AN76405 describes the boot options—over USB, I²C, serial peripheral interface (SPI), and synchronous Address Data Multiplexed (ADMux) interfaces—available for the EZ-USB™ FX3 peripheral controller. This application note is also applicable to FX3S and CX3 peripheral controllers.

This application note is primarily intended for EZ-USB™ FX3/FX3S/CX3 users.

For a consolidated list of USB SuperSpeed Code Examples, visit .

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 USB, I^2C, SPI, and synchronous ADMux interfaces.

The default state of the FX3 I/Os during boot is also documented. covers the stepwise sequence for testing the different boot modes using the .

Infineon provides a wealth of data at to help you to select the right device for your design, and to help you to integrate the device into your design quickly and effectively.

- Overview: ,
- USB 3.0 product selectors: , , ,
- Application notes: Infineon offers a large number of USB application notes covering a broad range of topics, from basic to advanced level. Recommended application notes for getting started with FX3 are:
  - – Getting started with EZ-USB™ FX3
  - – Getting started with EZ-USB™ SX3
  - – EZ-USB™ FX3 /FX3S hardware design guidelines and schematic checklist
  - – Designing with the EZ-USB™ FX3 slave FIFO interface
  - – How to implement an image sensor Interface with EZ-USB™ FX3 in a USB Video Class (UVC) framework
  - – Optimizing USB 3.0 throughput with EZ-USB™ FX3
  - – Configuring an FPGA over USB using EZ-USB™ FX3
  - – Slave FIFO interface for EZ-USB™ FX3 : 5-bit address mode
  - – Differences in implementation of EZ-USB™ FX2LP and EZ-USB™ FX3 applications
  - – USB RAID 1 disk design using EZ-USB™ FX3S
- Code examples:
  -
  -
  -
- Technical reference manual (TRM):
  -
- Development kits:
  -
  -
  - Models:

Infineon delivers the complete software and firmware stack for FX3 to easily integrate SuperSpeed USB into any embedded application. The (SDK) comes with tools, drivers, and application examples, which help accelerate application development.

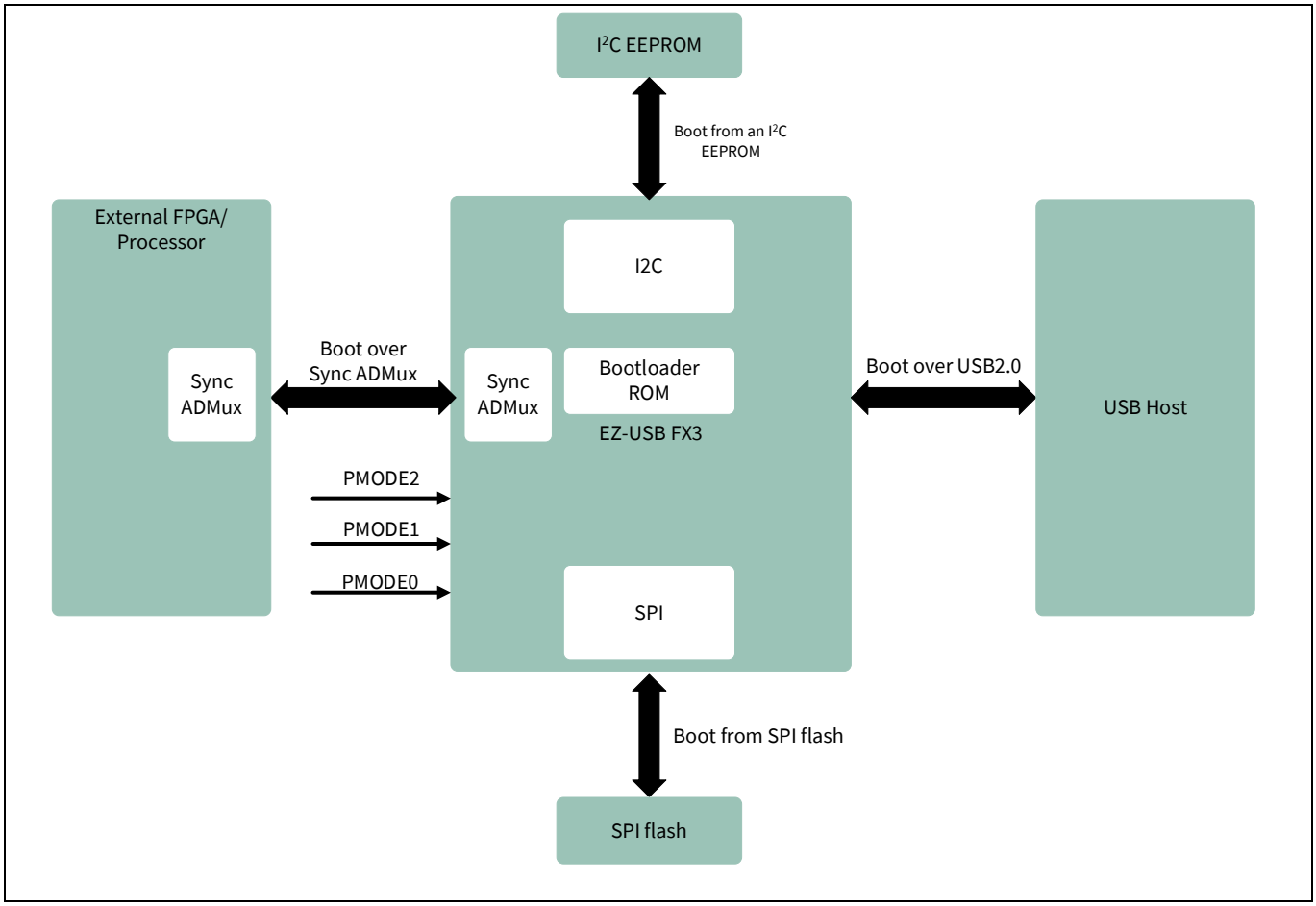
The is a graphical software that allows designers to configure the GPIF II interface of the EZ-USB™ FX3 USB 3.0 Device Controller.

The tool allows users the ability to select from one of five Infineon-supplied interfaces, or choose to create their own GPIF II interface from scratch. Infineon has supplied industry-standard interfaces such as asynchronous and synchronous Slave FIFO, and asynchronous and synchronous SRAM. Designers who already have one of these pre-defined interfaces in their system can simply select the interface of choice, choose from a set of standard parameters such as bus width (x8, 16, x32) endianness, clock settings, and then compile the interface.

The tool has a streamlined three-step GPIF interface development process for users who need a customized interface. Users can first select their pin configuration and standard parameters. Secondly, they can design a virtual state machine using configurable actions. Finally, users can view the output timing to verify that it matches the expected timing. After this three-step process is complete, the interface can be compiled and integrated with FX3.

FX3 integrates a bootloader that resides in the masked ROM. The function of the bootloader is to download the FX3 firmware image from various interfaces such as USB, I²C, SPI, or GPIF II (for example, synchronous ADMux).

The FX3 bootloader uses the three PMODE input pins of FX3 to determine the boot option to be used. shows the boot options discussed in this application note. lists these boot options along with the required PMODE pin settings.



Z	0	0	Sync ADMux (16-bit)	No
Z	1	1	USB Boot	Yes
1	Z	Z	I²C	No
Z	1	Z	I²C → USB	Yes
0	Z	1	SPI → USB	Yes

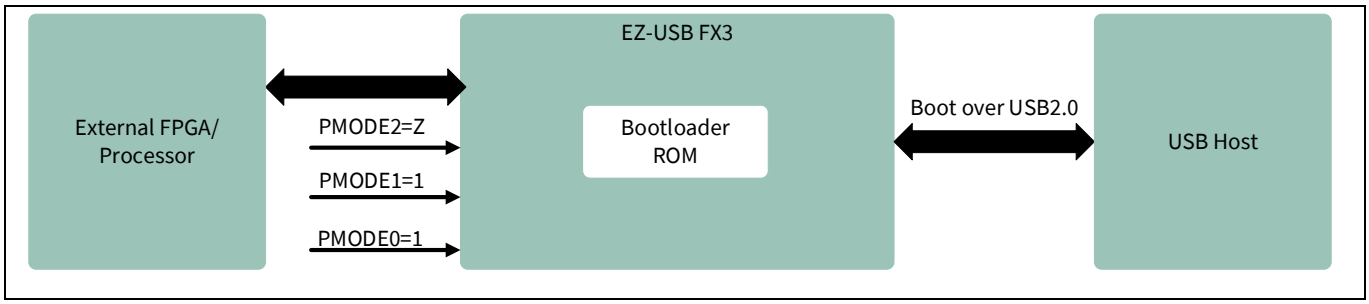
Other combinations are reserved.

Note: ** Z = Float. The PMODE pin can be made to float either by leaving it unconnected or by connecting it to an FPGA I/O and then configuring that I/O as an input to the FPGA.*

The following sections describe the boot options supported by FX3:

- : The FX3 firmware image is downloaded into the FX3 system RAM from the USB Host.
- : The FX3 firmware image is programmed into an external I^2C EEPROM, and on reset, the FX3 bootloader downloads the firmware over I^2C.
- : The FX3 firmware image is programmed into an external SPI flash or SPI EEPROM, and on reset, the FX3 bootloader downloads the firmware over SPI.
- : The FX3 firmware image is downloaded from an external processor or an FPGA connected to the FX3 GPIF II interface.

shows the system diagram for FX3 when booting over USB.



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

Z	1	1

Note: *Z = Float*

The external USB Host can download the firmware image to FX3 in USB 2.0 mode. FX3 enumerates as a USB Vendor class device with bus-powered support.

The state of FX3 in USB boot mode is as follows:

- USB 3.0 (SuperSpeed) signaling is disabled.
- USB 2.0 (High Speed/Full Speed) is enabled.
- FX3 uses the vendor command A0h for firmware download/upload. This vendor command is implemented in the bootloader.

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 be used only for development purposes. Users must use their own VID/PID for final products. A VID is obtained through registration with the USB-IF.

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

The bootloader revision is stored in the ROM area at the address FFFF_0020h, as shown in .

Major revision	FFFF_0021h
Reserved bytes	FFFF_0022h, FFFF_0023h

ReNumeration feature is supported in FX3 and is controlled by firmware.

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

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

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 may be slightly higher than the USB boot mode due to other clock sources being turned ON.

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

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

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

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

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 the **USB 2.0 specification** for the bitwise explanation.*

The FX3 bootloader handles the commands in .

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 so on
A1h-FFh	Reserved: Returns STALL

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

0	***BmRequestType***	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

0	***BmRequestType***	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

0	***bmRequestType***	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, the bootloader will turn off all the interrupts and disconnect the USB. Three examples of vendor command subroutines follow.

. 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 a length of transfer equal to wLength and a DATA IN Zero length packet.

. 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 a length of transfer equal to wLength and a DATA OUT Zero length packet.

. Jump to Program Entry With 8-Byte Setup Packet (refer to .)

bmRequestType = 0x40

bRequest = 0xA0;

wValue = Program Entry (16-bit LSB)

wIndex = Program Entry >>16 (16-bit MSB)

wLength = 0

Note: FX3 uses 32-bit addressing. Addresses should be written to the wValue and wIndex fields of the command.

To download the code, the application should read the firmware image file and write 4K 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 **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);/*fread(void *ptr, size_t size, size_t count, FILE
*stream)
                                  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 in the format specified in .

Note: *For an example implementation, see USB Control Center application source code project provided along with EZ-USB FX3 SDK.*

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

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, 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 about 10K 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).

The FX3 bootloader allows read access from the ROM, MMIO, SYSMEM, ITCM, and DTCM memory spaces. The bootloader allows write access to the MMIO, SYSMEM, ITCM, and DTCM memory spaces except for the first 1280-byte of DTCM and first 10K of system memory. When writing to the MMIO space, the expected transfer length for Bootloader must be four (equal to LONG word), and the address should be aligned by 4 bytes.

The FX3 bootloader can boot with your choice of VID and PID by scanning the eFUSE (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.

The FX3 bootloader does not support USB On-The-Go (OTG) protocol. It operates as a USB bus-powered device.

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

The 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 clock input for the USB core if the external 32-kHz clock is not present.

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

The following tables list the FX3 bootloader descriptors for High Speed and Full-Speed.

Note: The Device Qualifier is not available in the Full-Speed mode.

0	***bLength***	12h	Length of this descriptor = 18 bytes
1	***bDescType***	01	Descriptor type = Device
2-3	***wBCDUSB***	0200h	USB Specification version 2.0
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***	04B4h	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

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

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

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

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”

For USB boot, the bootloader expects the firmware image file to be in the format shown in . The provides a software utility that can be used to generate a firmware image in the format required for USB boot. Refer to the *elf2img* utility located in the *C:\Program Files\Cypress\EZ-USB FX3 SDK\1.3\util\elf2img* directory after installing the SDK. For 64-bit systems, the first folder in the path is Program Files(x86). The number *1.3* in the directory path is the version number of the SDK, and it can vary based on the latest release of the FX3 SDK. For more details on using the elf2img utility, see and in .

wSignature	1	Signature 2 bytes initialize with “CY” ASCII text.
bImageCTL;	½	Bit0 = 0: Execution binary file; 1: data file type Bit3:1 No use when booting in SPI EEPROM Bit5:4(SPI speed):

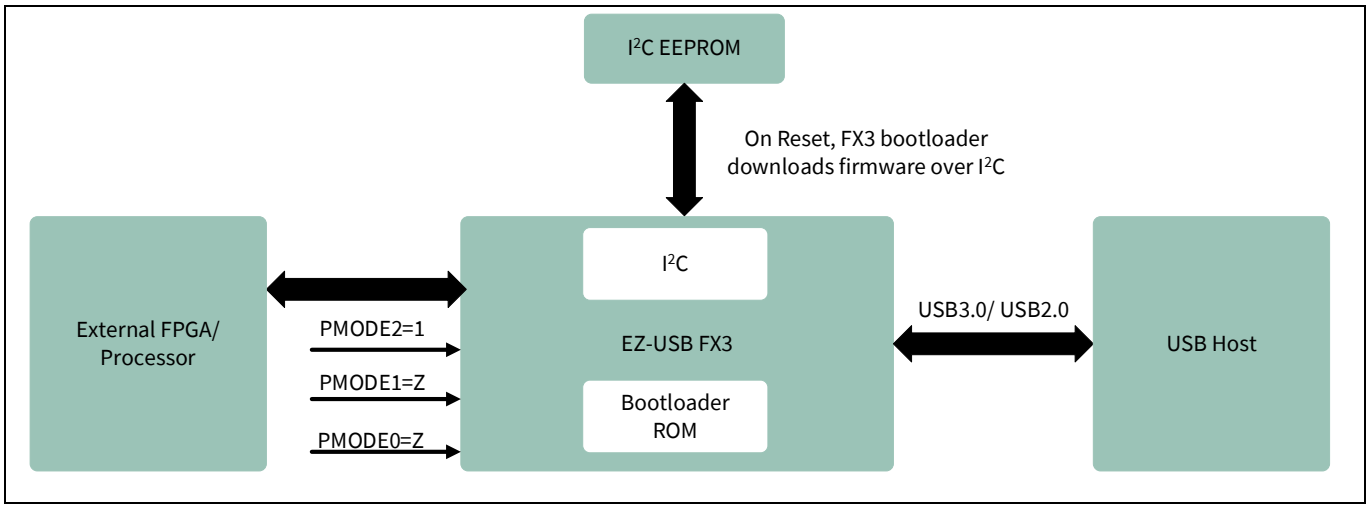
		00: 10 MHz 01: 20 MHz 10: 30 MHz 11: Reserved Bit7:6: Reserved, should be set to zero
bImageType;	½	bImageType = 0xB0: Normal FW binary image with checksum bImageType = 0xB2: I²C/SPI boot with new VID and PID
dLength 0	2	First section length, in long words (32-bit) When bImageType = 0xB2, the dLength 0 will contain PID and VID. Bootloader ignores the rest of the following data.
dAddress 0	2	First 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, that is, 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 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 first section to the termination section. The checksum will not include the dLength, dAddress, and Image Header.

```
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
Location14: 0x6AF37AF2      //Checksum (0x12345678 + 0x9ABCDEF1 + 0x23456789 +
              0xABCDEF12+ 0xDDCCBBAA +0x11223344)
```

The stepwise sequence for testing the USB boot mode using the is shown in the section of .

shows the system diagram for FX3 when booting over I²C.



For I²C EEPROM boot, the state of the PMODE[2:0] pins should be *1ZZ*, as shown .

1	Z	Z

The pin mapping of the FX3 I²C interface is shown in .

