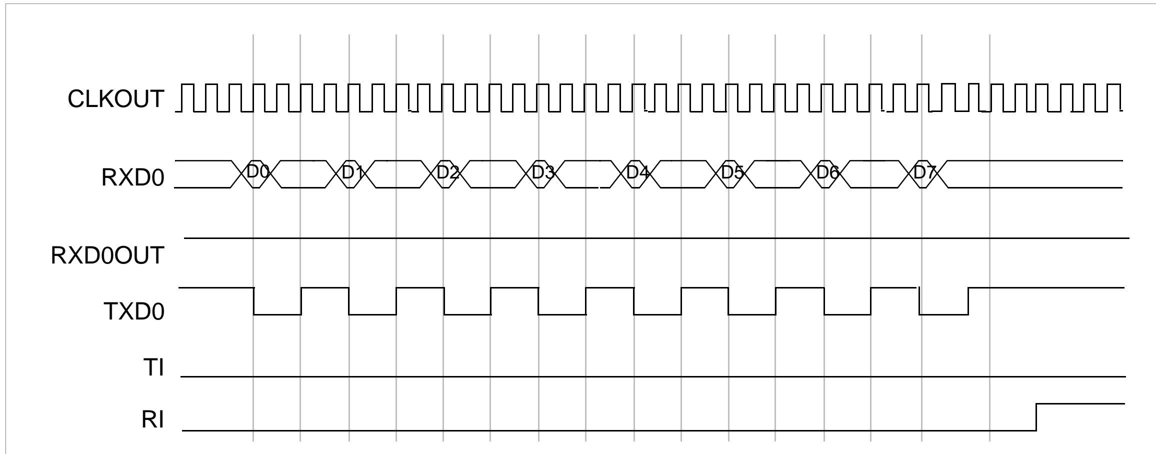





Exhibit 2033 - Page 228 of 346

Figure 14-8. Serial Port Mode 0 Receive Timing - High-Speed Operation




At both low and high-speed in Mode 0, data on RXD0 is sampled two CLKOUT cycles **before** the rising clock edge on TXD0.

