AN76405

EZ-USB® FX3 Boot Options

Author: Sonia Gandhi
Associated Project: No
Associated Part Family: EZ-USB® FX3
Software Version: EZ-USB FX3 SDK1.2.1
Related Application Notes: AN75705-Getting Started with FX3

AN76405 describes the different boot options available for the Cypress EZ-USB® FX3 peripheral controller.

Contents

Introduction 1
FX3 Boot Options 1
USB Boot 2
PMODE Pins 3
Features of USB Boot 3
USB Checksum Calculation 6
Boot Image Format 9
I^2C EEPROM Boot 10
Features 10
Storing Firmware Image on EEPROM 11
Boot Image Format For I^2C EEPROM Boot 12
Checksum Calculation 13
I^2C EEPROM Boot with USB Fallback 15
SPI Boot 16
Features 16
Storing Firmware Image on SPI Flash/EEPROM 17
Boot Image Format 18
Checksum Calculation 19
SPI Boot with USB Fallback 20
Sync ADMUX Boot 21
Boot Image Format 31
Default State of IOs During Boot 32
Appendix 34
Steps for Booting using FX3 DVK Board 34
USB Boot 35
I^2C Boot 41
SPI Boot 47
Document History 54
Worldwide Sales and Design Support 55

Introduction

EZ-USB® FX3 is the next generation USB 3.0 peripheral controller, providing highly integrated and flexible features that enable developers to add USB 3.0 functionality to a wide range of applications.

FX3 supports several boot options including booting over I^2C, SPI, USB, Synchronous ADMux and Asynchronous SRAM interfaces. This application note describes the details of the different booting options for FX3.

The default state of the FX3 IOs during boot are also documented. The Appendix describes the step-wise sequence for testing the different boot modes using the FX3 DVK.

FX3 Boot Options

FX3 integrates a bootloader which resides in masked ROM. The function of the bootloader is to download the FX3 firmware image from various interfaces such as USB, I^2C, SPI or GPIF II interfaces (for example Synchronous ADMux, Asynchronous SRAM or Asynchronous ADMux).

The FX3 bootloader uses the three PMODE input pins of FX3 to determine the boot option to be used.

This application note describes the details of the USB boot, I^2C boot, SPI boot and Synchronous ADMux boot options. Table 1 lists these boot options along with the required PMODE pin settings.

Table 1. Boot Options for FX3

PMODE[2:0] Pins			Boot Option	USB Fallback
PMODE[2]	PMODE[1]	PMODE[0]		
Z	0	0	Sync ADMUX (16-bit)	No
Z	1	1	USB Boot	Yes
1	Z	Z	I^2C	No
Z	1	Z	I^2C => USB	Yes
0	Z	1	SPI => USB	Yes
Other combinations are reserved.				

(Z=Float. The PMODE pin can be made to float either by leaving it unconnected, or by connecting it to an FPGA IO and then configuring that IO as an input to the FPGA.)

Note In addition to the above, FX3 also supports booting from Asynchronous SRAM and Asynchronous ADMux interfaces. Please contact Cypress Applications Support for details.

The following sections of this application note describe the boot options supported by FX3:

1. USB Boot

   The FX3 firmware image is downloaded into FX3's system RAM from the USB host.

2. I2C Boot

   FX3's firmware image is programmed into an external I^2C EEPROM and, on reset, FX3's bootloader downloads the firmware over I2C.

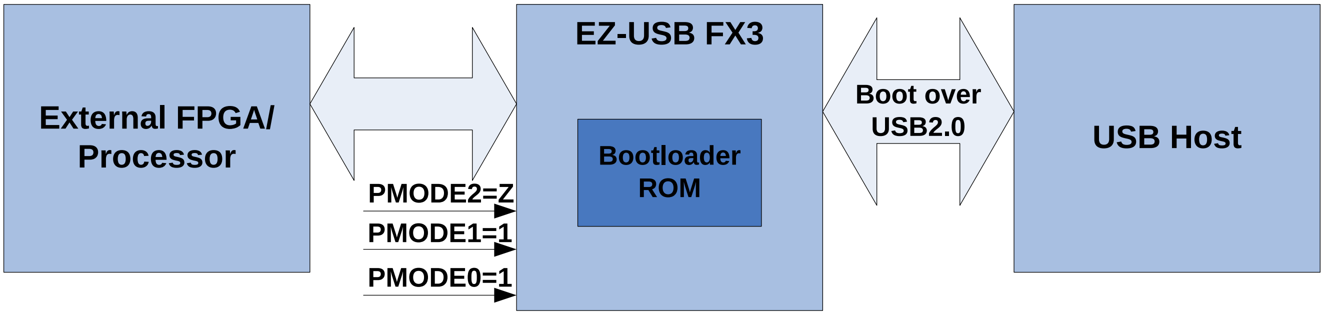
3. SPI Boot

   FX3's firmware image is programmed into an external SPI Flash or SPI EEPROM and, on reset, FX3's bootloader downloads the firmware over SPI.

USB Boot

Figure 1 shows the system diagram for FX3 when booting over USB.

Figure 1. FX3 System Diagram



PMODE Pins

For USB boot, the state of the PMODE[2:0] pins should be Z11.

Table 2. PMODE Pins for USB Boot

PMODE[2]	PMODE[1]	PMODE[0]
Z	1	1

Note Z=Float

Features of USB Boot

The external USB host can download the firmware image to FX3 in USB 2.0 mode. FX3 enumerates in USB Vendor Specific mode with bus-powered support.

The state of FX3 in USB boot mode follows:

- USB3.0 (Super Speed) signaling is disabled
- USB2.0 (High-speed/Full Speed) is enabled
- The FX3 uses the vendor command A0h for firmware download/upload. This vendor command is implemented in the bootloader. (Unlike FX2LP, the A0h vendor command is implemented in firmware i.e. in the bootloader code.)

Default Silicon ID

By default, FX3 has the default Cypress Semiconductor VID=04B4h and PID=00F3h stored in the ROM space. This VID/PID is used for default USB enumeration unless the eFUSE[1] VID/PID is programmed. The default Cypress ID values should only be used for development purposes. Users must use their own VID/PID for final products. A VID is obtained through registration with the USB-IF.

Bootloader Revision

The bootloader revision is stored in the ROM area at the address: FFFF_0020h.

Table 3. Bootloader Revision

Minor Revision	FFFF_0020h
Major Revision	FFFF_0021h
Reserved bytes	FFFF_0022h, FFFF_0023h

ReNumeration™

Cypress's ReNumeration feature is supported in FX3 and is controlled by firmware.

[1] eFUSE is the technology which allows reprogramming of certain circuits in the chip. Contact your Cypress representative for details on eFUSE programming.

When first plugged into a USB host, the FX3 enumerates automatically with its default USB descriptors. Once firmware is downloaded, the FX3 enumerates again, this time as a device defined by the downloaded USB descriptor information. This two-step process is called ReNumeration.

Bus-Powered Applications

The bootloader enumerates in bus-powered mode. The FX3 can fully support bus-powered designs by enumerating with less than 100 mA, as required by the USB 2.0 specification.

USB Fallback Options (=> USB)

When booting over other options with USB fallback enabled, FX3 will fall back to the same USB boot mode described in this section. The operating current might be slightly higher than USB boot mode due to other clock sources being turned on.

USB with VID/PID Options

The bootloader supports booting with a new VID/PID that may be stored in:

- I^2C EEPROM (see the I^2C Boot Option section of this application note)
- SPI EEPROM (see the SPI Boot Option section of this application note)
- eFUSE (VID/PID): Contact Cypress Sales for custom eFUSE VID/PID programming.

USB Default Device

The FX3 bootloader consists of a single USB configuration containing one interface (interface 0) and alternative setting of 0. In this mode, only endpoint 0 is enabled. All other endpoints are turned off.

USB Setup Packet

The FX3 bootloader decodes the SETUP packet that contains an 8-byte data structure defined as follows.

Table 4. Setup Packet

Byte	Field	Description
0	**bmRequestType**	Request Type: Bit7: Direction Bit6-0: Recipient.
1	**bRequest**	This byte will be A0h for firmware download/upload vendor command
2-3	**wValue**	16-bit value (little endian format)
4-5	**wIndex**	16-bit value (little endian format)
6-7	**wLength**	Number of bytes

Note Refer to USB 2.0 spec for the bit-wise explanation

USB Chapter9 and Vendor Commands

FX3 bootloader handles the following commands:

Table 5. USB Commands

bRequest	Descriptions
00	GetStatus: Device, Endpoints and Interface
01	ClearFeature: Device, Endpoints
02	Reserved: returns STALL
03	SetFeature: Device, Endpoints
04	Reserved: returns STALL
05	SetAddress: Handle in FX3 hardware
06	GetDescriptor: Devices descriptors in ROM
07	Reserved: returns STALL
08h	GetConfiguration: returns internal value
09h	SetConfiguration: sets internal value
0Ah	GetInterface: returns internal value
0Bh	SetInterface: sets internal value
0Ch	Reserved: returns STALL
20h-9Fh	Reserved: returns STALL
A0h	Vendor Commands: firmware upload/download, and etc.
A1h-FFh	Reserved: returns STALL

USB Vendor Commands

The bootloader supports the A0h vendor command for firmware download and upload. The fields for the command are shown as follows:

Table 6. Command Fields for Firmware Download

Byte	Field	Value	Description
0	***bmRequest Type***	40h	Request Type: Bit7: Direction Bit6-0: Recipient.
1	***bRequest***	A0h	This byte will be A0 for firmware download/upload vendor command
2-3	***wValue***	AddrL (LSB)	16-bit value (little endian format)
4-5	***wIndex***	AddrH (MSB)	16-bit value (little endian format)
6-7	***wLength***	Count	Number of bytes

Table 7. Command Fields for Firmware Upload

Byte	Field	Value	Description
0	***bmRequest Type***	C0h	Request Type: Bit7: Direction Bit6-0: Recipient.
1	***bRequest***	A0h	This byte will be A0 for firmware download/upload vendor command
2-3	***wValue***	AddrL (LSB)	16-bit value (little endian format)
4-5	***wIndex***	AddrH (MSB)	16-bit value (little endian format)
6-7	***wLength***	Count	Number of bytes

Table 8. Command fields for Transfer of Execution to Program Entry

byte	Field	Value	Description
0	***bmRequest Type***	40h	Request Type: Bit7: Direction Bit6-0: Recipient.
1	***bRequest***	A0h	This byte will be A0 for firmware download/upload vendor command
2-3	***wValue***	AddrL (LSB)	32-bit Program Entry
4-5	***wIndex***	AddrH (MSB)	32-bit Program Entry>>16
6-7	***wLength***	0	This field must be zero

In the transfer execution entry command, bootloader will turn off all the interrupts and disconnect the USB.

Three examples of vendor command subroutines follow:

Example 1.Vendor command Write data protocol with 8–byte setup packet:

bmRequestType=0x40
bRequest = 0xA0;
wValue = (WORD)address;
wIndex = (WORD)(address>>16);
wLength = 1 to 4K-byte max

This command will send DATA OUT packets with length of transfer equal to wLength and a DATA IN Zero length packet.

Example 2. Reading Bootloader revision with setup packet:

bmRequestType= 0xC0
bRequest = 0xA0;
wValue = (WORD)0x0020;
wIndex = (WORD)0xFFFF;
wLength = 4

This command will issue DATA IN packets with length of transfer equal to wLength and a DATA OUT Zero length packet.

Example 3. Jump to program entry with 8–byte Setup packet:
bmRequestType= 0x40
bRequest = 0xA0;
wValue = Program Entry (16-bit LSB)
wIndex = Program Entry>>16 (16-bit MSB)
wLength = 0

Note The FX2LP only uses16-bit address, but 32-bit addressing is used in FX3. Addresses should be written to wValue and wIndex fields of the command.

USB Download Sample Code

Note To download the code, the application should read the firmware image file and write 4 K sections at a time using the vendor write command. The size of the section is limited to the size of the buffer used in the bootloader. Note the firmware image must be in the format specified in

Note Table 14.

The following is an example of how the firmware download routine can be implemented:

```
DWORD dCheckSum, dExpectedCheckSum, dAddress, i, dLen;
WORD wSignature, wLen;
DWORD dImageBuf[512*1024];
BYTE  *bBuf, rBuf[4096];

fread(&wSignature,1,2,input_file); // read signature bytes
if (wSignature != 0x5943)          // check 'CY' signature byte
{
   printf("Invalid image");
   return fail;
}
fread(&i, 2, 1, input_file);     // skip 2 dummy bytes
dCheckSum = 0;
while (1)
{
   fread(&dLength,4,1,input_file);  // read dLength
   fread(&dAddress,4,1,input_file); // read dAddress
   if (dLength==0) break;           // done
   // read sections
```

```
    fread(dImageBuf, 4, dLength,  input_file);
    for (i=0; i<dLength; i++) dCheckSum += dImageBuf[i];
    dLength <<= 2;  // convert to Byte length
    bBuf = (BYTE*)dImageBuf;

    while (dLength > 0)
    {
        dLen = 4096;  // 4K max
        if (dLen > dLength) dLen = dLength;
        VendorCmd(0x40, 0xa0, dAddress, dLen, bBuf); // Write data
        VendorCmd(0xc0, 0xa0, dAddress, dLen, rBuf); // Read data
        // Verify data: rBuf with bBuf
        for (i=0; i<dLen; i++)
        {
           if (rBuf[i] != bBuf) { printf("Fail to verify image"); return fail; }
       }
       dLength -= dLen;
       bBuf += dLen;
       dAddress += dLen;
    }
}

// read pre-computed checksum data
fread(&dExpectedChecksum, 4, 1, input_file);
if  (dCheckSum != dExpectedCheckSum)
{
   printf("Fail to boot due to checksum error\n");
  return fail;
}
// transfer execution to Program Entry
VendorCmd(0x40, 0xa0, dAddress, 0, NULL);
```

input_file is the FILE pointer that points to the firmware image file which is of the format specified in

Table 14.

USB Checksum Calculation

In the USB download, the download tool is expected to handle the checksum computation as shown in the USB download sample code.

USB Bootloader Memory Allocation

The FX3 bootloader allocates 1280 bytes of data tightly-coupled memory (DTCM) from 0x1000_0000 to 0x1000_04FF for its variables and stack. The firmware application can use it as long as this area remains uninitialized that is, (un-initialized local variables) during the firmware download.

The bootloader allocates the first 16-byte from 0x4000_0000 to 0x4000_000F for Warm-boot and standby boot. These bytes should not be used by firmware applications.

The bootloader allocates around 10 K bytes from 0x4000_23FF for its internal buffers. The firmware application can use this area as the uninitialized local variables/buffers.

The bootloader does not use the instruction tightly-coupled memory (ITCM).

USB eFUSE VID/PID Boot Option

The FX3 bootloader can boot with user's choice of VID and PID by scanning the e-Fuse (eFUSE_USB_ID) to see whether the USB_VID bits are programmed. If they are, the bootloader will use the eFUSE value for VID and PID.

USB Memory Access

The FX3 bootloader allows read access from ROM, MMIO, SYSMEM, ITCM, and DTCM memory spaces.

USB Registers/Memory Access

The FX3 bootloader allows read access from ROM, MMIO, SYSMEM, ITCM, and DTCM memory spaces.

The bootloader allows write access to MMIO, SYSMEM, ITCM, and DTCM memory spaces except the first 1280-byte of DTCM and first 10 K of system memory. When writing to MMIO space, the expected transfer length for Bootloader must be four (equal to LONG word) and the address should be aligned by 4 bytes.

USB OTG

The FX3 bootloader does not support OTG protocol. It always operates as a USB Bus power device.

Boot Loader Limitations

The FX3 bootloader handles limited checking of the address range. Accessing non-existing addresses can lead to unpredictable results.

The bootloader does not check the Program Entry. Invalid Program Entry can lead to unpredictable results.

The bootloader allows write access to MMIO register spaces. Write accesses to invalid addresses can lead to unpredictable results.

USB Watchdog timer

FX3 USB hardware requires a 32-kHz clock input to the USB Core hardware. The bootloader will configure the watchdog timer to become the internal 32 kHz for the USB core if the external 32-kHz clock is not present.

USB Suspend/Resume

The FX3 bootloader will enter suspend if there is no activity on USB. It will resume when the PC resumes the USB operation.

USB Additional PID

The bootloader may boot with VID=0x04B4/PID=0x00BC or VID=0x04B4/PID=0x0053 based on the setting of the PMODE pins.

USB Wall Charger Detection

When connecting FX3 to a wall charger, the bootloader will enter suspend mode and set O[60] (Charger detection output) pin to logic ‘1’. When connecting FX3 to a USB host, the bootloader will resume normal operation and set O[60] pin to logic ‘0’.

USB Device Descriptors

The following tables are the FX3 descriptors for high-speed and full speed.

Note The Device Qualifier is not available in the full speed mode.

Table 9. Device Descriptor

offset	Field	Value	Description
0	***bLength***	12h	Length of this descriptor=18 bytes
1	***bDescType***	01	Descriptor Type = Device
2-3	***wBCDUSB***	0200h	USB Specification Version 2.00
4	***bDevClass***	00	Device Class (No Class Specific protocol is implemented)
5	***bDevSubClass***	00	Device Subclass (No Class Specific protocol is implemented)
6	***bDevProtocol***	00	Device Protocol (No Class Specific Protocol is implemented)
7	***bMaxPktSize***	40h	Endpoint0 packet size is 64
8-9	***wVID***	4B4h	Cypress Semiconductor VID
10-11	***wPID***	00F3h	FX3 Silicon
12-13	***wBCDID***	0100h	FX3 bcdID
14	***iManufacture***	01h	Manufacturer Index String = 01
15	***iProduct***	02h	Serial Number Index String = 02
16	***iSerialNum***	03h	Serial Number Index String = 03
17	***bNumConfig***	01h	One configuration

Table 10. Device Qualifier

offset	Field	Value	Description
0	***bLength***	0Ah	Length of this descriptor=10 bytes
1	***bDescType***	06	Descriptor Type = Device Qualifier
2-3	***wBCDUSB***	0200h	USB Specification Version 2.00
4	***bDevClass***	00	Device Class (No Class Specific protocol is implemented)
5	***bDevSubClass***	00	Device Subclass (No Class Specific protocol is implemented)
6	***bDevProtocol***	00	Device Protocol (No Class Specific Protocol is implemented)
7	***bMaxPktSize***	40h	Endpoint0 packet size is 64
8	***bNumConfig***	01h	One configuration
9	***bReserved***	00h	Must be zero

Table 11. Configuration Descriptor

offset	Field	Value	Description
0	***bLength***	09h	Length of this descriptor=10 bytes
1	***bDescType***	02h	Descriptor Type = Configuration
2-3	***wTotalLength***	0012h	Total Length
4	***bNumInterfaces***	01	Number of Interfaces in this Configuration
5	***bConfigValue***	01	Configuration value used by SetConfiguration Request to select this interface
6	***bConfiguration***	00	Index of String Describing this Configuration = 0
7	***bAttribute***	80h	Attributes - Bus Powered, No Wakeup
8	***bMaxPower***	64h	Maximum Power - 200 mA

Table 12. Interface Descriptor (Alt. Setting 0)

offset	Field	Value	Description
0	***bLength***	09h	Length of this descriptor=10 bytes
1	***bDescType***	04h	Descriptor Type = Interface
2	***bInterfaceNum***	00h	Zero based index of this interface=0
4	***bAltSetting***	00	Alternative Setting value = 0
5	***bNumEndpoints***	00	Only endpoint0
6	***bInterfaceClass***	FFh	Vendor Command Class
7	***bInterfaceSubClass***	00h	
8	***bInterfaceProtocol***	00h	
9	***iInterface***	00h	None

Table 13. String Descriptors

offset	Field	Value	Description
0	***bLength***	04h	Length of this descriptor=04 bytes
1	***bDescType***	03h	Descriptor Type = String
2-3	***wLanguage***	0409h	Language = English
4	***bLength***	10h	Length of this descriptor=16 bytes
5	***bDescType***	03h	Descriptor Type = String
6-21	***wStringIdx1***	--	“Cypress”
22	***bLength***	18h	Length of this descriptor=24 bytes
23	***bDescType***	03h	Descriptor Type = String
24-47	***wStringIdx2***	--	“WestBridge ”
48	***bLength***	1Ah	Length of this descriptor=26 bytes
49	***bDescType***	03h	Descriptor Type = String
50-75	***wStringIdx3***	--	“0000000004BE”

Boot Image Format

For USB boot, the booloader expects the firmware image file to be in the format shown in Table 14. The EZ-USB FX3 SDK provides a software utility that can be used to generate a firmware image in the format required for USB boot. Please refer to the elf2img utility found in the [install path] \util\elf2img directory after installing the SDK.