I2C_GPIO[58]	I2C_SCL
I2C_GPIO[59]	I2C_SDA

- FX3 boots from I²C EEPROM devices through a two-wire I²C 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
  - 2048 Kb or 256 KB

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

Note: *It is recommended to use the firmware image built in Release mode, as the size of the generated image file in the Release version is smaller than that in the Debug version.*

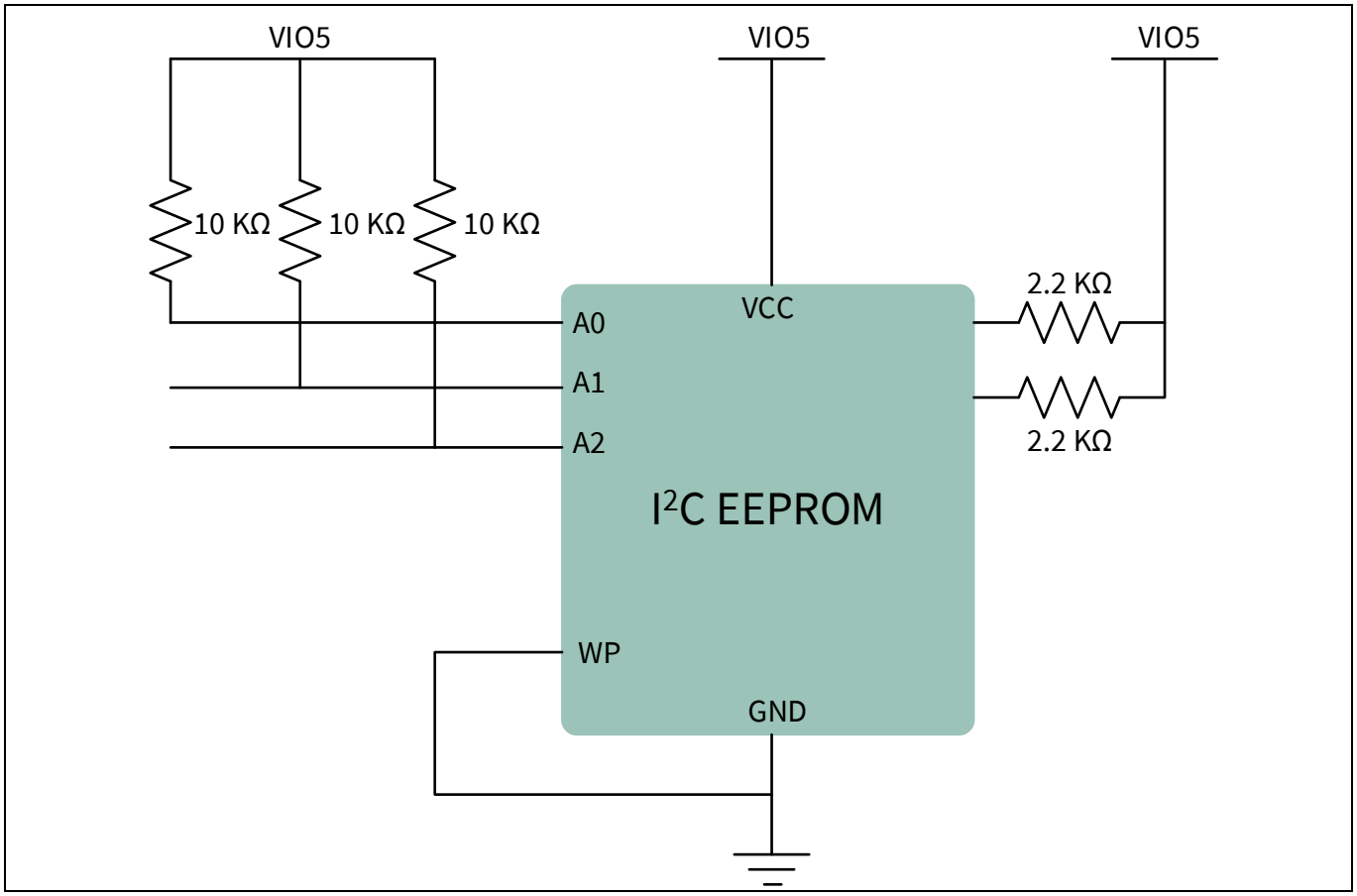
- ATMEL, Microchip, and ST Electronics devices have been tested (for example M24M02, AT24C1024, etc)
- 100 kHz, 400 kHz, and 1 MHz I^2C frequencies are supported during boot. Note that when V_{IO5} is 1.2 V, the maximum operating frequency supported is 100 kHz. When V_{IO5} is 1.8 V, 2.5 V, or 3.3 V, the operating frequencies supported are 400 kHz and 1 MHz. (V_{IO5} is the I/O voltage for I^2C interface)
- The FX3 bootloader only supports an I^2C EEPROM with slave address 0xA0
- Boot from multiple I^2C EEPROM devices of the same size is supported. When the I^2C EEPROM is smaller than the firmware image, multiple I^2C EEPROM devices must be used. The bootloader supports loading the image across multiple I^2C EEPROM devices. SuperSpeed Explorer CYUSB3KIT-003 uses a 256 KB EEPROM (M24M02) from ST Electronics. The bootloader can support up to eight I^2C EEPROM devices smaller than 128 KB. The bootloader can support up to four I^2C EEPROM devices of 128 KB
- Only one firmware image can be stored on I^2C EEPROM. No redundant images are allowed
- The bootloader does not support the multimaster I^2C feature of FX3. Therefore, during the FX3 I^2C booting process, other I^2C masters should not perform any activity on the I^2C bus

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

The I^2C 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 to EEPROM.

The A0, A1, and A2 pins are the address lines. They set the slave device address from 000 to 111. This makes it possible to address eight I^2C EEPROMs of the same size. These lines should be pulled HIGH or LOW based on the address required.



In the case of a 128-KB I²C 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.

shows how four Microchip 24LC1025 EEPROM devices can be connected.

1	0x00000-0x1FFFF	Vcc 0 0	128 KB
2	0x20000-0x3FFFF	Vcc 0 1	128 KB
3	0x40000-0x5FFFF	Vcc 1 0	128 KB
4	0x60000-0x7FFFF	Vcc 1 1	128 Kbytes

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

1	0x00000-0x1FFFF	0 0 NC	128 KB

2	0x20000-0x3FFFF	0 1 NC	128 KB
3	0x40000-0x5FFFF	1 0 NC	128 KB
4	0x60000-0x7FFFF	1 1 NC	128 KB

Note: NC indicates no connection.

For example, if the firmware code size is greater than 128 KB, then you must use two I²C EEPROMs, with the addressing schemes corresponding to that EEPROM, as shown in the previous two tables. The firmware image should be stored across the EEPROMs as a contiguous image as in a single I²C EEPROM.

The bootloader expects the firmware image file to be in the format shown in . The provides a software utility that can be used to generate a firmware image in the format required for I²C EEPROM boot. Refer to the *elf2img* utility located in the *C:\Program Files\Cypress\EZ-USB FX3 SDK\1.3\util\elf2img* directory after installing the SDK. For 64-bit systems, the first folder in the path is Program Files(x86). The number *1.3* in the directory path is the version number of the SDK, and it can vary based on the latest release of the FX3 SDK. For more details on using the elf2img utility, see and in .

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: 128 KB (microchip) 6: 64 KB (128K ATMEL and 256K ST Electronics) 5: 32 KB 4: 16 KB 3: 8 KB 2: 4 KB *Note: Options 1 and 0 are reserved for future usage. Unpredicted results will occur when booting in these modes.* *Bit5: 4 (I²C speed)* *00: 100 kHz* *01: 400 kHz* *10: 1 MHz* *11: Reserved*

		Note: The bootloader power-up default will be set at 100 kHz, 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 = 0xB2: I²C boot with new VID and PID
dLength 0	2	First section length, in long words (32-bit) When bImageType = 0xB2, the dLength 0 will contain PID and VID. The bootloader will ignore the rest of the following data.
dAddress 0	2	First section address of Program Code, not the I²C address *Note: The internal ARM address is byte addressable, so the address for each section should be 32-bit aligned.*
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 startup code, that is, 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 the ITCM area or SYSTEM RAM area.* *The bootloader does not validate the Program Entry*
dCheckSum	2	The 32-bit unsigned little-endian checksum data will start from the First sections to the termination section. The checksum will not include the dLength, dAddress, and Image Header.

The binary image file is stored in the I²C EEPROM in the following order:

Byte0: “C”

Byte1: “Y”

Byte2: bImageCTL

Byte3: bImageType

…..

Byte N: Checksum of Image

Attention:

- Bootloader default boot speed = 100 kHz; to change the speed from 100 kHz to 1 MHz, bImageCTL<5:4> should be set to 10.
- To select the I²C EEPROM size, bImageCTL[3:1]should be used.
- The addressing for the Microchip EEPROM 24LC1026 is different from the addressing of other 128-KB Microchip EEPROMs. If using Microchip EEPROM 24LC1026, the I²C EEPROM size field, for example, bImageCTL[3:1], should be set to 6.
- Default value of bImageCTL byte in .img files generated by the FX3 SDK is 0x1C (64KB size and 400 KHz speed). See for details on how to generate image file for a different I²C configuration.

The bootloader computes the checksum when loading the binary image in the I²C 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.

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)
```

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 stepwise sequence for testing the USB boot mode using the is shown in the section of .

The following is 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 USB fallback enabled.

For the I²C EEPROM boot with USB fallback, the state of the PMODE[2:0] pins should be Z1Z, as shown in
.

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.

The I²C EEPROM boot fails under the following conditions:

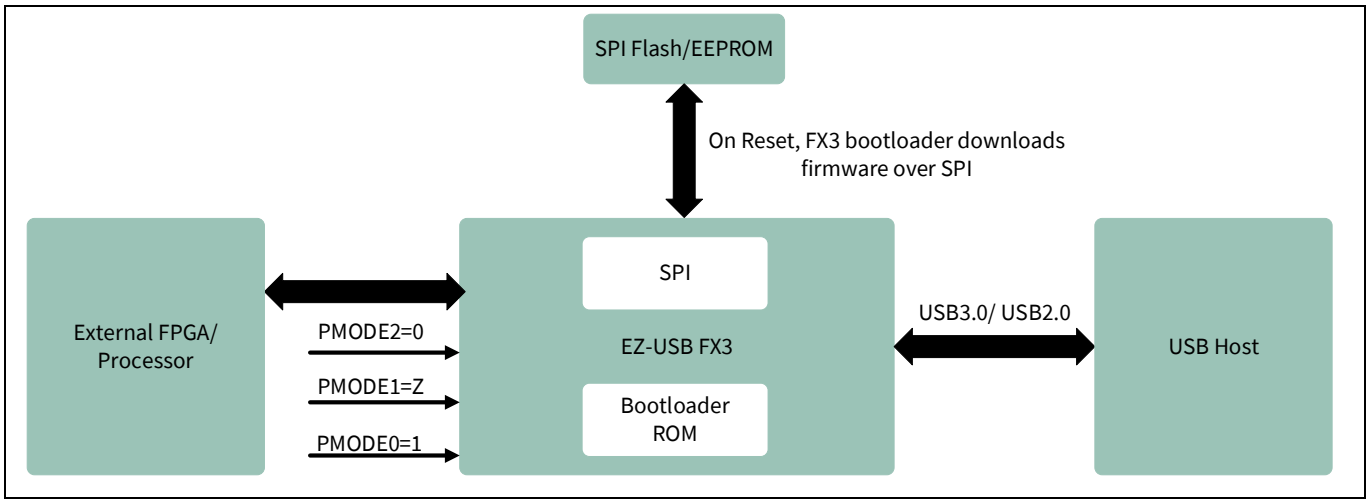
- I²C address cycle or data cycle error
- Invalid signature in FX3 firmware image
- Invalid image type

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.

- 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 I²C EEPROM.
- On USB fallback, 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, which will consume about 200 mA. However, the FX3 chip is typically observed to consume about 100 mA.

```
Location1: 0xB2 0x1A 'Y' 'C' //CY Signature,32k EEPROM,400Khz,0xB2 Image
Location2: 0x04B40008        //VID = 0x04B4 | PID=0x0008
```

shows the system diagram for FX3 when booting over SPI.



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

0	Z	1

The pin mapping of the FX3 SPI interface is shown in .

GPIO[53]	SPI_SCK
GPIO[54]	SPI_SSN
GPIO[55]	SPI_MISO
GPIO[56]	SPI_MOSI

FX3 boots from SPI flash/EEPROM devices through the 4-wire SPI interface.

- SPI flash/EEPROM devices from 1 Kb to 128 Mb in size are supported for boot.

  Supported SPI Flash parts:

  - SPI Flash (S25FS064S (64-Mbit), S25FL064L (64-Mbit) and S25FS128S (128-Mbit))
  - Winbond W25Q32FW (32-Mbit)
- SPI frequencies supported during boot are ~10 MHz, ~20 MHz, and ~30 MHz.
- Note that the SPI frequency may 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, refer to the SPI timing in the .)
- USB fallback is supported and used for storing new VID/PID information for USB boot. See the section in this application note for more information.

SPI flash should support the following commands to support FX3 boot.

- Read data: 03h with 3-byte addressing
- Read Status register: 05h
- Write Enable: 06h
- Write data (Page Program): 02h
- Sector Erase: D8h

SPI flash can be used for FX3 boot as long as the read commands match. If there are any differences in the write commands, then programming of that SPI flash will not be successful with the provided *CyBootProgrammer.img* (located at *C:\Program Files (x86)\Cypress\EZ-USB FX3 SDK\1.3\bin*); it requires changing the SPI write commands used in the USBFlashProg example project of the FX3 SDK. The image file created after building the modified USBFlashProg project should replace the default *CyBootProgrammer.img* (with the same name) for successful programming of the SPI flash.

Certain SPI Flash devices (like S25FS128S/S25FS256S etc) may support hybrid sectors. For example, the first sector of 64KB may be divided into 9 sectors (8 sectors of 4KB and one 32KB sector). All remaining sectors are uniform (64KB each). When USB Control Center application from FX3 SDK is used for programming SPI flash, the internal programming utility (*CyBootProgrammer.img*) assumes uniform sector size for the flash device; this leads to corrupted data in the first sector, which is a hybrid sector. To prevent this, configure the flash device to use uniform sectors of 64KB each by modifying the appropriate configuration register.

See to learn more about using hybrid SPI flash devices with FX3.

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. shows the pinout of a typical SPI flash/EEPROM.

The SPI EEPROM interface consists of four active wires:

- CS#: Chip Select
- SO: Serial Data Output (master in, slave out (MISO))
- SI: Serial Data Input (master out, slave in (MOSI))
- SCK: Serial Clock input

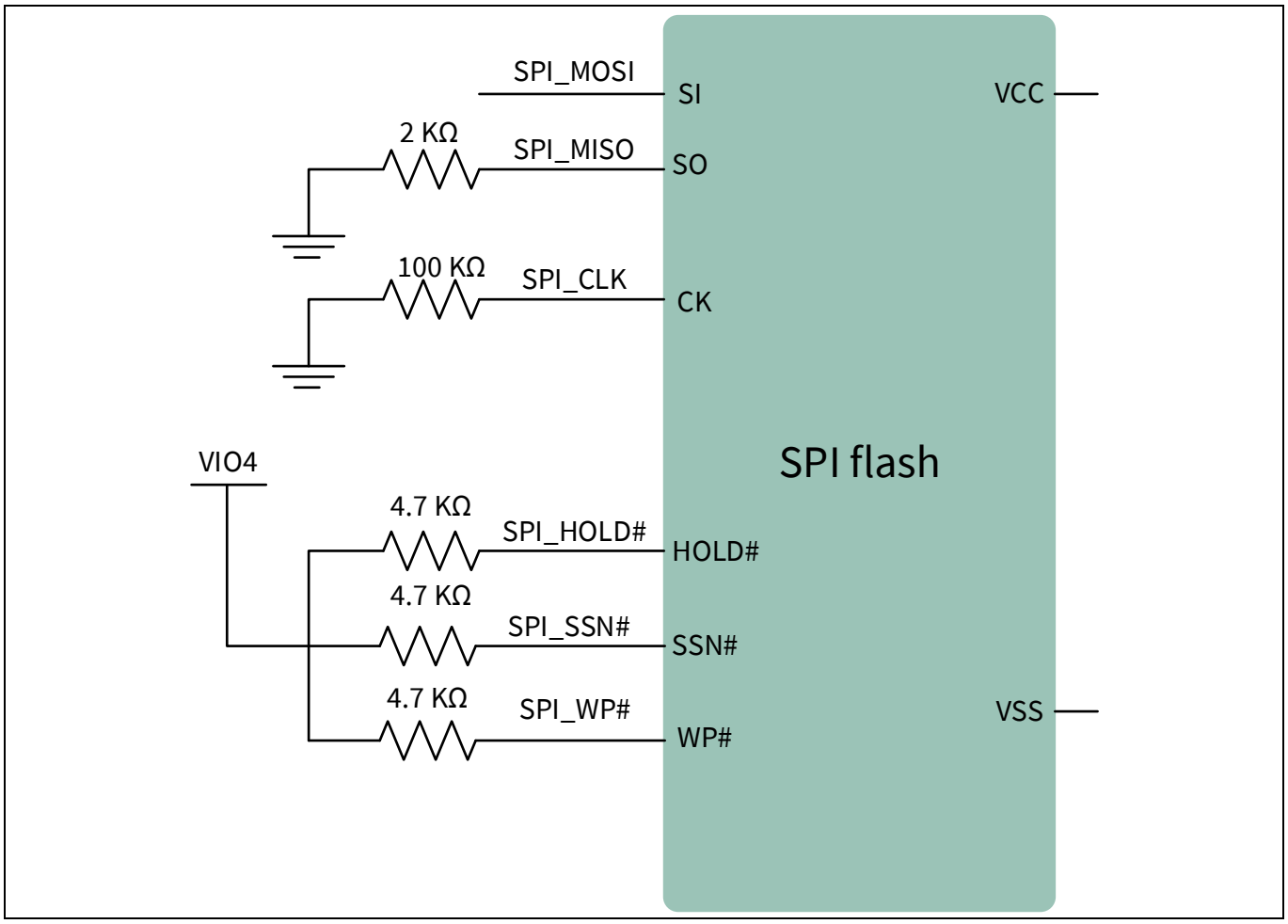
The HOLD# signal should be tied to VCC while booting or reading from the SPI device

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

Note: *External pull-ups should not be connected on the MOSI and MISO signals, as shown in **Figure 6**.*

Note: *It is recommended to include a pull-down resistor (2 K) on the MISO line.*

Note: *Refer to the "SPI and UART" section of **AN70707 EZ-USB FX3/FX3S hardware design guidelines and schematic checklist** for recommendations on SPI pin connections.*



For SPI boot, the bootloader expects the firmware image file to be in the format shown in . The provides a software utility that can be used to generate a firmware image in the format required for SPI boot. Refer to the *elf2img* utility located in the *C:\Program Files\Cypress\EZ-USB FX3 SDK\1.3\util\elf2img* directory after installing the SDK. For 64-bit systems, the first folder in the path is Program Files(x86). The number *1.3* in the directory path is the version number of the SDK, and it can vary based on the latest release of the FX3 SDK. For more details on using the elf2img utility, see and in .

wSignature	1	Signature 2 bytes initialize with “CY” ASCII text
bImageCTL	½	Bit0:0: execution binary file; 1: data file type Bit3:1 Not used 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 run only up to 33 MHz.* *Bit7:6: Reserved. Should be set to zero.*
bImageType	½	bImageType = 0xB0: Normal firmware binary image with checksum bImageType = 0xB2: SPI boot with new VID and PID
dLength 0	2	First 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	First section address of program code *Note: The 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, that is, 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 the 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 first section to the termination section. The checksum will not include the dLength, dAddress, and Image Header.

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 Point 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.

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 the 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 section to the termination section. The checksum will not include the dLength, dAddress, and Image Header. Refer to the section for the sample code to calculate the checksum.

The following is an example of a firmware image stored only at one section in the 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)
```