Figure 14-9. Serial Port Mode 0 Transmit Timing - Low Speed Operation

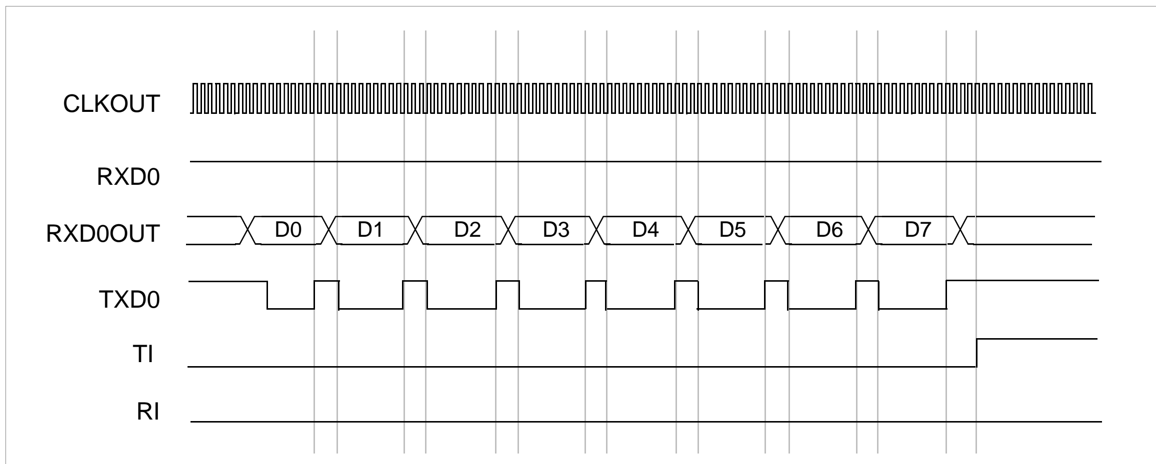
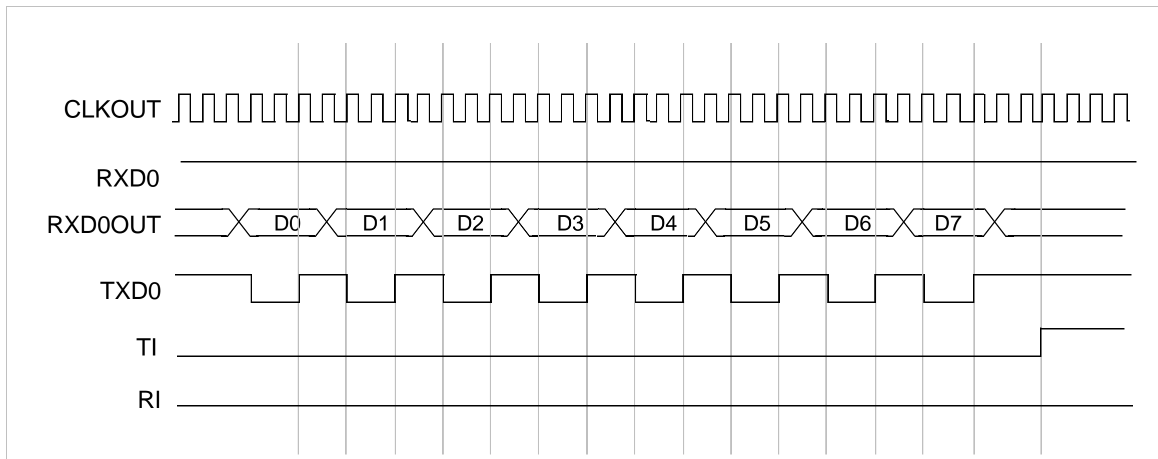





Exhibit 2033 - Page 229 of 346

Figure 14-10.  Serial Port Mode 0 Transmit Timing - High-Speed Operation




## 14.3.4 Mode 1

Mode 1 provides standard asynchronous, full-duplex communication, using a total of 10 bits: 1 start bit, 8 data bits, and 1 stop bit. For receive operations, the stop bit is stored in RB8_0 (or RB8_1). Data bits are received and transmitted LSB first.

Mode 1 operation is identical to that of the standard 8051 when Timer 1 uses CLKOUT/12, (T1M=0, the default).

### *14.3.4.1 Mode 1 Baud Rate*

The mode 1 baud rate is a function of timer overflow. Serial Port 0 can use either Timer 1 or Timer 2 to generate baud rates. Serial Port 1 can only use Timer 1. The two serial ports can run at the same baud rate if they both use Timer 1, or different baud rates if Serial Port 0 uses Timer 2 and Serial Port 1 uses Timer 1.

Each time the timer increments from its maximum count (0xFF for Timer 1 or 0xFFFF for Timer 2), a clock is sent to the baud rate circuit. That clock is then divided by 16 to generate the baud rate.

When using Timer 1, the SMOD0 (or SMOD1) Bit selects whether or not to divide the Timer 1 rollover rate by 2. Therefore, when using Timer 1, the baud rate is determined by the equation:

$$Baud\ Rate = \frac{2^{SMODx}}{32} \times Timer\ 1\ Overflow \qquad \textbf{Equation 1}$$

When using Timer 2, the baud rate is determined by the equation:

$$Baud\ Rate = \frac{Timer\ 2\ Overflow}{16} \qquad \textbf{Equation 2}$$

To use Timer 1 as the baud rate generator, it is generally best to use Timer 1 mode 2 (8-bit counter with auto-reload), although any counter mode can be used. In mode 2, the Timer 1 reload value is stored in the TH1 register, which makes the complete formula for Timer 1:

$$Baud\ Rate = \frac{2^{SMODx}}{32} \times \frac{CLKOUT}{(12 - 8 \times T1M) \times (256 - TH1)} \qquad \textbf{Equation 3}$$