Table 14. Boot Image Format

Binary Image Header	Length (16-bit)	Description
wSignature	1	Signature 2 bytes initialize with “CY” ASCII text
bImageCTL;	½	Bit0 = 0: execution binary file; 1: data file type Bit3:1 Not use when booting in SPI EEPROM Bit5:4(SPI speed): 00: 10 MHz 01: 20 MHz 10: 30 MHz 11: Reserved Bit7:6: Reserved should be set to zero
bImageType;	½	bImageType = 0xB0: normal FW binary image with checksum bImageType = 0xB1: Reserved for security image type bImageType = 0xB2: SPI boot with new VID and PID
dLength 0	2	1st section length, in long words (32-bit) When bImageType = 0xB2, the dLength 0 will contain PID and VID. Bootloader ignores the rest of the any following data.
dAddress 0	2	1st section address of Program Code. Note: Internal ARM address is byte addressable, so the address for each section should be 32-bit aligned
dData[dLength 0]	dLength 0*2	Image Code/Data must be 32-bit aligned
…		More sections
dLength N	2	0x00000000 (Last record: termination section)
dAddress N	2	Should contain valid Program Entry (Normally, it should be the Startup code i.e. the RESET Vector) Note: if bImageCTL.bit0 = 1, the bootloader will not transfer the execution to this Program Entry. If bImageCTL.bit0 = 0, the bootloader will transfer the execution to this Program Entry: This address should be in ITCM area or SYSTEM RAM area Boot Loader does not validate the Program Entry
dCheckSum	2	32-bit unsigned little endian checksum data will start from the 1st sections to termination section. The checksum will not include the dLength, dAddress, and Image Header

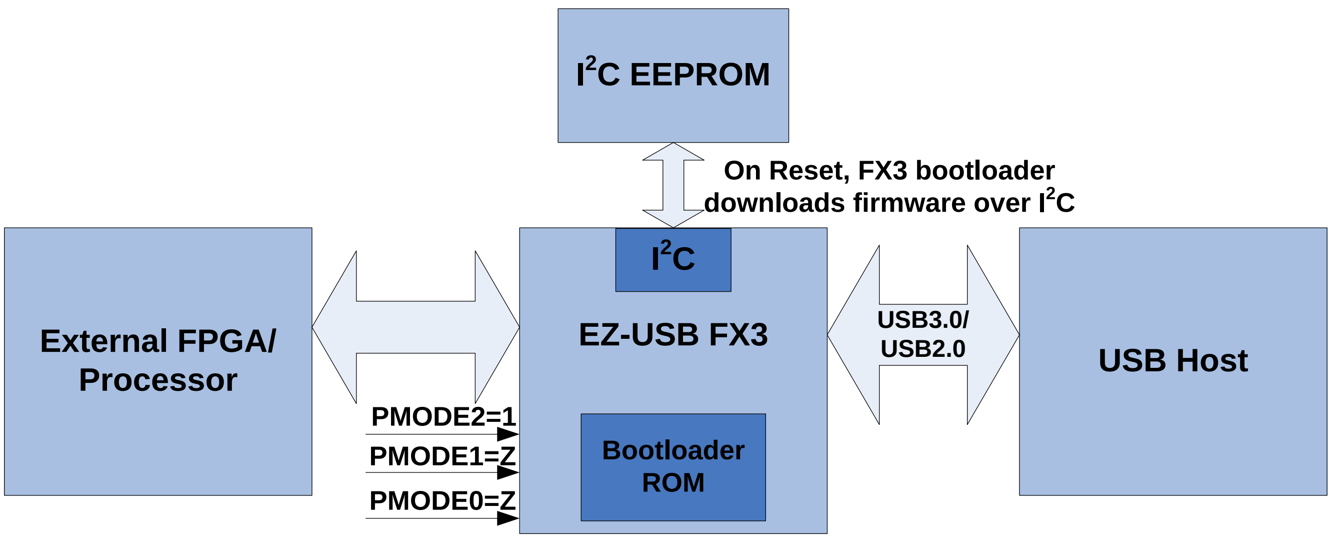
Example of boot image format organized in long word format:

Location1: 0xB0 0x10 'Y' 'C'	//CY Signature, 20 MHz, 0xB0 Image
Location2: 0x00000004	//Image length of section 1 = 4
Location3: 0x40008000	//1st section stored in SYSMEM RAM at 0x40008000
Location4: 0x12345678	//Image starts (Section1)
Location5: 0x9ABCDEF1	
Location6: 0x23456789	
Location7: 0xABCDEF12	//Section 1 ends
Location8: 0x00000002	//Image length of section 2 = 2
Location9: 0x40009000	//2nd section stored in SYSMEM RAM at 0x40009000
Location10: 0xDDCCBBAA	//Section 2 starts
Location11: 0x11223344	
Location12: 0x00000000	//Termination of Image
Location13: 0x40008000	//Jump to 0x40008000 on FX3 System RAM
Location 14: 0x6AF37AF2	//Checksum (0x12345678 + 0x9ABCDEF1 + 0x23456789 + 0xABCDEF12+ 0xDDCCBBAA +0x11223344)

I^{2}C EEPROM Boot

Figure 2 shows the system diagram for FX3 when booting over I^{2}C.

Figure 2. FX3 System Diagram for I^{2}C Boot



For I^{2}C EEPROM boot, the state of the PMODE[2:0] pins should be 1ZZ.

Table 15. PMODE Pins for I^{2}C Boot

PMODE[2]	PMODE[1]	PMODE[0]
1	Z	Z

Features

- FX3 boots from I^{2}C EEPROM devices through a two-wire I^{2}C interface.
- EEPROM[2] device sizes supported are:
  - 32 Kilobit (Kb) or 4 Kilobyte (KB)
  - 64 Kb or 8 KB
  - 128 Kb or 16 KB
  - 256 Kb or 32 KB
  - 512 Kb or 64 KB
  - 1024 Kb or 128 KB
- ATMEL and Microchip devices are supported
- 100 kHz, 400 kHz, and 1 MHz I^{2}C frequencies are supported during boot. Note that when V_{IO5} is 1.2 V, the maximum operating frequency supported is

[2] Only 2-byte I2C addressees are supported. Single byte address is not supported for any I2C EEPROM size less than 32 Kbit

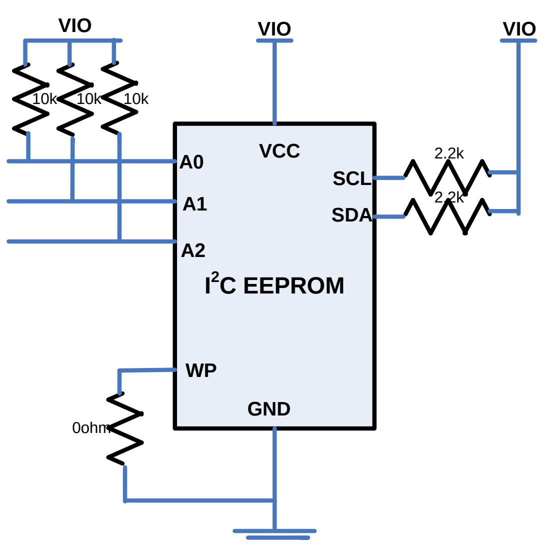
100 kHz. When V_{IO5} is 1.8 V, 2.5 V, or 3.3 V, the operating frequencies supported are 400 kHz and 1 MHz. (V_{IO5} is the I/O voltage for I^{2}C interface)

- Supports boot from multiple I^{2}C EEPROM devices of the same size. When I^{2}C EEPROM is smaller than the firmware image, multiple I^{2}C EEPROM devices must be used. The bootloader supports loading the image across multiple I^{2}C EEPROM devices. The bootloader can support up to eight I^{2}C EEPROM devices smaller than 128 Kbytes. The bootloader can support up to four I^{2}C EEPROM devices of128-Kbyte.
- Only one firmware image can be stored on I^{2}C EEPROM. No redundant images are allowed.
- Bootloader does not support the multimaster I^{2}C feature of FX3. Therefore, during the FX3 I^{2}C booting process, other I^{2}C masters should not perform any activity on the I^{2}C bus.

Storing Firmware Image on EEPROM

The FX3 bootloader supports a master I^{2}C interface for external serial I^{2}C EEPROM devices. The serial I^{2}C EEPROM can be used to store application specific code and data. The following diagram shows the pin connections of a typical I^{2}C EEPROM.

Figure 3. Pin Connections of a Typical I^{2}C EEPROM



The I^{2}C EEPROM interface consists of two active wires: serial clock line (SCL) and serial data line (SDA).

The Write Protect (WP) pin should be pulled low while writing the firmware image on to EEPROM.

The A0, A1, and A2 pins are the address lines. They set the slave device address from 000 to111. This makes it possible to address eight I^{2}C EEPROMs of the same size.

These lines should be pulled high or low based on the address required. Table 16 shows how eight 24LC256 EEPROM devices can be connected.

Table 16. 24LC256 EEPROM Device Connections

Device No.	Address Range	A2	A1	A0	Size
1	0x0000-0x7FFF	0	0	0	32 Kbytes
2	0x7FFF-0xFFFF	0	0	1	32 Kbytes
3	0xFFFF-0x17FFF	0	1	0	32 Kbytes
4	0x17FFF-0x1FFFF	0	1	1	32 Kbytes
5	0x1FFFF-0x27FFF	1	0	1	32 Kbytes
6	0x27FFF-0x2FFFF	1	1	0	32 Kbytes
7	0x2FFFF-0x37FFF	1	0	1	32 Kbytes
8	0x37FFF-0x3FFFF	1	1	1	32 Kbytes

For example, if the firmware code is 60 Kbyte, you must use two I^{2}C EEPROMs, with the first EEPROM having A[2:0] = b000 and second having A[2:0] = b001. The firmware image should be stored across the EEPROMs as a contiguous image as in a single I^{2}C EEPROM.

Important Points to Note on 128 Kbyte EEPROM Addressing

In the case of a 128-Kbyte I^{2}C EEPROM, the addressing style is not standard across EEPROMs. For example, Microchip EEPROMs use pins A1 and A0 for chip select and pin A2 is unused. However, Atmel EEPROMs use A2 and A1 for chip select and A0 is unused. Both these cases are handled by the bootloader. The addressing style can be indicated in the firmware image header.

Table 17 shows how four Microchip 24LC1024 EEPROM devices can be connected.

Table 17. Microchip 24LC1024 EEPROM Device Connections

Device No.	Address Range	A2	A1	A0	Size
1	0x00000-0x1FFFF	Vcc	0	0	128 Kbytes
2	0x20000-0x3FFFF	Vcc	0	1	128 Kbytes
3	0x40000-0x5FFFF	Vcc	1	0	128 Kbytes
4	0x60000-0x7FFFF	Vcc	1	1	128 Kbytes

Table 18 shows how four Atmel 24C1024 EEPROM devices can be connected.

Table 18. ATMEL 24C1024 EEPROM Device Connections

Device No.	Address Range	A2 A1 A0	Size
1	0x00000-0x1FFFF	0 0 NC	128 Kbytes
2	0x20000-0x3FFFF	0 1 NC	128 Kbytes
3	0x40000-0x5FFFF	1 0 NC	128 Kbytes
4	0x60000-0x7FFFF	1 1 NC	128 Kbytes

Note NC indicates No Connection

Boot Image Format For I²C EEPROM Boot

The booloader expects the firmware image file to be in the format shown in Table 19. The EZ-USB FX3 SDK provides a software utility that can be used to generate a firmware image in the format required for I²C EEPROM boot. Please refer to the elf2img utility found in the [install path] \util\elf2img directory after installing the SDK.

Table 19. Firmware Image Storage Format

Binary Image Header	Length (16-bit)	Description
wSignature	1	Signature 2 bytes initialize with “CY” ASCII text
bImageCTL;	½	Bit0 = 0: execution binary file; 1: data file type Bit3:1 (I2C size) 7: 128KB (Micro chip) 6: 64KB (128K ATMEL) 5: 32KB 4: 16KB 3: 8KB 2: 4KB **Note** Options 1 and 0 are reserved for future usage. Unpredicted result will occurred when booting in these modes. Bit5:4(I2C speed): 00: 100KHz 01: 400KHz 10: 1MHz 11: Reserved **Note** Bootloader power-up default will be set at 100KHz and it will adjust the I²C speed if needed. Bit7:6: Reserved should be set to zero
bImageType;	½	bImageType=0xB0: normal FW binary image with checksum bImageType=0xB1: Reserved for security image type bImageType=0xB2: I²C boot with new VID and PID
dLength 0	2	1st section length, in long words (32-bit) When bImageType = 0xB2, the dLength 0 will contain PID and VID. Boot Loader will ignore the rest of the following data.

Binary Image Header	Length (16-bit)	Description
dAddress 0	2	1st sections address of Program Code not the I2C address. **Note** Internal ARM address is byte addressable, so the address for each section should be 32-bit align
dData[dLength 0]	dLength 0*2	All Image Code/Data also must be 32-bit aligned
…		More sections
dLength N	2	0x00000000 (Last record: termination section)
dAddress N	2	Should contain valid Program Entry (Normally, it should be the Start up code i.e. the RESET Vector) **Note** if bImageCTL.bit0 = 1, the Boot Loader will not transfer the execution to this Program Entry. If bImageCTL.bit0 = 0, the Boot Loader will transfer the execution to this Program Entry: This address should be in ITCM area or SYSTEM RAM area Boot Loader does not validate the Program Entry
dCheckSum	2	32-bit unsigned little endian checksum data will start from the 1st sections to termination section. The checksum will not include the dLength , dAddress and Image Header

Example: The binary image file is stored in the I^{2}C EEPROM in the following order:

Byte0: "C"
Byte1: "Y"
Byte2: bImageCTL
Byte3: bImageType
…..
Byte N: Checksum of Image

Important Notes

- Bootloader default boot speed = 100 KHz; to change the speed from 100 KHz to 1 MHz, the bImageCTL<5:4> should be set to 10.
- To select I^{2}C EEPROM size, the bImageCTL[3:1]should be used.
- The addressing for the Microchip EEPROM 24LC1026 is different from the addressing of other 128kB Microchip EEPROMs. If using Microchip EEPROM 24LC1026, the I^{2}C EEPROM size field, for example, bImageCTL[3:1] should be set to 6.

Checksum Calculation

The bootloader computes the checksum when loading the binary image I^{2}C EEPROM. If the checksum does not match the one in the image, the bootloader does not transfer execution to the program entry.

The bootloader operates in little endian mode; for this reason, the checksum must also be computed in little endian mode.

The 32-bit unsigned little endian checksum data starts from the first sections to the termination section. The checksum does not include the dLength, dAddress and Image Header.

First Example Boot Image

The following image is stored only at one section in the system RAM of FX3 at the location 0x40008000:

Location1: 0xB0 0x1A 'Y' 'C'	//CY Signature, 32KB EEPROM,400Khz,0xB0 Image
Location2: 0x00000004	//Image length =4
Location3: 0x40008000	// 1st section stored in FX3 System RAM at 0x40008000
Location4: 0x12345678	//Image starts
Location5: 0x9ABCDEF1	
Location6: 0x23456789	
Location7: 0xABCDEF12	
Location8: 0x00000000	//Termination of Image
Location9: 0x40008000	//Jump to 0x40008000 in FX3 System RAM
Location 10: 0x7C048C04	//Check sum (0x12345678 + 0x9ABCDEF1 + 0x23456789 + 0xABCDEF12)

Second Example Boot Image

The following image is stored at two sections in the system RAM of FX3 at the locations 0x40008000 and 0x40009000:

Location1: 0xB0 0x1A 'Y' 'C'	//CY Signature, 32KB EEPROM,400Khz,0xB0 Image
Location2: 0x00000004	//Image length of section 1 =4
Location3: 0x40008000	// 1st section stored in FX3 System RAM at 0x40008000
Location4: 0x12345678	//Image starts (Section1)
Location5: 0x9ABCDEF1	
Location6: 0x23456789	
Location7: 0xABCDEF12	//Section 1 ends
Location8: 0x00000002	//Image length of section 2 =2
Location9: 0x40009000	// 2nd section stored in FX3 System RAM at 0x40009000
Location10: 0xDDCCBBAA	//Section 2 starts
Location11: 0x11223344	
Location12: 0x00000000	//Termination of Image
Location13 0x40008000	//Jump to 0x40008000 in FX3 System RAM
Location 14: 0x6AF37AF2	//Check sum (0x12345678 + 0x9ABCDEF1 + 0x23456789 + 0xABCDEF12+ 0xDDCCBBAA +0x11223344)

Similarly, you can have N sections of an image stored using one boot image.

The following section shows the checksum sample code:

```
// Checksum sample code
DWORD dCheckSum, dExpectedCheckSum;
WORD wSignature, wLen;
DWORD dAddress, i;
DWORD dImageBuf[512*1024];

fread(&wSignature,1,2,input_file); // read signature bytes
if (wSignature != 0x5943)          // check 'CY' signature byte
{
   printf("Invalid image");
   return fail;
}
fread(&i, 2, 1, input_file);       // skip 2 dummy bytes
dCheckSum = 0;
while (1)
```

```
{
   fread(&dLength,4,1,imput_file);  // read dLength
   fread(&dAddress,4,1,input_file); // read dAddress
   if (dLength==0) break;           // done
   // read sections
   fread(dImageBuf, 4, dLength,  input_file);
   for (i=0; i<dLength; i++) dCheckSum += dImageBuf[i];
}
// read pre-computed checksum data
fread(&dExpectedChecksum, 4, 1, input_file);
if  (dCheckSum != dExpectedCheckSum)
{
   printf("Fail to boot due to checksum error\n");
  return fail;
      }
```

This section described the details of the I²C boot option. The next section describes the I²C boot option with the USB fallback enabled.

I²C EEPROM Boot with USB Fallback

For the I²C EEPROM boot, the state of the PMODE[2:0] pins should be Z1Z.

Table 20. PMODE Pins for I²C Boot with USB Fallback

PMODE[2]	PMODE[1]	PMODE[0]
Z	1	Z

In all USB Fallback modes (denoted as "=>USB"), USB enumeration occurs if 0xB2 boot is selected or an error occurs. After USB enumeration, the external USB host can boot FX3 using USB Boot. I²C EEPROM boot with USB Fallback (I2C => USB) may also be used to store only Vendor Identification (VID) and Product Identification (PID) for USB Boot.

I²C EEPROM boot fails under the following conditions:

- I²C address cycle or data cycle error.
- Invalid signature in FX3 firmware image.
- Invalid image type.

A special image type is used to denote that instead of the FX3 firmware image, data on EEPROM is the VID and PID for USB boot. This helps in having a new VID and PID for USB Boot.

Features

- In case of USB boot, the bootloader supports only USB2.0. USB 3.0 is not supported.
- If the 0xB2 boot option is specified, the USB descriptor uses the customer defined VID and PID stored as part of the 0xB2 image in the I²C EEPROM.
- On USB fall back, when any error occurs during I²C boot, the USB descriptor uses the VID=0x04B4 and PID=0x00F3.
- The USB Device descriptor is reported as bus-powered, that will consume around 200 mA. However, the FX3 chip is typically observed to consume around 100 mA.