The following is an example of a firmware image stored at two sections in the 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 stepwise sequence for testing the USB boot mode using the is shown in the section of .

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 VID and 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

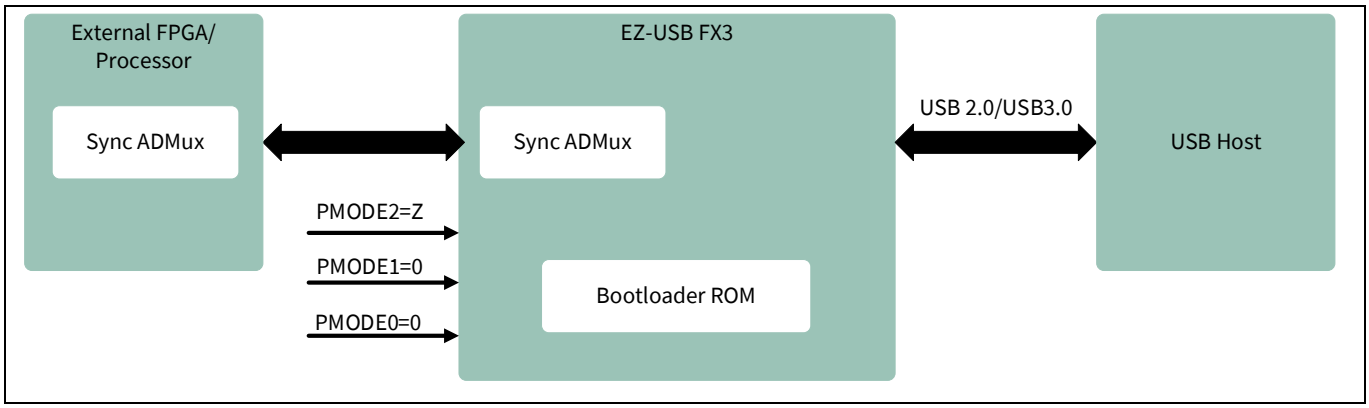
A special image type is used to denote that instead of the FX3 firmware image, data on SPI flash/EEPROM is the VID and PID for USB boot. This helps in having a new VID and PID for USB boot.

- In the 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 fallback, 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, which will consume about 200 mA. However, the FX3 chip is typically observed to consume about 100 mA.

```
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.

shows the FX3 system diagram when booting over the synchronous ADMux interface.



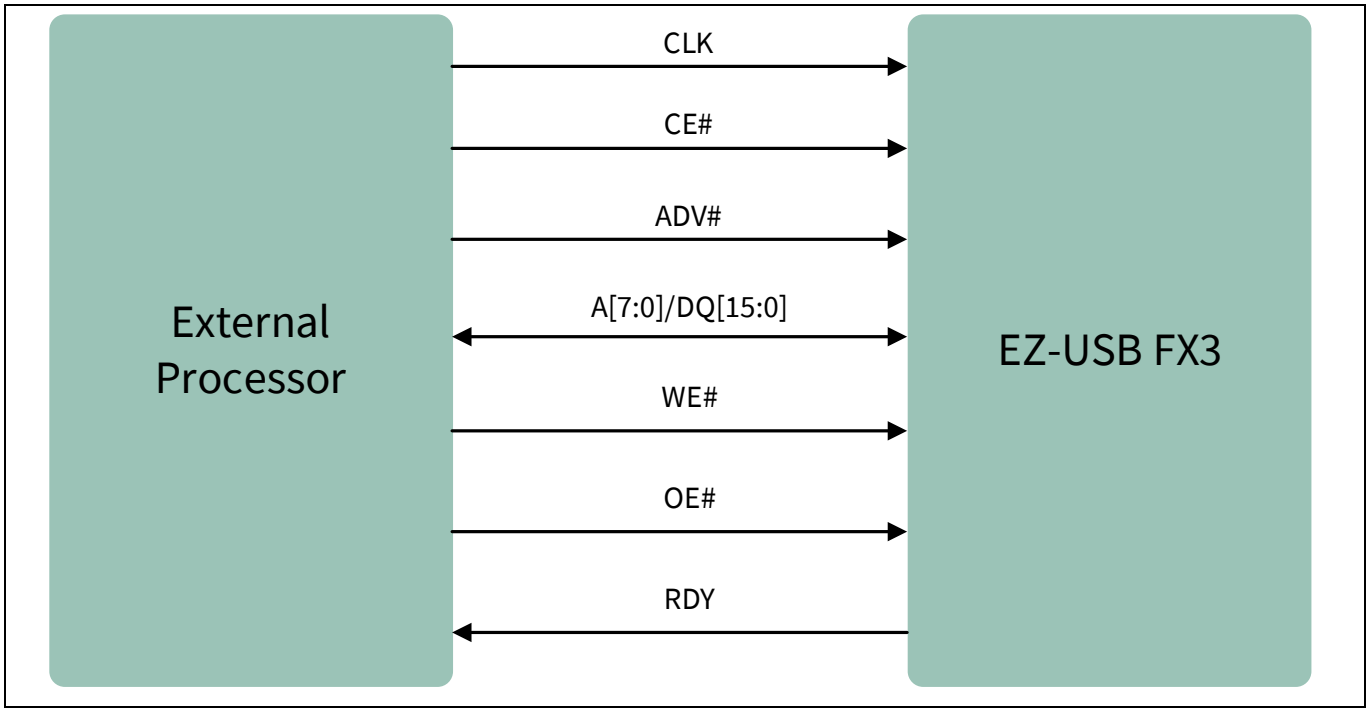
For booting over the synchronous ADMux interface, the state of the PMODE[2:0] pins should be Z00, as shown in
.

Z	0	0

The FX3 GPIF II interface supports a synchronous ADMux 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 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

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



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.

The pin mapping of the FX3 sync ADMux interface is shown in .

GPIO[7:0]/GPIO[15:0]	A[7:0]/DQ[15:0]
GPIO[16]	CLK
GPIO[17]	CE#
GPIO[18]	WE#
GPIO[19]	OE#
GPIO[25]	RDY
GPIO[27]	ADV#

For details on the sync ADMux timing diagrams (synchronous ADMux interface—read cycle timing and write cycle timing) and timing parameters, see , , and .

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 GPIF II for the sync ADMux interface. This process can take a few hundred microseconds. Read/write access to FX3 should be performed only after the bootloader has completed the configuration. Otherwise, data corruption can result. To avoid it, use one of the following schemes:

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

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

When a 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 GPIF II configuration is preserved.

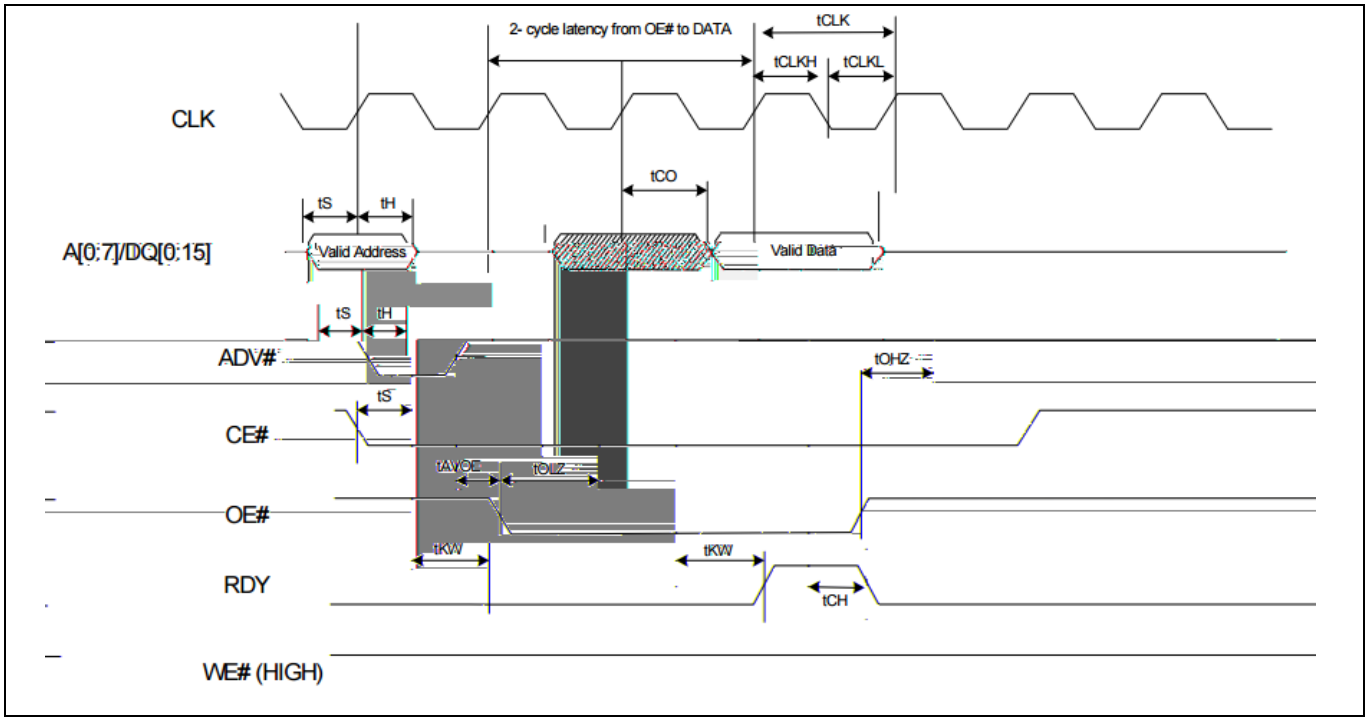
After a wakeup from standby, the application firmware is responsible for configuring and restoring the hardware registers, GPIF II configuration, ITCM, or DTCM.

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

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

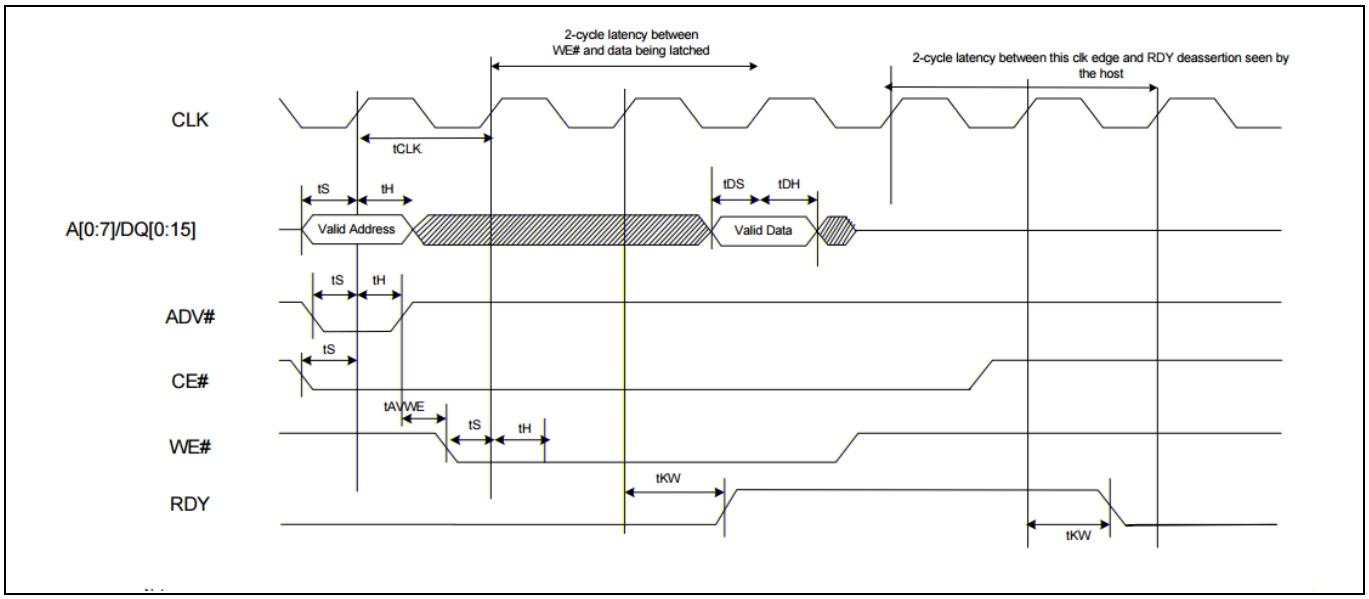
- Invalidates both DCACHE and ICACHE
- Turns ON ICACHE
- Disables MMU
- Turns ON DTCM and ITCM
- Sets up the stack using the DTC

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.



Note:

1. *The External P-Port processor and FX3S operate on the same clock edge.*
2. *The External processor sees RDY assert two cycles after OE# asserts and sees RDY deassert a cycle after the data appears on the output.*
3. *Valid output data appears two cycles after OE# is asserted. The data is held until OE# deasserts.*
4. *Two-cycle latency is shown for 0-100 MHz operation. Latency can be reduced by 1 cycle for operations at less than 50 MHz. (This 1-cycle latency is not supported by the bootloader.)*



Note:

1. *The External P-Port processor and FX3S operate on the same clock edge.*
2. *The External processor sees RDY assert two cycles after WE# asserts and deasserts three cycles after the edge sampling the data.*
3. *Two-cycle latency is shown for 0-100 MHz operation. Latency can be reduced by 1 cycle for operations at less than 50 MHz. (This 1-cycle latency is not supported by the bootloader.)*

FREQ	Interface Clock frequency	-	100	MHz
tCLK	Clock period	10	-	ns
tCLKH	Clock HIGH time	4	-	ns
tCLKL	Clock LOW time	4	-	ns
tS	CE#/WE#/DQ setup time	2	-	ns
tH	CE#/WE#/DQ hold time	0.5	-	ns
tCH	Clock to data output hold time	0	-	ns
tDS	Data input setup time	2	-	ns
tDH	Clock to data input hold time	0.5	-	ns
tAVDOE	ADV# HIGH to OE# LOW	0	-	ns
tAVDWE	ADV# HIGH to WE# LOW	0	-	ns
tHZ	CE# HIGH to Data HI-Z	-	8	ns
tOHZ	OE# HIGH to Data HI-Z	-	8	ns
tOLZ	OE# LOW to Data HI-Z	0	-	ns
tKW	Clock to RDY valid	-	8	ns

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 .

bSignature[2]	Sender initialize with “CY” The bootloader responds with “WB”
bCommand	Sender: 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	

	For bCommand 00: bLenStatus = 0 (the bootloader will jump to addr in dAddr if bCommand is WRITE_DATA_CMD and ignore value for all other commands) bCommand 01: Length in Long Word (Max = (512-8)/4) bCommand 02: Number of 512 byte blocks (Max = 16) bCommand 03: Length in Long Word (Max = (512-8)/4) Bootloader responds 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 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	Sender: Address used by command 1 and 3
dData[bLenStatus]	Data length determine by bLenStatus Sender: Data to be filled by the Sender

Note:

1. *The error code bLenStatus will be reported on the mailbox of the GPIF II.*
2. *When downloading firmware to FX3 using sync ADMUX, the external AP should ensure the following:*
   a) Command block length is exactly 512 bytes
   b) Response block length is exactly 512 bytes
   c) The bootloader binary image is converted to a data stream that is segmented in multiples of 512 bytes.
   d) The data chunk of the bootloader image is not larger than 8K. For instance, on the command 0x02, the bLenStatus should not be larger than 16 blocks (8K bytes). The limitation stated above is due to the maximum DMA buffer size. The maximum DMA_SIZE that the bootloader supports is 8K and thus, the AP can send only 8K data (max) per transfer. If the firmware size is greater than 8K, multiple transfers are needed to download the complete firmware to FX3 . i.e, step 4 of section should be repeated for each transfer until the complete firmware is transferred
   e) The host does not send more than the total image size.
3. *The bootloader does not support queuing commands. Therefore, every time a command is sent, the host must read the response.*
4. *To prevent the corruption of the 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 also not be used (0x1000_0000 – 0x1000_04FF).*
5. *On the WRITE_DATA_CMD: When bLenStatus = 0, the bootloader jumps to the Program Entry of the dAddr.*

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

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

Note: *Refer to the "FX3 Terminology" section in the **Getting started with EZ-USB™ FX3** application note to learn about the concept of sockets in FX3 .*

The host processor uses available GPIF II sockets to transfer blocks of data into and out of FX3 . The 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 the 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 four 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 the following sequence of socket accesses:

1. One command socket writes 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 reads 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 writes 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 be 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.

The register list shown in indicates how the PP_xxx registers are mapped on the external P-Port address space. Addresses in this space indicate a word, not a byte address. The sync ADMux interface provides eight address lines to access these registers.

PP_ID	0x80	16	P-Port Device ID Register. Provides device ID information
PP_INIT	0x81	16	P-Port reset and power control. This register is used for reset and power control and determines the endian orientation of the P-Port.
PP_CONFIG	0x82	16	P-Port configuration register
PP_IDENTIFY	0x83	16	P-Port identification register. The lower 8 bits of this register are read-only and defaulted to 0x81.
PP_INTR_MASK	0x88	16	P-Port Interrupt Mask Register. This register has the same layout as PP_EVENT and masks the events that lead to assertion of interrupt.
PP_DRQR5_MASK	0x89	16	P-Port DRQ/R5 Mask Register. This register has the same layout as PP_EVENT and masks the events that lead to assertion of interrupt or DRQ/R5 respectively.
PP_ERROR	0x8C	16	P-Port error indicator register
PP_DMA_XFER	0x8E	16	P-Port DMA transfer register. This register is used to set up and control a DMA transfer.
PP_DMA_SIZE	0x8F	16	P-Port DMA Transfer Size Register. This register indicates the (remaining) size of the transfer.
PP_WR_MAILBOX	0x90	64	P-Port Write Mailbox Registers. These registers contain a message of up to eight bytes from the AP to FX3 firmware.
PP_MMIO_ADDR	0x94	32	P-Port MMIO Address Registers. These registers together form a 32-bit address for accessing the FX3 internal MMIO space.
PP_MMIO_DATA	0x96	32	P-Port MMIO Data Registers. These registers together form a 32-bit data for accessing the FX3 internal MMIO space.
PP_MMIO	0x98	16	P-Port MMIO Control Register. This register controls the access to the FX3 MMIO space.
PP_EVENT	0x99	16	P-Port Event Register. This register indicates all types of events that can cause interrupt or DRQ to assert.
PP_RD_MAILBOX	0x9A	64	P-Port Read Mailbox Registers. These registers contain a message of up to eight bytes from FX3 firmware to the AP.
PP_SOCK_STAT	0x9E	32	P-Port Socket Status Register. These registers contain 1 bit for each of the 32 sockets in the P-port, indicating the buffer availability of each socket.

Refer to the “Registers” chapter in the for the bit field definitions of these registers.

Before delving into the details of the FX3 firmware download, note that the following functions are frequently used in the example implementation in this document and are platform-dependent. Contact 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
```