To derive the required TH1 value from a known baud rate when T1M=0, use the equation:

$$TH1 = 256 - \frac{2^{SMODx} \times CLKOUT}{384 \times Baud\ Rate} \qquad \textbf{Equation 4}$$



Exhibit 2033 - Page 230 of 346

To derive the required TH1 value from a known baud rate when T1M=1, use the equation:

$$TH1 = 256 - \frac{2^{SMODx} \times CLKOUT}{128 \times Baud\ Rate}$$ **Equation 5**

**Note** Very low serial port baud rates may be achieved with Timer 1 by enabling the Timer 1 interrupt, configuring Timer 1 to mode 1, and using the Timer 1 interrupt to initiate a 16-bit software reload.

Table 14-15 lists sample reload values for a variety of common serial port baud rates, using Timer 1 operating in mode 2 (TMOD.5:4=10) with a CLKOUT/4 clock source (T1M=1) and the full timer rollover (SMOD1=1).

Table 14-15. Timer 1 Reload Values for Common Serial Port Mode 1 Baud Rates

| Nominal Rate | CLKOUT = 12 MHz | | | CLKOUT = 24 MHz | | | CLKOUT = 48 MHz | | |
|---|---|---|---|---|---|---|---|---|---|
| | TH1 Reload Value | Actual Rate | Error | TH1 Reload Value | Actual Rate | Error | TH1 Reload Value | Actual Rate | Error |
| 57600 | FD | 62500 | +8.50% | F9 | 53571 | -6.99% | F3 | 57692 | +0.16% |
| 38400 | FB | 37500 | -2.34% | F6 | 37500 | -2.34% | EC | 37500 | -2.34% |
| 28800 | F9 | 26786 | -6.99% | F3 | 28846 | +0.16% | E6 | 28846 | +0.16% |
| 19200 | F6 | 18750 | -2.34% | EC | 18750 | -2.34% | D9 | 19230 | +0.16% |
| 9600 | EC | 9375 | -2.34% | D9 | 9615 | +0.16% | B2 | 9615 | +0.16% |
| 4800 | D9 | 4807 | +0.16% | B2 | 4807 | +0.16% | 64 | 4807 | +0.16% |
| 2400 | B2 | 2403 | +0.16% | 64 | 2403 | +0.16% | — | — | — |
| Settings: SMOD=1, C/$\overline{T}$=0, Timer1 Mode=2, T1M=1 | | | | | | | | | |
| **Note** Using rates that are off by 2% or more will not work in all systems. | | | | | | | | | |

More accurate baud rates may be achieved by using Timer 2 as the baud rate generator. To use Timer 2 as the baud rate generator, configure Timer 2 in auto-reload mode and set the TCLK and/or RCLK bits in the T2CON SFR. TCLK selects Timer 2 as the baud rate generator for the transmitter; RCLK selects Timer 2 as the baud rate generator for the receiver. The 16-bit reload value for Timer 2 is stored in the RCAP2L and RCA2H SFRs, which makes the equation for the Timer 2 baud rate:

$$Baud\ Rate = \frac{CLKOUT}{32 \times (65536 - (256 \times RCAP2H + RCAP2L))}$$ **Equation 6**

To derive the required RCAP2H and RCAP2L values from a known baud rate, use the equation:

$$RCAP2H{:}L = 65536 - \frac{CLKOUT}{32 \times Baud\ Rate}$$ **Equation 7**

When either RCLK or TCLK is set, the TF2 flag is not set on a Timer 2 rollover and the T2EX reload trigger is disabled.

Table 14-16 lists sample RCAP2H:L reload values for a variety of common serial baud rates.