Example Image for boot with VID and PID

Location1: 0xB2 0x1A 'Y' 'C'   //CY Signature,32k EEPROM,400Khz,0xB2 Image

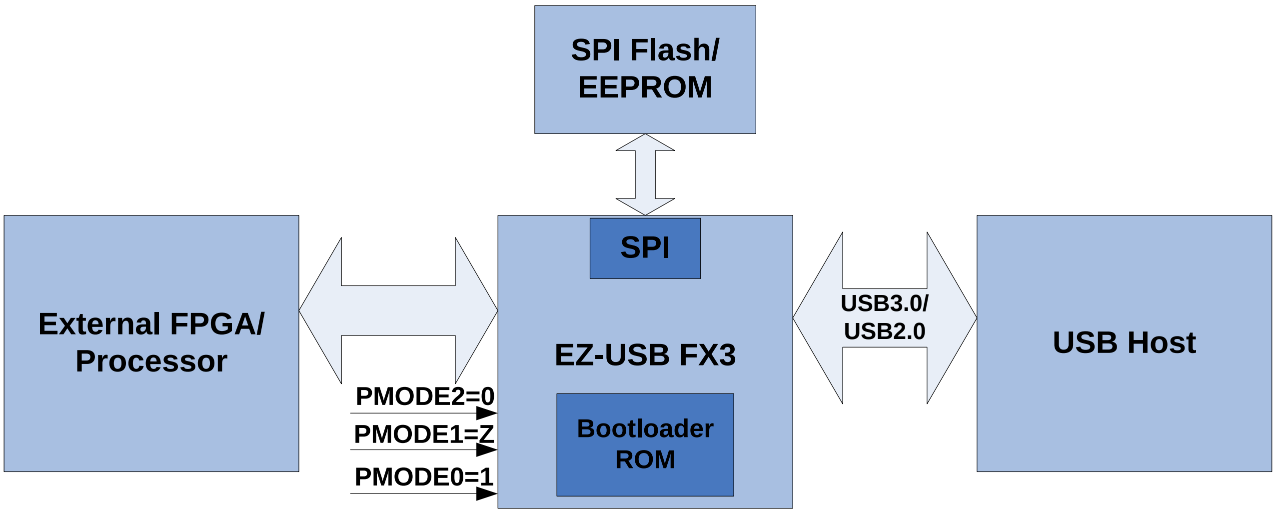
Location2: 0x04B40008   // VID = 0x04B4 | PID=0x0008

.

SPI Boot

Figure 4 shows the system diagram for FX3 when booting over SPI.

Figure 4. System Diagram for SPI Boot



For SPI boot, the state of the PMODE[2:0] pins should be 0Z1.

Table 21. MODE Pins for SPI Boot

PMODE[2]	PMODE[1]	PMODE[0]
0	Z	1

Features

FX3 boots from SPI Flash/EEPROM devices through a 4-wire SPI interface.

- SPI Flash/EEPROM devices from 1 Kbit to 32 Mbit in size are supported for boot.
- SPI devices from Numonyx, Atmel and Microchip are supported. (Please note, SPI boot has been tested with the part numbers M25P16 (16 Mbit), M25P80 (8 Mbit), and M25P40 (4 Mbit), but the equivalents of these parts may also be used.)
- SPI frequencies supported during boot are ~10 MHz, ~20 MHz, and ~30 MHz.

  Please note, the SPI frequency might vary due to a rounding off on the SPI clock divider and clock input.

  - When the crystal or clock input to FX3 is 26 MHz or 52 MHz, the internal PLL runs at 416 MHz. SPI frequencies with PLL_CLK = 416 MHz can be 10.4 MHz, 20.8 MHz, or 34.66 MHz.
  - When the crystal or clock input to FX3 is 19.2 MHz or 38.4 MHz, the internal PLL runs at 384 MHz. SPI frequencies with PLL_CLK = 384 MHz can be: 9.6 MHz, 19.2 MHz, and 32 MHz.
- Operating voltages supported are 1.8 V, 2.5 V, and 3.3 V.
- Only one firmware image is stored on an SPI Flash/EEPROM. No redundant image is allowed.
- For SPI boot, the bootloader sets CPOL=0 and CPHA=0. (For the timing diagram of this SPI mode, please refer to the SPI timing in the FX3 datasheet.)
- USB Fallback is supported used for storing new VID/PID information for USB boot. See the SPI Boot with USB Fallback section in this application note for more information.

Storing Firmware Image on SPI Flash/EEPROM

The FX3 bootloader supports a master SPI controller for interfacing with external serial SPI Flash/EEPROM devices. The SPI Flash/EEPROM can be used to store application specific code and data. Figure 5 shows the pinout of a typical SPI Flash/EEPROM.

The SPI EEPROM interface consists of four active wires:

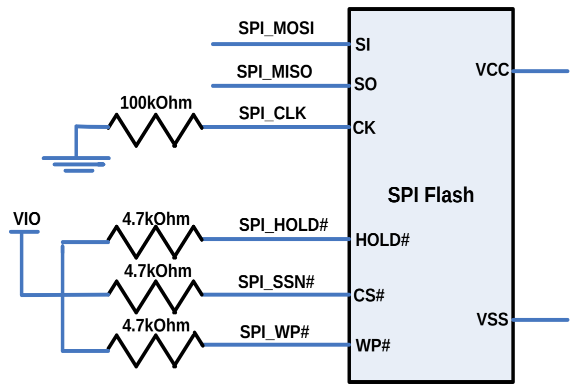
1. CS#: Chip Select
2. SO: Serial Data Output (Master In Slave Out (MISO))
3. SI: Serial Data Input (Master Out Slave In (MOSI))
4. SCK: Serial Clock input

The HOLD# signal should tied to VCC while booting/ or reading from the SPI device

Write Protect (WP#) and HOLD# signals should tied to VCC while writing the image onto EEPROM.

Please note, external pull-ups should not be connected on the MOSI and MISO signals, as shown in Figure 5.

Figure 5. Pin Connections of a Typical SPI Flash



Boot Image Format

For SPI boot, the booloader expects the firmware image file to be in the format shown in Table 22. The EZ-USB FX3 SDK provides a software utility that can be used to generate a firmware image in the format required for SPI boot. Please refer to the elf2img utlity found in the [install path] \util\elf2img directory after installing the SDK.

Table 22. Boot Image Format for SPI Boot Option

Binary Image Header	Length (16-bit)	Description
wSignature	1	Signature 2 bytes initialize with "CY" ASCII text
bImageCTL	½	Bit0 = 0: execution binary file; 1: data file type Bit3:1 Not use when booting from SPI Bit5:4(SPI speed): - 00: 10 MHz - 01: 20 MHz - 10: 30 MHz - 11: Reserved **Note**: Bootloader power-up default is set to 10 MHz and it will adjust the SPI speed if needed. The FX3 SPI hardware can only run up to 33 MHz. Bit7:6: Reserved. Should be set to zero.
bImageType	½	bImageType = 0xB0: normal firmware binary image with checksum bImageType = 0xB1: Reserved for security image type bImageType = 0xB2: SPI boot with new VID and PID
dLength 0	2	1st section length, in long words (32-bit) When bImageType = 0xB2, the dLength 0 will contain PID and VID. Bootloader ignores the rest of any following data.
dAddress 0	2	1st section address of program code Note: Internal ARM address is byte addressable, so the address for each section should be 32-bit aligned
dData[dLength 0]	dLength 0*2	Image Code/Data must be 32-bit aligned
…		More sections
dLength N	2	0x00000000 (Last record: termination section)
dAddress N	2	Should contain valid Program Entry (Normally, it should be the Startup code i.e. the RESET Vector) **Note:** if bImageCTL.bit0 = 1, the bootloader will not transfer the execution to this Program Entry. If bImageCTL.bit0 = 0, the bootloader will transfer the execution to this Program Entry: This address should be in ITCM area or SYSTEM RAM area Bootloader does not validate the Program Entry
dCheckSum	2	32-bit unsigned little endian checksum data will start from the 1st section to termination section. The checksum will not include the dLength, dAddress, and Image Header

Example: The binary image file is stored in the SPI EEPROM in the following order:

Byte0: “C”
Byte1: “Y”
Byte2: bImageCTL
Byte3: bImageType
…..
Byte N: Checksum of Image

Important Points to Note:

- Bootloader default boot speed = 10 MHz; to change the speed from 10 MHz to 20 MHz the bImageCTL[5:4] should be set to 01.

Checksum Calculation

The bootloader computes the checksum when loading the binary image over SPI. If the checksum does not match the one in the Image, the bootloader will not transfer execution to the program entry.

The bootloader operates in little endian mode; for this reason, the checksum must also be computed in little endian mode.

32-bit unsigned little endian checksum data starts from the first section to the termination section. The checksum will not include the dLength, dAddress, and Image Header.

Example 1. Following is an example of a firmware image stored only at one section in System RAM of FX3 at location 0x40008000:

Location1: 0xB0 0x10 ’Y’ ’C’	//CY Signature, 20 MHz, 0xB0 Image
Location2: 0x00000004	//Image length = 4
Location3: 0x40008000	//1st section stored in FX3 System RAM at 0x40008000
Location4: 0x12345678	//Image starts
Location5: 0x9ABCDEF1	
Location6: 0x23456789	
Location7: 0xABCDEF12	
Location8: 0x00000000	//Termination of Image
Location9: 0x40008000	//Jump to 0x40008000 in FX3 System RAM
Location 10: 0x7C048C04	//Checksum (0x12345678 + 0x9ABCDEF1 + 0x23456789 + 0xABCDEF12)

Example 2. Following is an example of a firmware image stored at two sections in System RAM of FX3 at location 0x40008000 and 0x40009000:

Location1: 0xB0 0x10 ’Y’ ’C’	//CY Signature, 20MHz, 0xB0 Image
Location2: 0x00000004	//Image length of section 1 = 4
Location3: 0x40008000	//1st section stored in FX3 System RAM at 0x40008000
Location4: 0x12345678	//Image starts (Section1)
Location5: 0x9ABCDEF1	
Location6: 0x23456789	
Location7: 0xABCDEF12	//Section 1 ends
Location8: 0x00000002	//Image length of section 2 = 2
Location9: 0x40009000	//2nd section stored in FX3 System RAM at 0x40009000
Location10: 0xDDCCBBAA	//Section 2 starts
Location11: 0x11223344	
Location12: 0x00000000	//Termination of Image
Location13: 0x40008000	//Jump to 0x40008000 in FX3 System RAM
Location 14: 0x6AF37AF2	//Checksum (0x12345678 + 0x9ABCDEF1 + 0x23456789 + 0xABCDEF12+

0xDDCCBBAA +0x11223344)

Similarly you can have N sections of an image stored using one boot image.

The following section shows the checksum sample code:

```
// Checksum sample code
DWORD dCheckSum, dExpectedCheckSum;
WORD wSignature, wLen;
DWORD dAddress, i;
DWORD dImageBuf[512*1024];

fread(&wSignature,1,2,input_file); // read signature bytes
if (wSignature != 0x5943)         // check 'CY' signature byte
{
   printf("Invalid image");
   return fail;
}
fread(&i, 2, 1, input_file);       // skip 2 dummy bytes
dCheckSum = 0;
while (1)
{
   fread(&dLength,4,1,imput_file);  // read dLength
   fread(&dAddress,4,1,input_file); // read dAddress
   if (dLength==0) break;           // done
  // read sections
   fread(dImageBuf, 4, dLength,  input_file);
   for (i=0; i<dLength; i++) dCheckSum += dImageBuf[i];
}

// read pre-computed checksum data
fread(&dExpectedChecksum, 4, 1, input_file);

if  (dCheckSum != dExpectedCheckSum)
{
   printf("Fail to boot due to checksum error\n");
   return fail;
}
```

SPI Boot with USB Fallback

In all USB Fallback ("=>USB") modes, USB enumeration occurs if 0xB2 boot is selected or an error occurs. After USB enumeration occurs, the external USB host can boot FX3 using USB Boot. SPI boot with USB Fallback (SPI => USB) is also used to store Vendor Identification (VID) and Product Identification (PID) for USB Boot.

SPI boot fails under the following conditions:

- SPI address cycle or data cycle error.
- Invalid signature on FX3 firmware. Invalid image type.

A special image type is used to denote that instead of the FX3 firmware image, data on SPI Flash/EEPROM is the VID and PID for USB boot. This helps in having a new VID and PID for USB Boot.

- In case of USB boot, the bootloader supports only USB 2.0. USB 3.0 is not supported.
- If the 0xB2 boot option is specified, the USB descriptor uses the customer defined VID and PID stored as part of the 0xB2 image in the SPI Flash/ EEPROM.
- On USB fall back, when any error occurs during I^2C boot, the USB descriptor uses the VID=0x04B4 and PID=0x00F3.
- The USB Device descriptor is reported as bus-powered, that will consume around 200 mA. However, the FX3 chip is typically observed to consume around 100 mA.

Example Image for Boot with VID and PID

Location1: 0xB2 0x10 'Y' 'C' //CY Signature, 20 MHz, 0xB2 Image

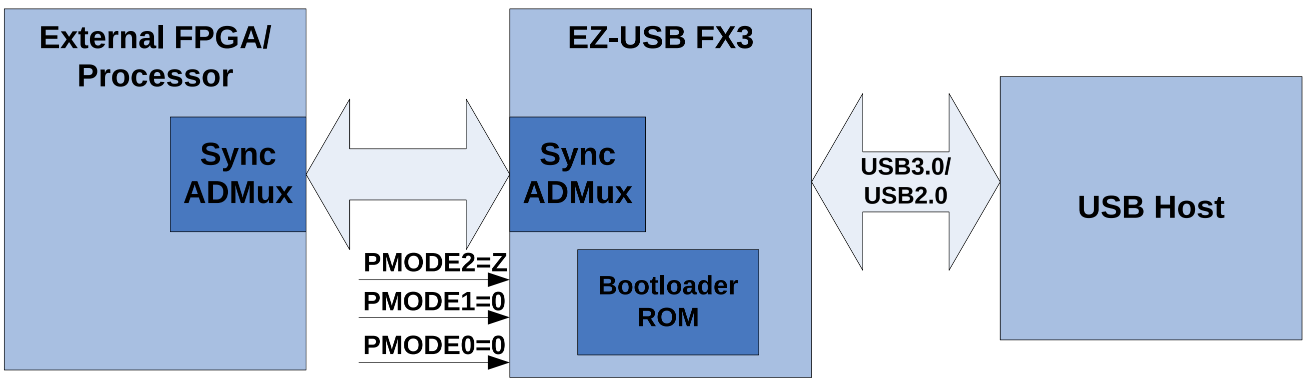
Location2: 0x04B40008 //VID = 0x04B4 | PID = 0x0008

The next section describes the details of the Synchronous ADMux interface and booting over the Synchronous ADMux interface.

Sync ADMUX Boot

Figure 6 shows the FX3 system diagram when booting over the Sync ADMux interface.

Figure 6. System Diagram for Sync ADMux Boot



For booting over the Synchronous ADMux interface, the state of the PMODE[2:0] pins should be Z00.

Table 23. PMODE Pins for Sync ADMux Boot

PMODE[2]	PMODE[1]	PMODE[0]
Z	0	0

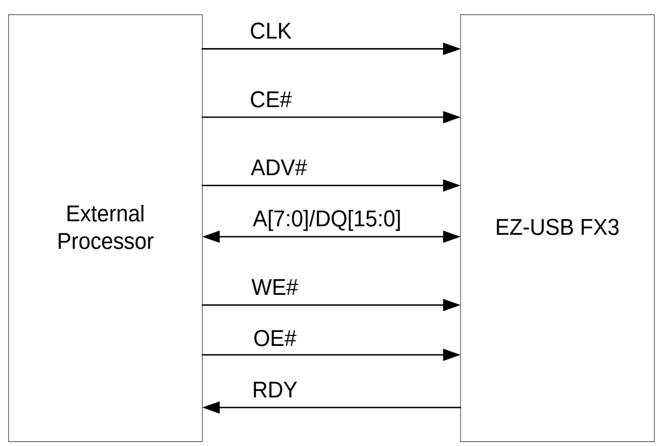
The FX3 GPIFII interface supports a Synchronous Address Data Multiplexed interface, which may be used for downloading a firmware image from an external processor or FPGA. The Synchronous ADMux interface configured by the bootloader consists of the following signals:

- PCLK: This must be a clock input to FX3. The maximum frequency supported for of the clock input is 100 MHz.
- DQ[15:0]: 16 bit data bus
- A[7:0]: 8-bit address bus:
- CE#: Active low chip enable
- ADV#: Active low address valid
- WE#: Active low write enable
- OE#: Active low output enable
- RDY: Active high ready signal

Figure 7 shows the typical interconnect diagram for the Sync ADMux interface configured by the bootloader and connected with an external processor.

Interface Signals

Figure 7. Sync ADMUX Interface



For Read operations, both CE# and OE# must be asserted.

For Write operations, both CE# and WE# are asserted.

ADV# must be low during the Address phase of a read/write operation. ADV# must be high during the data phase of a read/write operation.

The RDY output signal from the FX3 device indicates that data is valid for read transfers.

For details on the Sync ADMux timing diagrams and timing parameters, please refer to the FX3 Datasheet

Sync ADMUX Mode Power-up Delay

On power-up or a hard reset on the RESET# line, the bootloader will take some time to configure GPIFII for the Sync ADMux interface. This process can take a few hundred microseconds. Read/write access to FX3 should be performed only after the the bootloaser has completed the configuration. Otherwise data corruption can result. To avoid this, use one of the following schemes:

- Wait for 1 ms after RESET# de-assertion
- Keep polling the PP_IDENTIFY register until the value 0x81 is read back
- Wait for the INT# signal to assert, then read the RD_MAILBOX registers and verify the value read back equals 0x42575943 (that is, 'CYWB')

USB Fallback (=>USB)

The USB fallback will not be active during Sync ADMUX boot even if an error occurs on the commands.

Warm Boot

When warm boot is detected, the bootloader will transfer execution to the previously stored "Program Entry", which could be the user's RESET vector. In this case, the GPIFII configuration is preserved.

Wakeup/Standby

After wakeup from standby, the application firmware is responsible for configuring and restoring the hardware registers, GPIFII configuration, instruction tightly-coupled memory (ITCM), or data tightly-coupled memory (DTCM).

After wakeup from standby, the bootloader checks that both ITCM and DTCM are enabled.

Note When the bootloader wakes up from standby mode or the warm boot process, the bootloader jumps to the reset interrupt service subroutine and does the following:

- Invalidates both DCACHE and ICACHE
- Turns on ICACHE
- Disables MMU
- Turns on DTCM and ITCM
- Sets up the stack using the DTC

The bootloader allocates 0x500 bytes from 0x1000_0000 – 0x1000_04FF, so 0x1000_0500 – 0x1000_1FFF is available for downloading firmware. When the download application takes over, the memory from 0x1000_0000 – 0x1000_04FF can be used for other purposes.

GPIF II API Protocol

This protocol is used only in GPIF II boot mode. After reset, the external application processor (AP) communicates with the bootloader using the command protocol defined in the following table.

Table 24. GPIF II API Protocol

Field	Description
bSignature[2]	**2-byte** Sender initialize with "CY" The bootloader responses with "WB"
bCommand	Sender: **1-byte** Command 0x00: NOP 0x01: WRITE_DATA_CMD: Write Data Command 0x02: Enter Boot mode 0x03: READ_DATA_CMD: Read Data Command The bootloader treats all others as no operation and return error code in bLenStatus
bLenStatus	**Input: (1-byte)** 00: bLenStatus = 0 (the bootloader will jump to addr in dAddr if bCommand is WRITE_DATA_CMD, and ignore value for all other commands;) 01: Length in Long Word (Max = (512-8)/4) 02: Number of 512 byte blocks (Max = 16) 03: Length in Long Word (Max = (512-8)/4) Bootloader responses with the following data in the PIB_RD_MAILBOX1 register: 0x00: Success 0x30: Fail on Command process encounter error 0x31: Fail on Read process encounter error 0x32: Abort detection 0x33: PP_CONFIG.BURSTSIZE mailbox notification from the bootloader to application. The PIB_RD_MAILBOX0 will contain the GPIF_DATA_COUNT_LIMIT

Field	Description
	register. 0x34: the bootloader detects DLL _LOST_LOCK. The PIB_RD_MAILBOX0 will contain the PIB_DLL_CTRL register. 0x35: the bootloader detects PIB_PIB_ERR bit. The PIB_RD_MAILBOX0 will contain the PIB_PIB_ERROR register. 0x36: the bootloader detects PIB_GPIF_ERR bit. The PIB_RD_MAILBOX0 will contain the PIB_PIB_ERROR register.
dAddr	**4-byte** Sender: address use by command 1 and 3
dData[bLenStatus]	Data length determine by bLenStatus Sender: Data to be filled by the Sender

Note The error code bLenStatus will be reported on the mailbox of the GPIF II.