Note: *While performing register access, the most significant bit of the 8-bit address should be 1, notifying FX3 that it is register access operation. Similarly, for performing socket access, the most significant bit should be set to 0.*

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

The following 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.

```
/* Register addresses and the constants used in the code shown below. */
#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


// All register addresses defined with bit 7 set to indicate Register access (not
Socket)
#define PP_CONFIG             0x82
   #define CFGMODE            0x0040



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)== 0) {
                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);
```

```
        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;
}
```

The following 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 the 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 deasserts 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 (multibuffer) 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, the 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.

```
/* Register addresses and the constants used in the code shown below. */
#define PP_CONFIG              0x82
   #define CFGMODE             0x0040

#define PP_DRQR5_MASK          0x89
   #define DMA_WMARK_EV        0x0800

#define PP_DMA_XFER            0x8E
   #define LONG_TRANSFER       0x0400
   #define DMA_DIRECTION       0x0200
   #define DMA_ENABLE          0x0100
#define PP_EVENT               0x99
   #define DMA_READY_EV        0x1000

#define PP_RD_MAILBOX0         0x9A     // 64 Bit register accessed as 4 x 16 bit
registers
#define PP_RD_MAILBOX1         0x9B
#define PP_RD_MAILBOX2         0x9C
#define PP_RD_MAILBOX3         0x9D

#define PP_SOCK_STAT_L         0x9E     // LSB 16 bits of 32 bit register
#define PP_SOCK_STAT_H         0x9F     // MSB 16 bits of 32 bit register

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

   u32 count;
   u16 val, buf_sz;
   int i;

        buf_sz = 512;
        /* 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++); /* Write 512 bytes of data continuously to
data socket 16 bits at a time ( Sync ADMux has 16 data lines) */
                count += (buf_sz / 2);

                if (count < (sz/2))
                        do {
                                udelay(10);
                                val = IORD_REG16 (PP_SOCK_STAT_L); /* After writing 512
bytes to data socket of the device, P-Port Socket Status Register is read to check if
the Socket is available for reading or writing next set of 512 bytes data */
                        } while(!(val&(0x1<<sck)));/* You remain in this Do-while loop
till PP_SOCK_STAT_L register makes the bit corresponding to the socket as 1
indicating socket is now available for next read/write */

        } while (count < (sz/2)); /* sz is the total size of data to be written. In
case of firmware_download, sz will be total size of the firmware */

         /* 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 sck_bootloader_read(u8 sck, u32 sz, u16 *p)
{

   u32 count;
   u16 val, buf_sz;
   int i;
        buf_sz = 512;
        /* 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 troubleshooting the synchronous ADMux boot, please refer to

.

For sync ADMux boot, the bootloader expects the firmware image to be in the format shown in . The 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 utility located in the *C:\Program Files\Cypress\EZ-USB FX3SDK\1.3\util\elf2img* directory after installing the SDK. For 64-bit systems, the first folder in the path is Program Files(x86). The number *1.3* in the directory path is the version number of the SDK, and it can vary based on the latest release of the FX3 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 shown previously. shows how the elf2img post-build command is issued, followed by an example for printing the contents of the *.img* file into an array in ASCII format.

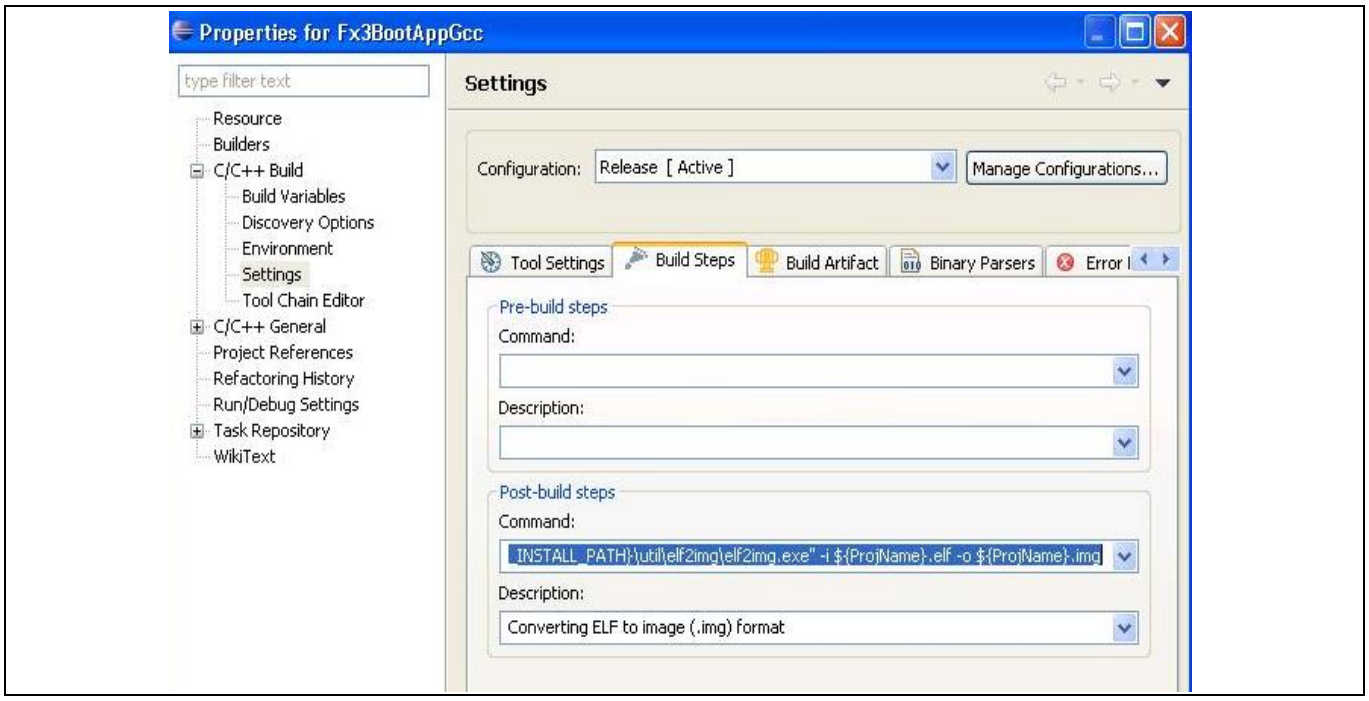
wSignature	1	Signature 2 bytes initialize with “CY” ASCII text
bImageCTL;	½	Bit0 = 0: execution binary file; 1: data file type Bit3:1 Do not use when booting in SPI EEPROM Bit5:4 (SPI speed): 00: 10 MHz 01: 20 MHz 10: 30 MHz 11: Reserved Bit7:6: Reserved, should be set to '0'

bImageType;	½	bImageType = 0xB0: Normal FW binary image with checksum bImageType = 0xB2: SPI boot with new VID and PID
dLength 0	2	First 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 following data.
dAddress 0	2	First section address of Program Code *Note: The 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, 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 bootloader 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 first section to the termination section. The checksum will not include the dLength, dAddress, and Image Header.

An 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)
```