Exhibit 2033 - Page 231 of 346

Table 14-16. Timer 2 Reload Values for Common Serial Port Mode 1 Baud Rates

| Nominal Rate | CLKOUT = 12 MHz | | | CLKOUT = 24 MHz | | | CLKOUT = 48 MHz | | |
|---|---|---|---|---|---|---|---|---|---|
| | RCAP2H:L Reload Value | Actual Rate | Error | RCAP2H:L Reload Value | Actual Rate | Error | RCAP2H:L Reload Value | Actual Rate | Error |
| 57600 | FFF9 | 53571 | -6.99% | FFF3 | 57692 | +0.16% | FFE6 | 57692 | +0.16% |
| 38400 | FFF6 | 37500 | -2.34% | FFEC | 37500 | -2.34% | FFD9 | 38461 | +0.16% |
| 28800 | FFF3 | 28846 | +0.16% | FFE6 | 28846 | +0.16% | FFCC | 28846 | +0.16% |
| 19200 | FFEC | 18750 | -2.34% | FFD9 | 19230 | +0.16% | FFB2 | 19230 | +0.16% |
| 9600 | FFD9 | 9615 | +0.16% | FFB2 | 9615 | +0.16% | FF64 | 9615 | +0.16% |
| 4800 | FFB2 | 4807 | +0.16% | FF64 | 4807 | +0.16% | FEC8 | 4807 | +0.16% |
| 2400 | FF64 | 2403 | +0.16% | FEC8 | 2403 | +0.16% | FD90 | 2403 | +0.16% |
| **Note** using rates that are off by 2.3% or more will not work in all systems. | | | | | | | | | |

#### *14.3.4.2 Mode 1 Transmit*

Figure 14-11 on page 233 illustrates the mode 1 transmit timing. In mode 1, the USART begins transmitting after the first roll-over of the divide-by-16 counter after the software writes to the SBUF0 (or SBUF1) register. The USART transmits data on the TXD0 (or TXD1) pin in the following order: start bit, 8 data bits (LSB first), stop bit. The TI_0 (or TI_1) bit is set 2 CLKOUT cycles after the stop bit is transmitted.

### 14.3.5 Mode 1 Receive

Figure 14-12 on page 233 illustrates the mode 1 receive timing. Reception begins at the falling edge of a start bit received on the RXD0 (or RXD1) pin, when enabled by the REN_0 (or REN_1) Bit. For this purpose, the RXD0 (or RXD1) pin is sampled 16 times per bit for any baud rate. When a falling edge of a start bit is detected, the divide-by-16 counter used to generate the receive clock is reset to align the counter rollover to the bit boundaries.

For noise rejection, the serial port establishes the content of each received bit by a majority decision of 3 consecutive samples in the middle of each bit time. For the start bit, if the falling edge on the RXD0 (or RXD1) pin is not verified by a majority decision of 3 consecutive samples (low), then the serial port stops reception and waits for another falling edge on the RXD0 (or RXD1) pin.

At the middle of the stop bit time, the serial port checks for the following conditions:

- RI_0 (or RI_1) = 0
- If SM2_0 (or SM2_1) = 1, the state of the stop bit is 1
  (If SM2_0 (or SM2_1) = 0, the state of the stop bit does not matter.

If the above conditions are met, the serial port then writes the received byte to the SBUF0 (or SBUF1) Register, loads the stop bit into RB8_0 (or RB8_1), and sets the RI_0 (or RI_1) Bit. If the above conditions are not met, the received data is lost, the SBUF Register and RB8 Bit are not loaded, and the RI Bit is not set.

After the middle of the stop bit time, the serial port waits for another high-to-low transition on the (RXD0 or RXD1) pin.