When downloading firmware to FX3 using Sync ADMUX, the external AP should do the following:

- Command block length should be exactly 512 bytes.
- Response block length should be exactly 512 bytes.
- The bootloader binary image should be converted to a data stream that is segmented in multiples of 512 bytes.
- The data chunk of the bootloader image should not be larger than 8 K. For instance, on the command 0x02, the bLenStatus should not be larger than 16 blocks (8 K bytes).
- The host should not send more than the total image size.

The bootloader does not support queuing commands. Therefore, every time a command is sent, the host must read the response.

Users should prevent the corruption of this API structure during the downloading process.

The host should not download firmware to the reserved bootloader SYSTEM address (0x4000_0000 to 0x4000_23FF). An error will be returned if the firmware application attempts to use this space.

The first 1280 bytes of DTCM should not be used (0x1000_0000 – 0x1000_04FF).

On the WRITE_DATA_CMD: When bLenStatus = 0**,** the bootloader jumps to the program entry of the dAddr.

Firmware Download Example

This section describes a simple way to implement firmware download from a host processor to FX3 via the 16-bit synchronous ADMux interface.

The host processor communicates with FX3 bootloader to perform the firmware download. The communication requires the host processor to read and write FX3 registers and data sockets.

The host processor uses available GPIF II sockets to transfer blocks of data into and out of FX3. FX3 bootloader maintains three data sockets to handle the firmware download protocol: one each for command, response and firmware data.

```
#define CY_WB_DOWNLOAD_CMD_SOCKET            (0x00)    // command block write only socket
#define CY_WB_DOWNLOAD_DATA_SOCKET           (0x01)    // data block read/write socket
#define CY_WB_DOWNLOAD_RESP_SOCKET           (0x02)    // response read only socket
```

The host processor communicates with FX3 bootloader via these data sockets to carry out the firmware download. The command and response are data structures used for the firmware download protocol. Both are 512 bytes in size. The bit fields are defined in these data structures to perform various functions by the FX3 bootloader. In the simple example implementation given in this document only the first 4 bytes of both command and response are actually used. The rest of the data bytes in the command and response are don't-cares.

From the high level FX3 firmware, the download requires the host processor to perform following sequence of socket accesses:

1. One command socket write with command block initialized as

```
command[0] = 'C';
command[1] = 'Y';        /* first two bytes are signature bytes with
                            constant value of "CY" */
command[2] = 0x02;       /* 0x2 is value for boot mode command. */
command[3] = 0x01;       /* 1 data block */
```

2. One response socket read that expects response block data as

```
response[0] = 'W';
response[1] = 'B';        /* first two bytes are signature bytes with
                            constant value of "WB" */
//response[2] = 0x0;     /* this byte is don't care. */
response[3] = 0x0;       /* indicate command is accepted */
```

3. One data socket write that transfers the entire firmware image in terms of byte array into FX3.

Note that once the firmware image has been completely transferred, the FX3 bootloader automatically jumps to the entry point of the newly downloaded firmware and starts executing. Before the host process can communicate with the downloaded firmware it is recommended to wait for a certain amount of time (depending on the firmware implementation) to allow the firmware to fully initialized. An even better option is to implement in the firmware a status update via mailbox registers after the initialization. In this case, the host processor is notified whenever the firmware is ready.

Before going into the details of the FX3 firmware download, note that the below functions are frequently used in the example implementation in this document and are platform dependent. Please contact Cypress Support for more information on how these can be implemented on a specific platform.

```
IORD_REG16(); // 16-bit read from GPIF II
IOWR_REG16(); // 16-bit write to GPIF II
IORD_SCK16(); // 16-bit read from active socket set in PP_DMA_XFER. The address driven on
              //   on the Sync ADMux bus during the address phase is treated as a
              //   don't-care
IOWR_SCK16(); // 16-bit write to active socket set in PP_DMA_XFER. The address driven on
              //   on the Sync ADMux bus during the address phase is treated as a
              //   don't-care
```

```
mdelay();      // millisecond delay
udelay();      // microsecond delay
```

Below is the example implementation of the fx3_firmware_download() function that takes a pointer to the firmware data array and the size of the firmware as parameters.

```
int fx3_firmware_download(const u8 *fw, u16 sz)
{
        u8  *command=0, *response=0;
        u16 val;
        u32 blkcnt;
        u16 *p = (u16 *)fw;
        int i=0;

        printf("FX3 Firmware Download with size = 0x%x\n", sz);

        /* Check PP_CONFIG register and make sure FX3 device is configured */
        /*   When FX3 bootloader is up with correct PMODE, bootloader configures */
        /*   the GPIF II into proper interface and sets the CFGMODE bit on PP_CONFIG */
        val = IORD_REG16(PP_CONFIG);
        if (val & CFGMODE) {
                printf("ERROR: WB Device CFGMODE not set !!! PP_CONFIG=0x%x\n", val);
                return FAIL;
        }

        /* A good practice to check for size of image */
        if (sz > (512*1024)) {
                printf("ERROR: FW size larger than 512kB !!!\n");
                return FAIL;
        }

        /* Allocate memory for command and response */
        /* Host processor may use DMA sequence to transfer the command and response */
        /* In that case make sure system is allocating contiguous physical memory area */
        command  = (u8 *) malloc(512);
        response = (u8 *) malloc(512);
        memset(command, 0, 512);
        memset(response, 0, 512);

        if (command==0 || response==0) {
                printf("ERROR: Out of memory !!!\n");
                return FAIL;
        }

        /* Initialize the command block */
        command[0] = 'C';
        command[1] = 'Y';
        command[2] = 0x02;      /* Enter boot mode command. */
        command[3] = 0x01;      /* 1 data block */

        /* Print the command block if you like to see it */
        for (i=0; i<512; i++) {
           if (!(i%16))
              printf("\n%.3x: ", i);
           printf("%.2x ",command[i]);
        }
    printf("\n");

     /* write boot command into command socket */
    sck_bootloader_write(CY_WB_DOWNLOAD_CMD_SOCKET, 512, (u16 *)command);
```

```
    /* read the response from response socket */
         sck_bootloader_read(CY_WB_DOWNLOAD_RESP_SOCKET, 512, (u16 *)response);

    /* Check if correct response */
        if ( response[3]!=0 || response[0]!='W' || response[1]!='B' ) {
                printf("ERROR: Incorrect bootloader response = 0x%x !!!\n",response[3]);
                for (i=0; i<512; i++) {
                        if (!(i%16))
                                printf("\n%.3x: ", i);
                        printf("%.2x ",response[i]);
                }
        printf("\n");
                kfree(command);
                kfree(response);
                return FAIL;
        }

        /* Firmware image transfer must be multiple of 512 byte */
        /* Here it rounds up the firmware image size */
        /* and write the array to data socket */
        blkcnt = (sz+511)/512;
        sck_bootloader_write(CY_WB_DOWNLOAD_DATA_SOCKET, blkcnt*512, p);

        /* Once the transfer is completed, bootloader automatically jumps to */
        /* entry point of the new firmware image and start executing */

        kfree(command);
        kfree(response);
        mdelay(2);         /* let the new image come up */
        return PASS;
}
```

Below is an example implementation of the socket write and socket read functions. Besides the data direction, function implementations for both socket write and read are based on the following command, configuration and status bits on PP_* register interface:

- PP_SOCK_STAT.SOCK_STAT[N]. For each socket this status bit indicates that a socket has a buffer available to exchange data (it has either data or space available).
- PP_DMA_XFER.DMA_READY. This status bit indicates whether the GPIF II is ready to service reads from or writes to the active socket (the active socket is selected through the PP_DMA_XFER register). PP_EVENT.DMA_READY_EV mirrors PP_DMA_XFER.DMA_READY with a short delay of a few cycles.
- PP_EVENT.DMA_WMARK_EV. This status bit is similar to DMA_READY, but it de-asserts a programmable number of words before the current buffer is completely exchanged. It can be used to create flow control signals with offset latencies in the signaling interface.
- PP_DMA_XFER.LONG_TRANSFER. This config bit indicates if long (multi-buffer) transfers are enabled. This bit is set by the application processor as part of transfer initiation.
- PP_CONFIG.BURSTSIZE and PP_CONFIG.DRQMODE. These config bits define and enable the size of the DMA burst. Whenever the PP_CONFIG register is updated successfully, FX3 bootloader responds with a value 0x33 in the PP_RD_MAILBOX register.
- PP_DMA_XFER.DMA_ENABLE. This command and status indicates that DMA transfers are enabled. This bit is set by the host processor as part of transfer initiation and cleared by FX3 hardware upon transfer completion for short transfers and by the application processor for long transfers.

```
static u32 sck_bootloader_write(u8 sck, u32 sz, u16 *p)
{
```

```
u32 count;
u16 val, buf_sz;
int i;

     /* Poll for PP_SOCK_STAT_L and make sure socket status is ready */
     do {
             val = IORD_REG16(PP_SOCK_STAT_L);
             udelay(10);
     } while(!(val&(0x1<<sck)));

     /* write to pp_dma_xfer to configure transfer
     socket number, rd/wr operation, and long/short xfer modes */
     val = (DMA_ENABLE | DMA_DIRECTION | LONG_TRANSFER | sck);
     IOWR_REG16(PP_DMA_XFER, val);

     /* Poll for DMA_READY_EV */
     count = 10000;
     do {
             val = IORD_REG16(PP_EVENT);
             udelay(10);
             count--;
     } while ((!(val & DMA_READY_EV)) && (count != 0));

     if (count == 0) {
             printf("%s: Fail timeout; Count = 0\n", __func__);
             return FAIL;
     }

     /* enable DRQ WMARK_EV for DRQ assert */
     IOWR_REG16(PP_DRQR5_MASK, DMA_WMARK_EV);

     /* Change FX3 FW to single cycle mode */
     val = IORD_REG16(PP_CONFIG);
     val = (val&0xFFF0)|CFGMODE;
     IOWR_REG16(PP_CONFIG, val);

     /* Poll for FX3 FW config init ready */
     count = 10000;
     do {
             val = IORD_REG16 (PP_RD_MAILBOX2);
             udelay(10);
    count --;
     } while ((!(val & 0x33)) || count==0);  /* CFGMODE bit is cleared by FW */

     if (count == 0) {
             printk("%s: Fail timeout; Count = 0\n", __func__);
             return FAIL;
     }

count=0;
do {
    for (i = 0; i < (buf_sz / 2); i++)
       IOWR_SCK16(*p++);
    count += (buf_sz / 2);

    if (count < (sz/2))
       do {
          udelay(10);
          val = IORD_REG16 (PP_SOCK_STAT_L);
       } while(!(val&(0x1<<sck)));
```

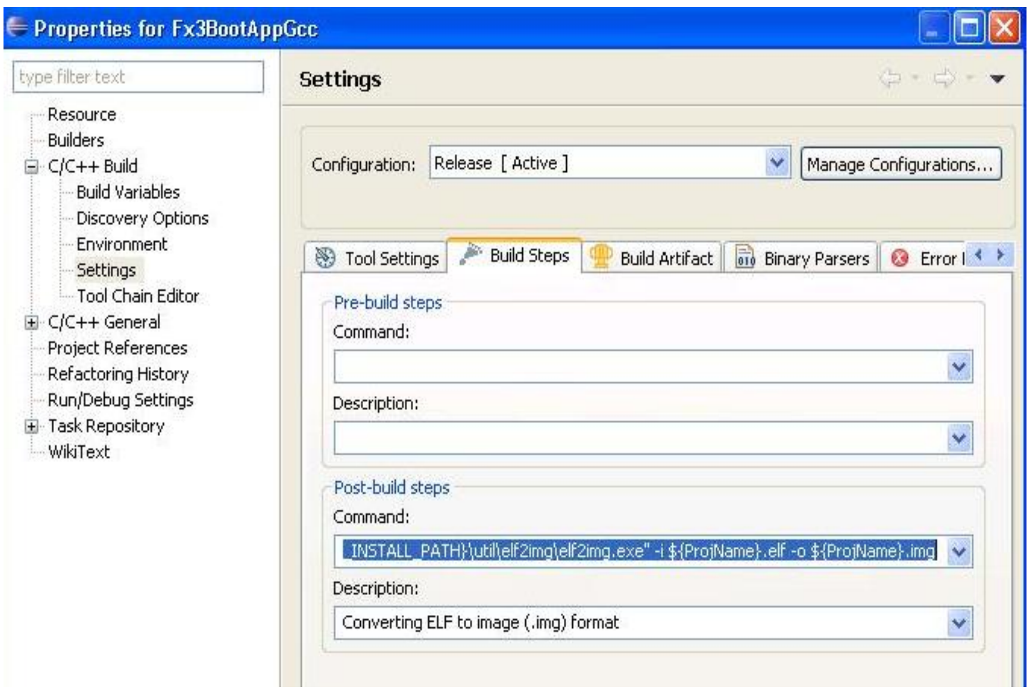
```
    } while (count < (sz/2));

        /* disable dma */
        val = IORD_REG16(PP_DMA_XFER);
        val &= (~DMA_ENABLE);
        IOWR_REG16(PP_DMA_XFER, val);

        printf("DMA write completed .....\n");
    return PASS;
}

static u32 pp_bootloader_read(u8 sck, u32 sz, u16 *p)
{

   u32 count;
   u16 val, buf_sz;
   int i;

        /* Poll for PP_SOCK_STAT_L and make sure socket status is ready */
        do {
                val = IORD_REG16(PP_SOCK_STAT_L);
                udelay(10);
        } while(!(val&(0x1<<sck)));

        /* write to PP_DMA_XFER to configure transfer
socket number, rd/wr operation, and long/short xfer modes */
        val = (DMA_ENABLE | LONG_TRANSFER | sck);
        IOWR_REG16(PP_DMA_XFER, val);

        /* Poll for DMA_READY_EV */
        count = 10000;
        do {
                val = IORD_REG16 (PP_EVENT);
                udelay(10);
                count--;
        } while ((!(val & DMA_READY_EV)) && (count != 0));

        if (count == 0) {
                printk("%s: Fail timeout; Count = 0\n", __func__);
                return FAIL;
        }

        /* enable DRQ WMARK_EV for DRQ assert */
        IOWR_REG16(PP_DRQR5_MASK, DMA_WMARK_EV);

        /* Change FX3 FW to single cycle mode */
        val = IORD_REG16(PP_CONFIG);
        val = (val&0xFFF0)|CFGMODE;
        IOWR_REG16(PP_CONFIG, val);

        /* Poll for FX3 FW config init ready */
        count = 10000;
        do {
                val = IORD_REG16 (PP_RD_MAILBOX2);
                udelay(10);
        count --;
        } while ((!(val & 0x33)) || count==0);  /* CFGMODE bit is cleared by FW */

        if (count == 0) {
                printk("%s: Fail timeout; Count = 0\n", __func__);
```

```
                return -1;
        }

    count=0;
    do {
        for (i = 0; i < (buf_sz / 2); i++) {
            p[count+i] = IORD_SCK16();
        }
        count += (buf_sz / 2);  /* count in words */

        if (count < (sz/2))
            do {
                    udelay(10);
            val = IORD_REG16 (PP_SOCK_STAT_L);
        } while(!(val&(0x1<<sck)));

        } while (count < (sz/2));

         /* disable dma */
         val = IORD_REG16(PP_DMA_XFER);
         val &= (~DMA_ENABLE);
         IOWR_REG16(PP_DMA_XFER, val);

         printf("DMA read completed .....\n");
    return PASS;
}
```

For Sync ADMux boot, the booloader expects the firmware image to be in the format shown in Table 25. The EZ-USB FX3 SDK provides a software utility that can be used to generate a firmware image in the format required for Sync ADMux boot. Please refer to the elf2img utlity found in the [install path] \util\elf2img directory after installing the SDK.

Note that the elf2img post-build command generates an .img fie. This then needs to be converted into an array that can be used for the download example just shown above. Figure 8 shows how the elf2img post build command is issued. An example for printing the contents of the .img file into an array in ASCII format follows in Figure 8.

Figure 8. Post Build Command in Eclipse IDE



The following is an example of code for printing the contents of the .img file into an array in ASCII format:

```
#include <stdio.h>
#include <stdint.h>
int main (int argc, char *argv[])
{
    char *filename = "firmware.img";
FILE *fp;
    int i = 0;
    uint32_t k;

    if (argc > 1)
        filename = argv[1];

    fprintf (stderr, "Opening file %s\n", filename);
    fp =  fopen (filename, "r");
    printf ("const uint8_t fw_data[] = {\n\t");

    while (!feof(fp))
    {
        fread (&k, sizeof (uint32_t), 1, fp);
        printf ("0x%02x, 0x%02x, 0x%02x, 0x%02x,",
                ((uint8_t *)&k)[0], ((uint8_t *)&k)[1],
                ((uint8_t *)&k)[2], ((uint8_t *)&k)[3]);
        i++;
        if (i == 4)
        {
            i = 0;
            printf ("\n\t");
        }
        else
            printf (" ");
    }

    printf ("\n};\n");

    fclose (fp);
    return 0;
    }
```

Bootloader Memory Allocation

The FX3 bootloader allocates 1280 bytes of DTCM from 0x1000_0000 to 0x1000_04FF for its variables and stack. The firmware application can use it as long as this area remains uninitialized (that is, uninitialized local variables) during the firmware download.

The bootloader allocates the first 16 bytes from 0x4000_0000 to 0x4000_000F for warm boot and standby boot. These bytes should not be used by firmware applications.

The bootloader allocates around 10 K bytes from 0x4000_23FF for its internal buffers. The firmware application can use this area as the uninitialized local variables/buffers.

The bootloader does not use instruction tightly-coupled memory (ITCM) memory.

USB Memory Access

The FX3 bootloader allows all the read access from ROM, MMIO, SYSMEM, ITCM, and DTCM memory spaces.

USB Registers/Memory Access

The FX3 bootloader allows read access from ROM, MMIO, SYSMEM, ITCM, and DTCM memory spaces.

The bootloader allows write access to MMIO, SYSMEM, ITCM, and DTCM memory spaces except for the first 1280 bytes of DTCM and the first 10 K of SYSTEM memory. When writing to MMIO space, the expected transfer length for the bootloader must be 4 (equal to LONG word) and the address should be aligned by 4 bytes.

Boot Image Format

For Sync ADMux boot, the booloader expects the firmware image file to be in the format shown in Table 25. The EZ-USB FX3 SDK provides a software utility that can be used to generate a firmware image in the format required for Sync ADMux boot. Please refer to the elf2img utlity found in the [install path] \util\elf2img directory after installing the SDK.