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;
}
```

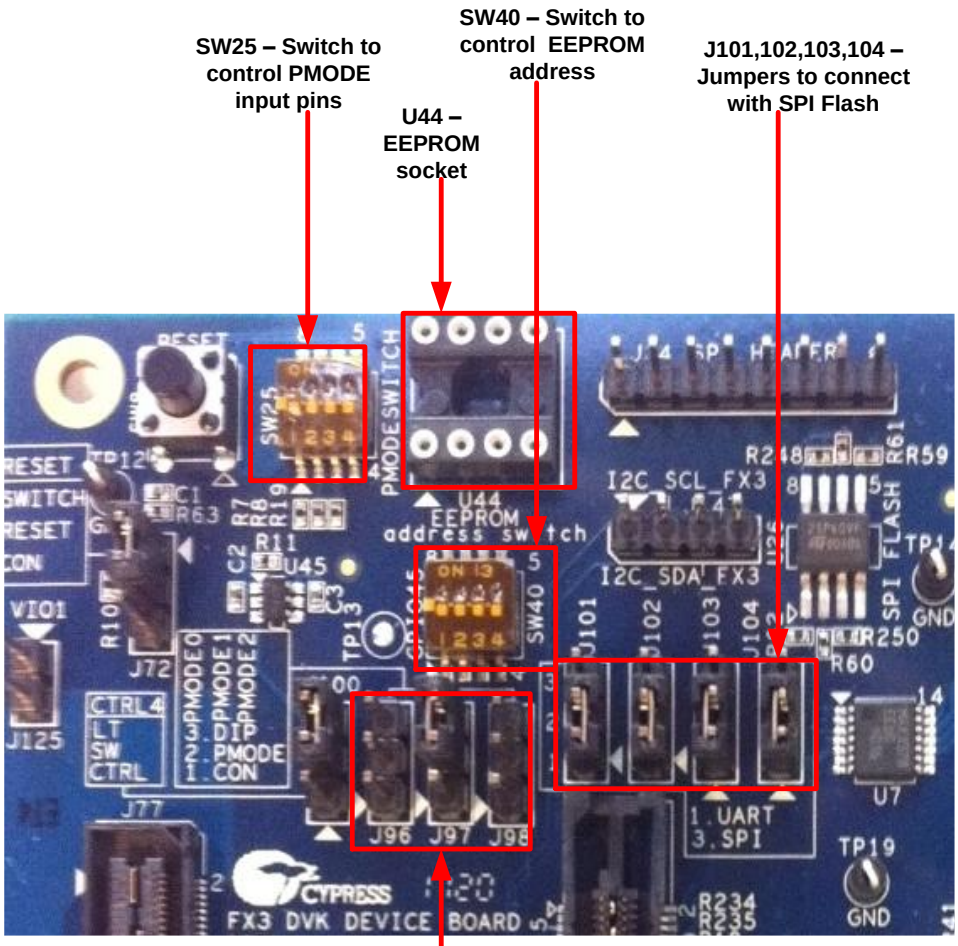
shows the default state of the FX3 I/Os for the different boot modes, while the bootloader is executing before application firmware download).

Note: *The default state of the GPIOs need not be same when FX3 is in reset and after the boot-loader finishes the configuration.*

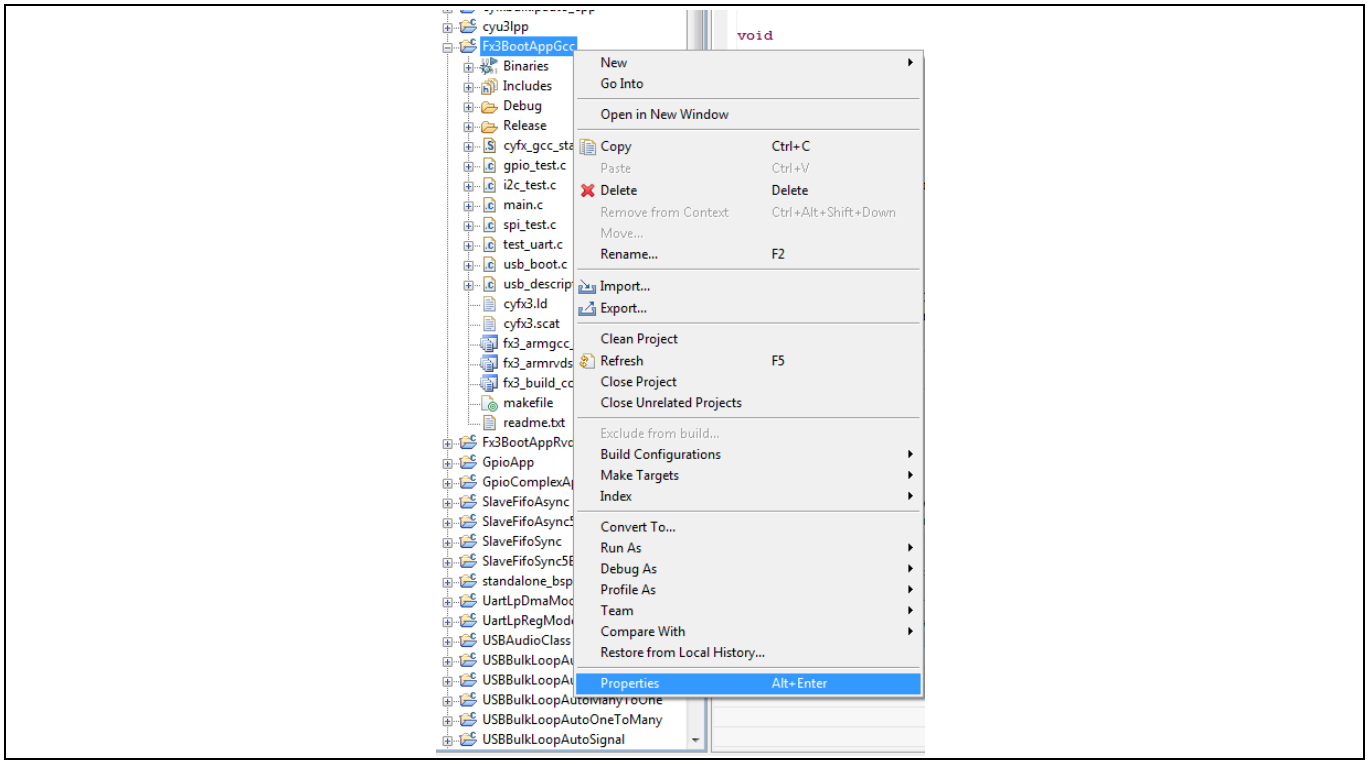
GPIO[0]	Tristate	Tristate	Tristate	Tristate
GPIO[1]	Tristate	Tristate	Tristate	Tristate
GPIO[2]	Tristate	Tristate	Tristate	Tristate
GPIO[3]	Tristate	Tristate	Tristate	Tristate
GPIO[4]	Tristate	Tristate	Tristate	Tristate
GPIO[5]	Tristate	Tristate	Tristate	Tristate
GPIO[6]	Tristate	Tristate	Tristate	Tristate
GPIO[7]	Tristate	Tristate	Tristate	Tristate
GPIO[8]	Tristate	Tristate	Tristate	Tristate
GPIO[9]	Tristate	Tristate	Tristate	Tristate
GPIO[10]	Tristate	Tristate	Tristate	Tristate
GPIO[11]	Tristate	Tristate	Tristate	Tristate
GPIO[12]	Tristate	Tristate	Tristate	Tristate
GPIO[13]	Tristate	Tristate	Tristate	Tristate
GPIO[14]	Tristate	Tristate	Tristate	Tristate
GPIO[15]	Tristate	Tristate	Tristate	Tristate
GPIO[16]	Tristate	Tristate	Tristate	CLK Input
GPIO[17]	Tristate	Tristate	Tristate	Input
GPIO[18]	Tristate	Tristate	Tristate	Input
GPIO[19]	Tristate	Tristate	Tristate	Input
GPIO[20]	Tristate	Tristate	Tristate	Input
GPIO[21]	Tristate	Tristate	Tristate	Output
GPIO[22]	Tristate	Tristate	Tristate	Tristate
GPIO[23]	Tristate	Tristate	Tristate	Input
GPIO[24]	Tristate	Tristate	Tristate	Tristate
GPIO[25]	Tristate	Tristate	Tristate	Tristate
GPIO[26]	Tristate	Tristate	Tristate	Tristate
GPIO[27]	Tristate	Tristate	Tristate	Input
GPIO[28]	Tristate	Tristate	Tristate	Tristate
GPIO[29]	Tristate	Tristate	Tristate	Tristate

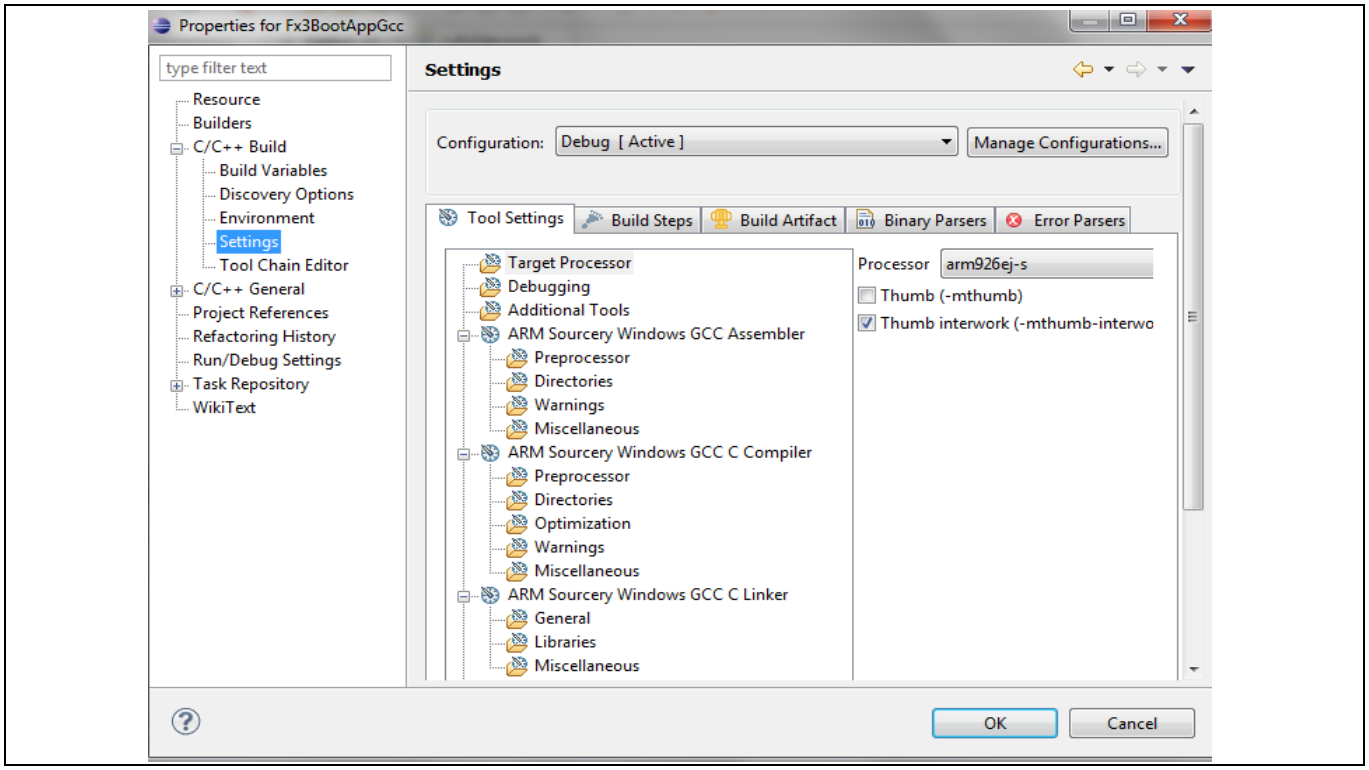
GPIO[30]	PMODE[0] I/P to FX3	PMODE[0] I/P to FX3	PMODE[0] I/P to FX3	PMODE[0] I/P to FX3
GPIO[31]	PMODE[1] I/P to FX3	PMODE[1] I/P to FX3	PMODE[1] I/P to FX3	PMODE[1] I/P to FX3
GPIO[32]	PMODE[2] I/P to FX3	PMODE[2] I/P to FX3	PMODE[2] I/P to FX3	PMODE[2] I/P to FX3
GPIO[33]	Tristate	Tristate	Tristate	Tristate
GPIO[34]	Tristate	Tristate	Tristate	Tristate
GPIO[35]	Tristate	Tristate	Tristate	Tristate
GPIO[36]	Tristate	Tristate	Tristate	Tristate
GPIO[37]	Tristate	Tristate	Tristate	Tristate
GPIO[38]	Tristate	Tristate	Tristate	Tristate
GPIO[39]	Tristate	Tristate	Tristate	Tristate
GPIO[40]	Tristate	Tristate	Tristate	Tristate
GPIO[41]	Tristate	Tristate	Tristate	Tristate
GPIO[42]	LOW	LOW	LOW	LOW
GPIO[43]	Tristate	Tristate	Tristate	Tristate
GPIO[44]	Tristate	Tristate	Tristate	Tristate
GPIO[45]	Tristate (HIGH if SPI boot fails)	HIGH	Tristate	HIGH
GPIO[46]	HIGH	Tristate	Tristate	Tristate
GPIO[47]	Tristate	Tristate	Tristate	Tristate
GPIO[48]	HIGH	Tristate	Tristate	Tristate
GPIO[49]	Tristate	Tristate	Tristate	Tristate
GPIO[50]	Tristate (LOW if SPI boot fails)	Tristate	Tristate	Tristate
GPIO[51]	LOW	LOW	LOW	LOW
GPIO[52]	HIGH	Tristate	Tristate	Tristate
GPIO[53]	LOW (toggles during SPI transactions)	HIGH	Tristate	HIGH
GPIO[54]	HIGH	Tristate	Tristate	Tristate
GPIO[55]	Tristate	HIGH	HIGH	HIGH
GPIO[56]	LOW	Tristate	Tristate	Tristate
GPIO[57]	LOW	Tristate	Tristate	Tristate
GPIO[58] I2C_SCL	Tristate	Tristate	Tristate (Toggles during transaction., then Tristated)	Tristate
GPIO[59] I2C_SDA	Tristate	Tristate	Tristate	Tristate

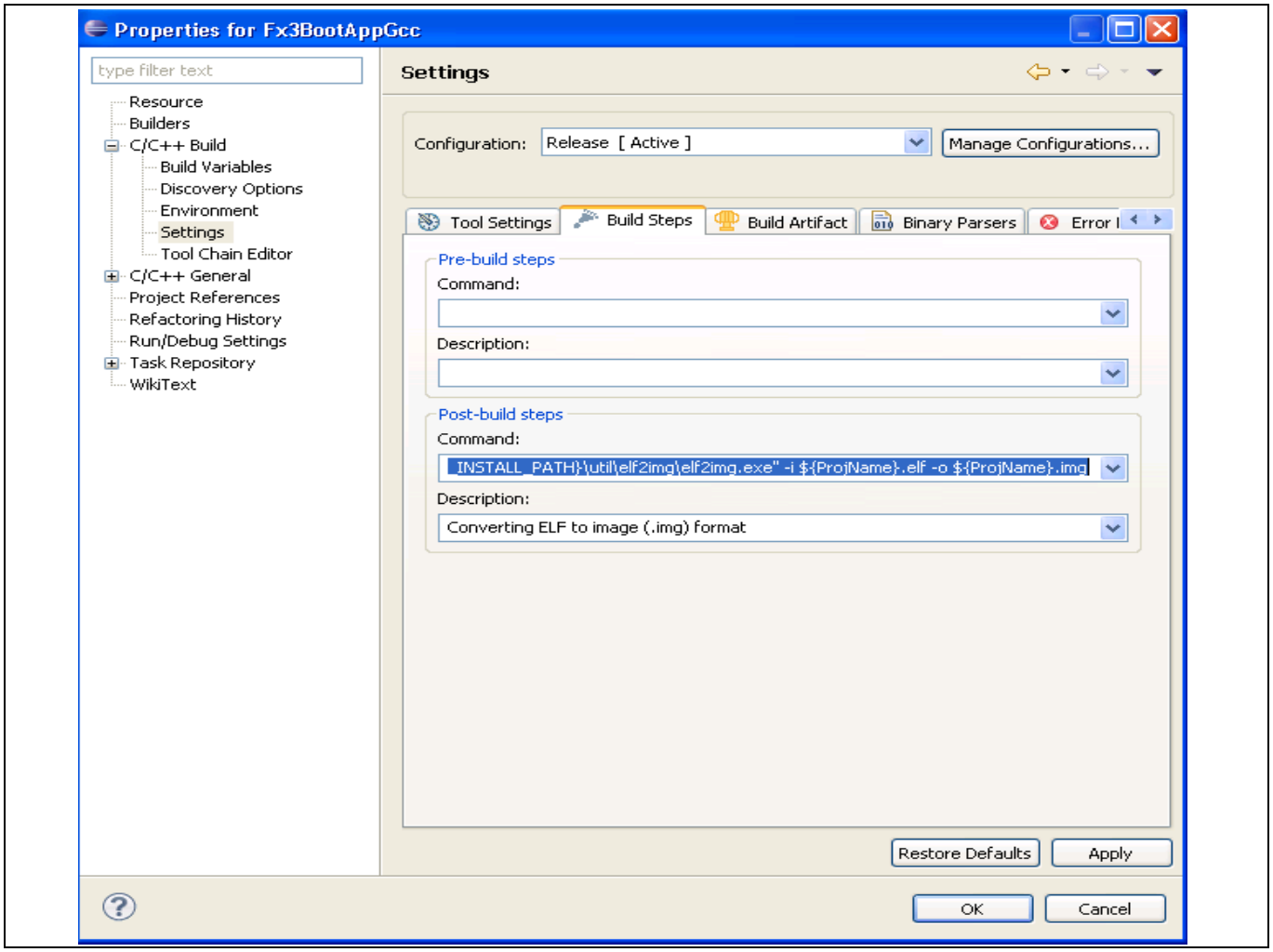
This appendix describes the stepwise sequence for exercising USB boot, I²C boot, and SPI boot using the FX3 DVK board. Figure 12 shows a part of the FX3 DVK board that contains switches and jumpers, which need to be configured appropriately for each boot option. The required settings for them are also described.



1. Build the firmware image in the Eclipse IDE as shown in , , and .


cyu3lpp
Fx3BootAppGcc
Binaries
Includes
Debug
Release
cyfx_gcc_sta
gpio_test.c
i2c_test.c
main.c
spi_test.c
test_uart.c
usb_boot.c
usb_descrip
cyfx3.ld
cyfx3.scat
fx3_armgcc_
fx3_armrvds
fx3_build_cc
makefile
readme.txt
Fx3BootAppRvc
GpioApp
GpioComplexA
SlaveFifoAsync
SlaveFifoAsync5
SlaveFifoSync
SlaveFifoSync5E
standalone_bsp
UartLpDmaMod
UartLpRegMod
USBAudioClass
USBBulkLoopA
USBBulkLoopA
USBBulkLoopAutoOneToMany
USBBulkLoopAutoSignal
void
New
Go Into
Open in New Window
Copy Ctrl+C
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Move...
Rename... F2
Import...
Export...
Clean Project
Refresh F5
Close Project
Close Unrelated Projects
Exclude from build...
Build Configurations
Make Targets
Index
Convert To...
Run As