Exhibit 2033 - Page 232 of 346

Figure 14-11. Serial Port 0 Mode 1 Transmit Timing

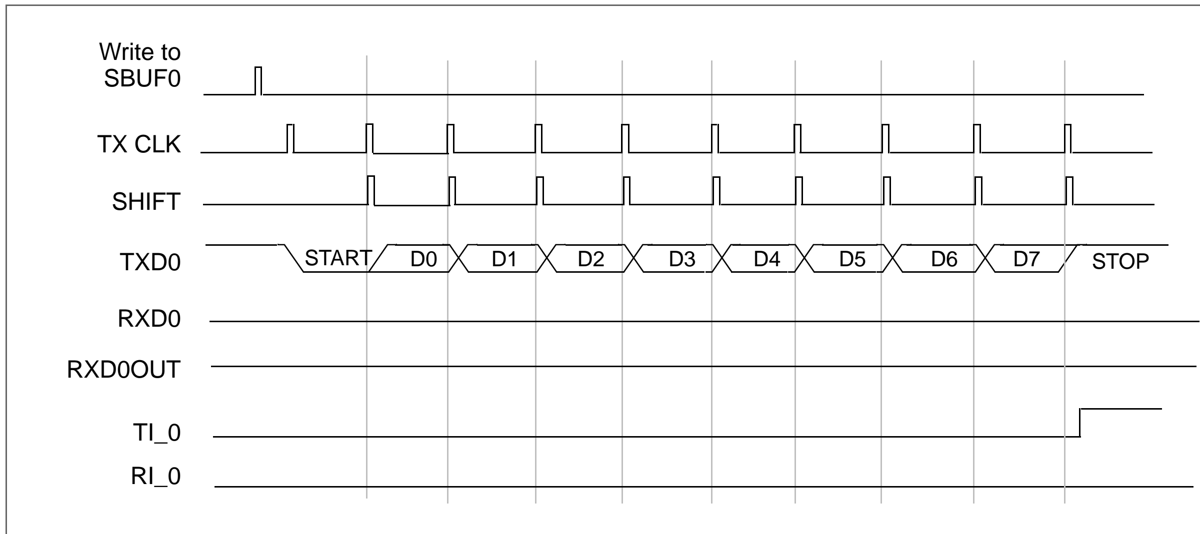



Figure 14-12. Serial Port 0 Mode 1 Receive Timing

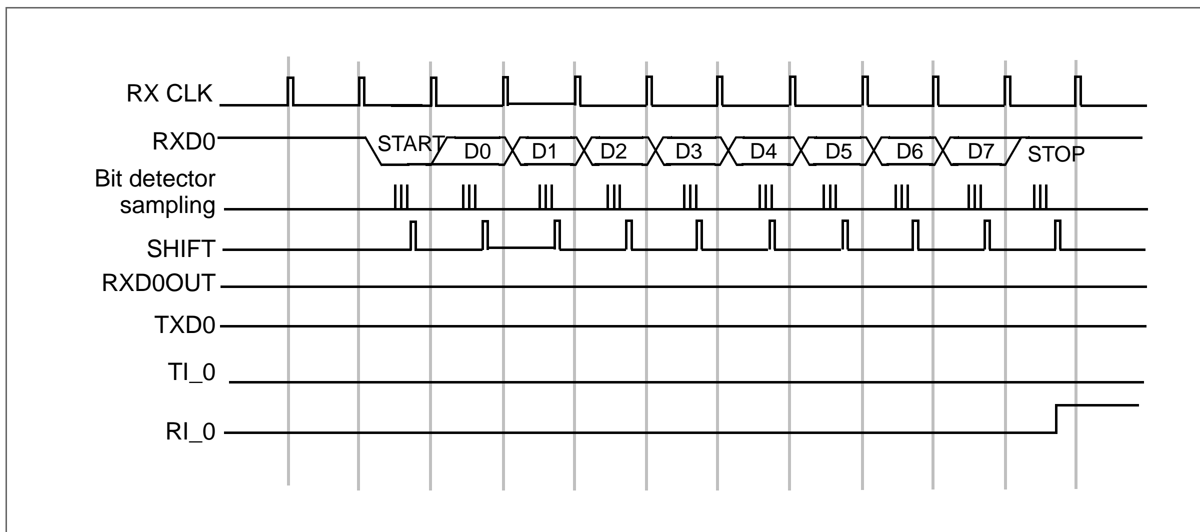





Exhibit 2033 - Page 233 of 346

### 14.3.6 Mode 2

Mode 2 provides asynchronous, full-duplex communication, using a total of 11 bits: 1 start bit, 8 data bits, a programmable 9th bit, and 1 stop bit. The data bits are transmitted and received LSB first. For transmission, the 9th bit is determined by the value in TB8_0 (or TB8_1). To use the 9th bit as a parity bit, move the value of the P bit (SFR PSW.0) to TB8_0 (or TB8_1).

The Mode 2 baud rate is either CLKOUT/32 or CLKOUT/64, as determined by the SMOD0 (or SMOD1) bit. The formula for the mode 2 baud rate is:

$$Baud\ Rate = \frac{2^{SMODx} \times CLKOUT}{64} \qquad \textbf{Equation 8}$$

Mode 2 operation is identical to the standard 8051.

#### *14.3.6.1 Mode 2 Transmit*

Figure 14-13 on page 235 illustrates the mode 2 transmit timing. Transmission begins after the first rollover of the divide-by-16 counter following a software write to SBUF0 (or SBUF1). The USART shifts data out on the TXD0 (or TXD1) pin in the following order: start bit, data bits (LSB first), 9th bit, stop bit. The TI_0 (or TI_1) Bit is set when the stop bit is placed on the TXD0 (or TXD1) pin.