Table 25. Boot Image Format for Sync ADMux boot option

Binary Image Header	Length (16-bit)	Description
wSignature	1	Signature 2 bytes initialize with “CY” ASCII text
bImageCTL;	½	Bit0 = 0: execution binary file; 1: data file type Bit3:1 Do not use when booting in SPI EEPROM Bit5:4(SPI speed): 00: 10 MHz 01: 20 MHz 10: 30 MHz 11: Reserved Bit7:6: Reserved should be set to zero
bImageType;	½	bImageType = 0xB0: normal FW binary image with checksum bImageType = 0xB1: Reserved for security image type bImageType = 0xB2: SPI boot with new VID and PID
dLength 0	2	1st section length, in long words (32-bit) When bImageType = 0xB2, the dLength 0 will contain PID and VID. the bootloader ignores the rest of the any following data.
dAddress 0	2	1st section address of Program Code. **Note** Internal ARM address is byte addressable, so the address for each section should be 32-bit align.
dData[dLength 0]	dLength 0*2	Image Code/Data must be 32-bit align.
…		More sections
dLength N	2	0x00000000 (Last record: termination section)
dAddress N	2	Should contain valid Program Entry (Normally, it should be the Startup code, for example, the RESET Vector) **Note** If bImageCTL.bit0 = 1, the bootloader will not transfer the execution to this Program Entry. If bImageCTL.bit0 = 0, the Boot Loader will transfer the execution to this Program Entry: This address should be in the ITCM area or SYSTEM RAM area. The bootloader does not validate the Program Entry
dCheckSum	2	32-bit unsigned little endian checksum data will start from the 1st sections to the termination section. The checksum will not include the dLength, dAddress and Image Header

Example of boot image format organized in long word formatLocation1: 0xB0 0x10 'Y' 'C' //CY Signature, 20 MHz, 0xB0 Image

Location2: 0x00000004	//Image length of section 1 = 4
Location3: 0x40008000	//1st section stored in SYSMEM RAM at 0x40008000
Location4: 0x12345678	//Image starts (Section1)
Location5: 0x9ABCDEF1	
Location6: 0x23456789	
Location7: 0xABCDEF12	//Section 1 ends
Location8: 0x00000002	//Image length of section 2 = 2
Location9: 0x40009000	//2nd section stored in SYSMEM RAM at 0x40009000
Location10: 0xDDCCBBAA	//Section 2 starts
Location11: 0x11223344	
Location12: 0x00000000	//Termination of Image
Location13: 0x40008000	//Jump to 0x40008000 on FX3 System RAM
Location 14: 0x6AF37AF2	//Checksum (0x12345678 + 0x9ABCDEF1 + 0x23456789 + 0xABCDEF12+ 0xDDCCBBAA +0x11223344)

Default State of IOs During Boot

Table 26 shows the default state of the FX3 IOs for the different boot modes, while the bootloader is executing (before application firmware download).

Table 26. Default State of IOs During Boot

GPIO	SPI Boot Default State	USB Boot Default State	I2C Boot Default State	Sync ADMux Boot Default State
GPIO[0]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[1]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[2]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[3]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[4]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[5]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[6]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[7]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[8]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[9]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[10]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[11]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[12]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[13]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[14]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[15]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[16]	Tri-state	Tri-state	Tri-state	CLK Input
GPIO[17]	Tri-state	Tri-state	Tri-state	Input
GPIO[18]	Tri-state	Tri-state	Tri-state	Input
GPIO[19]	Tri-state	Tri-state	Tri-state	Input
GPIO[20]	Tri-state	Tri-state	Tri-state	Input
GPIO[21]	Tri-state	Tri-state	Tri-state	Output
GPIO[22]	Tri-state	Tri-state	Tri-state	Tri-state

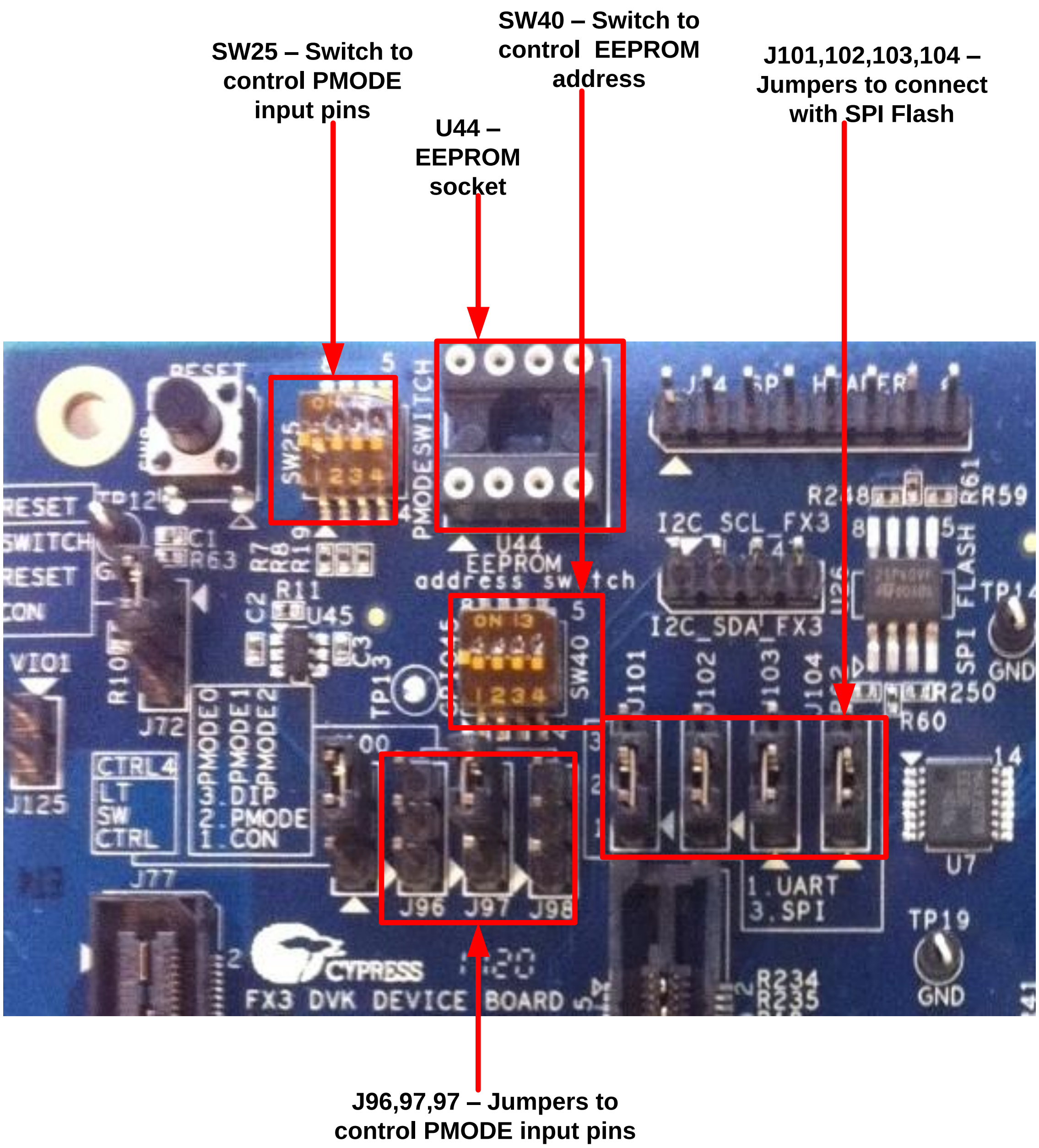
GPIO	SPI Boot Default State	USB Boot Default State	I2C Boot Default State	Sync ADMux Boot Default State
GPIO[23]	Tri-state	Tri-state	Tri-state	Input
GPIO[24]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[25]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[26]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[27]	Tri-state	Tri-state	Tri-state	Input
GPIO[28]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[29]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[30]	PMODE[0] I/P to FX3	PMODE[0] I/P to FX3	PMODE[0] I/P to FX3	PMODE[0] I/P to FX3
GPIO[31]	PMODE[1] I/P to FX3	PMODE[1] I/P to FX3	PMODE[1] I/P to FX3	PMODE[1] I/P to FX3
GPIO[32]	PMODE[2] I/P to FX3	PMODE[2] I/P to FX3	PMODE[2] I/P to FX3	PMODE[2] I/P to FX3
GPIO[33]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[34]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[35]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[36]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[37]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[38]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[39]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[40]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[41]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[42]	Low	Low	Low	Low
GPIO[43]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[44]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[45]	High	High	High	High
GPIO[46]	High	Tri-state	Tri-state	Tri-state
GPIO[47]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[48]	High	Tri-state	Tri-state	Tri-state
GPIO[49]	Tri-state	Tri-state	Tri-state	Tri-state
GPIO[50]	Low	Tri-state	Tri-state	Tri-state
GPIO[51]	Low	Low	Low	Low
GPIO[52]	High	Tri-state	Tri-state	Tri-state
GPIO[53]	Low (toggles during SPI transactions)	High	High	High
GPIO[54]	High	Tri-state	Tri-state	Tri-state
GPIO[55]	Tri-state	High	High	High
GPIO[56]	Low	Tri-state	Tri-state	Tri-state
GPIO[57]	Low	Tri-state	Tri-state	Tri-state
GPIO[58] I2C_SCL	Tri-state	Tri-state	Tri-state (Toggles during transaction., then tri-stated)	Tri-state
GPIO[59] I2C_SDA	Tri-state	Tri-state	Tri-state	Tri-state

Appendix

Steps for Booting using FX3 DVK Board

This section describes the step-wise sequence for exercising USB boot, I^{2}C boot and SPI boot using the FX3 DVK board. Figure 9 shows a part of the FX3 DVK board that contains switches and jumpers, which need to be configured appropriately for each boot option. The required settings for these are also described in the following sections.

Figure 9. FX3 DVK board - Essential Switches and Jumpers to be Configured for Boot



USB Boot

1. Build firmware image in Eclipe IDE as shown in Figure 10, Figure 11, and Figure 12:

Figure 10. Right Click on Project in Eclipse IDE

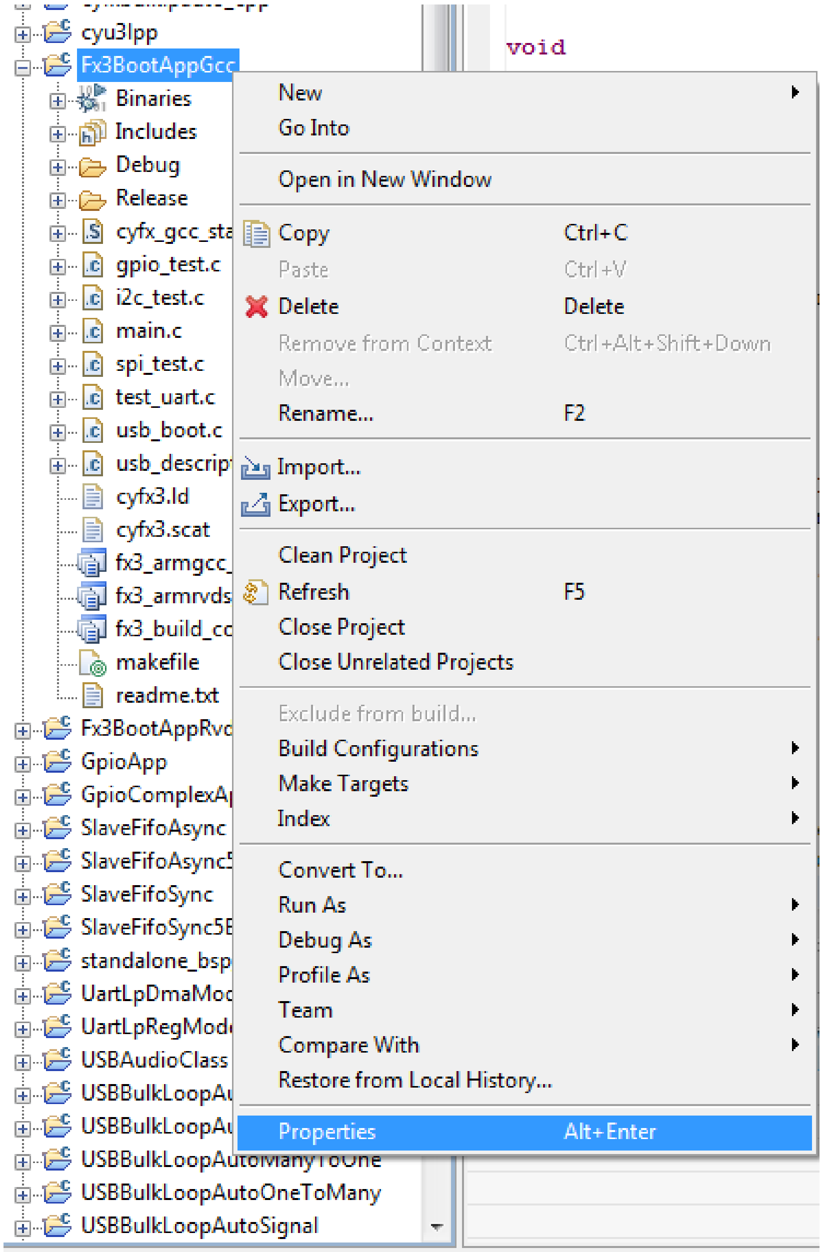


Figure 11. Select Settings

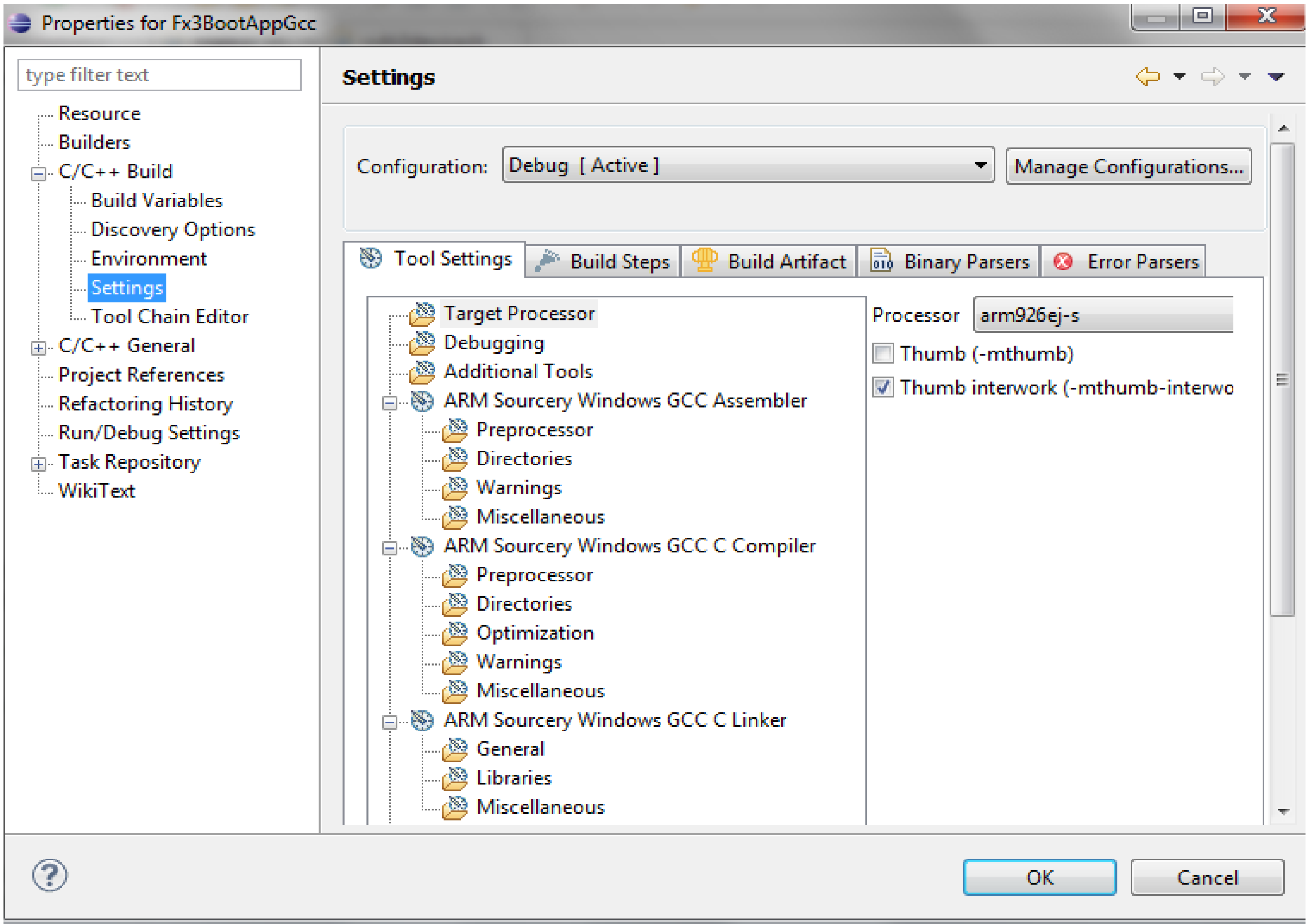
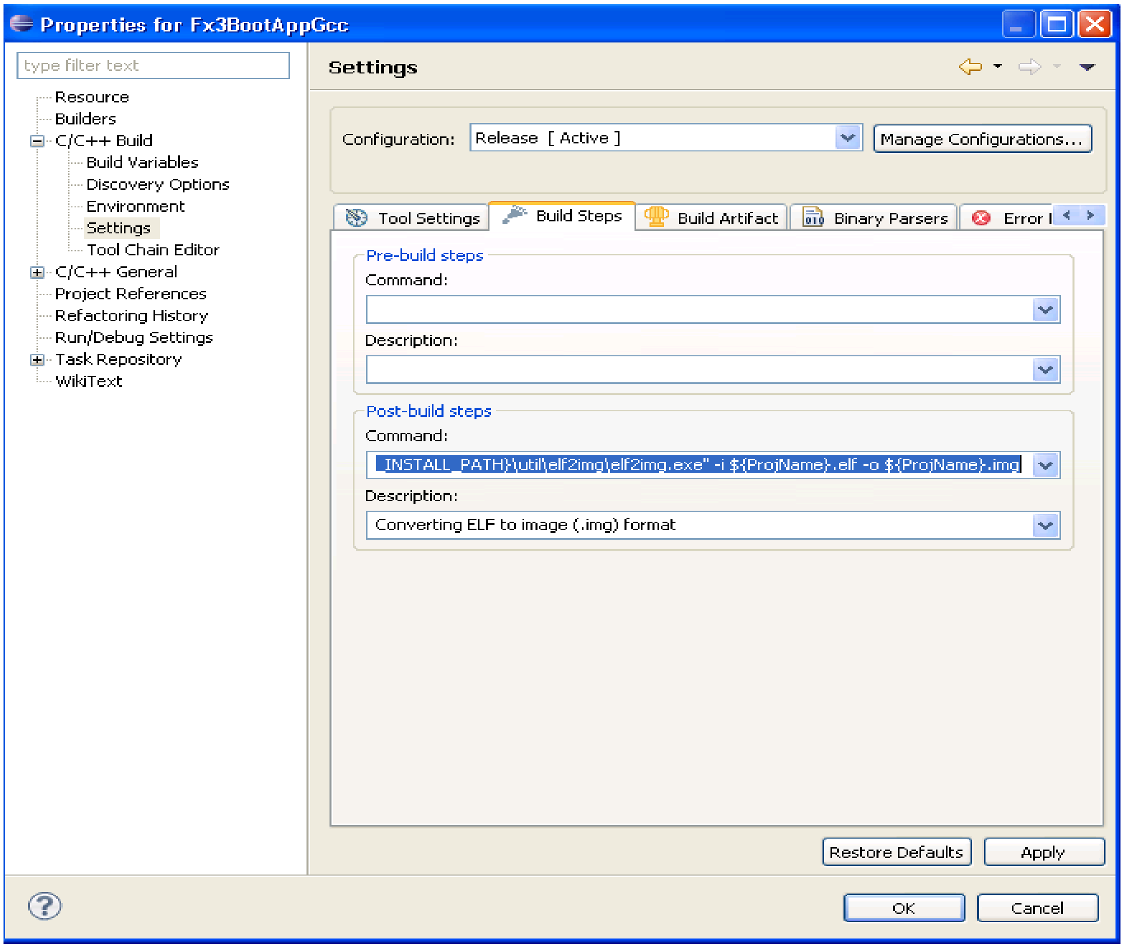