Debug As
Profile As
Team
Compare With
Restore from Local History...
Properties Alt+Enter

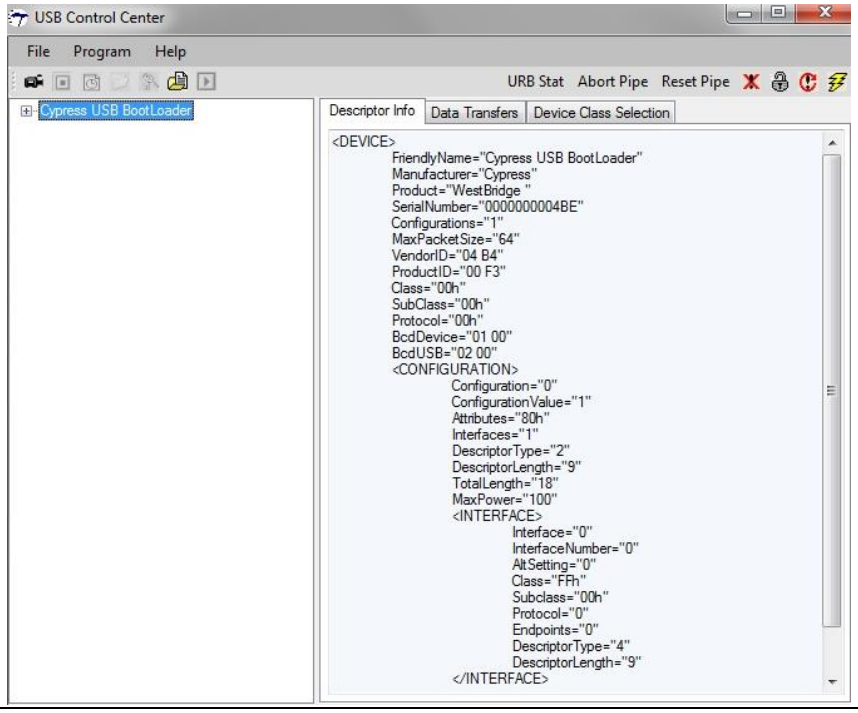

Properties for Fx3BootAppGcc
type filter text
Resource
Builders
C/C++ Build
Build Variables
Discovery Options
Environment
Settings
Tool Chain Editor
C/C++ General
Project References
Refactoring History
Run/Debug Settings
Task Repository
WikiText
Settings
Configuration: Debug [Active]
Manage Configurations...
Tool Settings
Build Steps
Build Artifact
Binary Parsers
Error Parsers
Target Processor
Debugging
Additional Tools
ARM Sourcery Windows GCC Assembler
Preprocessor
Directories
Warnings
Miscellaneous
ARM Sourcery Windows GCC C Compiler
Preprocessor
Directories
Optimization
Warnings
Miscellaneous
ARM Sourcery Windows GCC C Linker
General
Libraries
Miscellaneous
Processor arm926ej-s
Thumb (-mthumb)
Thumb interwork (-mthumb-interwo
OK
Cancel



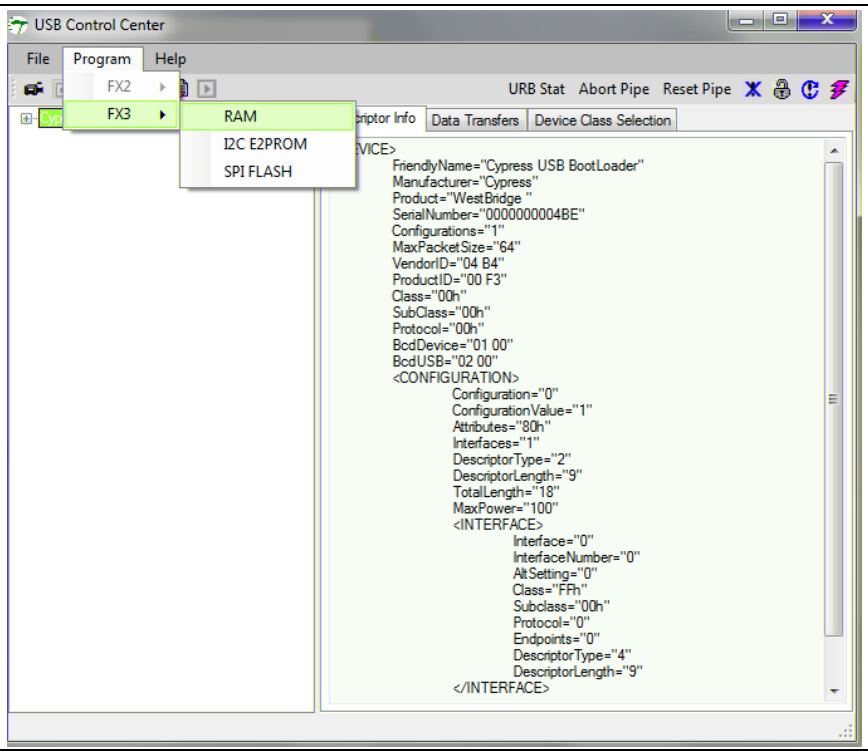
2. Enable USB boot by setting the PMODE[2:0] pins to Z11. On the DVK board, this is done by configuring the jumpers and switches as shown in .

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

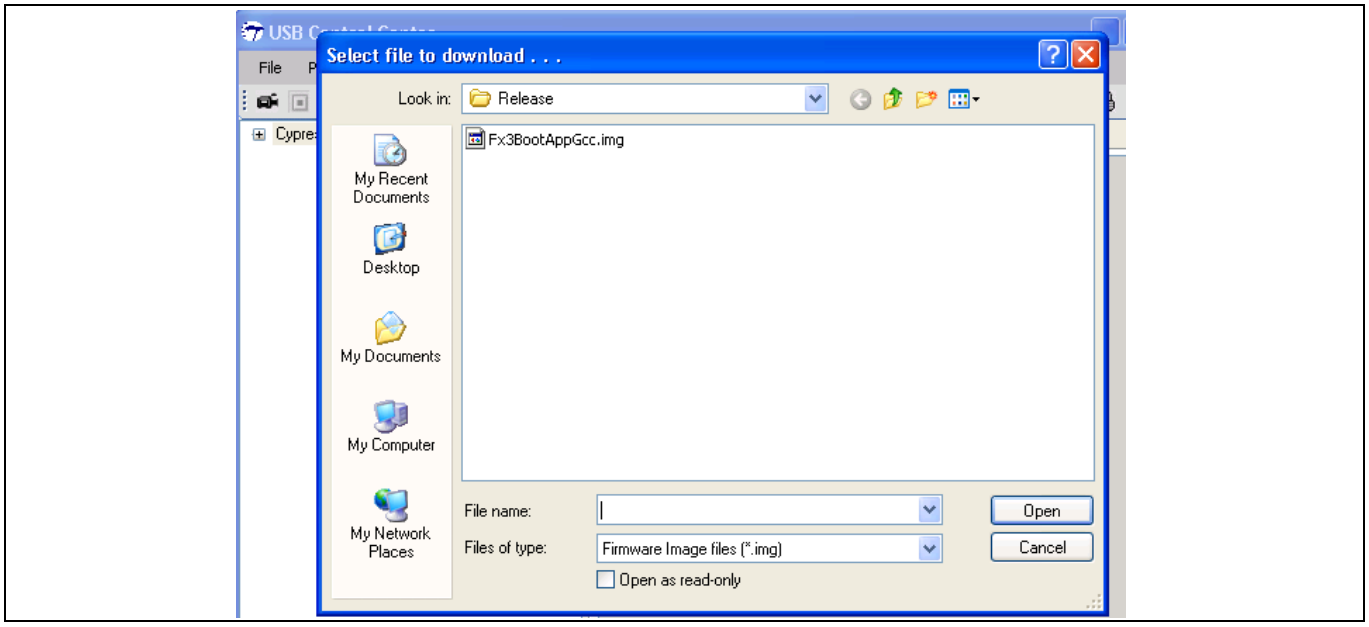
3. When connected to a USB Host, the FX3 device enumerates in the Control Center as “Cypress USB BootLoader,” as shown in .



4. In the Control Center, select the FX3 device by choosing as shown in .

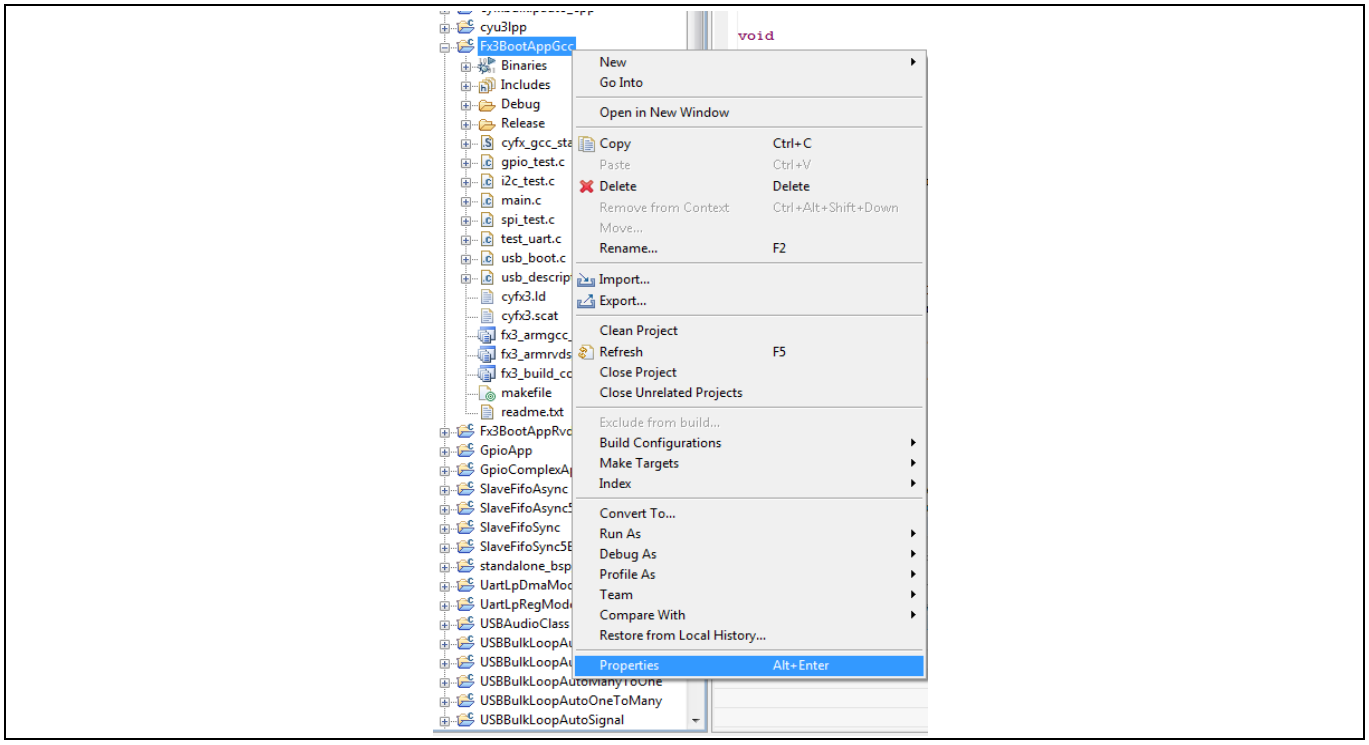


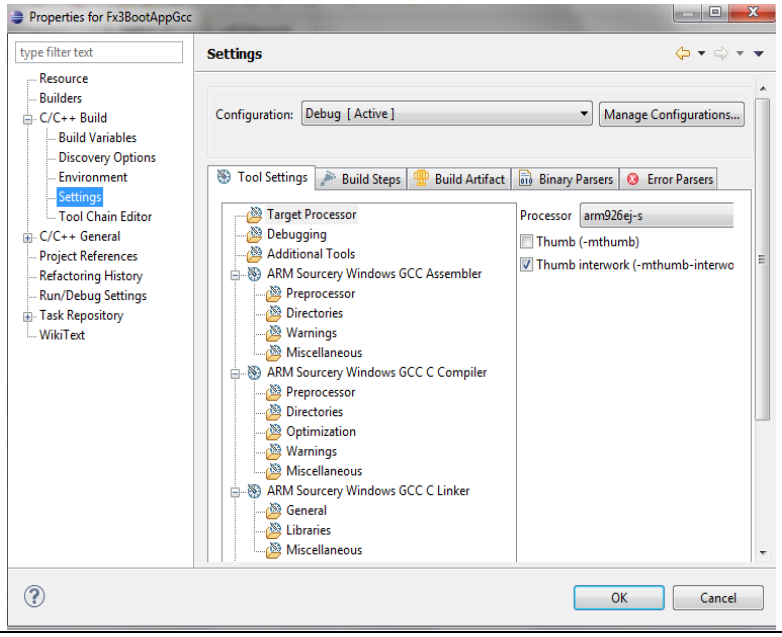
5. Next, browse to the *.img* file to be programmed into the FX3 RAM. Double-click on the *.img* file, as shown in .

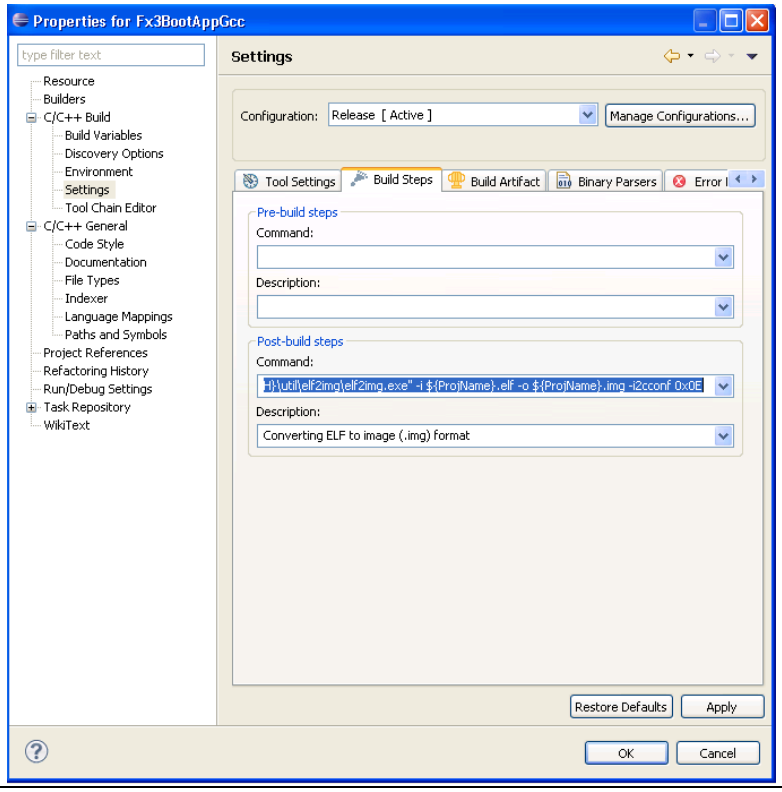


6. A “Programming Succeeded” message is displayed on the bottom left of the Control Center, and the FX3 device re-enumerates with the programmed firmware.

1. Build the firmware image in the Eclipse IDE as shown in , , and .



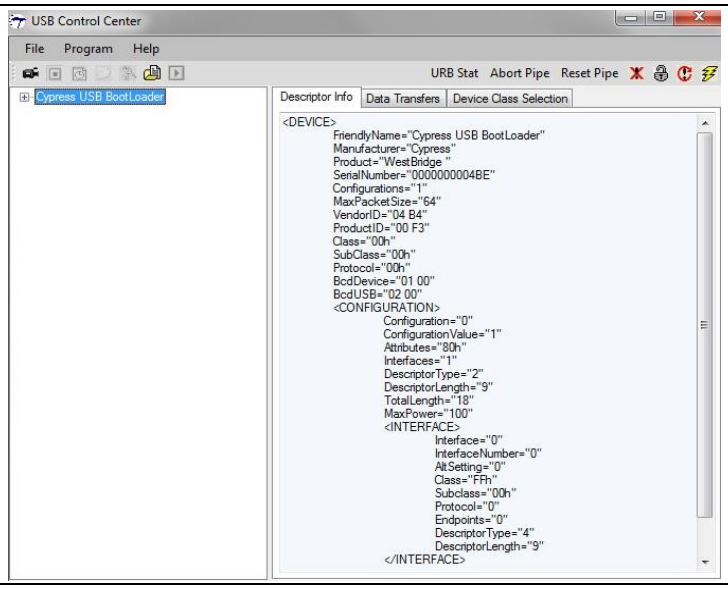




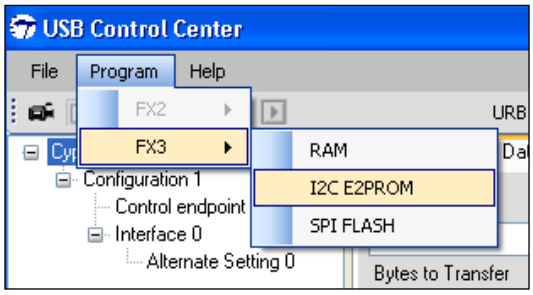
2. Enable USB boot, by setting the PMODE[2:0] pins to Z11. On the DVK board, this is done by configuring the jumpers and switches as shown in .

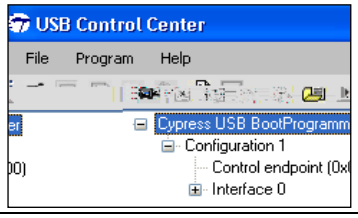
J96 (PMODE0)	2-3 Closed	PMODE0 controlled by SW25
J97 (PMODE1)	2-3 Closed	PMODE1 controlled by SW25
J98 (PMODE2)	Open	PMODE2 Floats
SW25.1-8 (PMODE0)	Open (OFF)	PMODE0 = 1
SW25.2-7 (PMODE1)	Open (OFF)	PMODE1 = 1
SW25.3-6 (PMODE2)	Don't care	PMODE2 Floats

3. When connected to a USB Host, the FX3 device enumerates in the Control Center as "Cypress USB BootLoader," as shown in .

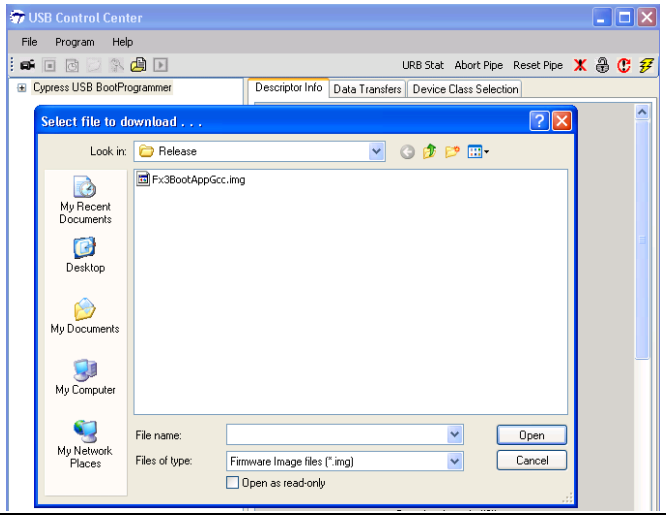


4. Before attempting to program the EEPROM, ensure that the address signals of the EEPROM are configured correctly using switch SW40 (For Microchip part 24AA1025, 1-8 ON, 2-7 ON, 3-6 OFF). Also, the I²C Clock (SCL) and data Line (SDA) jumpers J42 and J45 pins 1-2 should be shorted on the DVK board. In the Control Center, select the FX3 device. Next, choose Program > FX3 > I2C E2PROM, as shown in . This causes a special I²C boot firmware to be programmed into the FX3 device, which then enables programming of the I²C device connected to FX3. Now the FX3 device re-enumerates as "Cypress USB BootProgrammer," as shown in .

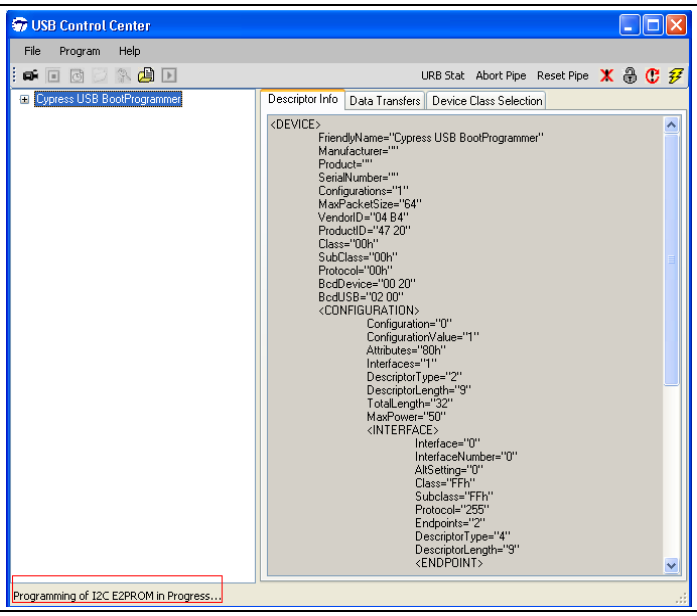




5. After the FX3 DVK board enumerates as “Cypress USB BootProgrammer,” the Control Center application prompts you to select the firmware binary to download. Browse to the *.img* file that is to be programmed into the I²C EEPROM, as shown in .



After programming is complete, the bottom left corner of the window displays “Programming of I2C EEPROM Succeeded,” as shown in .

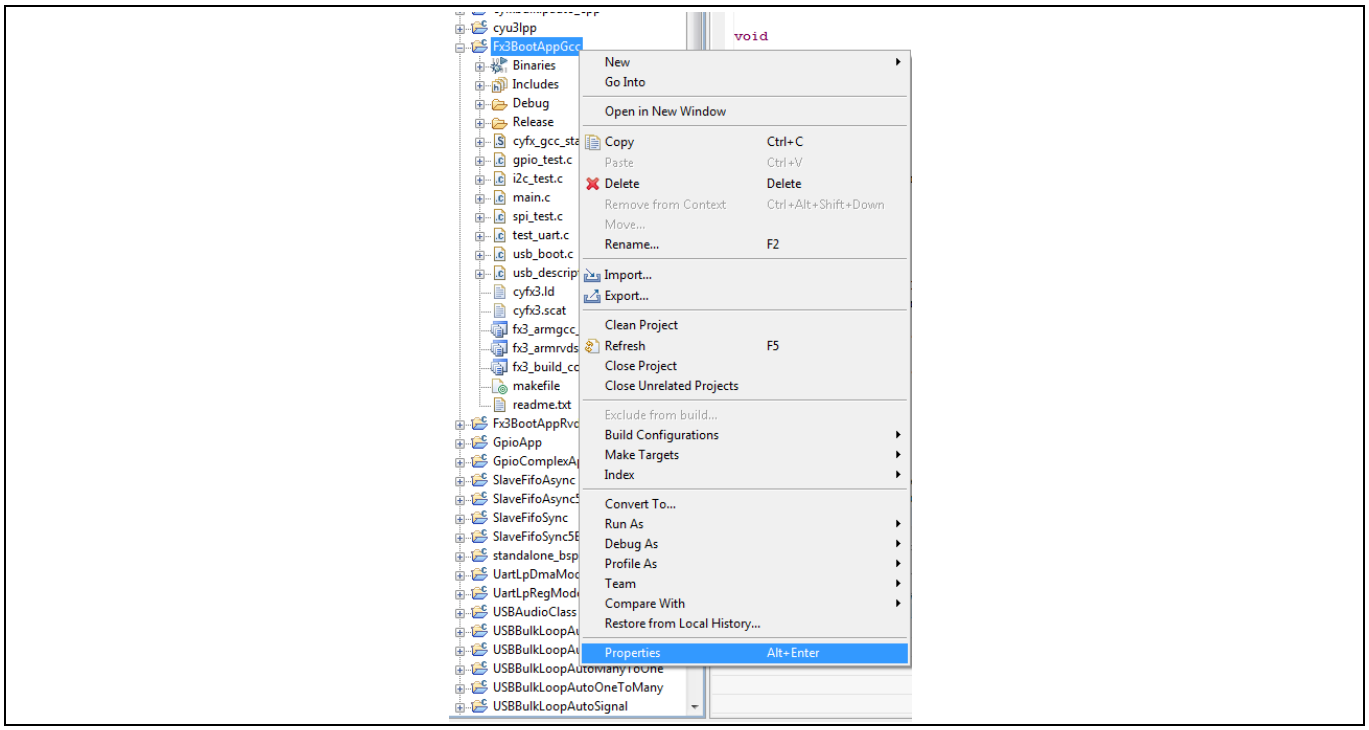


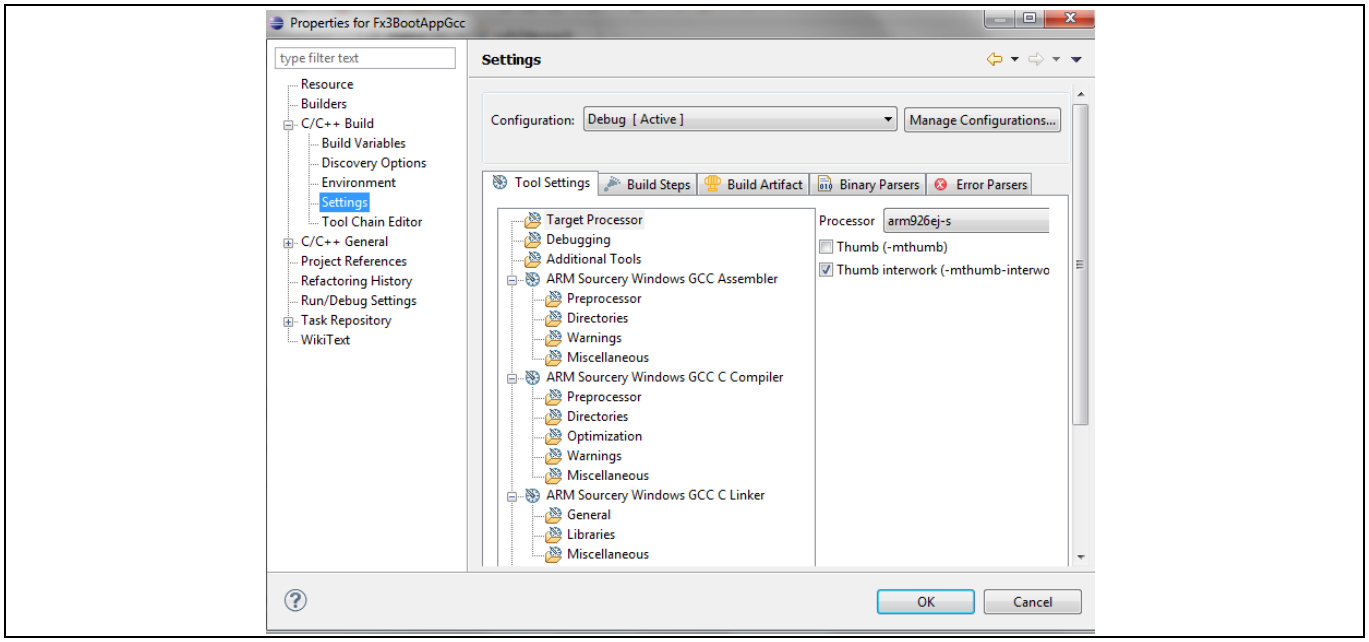
6. Change the PMODE pins on the DVK board to Z1Z to enable I²C boot. On the DVK board, this is done by configuring the jumpers and switches as shown in .

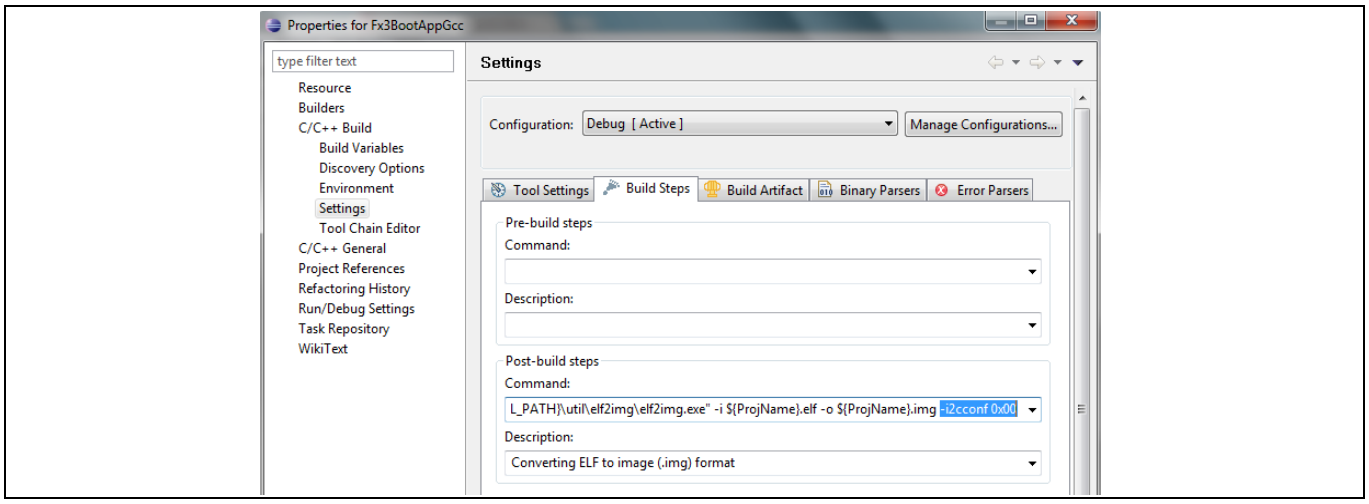
J96 (PMODE0)	Open	PMODE0 Floats
J97 (PMODE1)	2-3 Closed	PMODE1 controlled by SW25
J98 (PMODE2)	Open	PMODE2 Floats
SW25.1-8 (PMODE0)	Don't care	PMODE0 Floats
SW25.2-7 (PMODE1)	Open (OFF position)	PMODE1 = 1
SW25.3-6 (PMODE2)	Don't care	PMODE2 Floats

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

1. Build the firmware image in the Eclipse IDE as shown in , , and .



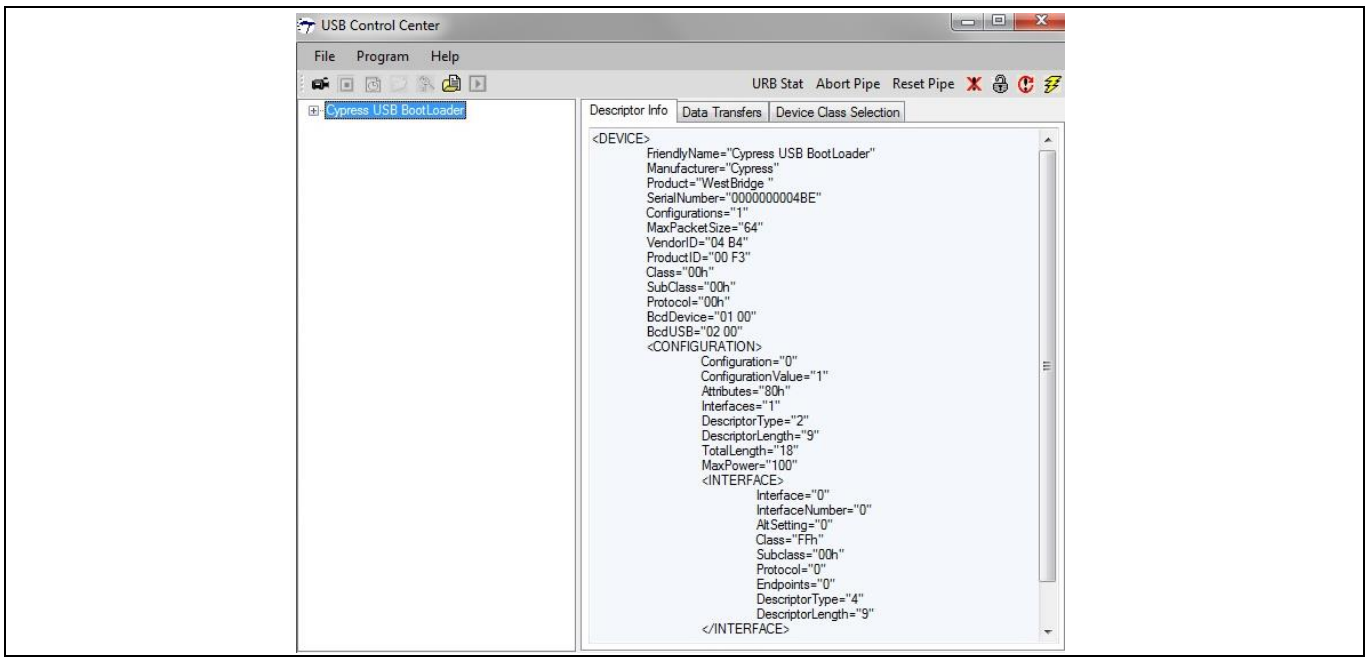




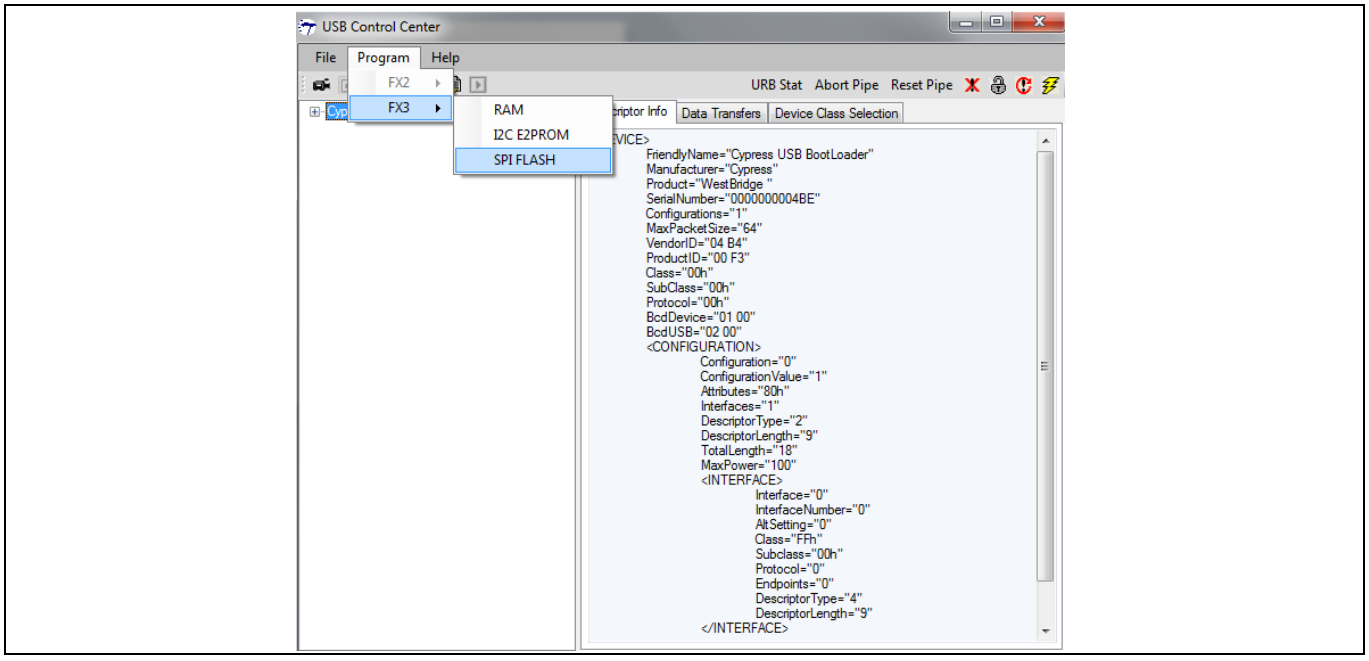
2. Enable USB boot by setting the PMODE[2:0] pins to Z11. On the DVK board, this is done by configuring the jumpers and switches as shown in .