#### *14.3.6.2 Mode 2 Receive*

Figure 14-14 on page 235 illustrates the mode 2 receive timing. Reception begins at the falling edge of a start bit received on the RXD0 (or RXD1) pin, when enabled by the REN_0 (or REN_1) Bit. For this purpose, the RXD0 (or RXD1) pin is sampled 16 times per bit for any baud rate. When a falling edge of a start bit is detected, the divide-by-16 counter used to generate the receive clock is reset to align the counter rollover to the bit boundaries.

For noise rejection, the serial port establishes the content of each received bit by a majority decision of 3 consecutive samples in the middle of each bit time. For the start bit, if the falling edge on the RXD0 (or RXD1) pin is not verified by a majority decision of 3 consecutive samples (low), then the serial port stops reception and waits for another falling edge on the RXD0 (or RXD1) pin.

At the middle of the stop bit time, the serial port checks for the following conditions:

- RI_0 (or RI_1) = 0
- If SM2_0 (or SM2_1) = 1, the state of the stop bit is 1.
  (If SM2_0 (or SM2_1) = 0, the state of the stop bit does not matter.)

If the above conditions are met, the serial port then writes the received byte to the SBUF0 (or SBUF1) Register, loads the stop bit into RB8_0 (or RB8_1), and sets the RI_0 (or RI_1) Bit. If the above conditions are not met, the received data is lost, the SBUF Register and RB8 Bit are not loaded, and the RI Bit is not set. After the middle of the stop bit time, the serial port waits for another high-to-low transition on the RXD0 *(*or RXD1) pin.