Figure 12. elf2img Command Configuration in Post-Build Steps for USB Boot Image



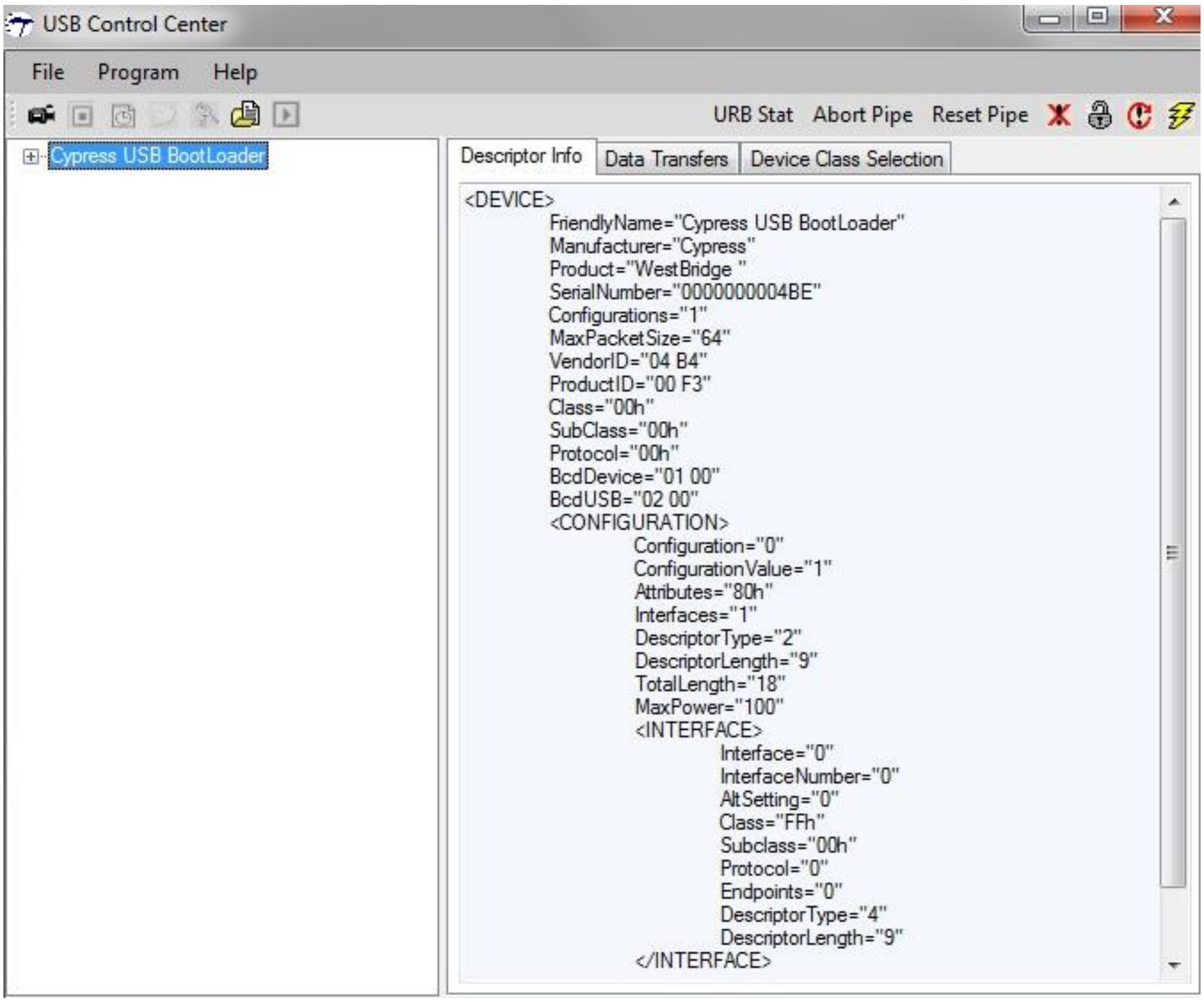
2. Enable USB boot, by setting the PMODE[2:0] pins to Z11. On the DVK board this is done by configuring the jumpers and switches as shown in Table 27.

Table 27. Jumper Configurations for USB Boot

Jumper/Switch	Position	State of Corresponding PMODE Pin
J96 (PMODE0)	2-3 Closed	PMODE0 controlled by SW25
J97 (PMODE1)	2-3 Closed	PMODE1 controlled by SW25
J98 (PMODE2)	Open	PMODE2 Floats
SW25.1-8 (PMODE0)	Open (OFF position)	PMODE0 = 1
SW25.2-7 (PMODE1)	Open (OFF position)	PMODE1 = 1
SW25.3-6 (PMODE2)	Don’t care	PMODE2 Floats

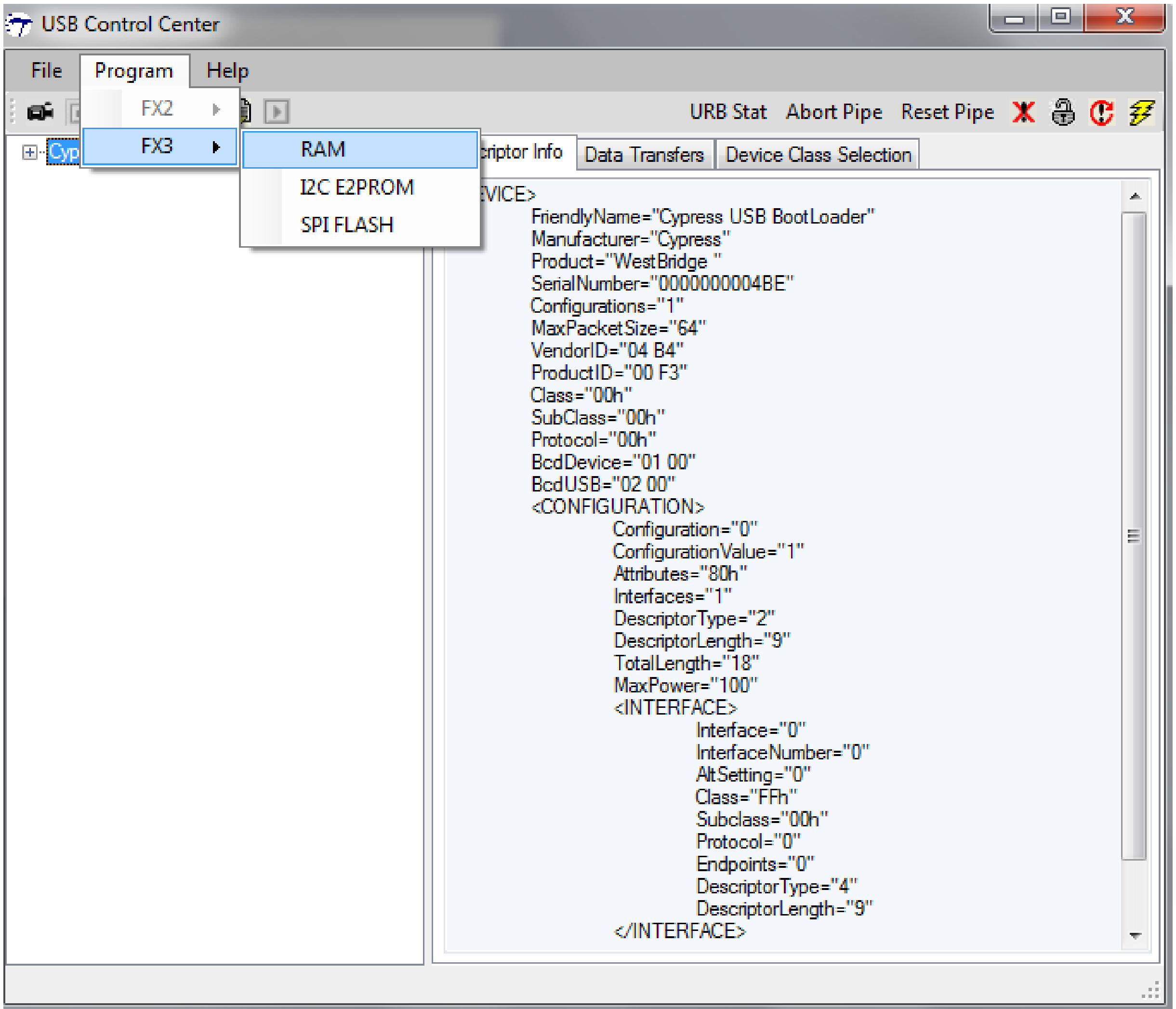
3. When connected to a USB host, the FX3 device enumerates in the Control Center as ‘Cypress USB Bootloader’

Figure 13. Cypress USB BootLoader Enumeration in Control Center



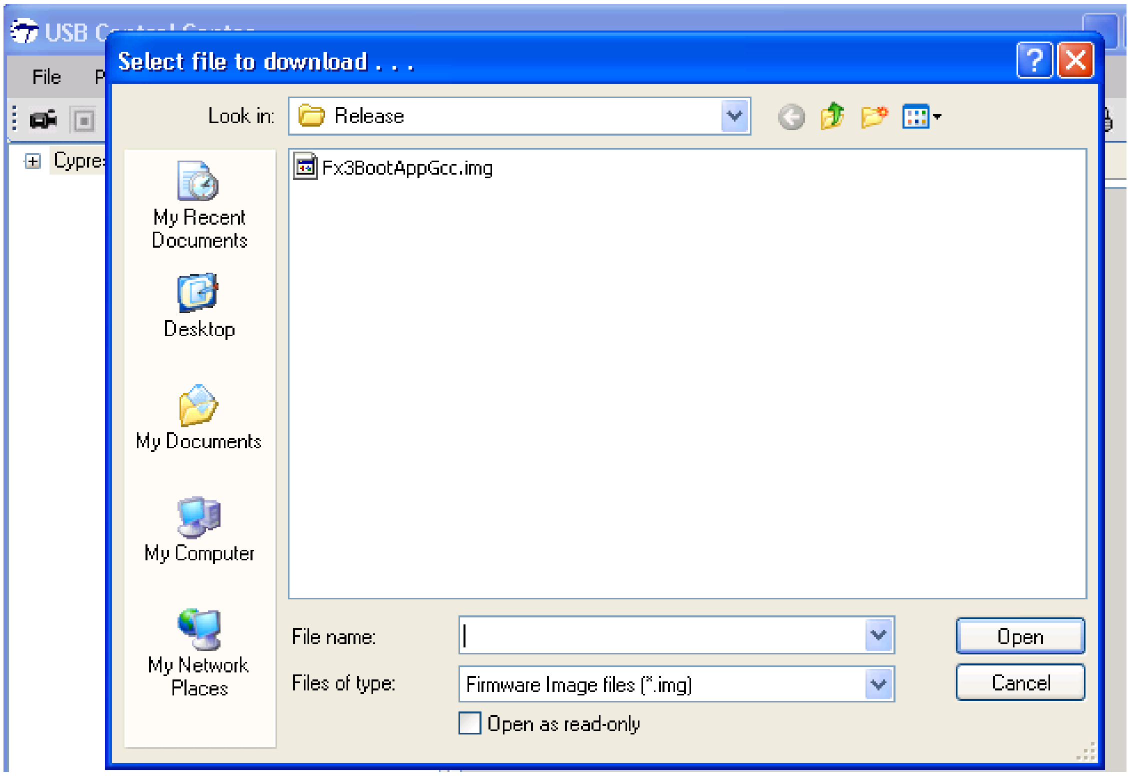
4. In Control Center select the FX3 device>FX3> RAM.

Figure 14. Select the Program from the Control Center



5. Next, browse to the .img file to be programmed into the FX3 RAM. Double click on the .img file.

Figure 15. Select .img File



6. A “Programming Succeeded” message is displayed on the bottom left of the Control Center and the FX3 device re-enumerates with the programmed firmware.

I²C Boot

1. Build the firmware image in Eclipe IDE as shown in Figure 16, Figure 17, and Figure 18.

Figure 16. Right Click on Project in Eclipse IDE

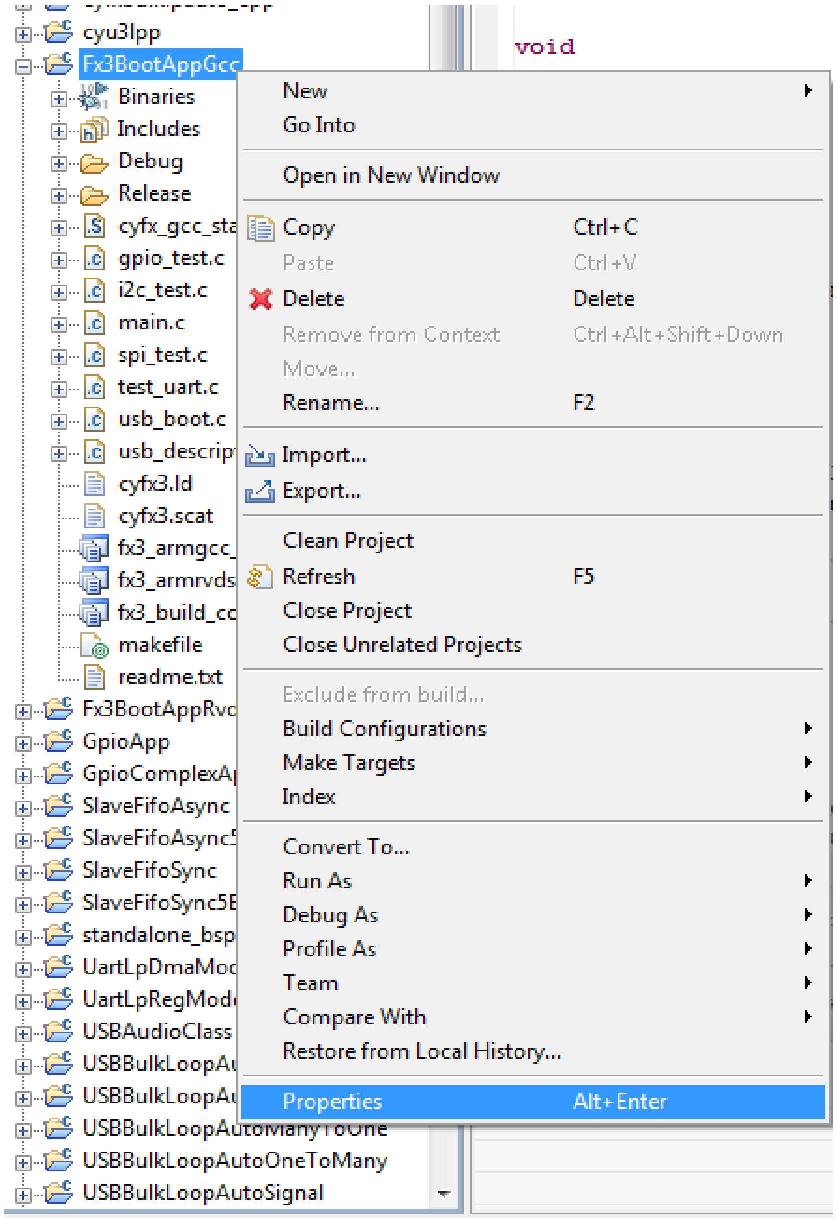


Figure 17. Properties of Fx3BootAppGcc

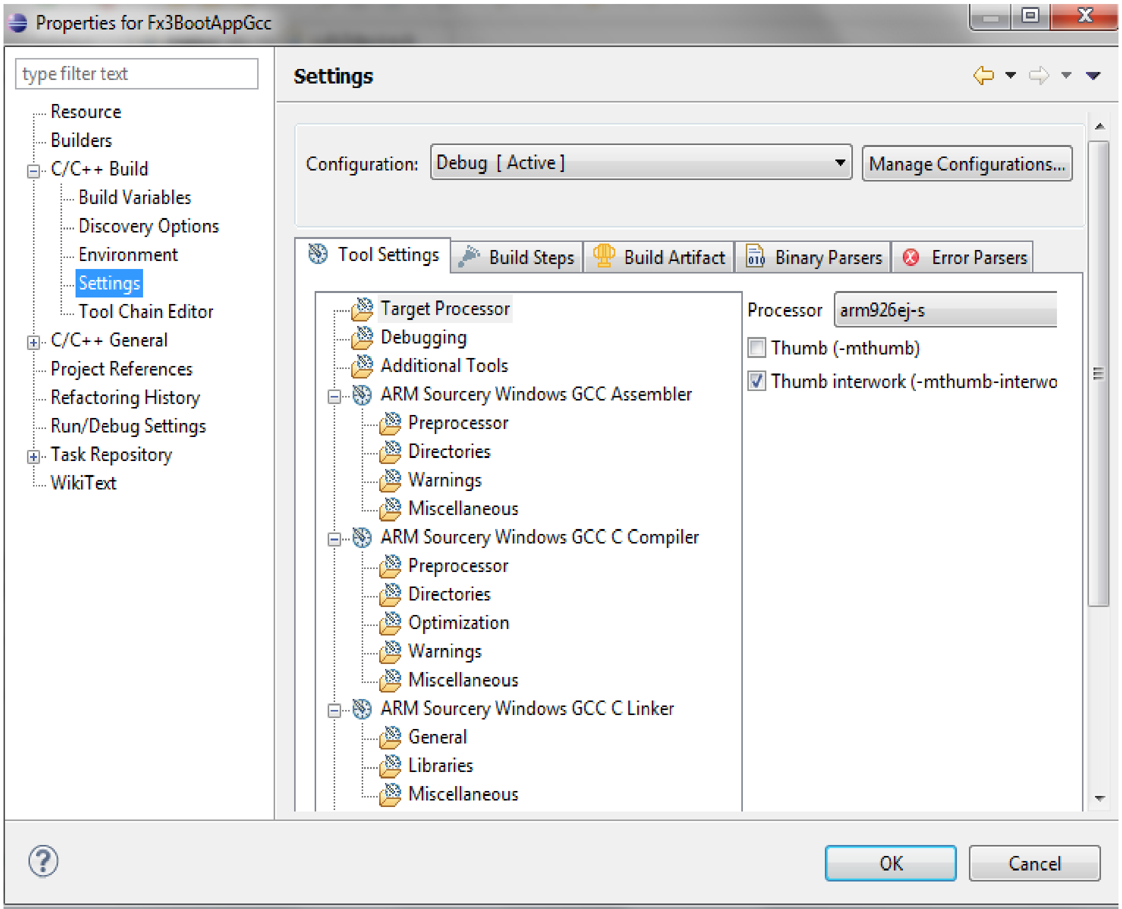
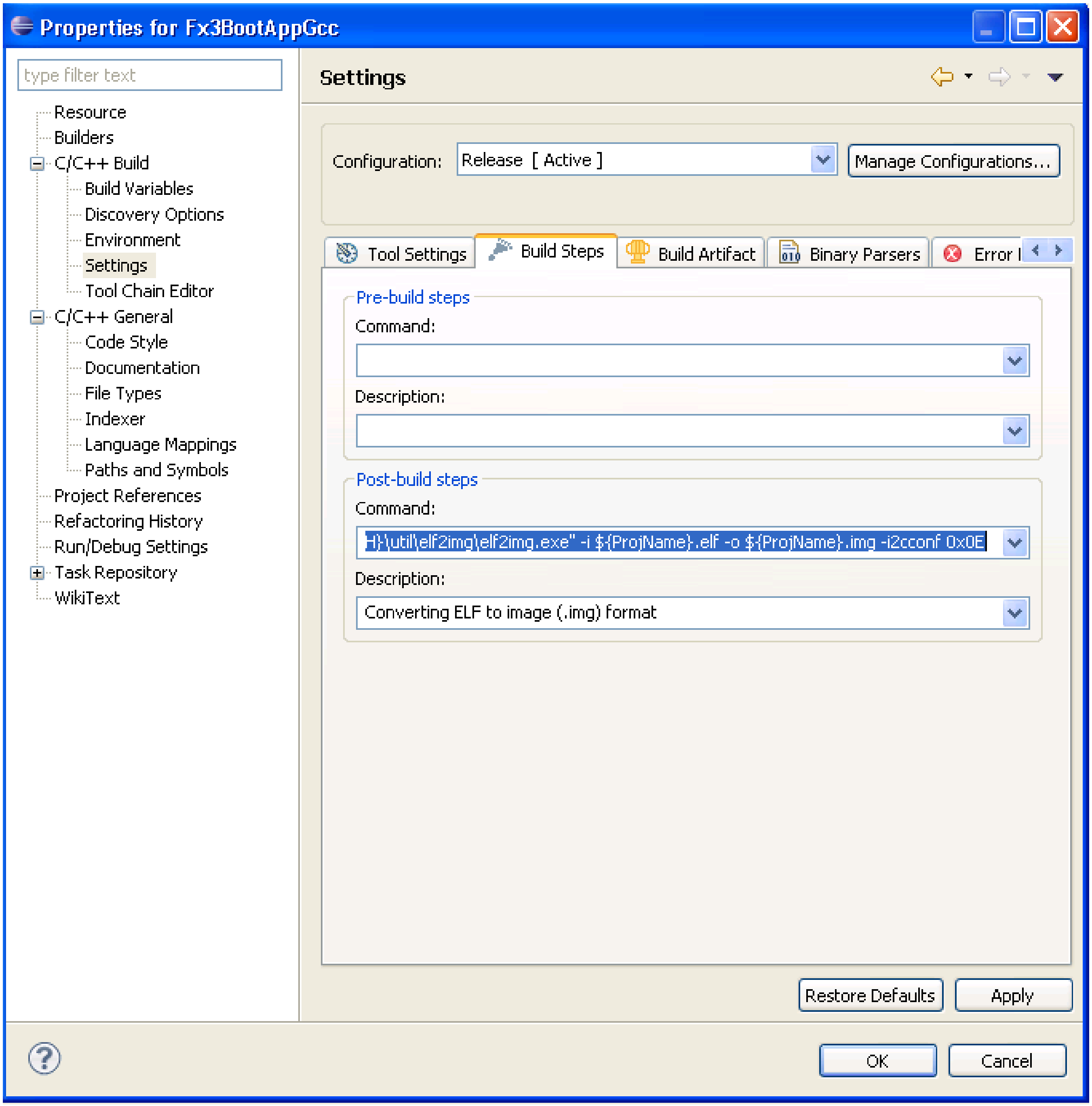