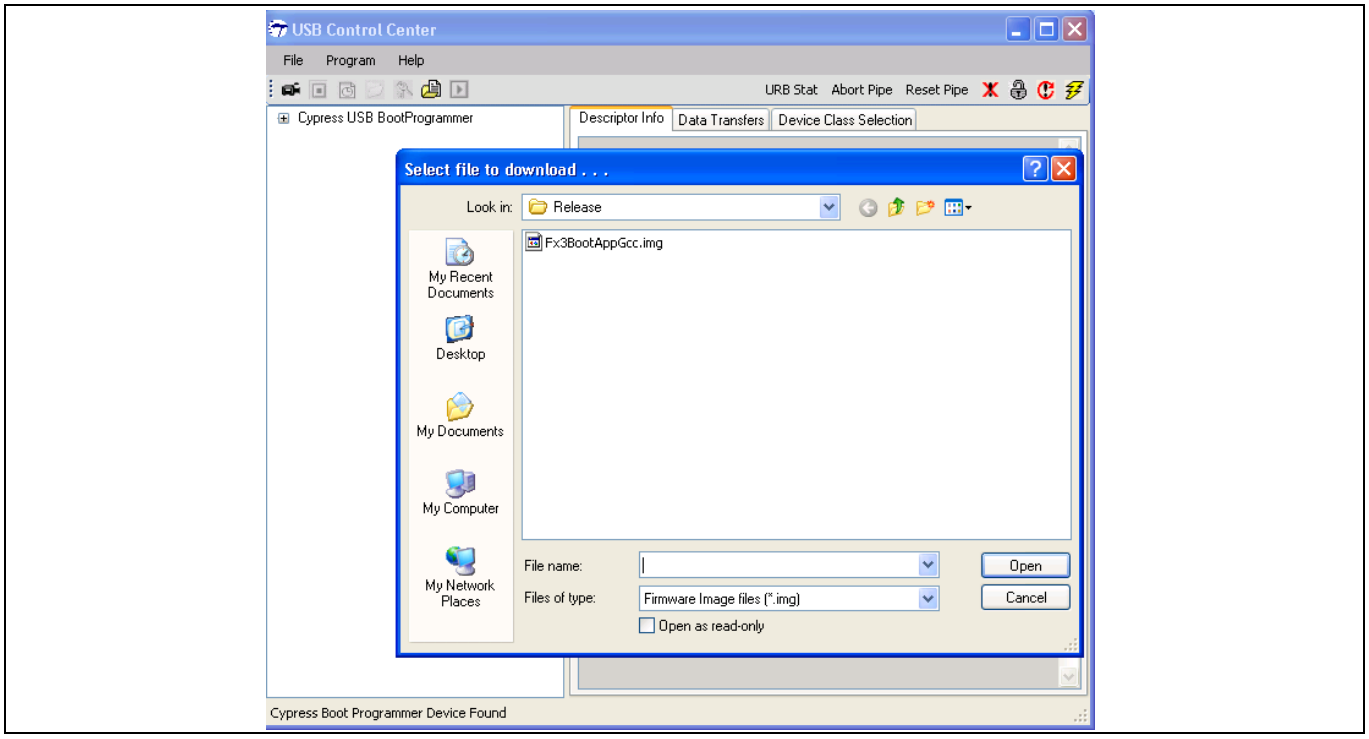
J96 (PMODE0)	2-3 Closed	PMODE0 controlled by SW25
J97 (PMODE1)	2-3 Closed	PMODE1 controlled by SW25
J98 (PMODE2)	Open	PMODE2 Floats
SW25.1-8 (PMODE0)	Open (OFF position)	PMODE0 = 1
SW25.2-7 (PMODE1)	Open (OFF position)	PMODE1 = 1
SW25.3-6 (PMODE2)	Don’t care	PMODE2 Floats

3. When connected to a USB Host, the FX3 device enumerates in the Control Center as “Cypress USB BootLoader, as shown in .

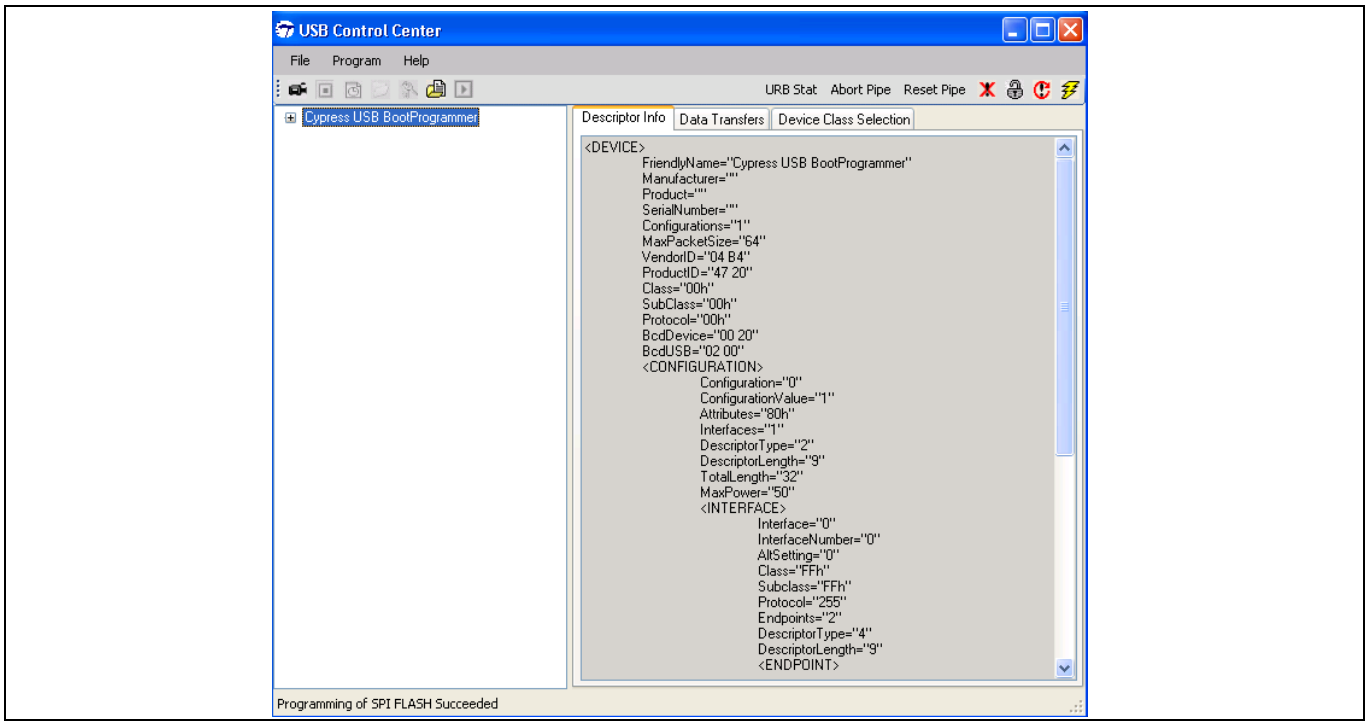


4. In the Control Center, select the FX3 device and then choose > > , as shown in . Browse to the *.img* file to be programmed into the SPI flash, as shown in .





5. After programming is complete, the bottom left corner of the window displays “Programming of SPI FLASH Succeeded,” as shown in .



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

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

Reset the DVK. Now the FX3 boots from the SPI flash.

This appendix describes the step-wise instructions on how to test and debug various sequences for sync ADmux boot.

Bytes 8 to 511 can contain random data

2. Read back the status of the write operation:
   a) Wait until bit 2 (Socket 2 Available) of PP_SOCK_STAT_L register (0x9E) is set.
   b) Write 0x0102 to the PP_DMA_XFER register (0x8E).
   c) Wait until Bit 12 of the PP_DMA_XFER register is set.
   d) Read the PP_DMA_SIZE register (0x8F) and verify that the value is 0x0200.
   e) Read 512[4] bytes of data (256 cycles) from the FX3 address 0x02.
   f) Verify that the first four bytes contain the pattern 0x57, 0x42, 0x01 (don't care), and 0x00.
3. Initiate a FIFO read command to read the data from address 0x40003000:
   a) Wait until bit 0 of PP_SOCK_STAT_L register (0x9E) is set.
   b) Write 0x0300 to the PP_DMA_XFER register (0x8E).
   c) Wait until Bit 12 and Bit 15 of the PP_DMA_XFER register are set.
   d) Write 512 bytes of data with the following format to the FX3 device address 0 (SOCKET 0):

      Byte 0 = 0x43

      Byte 1 = 0x59

      Byte 2 = 0x03 (read command)

      Byte 3 = 0x7E

      Byte 4 = 0x00 (LSB of address)

      Byte 5 = 0x30

      Byte 6 = 0x00

      Byte 7 = 0x40 (MSB of address)

      Bytes 8 to 511 are don't cares.

4. Read back the memory data from socket 2:
   a) Wait until bit 2 of PP_SOCK_STAT_L register (0x9E) is set.
   b) Write 0x0102 to the PP_DMA_XFER register (0x8E).
   c) Wait until Bit 12 of the PP_DMA_XFER register is set.
   d) Read the PP_DMA_SIZE register (0x8F) and verify that the value is 0x0200.
   e) Read 512 bytes of data (256 cycles) from FX3 address 0x02.
   f) Verify that the first 4 bytes contain the pattern 0x57, 0x42, 0x03, 0x00.
   g) Verify that bytes 8 to 511 match the random data written in step 1 above.
5. Repeat steps 1 to 4 for other memory addresses and data patterns.

If all of the above checks are good, you can proceed with the firmware download testing. The following sequence is to be used for firmware download[5]:

1. Read the content of the img file with the target firmware into a memory buffer. Pad the data to a multiple of 512 bytes as required. This is because bootloader is designed to support only full packets. The size of a full packet is specified in the bLenStatus field.
2. Follow step 1 in section to write the following data to socket 0: 0x43, 0x59, 0x02, 0x01, ... (remaining 508 bytes are don't care).
3. Follow step 2 in section to read the firmware download command status from socket 2. Verify that byte 3 (status) has the value 0x00.
4. Now, write the complete firmware content to socket 1, 512 bytes at a time. Follow the procedure given below to write each 512 bytes to socket 1.

   a) Wait until bit 1 of the PP_SOCK_STAT_L register (0x9E) is set.

   b) Write 0x0301 to the PP_DMA_XFER register (0x8E).

   c) Wait until Bit 12 and Bit 15 of the PP_DMA_XFER register are set.

   d) Write 512 bytes of data to the FX3 device address 1.

5 The steps mentioned in this section are based on bLenStatus=1 (single 512-byte block). If the bLenStatus is greater than 1, the data chunk size per transfer mentioned in steps 1-4 must be changed accordingly. Note that 512 bytes < Data chunk size per transfer < 8 KB.

This appendix describes how to use the elf2img utility (in the *util\elf2img* folder in the SDK installation path) to generate the firmware image for boot options mentioned in this application note.

The utility is a console application that needs to be invoked with the following options:

```
elf2img.exe -i <elf filename> -o <image filename> [-i2cconf <eeprom control>]
[-vectorload <vecload>] [-imgtype <image type>] [-v] [-h]
```

where,

`<elf filename>`: Input ELF file name with path

`<image filename>`: Output file name with path

`<eeprom control>`: I2C/SPI EEPROM control word in hexadecimal form

`<image type>`: Image type byte in hexadecimal form

`-v`: Enable verbose logs during the conversion process

`-h`: Print help information

The `<image type>` should be 0xB0 for all firmware applications. Other values are reserved.

The ARM926EJ-S core on the FX3 device has its reset and interrupt vectors stored in the first 256 bytes of the memory (address range 0x00–0x100). It is not advisable to load any code directly into this address range because it may interfere with the boot loader or active firmware operation. The FX3 firmware library and default linker settings ensure that no valid code is loaded directly into this address range. The interrupt vectors are safely copied into this area once the firmware starts running.

The elf2img utility in default mode removes any data in the 0x00–0x100 address range while generating the boot image. This is safe because the recommended linker settings ensure that no valid code/data is placed in this address range. This behavior can be overridden using the `-vectorload` command line option.

The `<vecload>` value is a yes/no string, which when set to "yes" causes the tool to retain any data in this address range in the boot image. The default value for this parameter is "no".

This parameter is only applicable in the case of boot from I2C EEPROM or SPI FLASH. If the FX3 is being booted via USB or the GPIF port, this field is not used and can be omitted while generating the img file.

In the case of I2C boot, the `<eeprom control>` byte specifies the type and speed of the EEPROM used.

In the case of SPI boot, the `<eeprom control>` byte specifies the speed at which the SPI boot from EEPROM should be performed. This byte is a don't care when any other boot mode is used.

The encoding in the case of I2C boot is as follows:

Bits 3 - 1	EEPROM size [7 = 128 KB, 6 = 64 KB, 5 = 32 KB, 4 = 16 KB, 3 = 8 KB, 2 = 4 KB]
Bits 5 - 4	EEPROM speed [0 = 100 KHz, 1 = 400 KHz, 2 = 1 MHz]
Bits 7 - 6	Must be zero

For example, a value of 0x1C corresponds to the use of 64 KB EEPROM at a frequency of 400 kHz.

The encoding in the case of SPI boot is as follows:

Bits 3 - 1	Don't care
Bits 5 - 4	SPI operating frequency [0 = 10 MHz, 1 = 20 MHz, 2 = 30 MHz]
Bits 7 - 6	Must be zero.

For example, a value of 0x1C will generate .img for SPI operating frequency of 20 MHz.

[1] - Getting started with EZ-USB™ FX3

[2]

**	2012-05-14	New application note
*A	2012-11-19	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	2012-12-10	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. Updated to new template.
*C	2013-04-12	Updated GPIO[55] in .
*D	2014-06-27	Added to show all the boot options discussed in this application note. Added pin mapping for I2C, SPI, and sync ADMux interfaces. Added command set of supported SPI flashes. Added the Processor Port register map. Pointed to for sync ADMux timing diagrams.
*E	2015-07-28	Added SPI flash part numbers supported by FX3 Updated the I2C EEPROM part number that is in production Added more information in Sync ADMux boot options Added read and write waveforms for Sync ADMux boot Updated GPIO[45] and GPIO[50] in Table 30. Corrected the RDY pin mapping for Sync ADMux. Removed secure boot (0xB1) format Added eMMC boot details Added FX3S and CX3 parts Changed the AN title by including FX3S Updated to new template.
*F	2017-01-20	Updated to new template. Added Appendix for ADMux troubleshooting. Completing Sunset Review.
*G	2017-04-18	Updated logo and copyright
*H	2018-07-24	Removed KBA Link (Design with FX3/FX3S) Removed Benicia References Removed Obsolete app note references Modified SPI flash limit to 128 Mbit Added supported SPI Flash parts Removed Ez-detect section Added Appendix C to provide more details on the elf2img utility

*I	2018-10-10	Updated Appendix B.
*J	2019-01-08	Updated GPIF II API Protocol section Updated Appendix B.
*K	2021-04-16	Removed mentions of asynchronous SRAM boot option Updated to Infineon template
*L	2021-12-10	Updated table 1. Deleted section 10. eMMc boot. Deleted note from section 1. Updated section 2 and 3. Added section 7.3. Added note to section 7.4.