Exhibit 2033 - Page 234 of 346

Figure 14-13. Serial Port 0 Mode 2 Transmit Timing

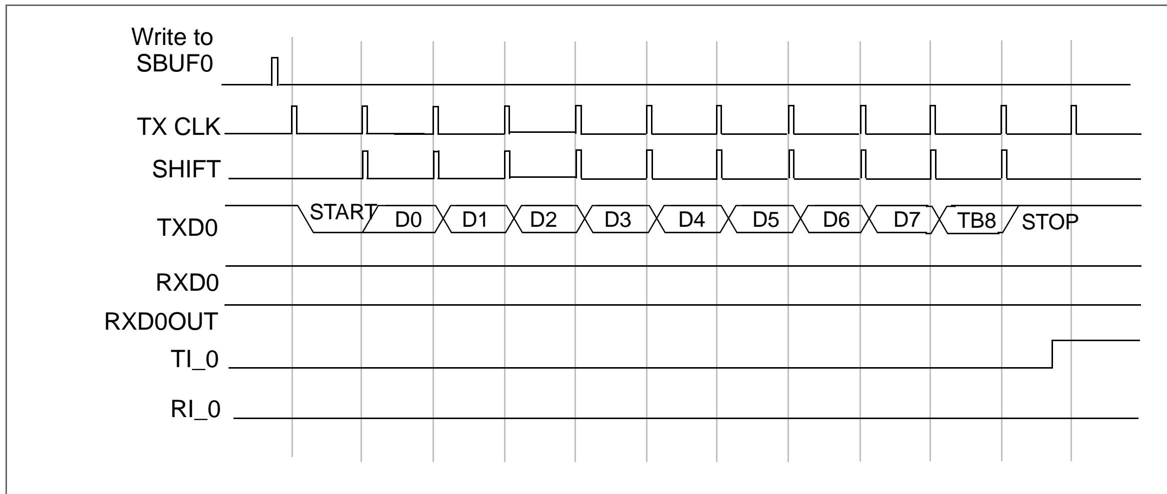



Figure 14-14. Serial Port 0 Mode 2 Receive Timing

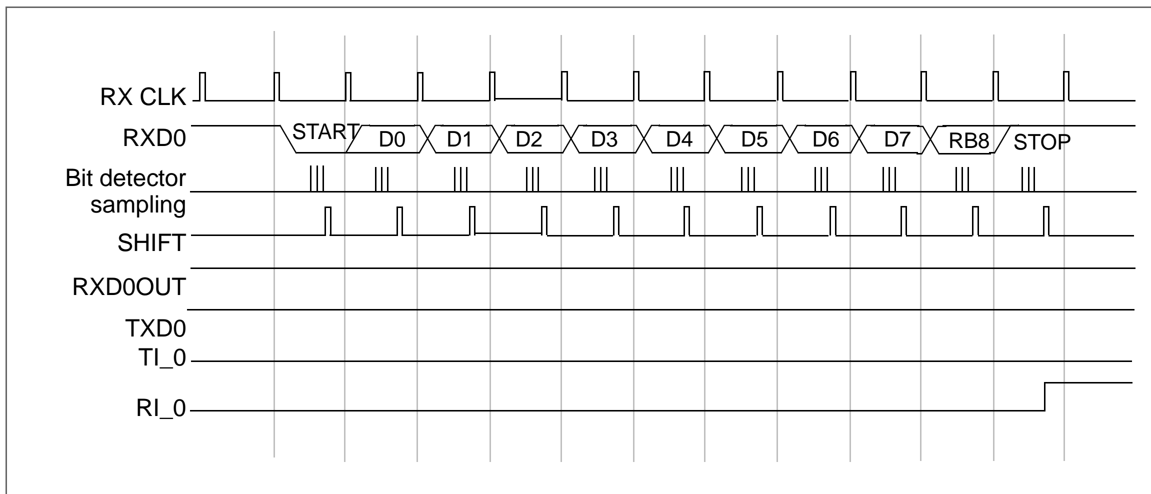





**Exhibit 2033 - Page 235 of 346**

### 14.3.7 Mode 3

Mode 3 provides asynchronous, full-duplex communication, using a total of 11 bits: 1 start bit, 8 data bits, a programmable 9th bit, and 1 stop bit. The data bits are transmitted and received LSB first.

The mode 3 transmit and operations are identical to mode 2. The mode 3 baud rate generation is identical to mode 1. That is, mode 3 is a combination of mode 2 protocol and mode 1 baud rate. Figure 14-15 illustrates the mode 3 transmit timing. Figure 14-16 illustrates the mode 3 receive timing.

Mode 3 operation is identical to that of the standard 8051 when Timer 1 uses CLKOUT/12, (T1M=0, the default).