Figure 18. ‘elf2img’ Command Configuration in Post-build Steps for I2C Boot Image



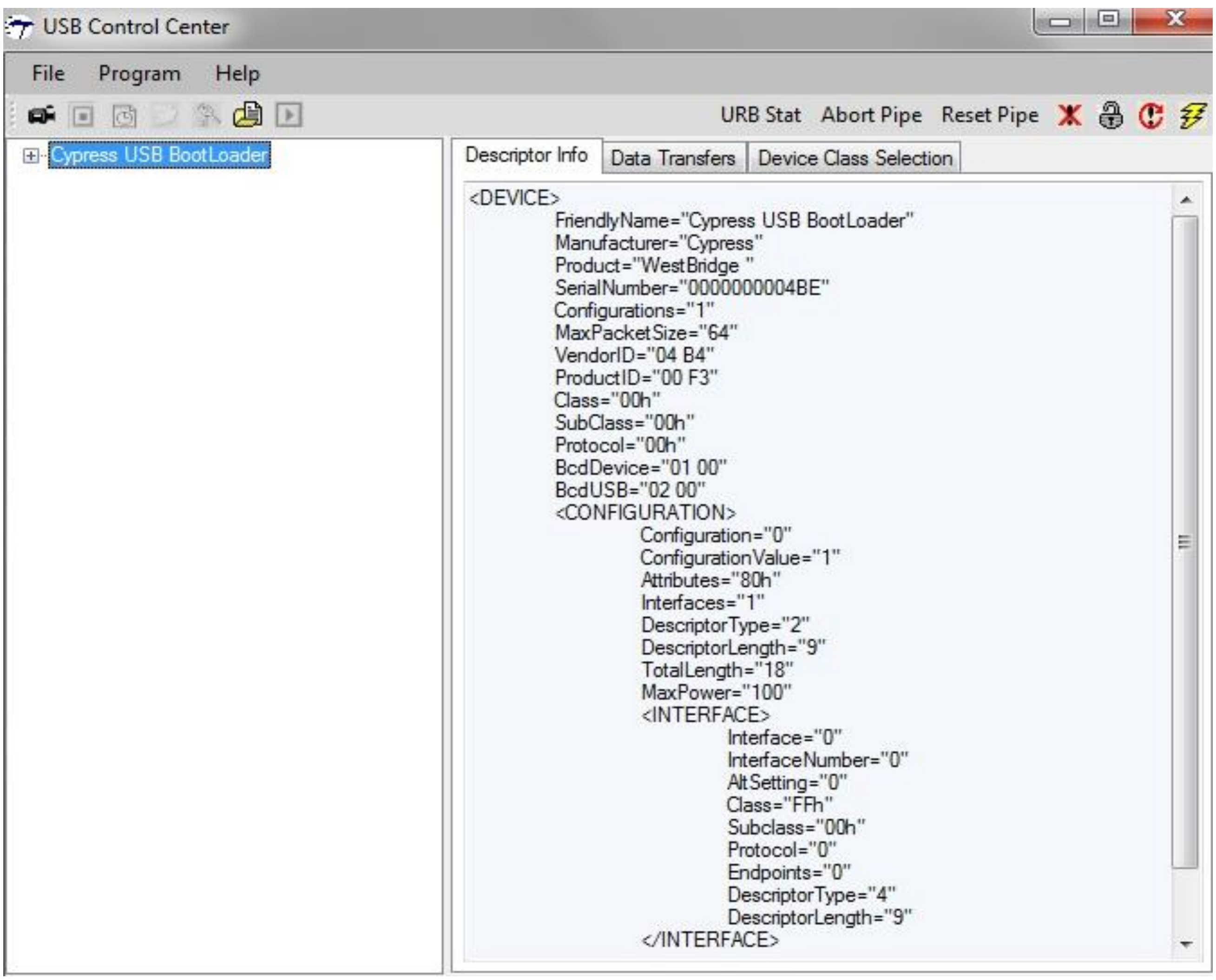
2. First enable USB boot, by setting the PMODE[2:0] pins to Z11. On the DVK board this is done by configuring the jumpers and switches as shown in Table 28.

Table 28. Jumper Configurations for USB Boot

Jumper/Switch	Position	State of Corresponding PMODE Pin
J96 (PMODE0)	2-3 Closed	PMODE0 controlled by SW25
J97 (PMODE1)	2-3 Closed	PMODE1 controlled by SW25
J98 (PMODE2)	Open	PMODE2 Floats
SW25.1-8 (PMODE0)	Open (OFF)	PMODE0 = 1
SW25.2-7 (PMODE1)	Open (OFF)	PMODE1 = 1
SW25.3-6 (PMODE2)	Don’t care	PMODE2 Floats

3. When connected to a USB host, the FX3 device enumerates in Control Center as ‘Cypress USB Bootloader’

Figure 19. Cypress USB BootLoader Enumeration in Control Center



4. Before attempting to program the EEPROM, ensure that the address signals of the EEPROM are configured correctly using the switch SW40 (For Microchip part 24AA1025, 1-8 ON, 2-7 ON, 3-6 OFF). Also, the I^2C Clock (SCL) and data Line (SDA) jumpers J42 and J45 pins 1-2 should be shorted on the DVK board. In the Control Center, select the FX3 device. Next, select Program FX3 and the I2C EEPROM Program. This causes a special I^2C boot firmware to be programmed into the FX3 device, which then enables programming of the I^2C device connected to FX3. Now the FX3 device re-enumerates as 'Cypress USB BootProgrammer' as shown in the following figures.

Figure 20. Select Program FX3 > I2C E2PROM

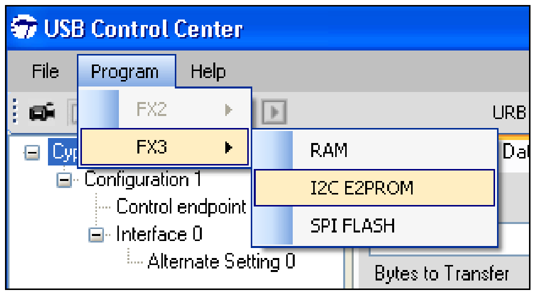
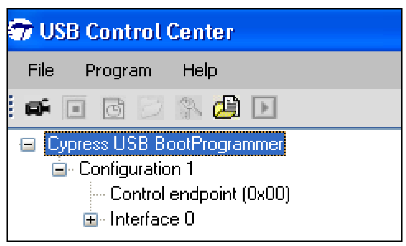
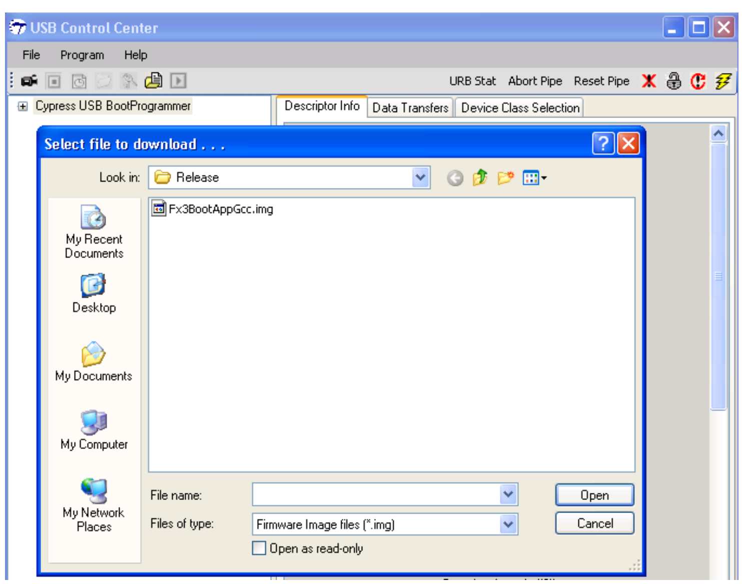


Figure 21. FX3 Re-enumerates as 'Cypress USB BootProgrammer'

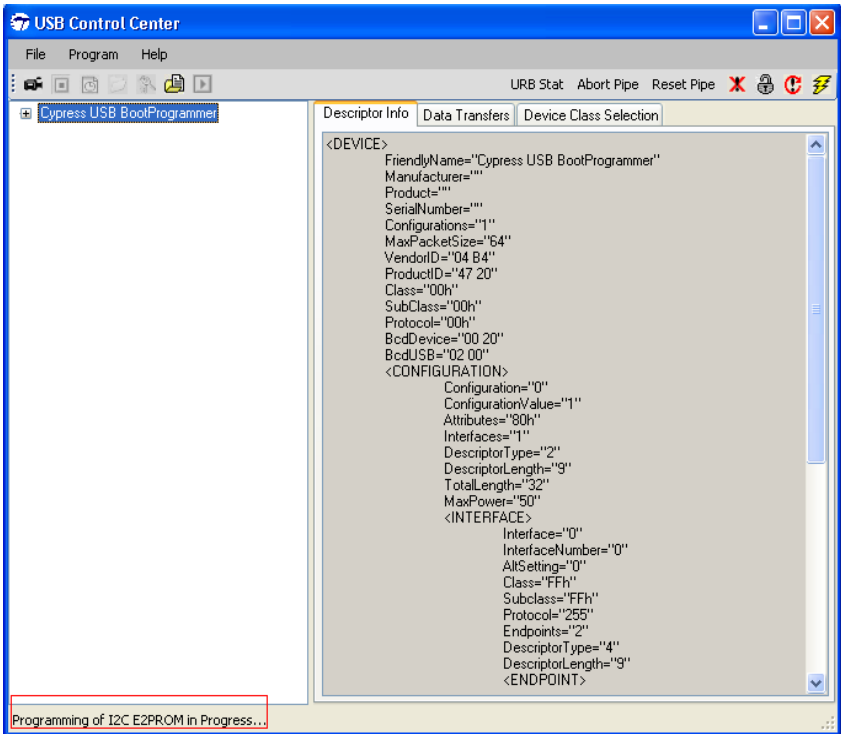


5. After the FX3 DVK board enumerates as 'Cypress USB BootProgrammer,' the Control Center application prompts the user to select the firmware binary to download. Browse to the .img file which is to be programmed into the I^2C EEPROM.

Figure 22. Select Firmware Image to Download



`Figure 23. I2C EEPROM Programming Update in Control Center



After the programming is complete, the bottom left corner of the window displays 'Programming of I2C E2PROM Succeeded**.'**

6. Change the PMODE pins on the DVK board to Z1Z in order to enable I^2C boot. On the DVK board this is done by configuring the jumpers and switches as shown in Table 29.

Table 29. Jumper Configurations for I2C Boot

Jumper/Switch	Position	State of Corresponding PMODE Pin
J96 (PMODE0)	Open	PMODE0 Floats
J97 (PMODE1)	2-3 Closed	PMODE1 controlled by SW25
J98 (PMODE2)	Open	PMODE2 Floats
SW25.1-8 (PMODE0)	Don't care	PMODE0 Floats
SW25.2-7 (PMODE1)	Open (OFF position)	PMODE1 = 1
SW25.3-6 (PMODE2)	Don't care	PMODE2 Floats

7. Reset the DVK. Now the FX3 device boots from the I2C EEPROM

SPI Boot

1. Build the firmware image in Eclipe IDE as shown in Figure 24, Figure 25 and Figure 26.

Figure 24. Right Click on Project in Eclipse IDE

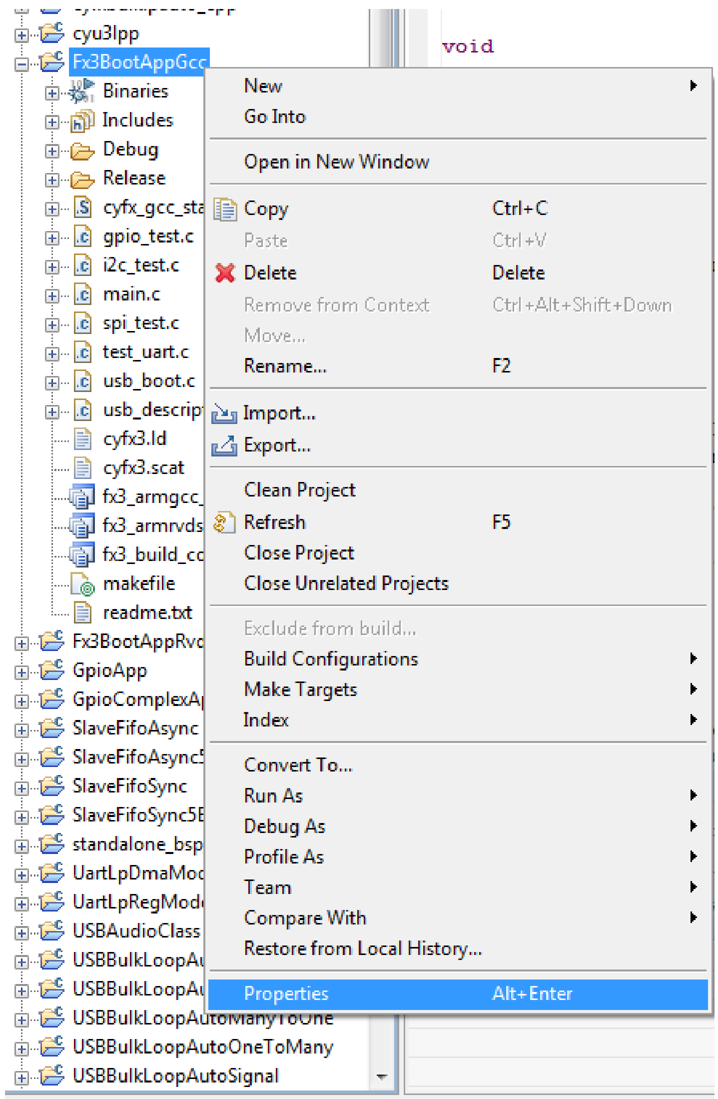


Figure 25. Select 'Settings'

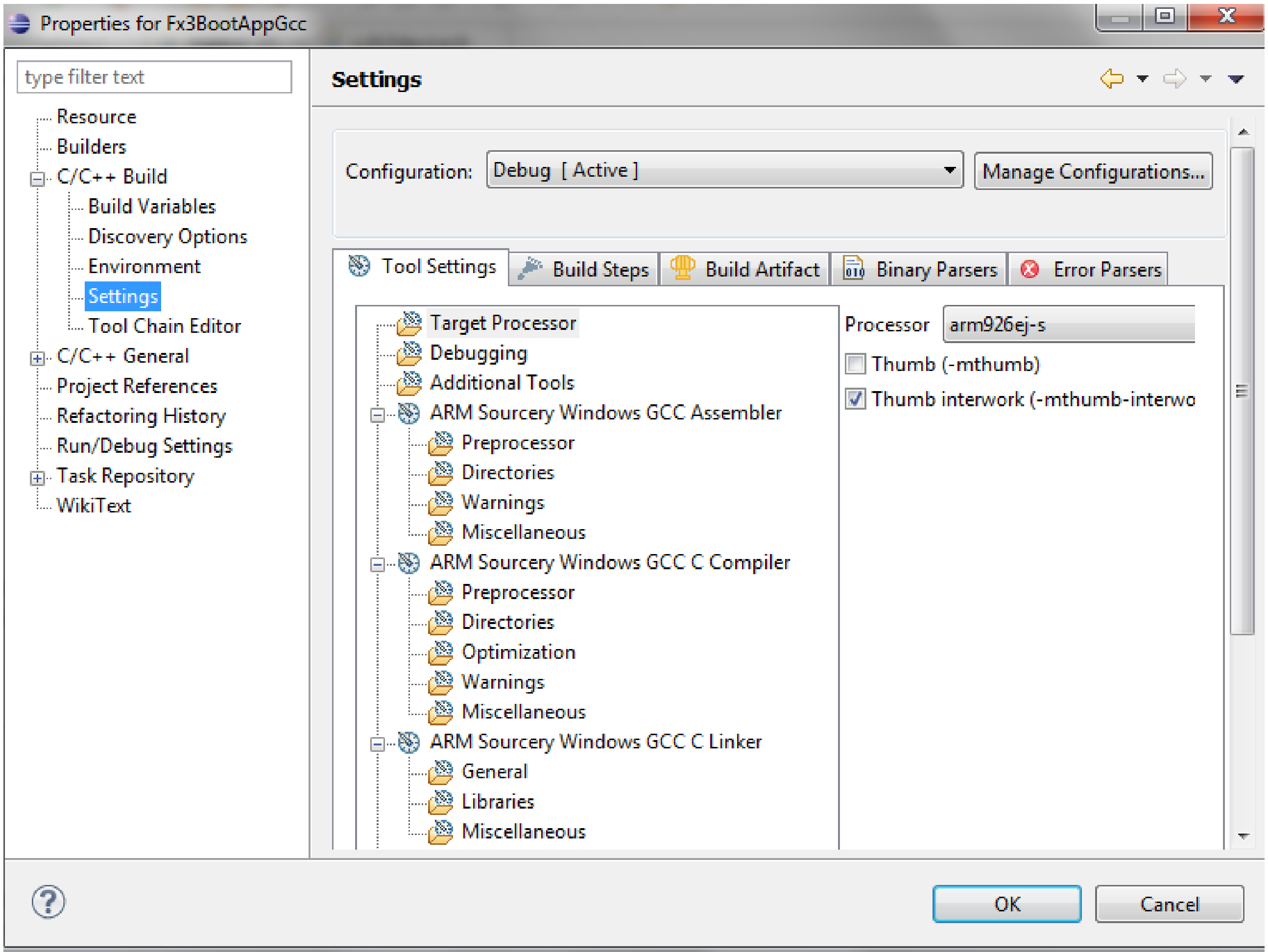
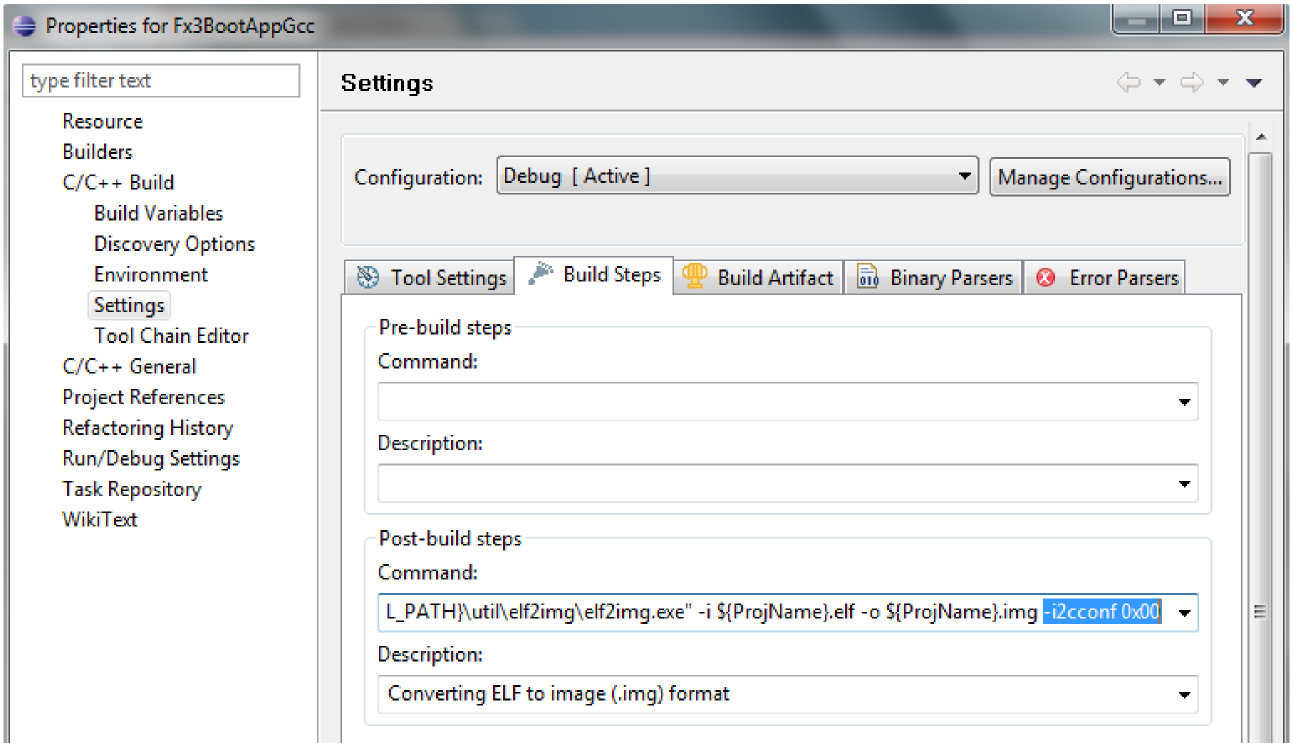


Figure 26. 'elf2img' Command Configuration in Post-build steps for SPI Boot Image



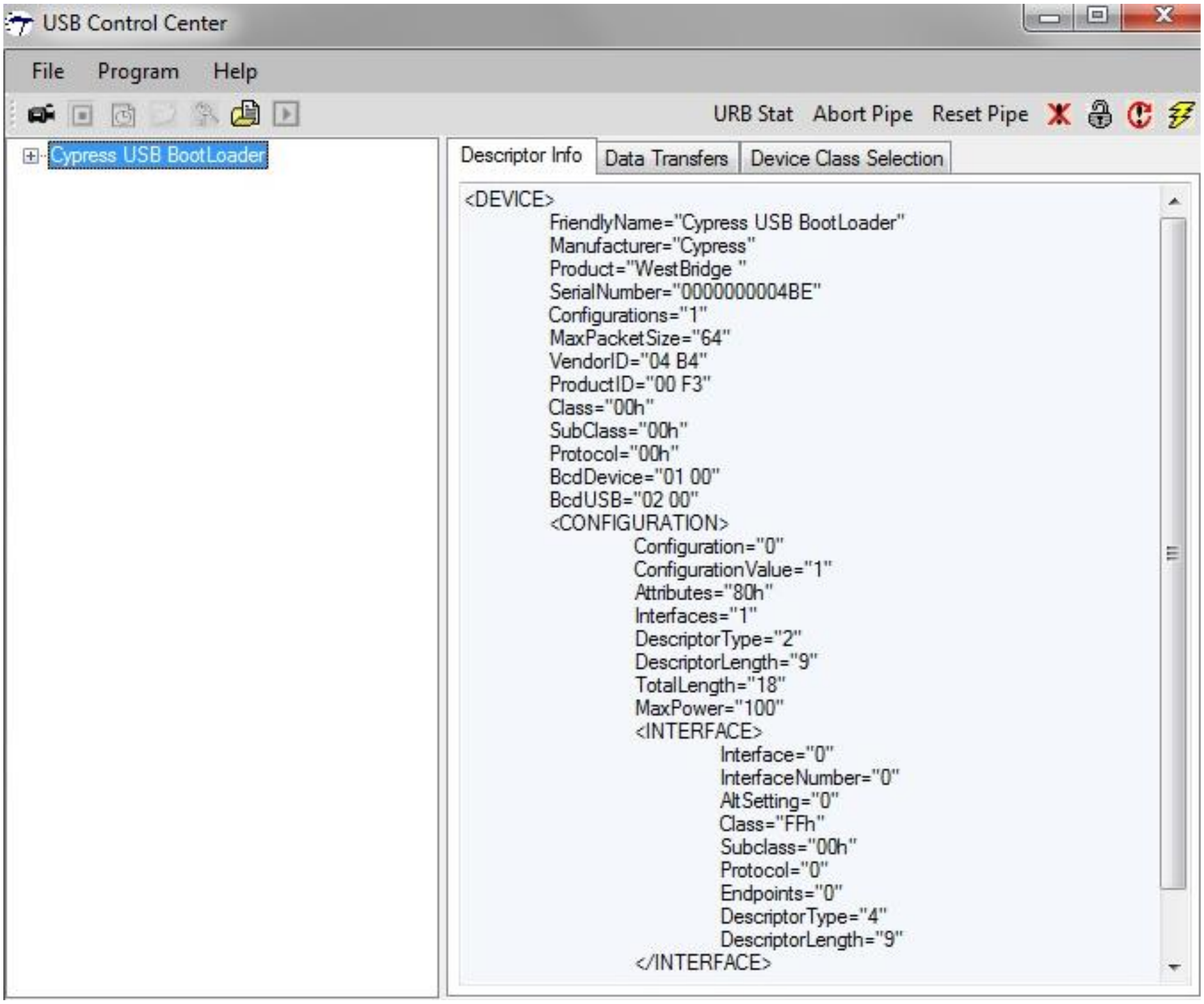
2. First enable USB boot, by setting the PMODE[2:0] pins to Z11. On the DVK board this is done by configuring the jumpers and switches as shown in Table 30.

Table 30. Jumper Configurations for USB Boot

Jumper/Switch	Position	State of Corresponding PMODE Pin
J96 (PMODE0)	2-3 Closed	PMODE0 controlled by SW25
J97 (PMODE1)	2-3 Closed	PMODE1 controlled by SW25
J98 (PMODE2)	Open	PMODE2 Floats
SW25.1-8 (PMODE0)	Open (OFF position)	PMODE0 = 1
SW25.2-7 (PMODE1)	Open (OFF position)	PMODE1 = 1
SW25.3-6 (PMODE2)	Don't care	PMODE2 Floats

3. When connected to a USB host, the FX3 device enumerates in Control Center as ‘Cypress USB Bootloader’

Figure 27. Cypress USB BootLoader Enumeration in Control Center



4. In Control Center select the FX3 device and then select Program FX3 and then SPI Flash. Then browse to the .img file to be programmed into the SPI Flash.

Figure 28. Select Program FX3>SPI Flash in Control Center

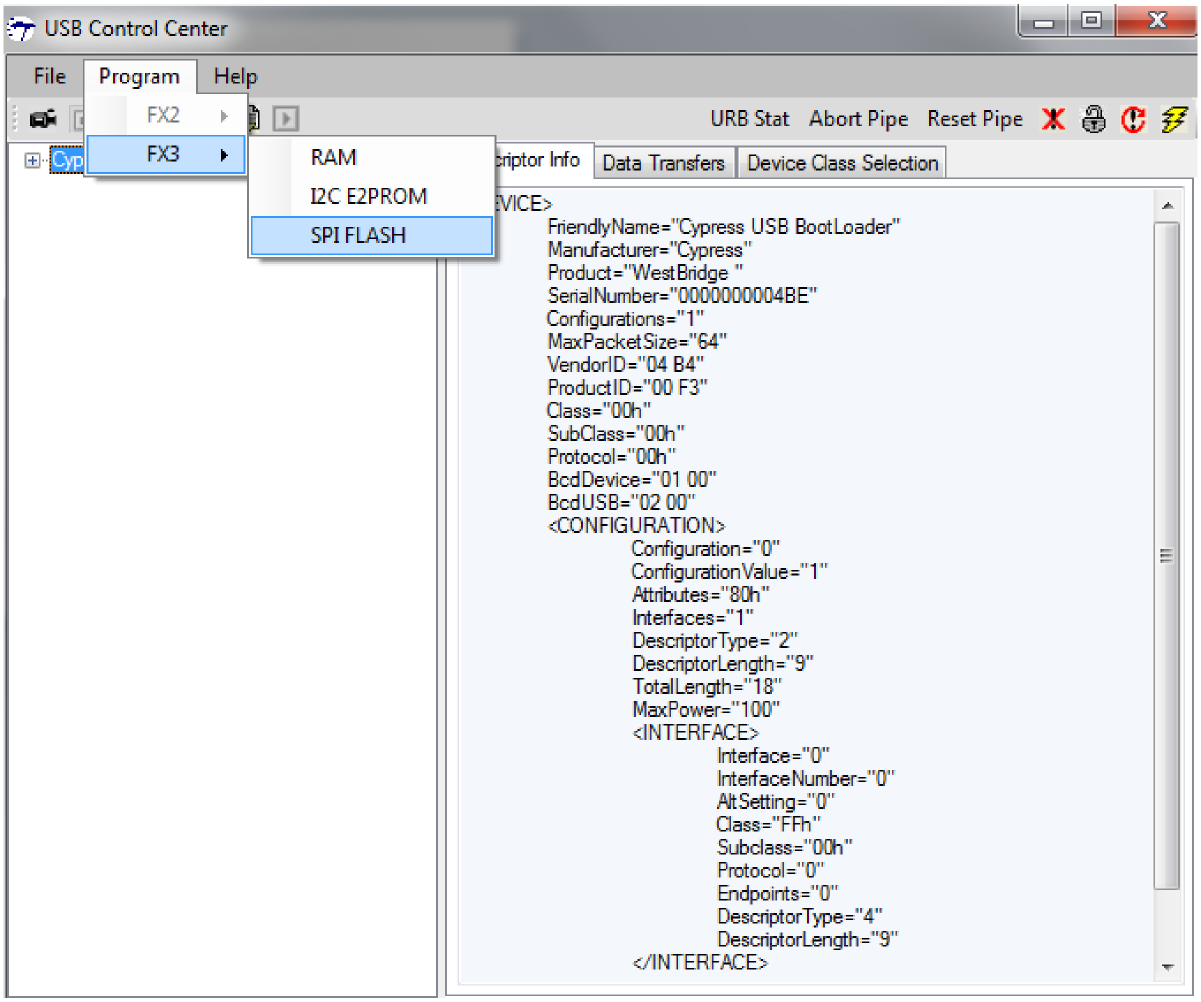


Figure 29. Double Click on the .img file to be Programmed into SPI Flash

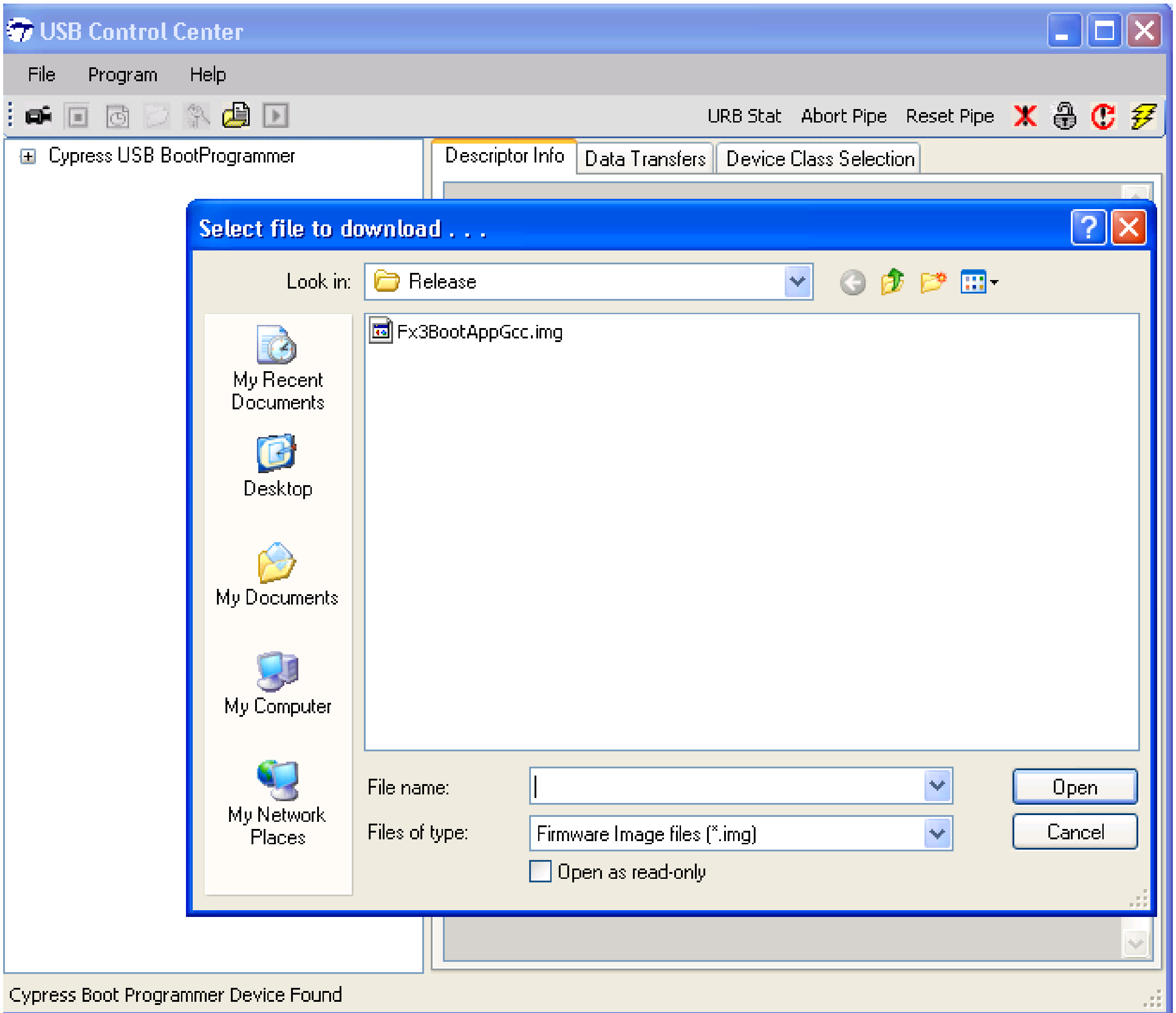
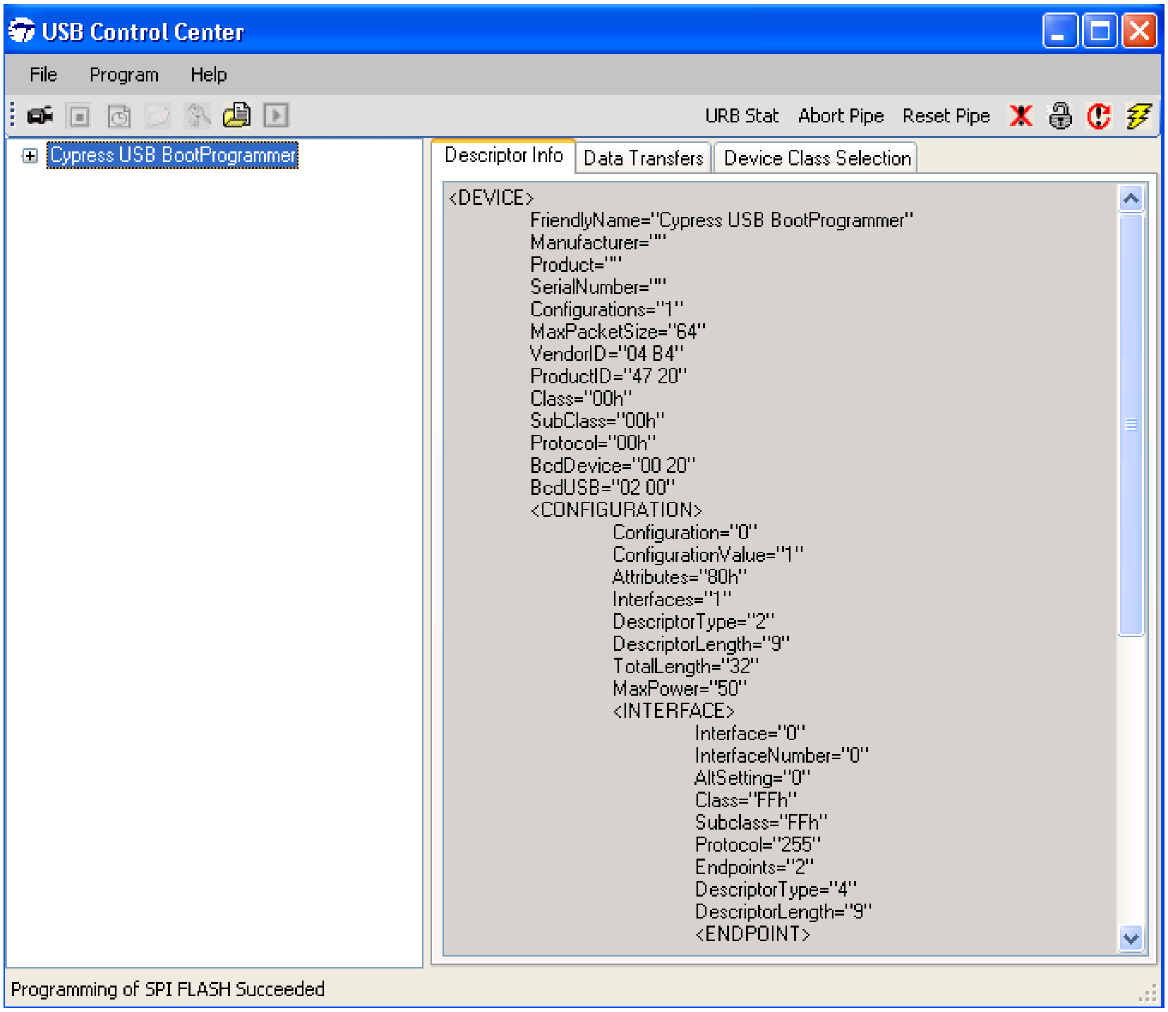


Figure 30. Successful Programming of SPI Flash Indicated at Bottom Left of Control Center



5. Change the PMODE[2:0] pins on the DVK board to 0Z1 in order to enable SPI boot. On the DVK board this is done by configuring the jumpers and switches as shown in Table 31.

Table 31. Jumper Configurations for SPI Boot

Jumper/Switch	Position	State of Corresponding PMODE Pin
J96 (PMODE0)	2-3 Closed	PMODE0 controlled by SW25
J97 (PMODE1)	Open	PMODE1 Floats
J98 (PMODE2)	2-3 Closed	PMODE2 controlled by SW25
SW25.1-8 (PMODE0)	Open (OFF position)	PMODE0 = 1
SW25.2-7 (PMODE1)	Don't care	PMODE1 Floats
SW25.3-6 (PMODE2)	Closed (ON position)	PMODE2 = 0

6. Reset the DVK. Now the FX3 boots from the SPI Flash.

Document History

Document Title: EZ-USB® FX3 Boot Options

Document Number: 001-76405

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3616262	VSO	05/14/2012	New application note
*A	3807283	OSG	11/19/2012	Merged the following application notes into AN76405: AN73150, AN70193, AN68914, and AN73304 Clarified the SPI FLash parts tested for boot Added an example for Sync ADMux firmware download implementation Added a step-by-step-sequence of instructions for testing boot options on the DVK Added a table with the default state of the GPIOs during boot
*B	3836755	OSG	12/10/12	Template updates. Table 26 – Updated default state of GPIO[33] for all boot modes Updated default states of GPIO[51], GPIO[55]-[57] for SPI boot mode.
*C	3964017	OSG	04/12/13	Updated GPIO[55] in Table 26.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5LP

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com