Figure 14-15. Serial Port 0 Mode 3 Transmit Timing

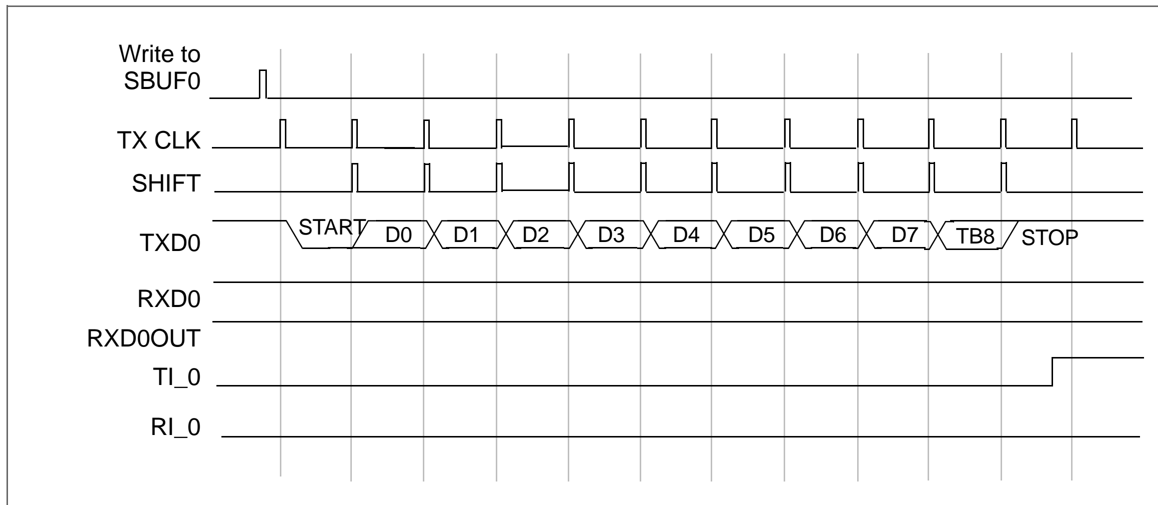



Figure 14-16. Serial Port 0 Mode 3 Receive Timing

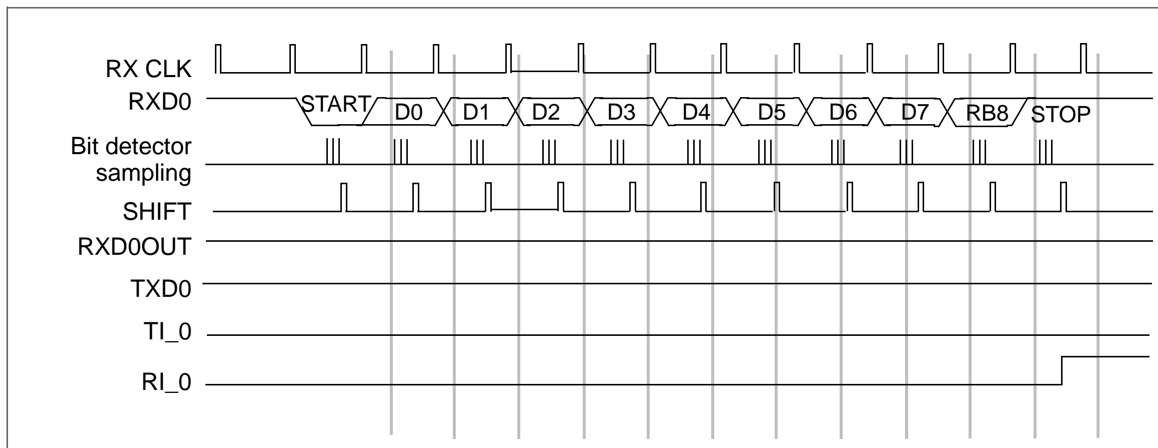





Exhibit 2033 - Page 236 of 346