restricted

EZ-USB™ FX2G3 SDK user guide

About this document

Scope and purpose

The EZ-USB™ FX2G3 SDK user guide provides comprehensive instructions and guidelines for effectively utilizing the FX2G3 family of USB device controllers.

Intended audience

This guide is intended for software developers, firmware engineers, and system integrators in the biometrics, scanner, camera, video, and imaging markets.

- Software developers: Professionals developing applications and firmware for the FX2G3 device controllers.
- Firmware engineers: Engineers responsible for writing and maintaining firmware for FX2G3-based systems.
- System integrators: Experts involved in integrating FX2G3 controllers into larger systems and ensuring compatibility.
- Technical project managers: Managers overseeing projects that involve the development and integration of FX2G3 controllers.

Table of contents

Table of contents

1 Introduction

The EZ-USB™ FX2G3 family of USB2 device controllers is designed to meet the demands of established USB2 applications in biometrics, scanners, cameras, video, and imaging markets. FX2G3 leverages Cortex®-M4 (CM4) and Cortex® M0+ (CM0) microcontrollers, providing robust features such as 512 KB flash memory, 128 KB SRAM, and 128 KB ROM.

The controllers also include serial communication blocks (SCB), a crypto engine for enhanced security, and a high-bandwidth data subsystem that facilitates DMA data transfers from LVCMOS input to USB output at speeds up to 480 Mbps, suitable for USB Hi-Speed based host systems. A 1024 KB SRAM is integrated within the high-bandwidth data subsystem to provide ample buffering for data.

1.1 Software prerequisites

Table 1 Prerequisites

#	Software	Version
1	ModusToolbox™	3.2 or later
2	Microsoft Visual C++ Redistributable	2015-2022

1.2 EZ-USB™ FX2G3 SDK components

The SDK includes reference projects for target applications that help you jump-start the process of developing EZ-USB™ FX2G3 controller applications. A comprehensive overview of each code example is provided in the corresponding *README.md* file.

- mtb-example-fx2g3-hello-world: Simple application which toggles GPIOs and outputs messages through a UART transmit pin periodically.
- mtb-example-fx2g3-usbhs-device: Implements a vendor-specific USB echo device which allows USB data transfers to be tested using a device-based loopback function or data source sink function. The EZ-USB™ FX Control Center GUI can be used to test out the loopback as well as data source sink functionality.
- mtb-example-fx2g3-slave-fifo-in: Implements the Infineon standard Synchronous Slave FIFO IN protocol over an LVCMOS interface and allows data transfer between an FPGA connected to the LVCMOS port and the USB host
- mtb-example-fx2g3-slave-fifo-out: Implements the Infineon standard Synchronous Slave FIFO OUT protocol over an LVCMOS interface and allows data transfer between the USB host and an FPGA connected to the LVCMOS port
- mtb-example-fx2g3-uvc-uac: Implements USB Video Class (UVC) and integrated USB Audio Class (UAC) functions. The UVC function streams the video data received over the LVCMOS interface from an FPGA. The UAC function streams audio data received from PDM microphones.
- mtb-example-fx2g3-flash-loader: Implements USB Vendor Class. The USB function is intended to program the FPGA bit file to the external SPI flash on the KIT_FX2G3_104LGA DVK using the EZ-USB™ FX Control Center GUI.

See the *README.md* file in each of these application folders for details of the application functionality.

1.3 Middleware libraries

The SDK uses the following middleware libraries to implement the key features supported by FX2G3 applications.

1. usbfxstack Middleware Library

   The usbfxstack Middleware Library has the following sub-components:

   - **FreeRTOS operating system:** Source files for FreeRTOS.
   - **LVDS/LVCMOS IP driver:** Driver for the Sensor Interface Port (SIP) on FX2G3, which can be configured to function using the LVCMOS interface.
   - **USB Stack:** An integrated USB stack that supports the Hi-Speed USB functionality on the FX2G3 device.
   - **DMA Manager:** Provides a generic DMA channel abstraction for the DMA resources that move data across the USB and LVCMOS interfaces on the FX2G3 device.
   - **Utility functions:** A set of utility functions that provide various functionalities including logging support, abstraction for USB data transfers, and fault handlers.

   All these functions are compiled into a common library, which can be linked with the application sources to create the firmware binaries.

2. mtb-pdl-cat1 Peripheral Driver Library

   The mtb-pdl-cat1 Peripheral Driver Library (PDL) simplifies the software development for EZ-USB™ FX devices. The PDL integrates device header files, startup code, and peripheral drivers into a single package. It is the driver library for the clock, GPIO, serial peripherals, TCPWM, USB Full-Speed, QSPI, PDM, and I2S interfaces on the EZ-USB™ FX2G3 device.

1.4 SDK dependencies

1.4.1 ModusToolbox™

ModusToolbox™ software is a collection of easy-to-use libraries and tools enabling rapid development with Infineon MCUs for applications ranging from wireless and cloud-connected systems, edge AI/ML, embedded sense and control, to wired USB connectivity using PSOC™ Industrial/IoT MCUs, AIROC™ Wi-Fi and Bluetooth® connectivity devices, XMC™ Industrial MCUs, and EZ-USB™/EZ-PD™ wired connectivity controllers.

The Eclipse IDE that comes as part of ModusToolbox™ is used to configure EZ-USB™ FX2G3 devices to develop and compile the firmware applications. This SDK version requires ModusToolbox™ v3.2 or higher.

ModusToolbox™ include configuration tools, low-level drivers, libraries, and operating system support, most of which are compatible with Linux, macOS, and Windows-hosted environments. For more details, see Getting Started with ModusToolbox™.

1.4.2 EZ-USB™ FX Control Center

The EZ-USB™ FX Control Center software helps to fetch the USB device information, perform data transfer, or data loopback over USB endpoints, and measure the USB data transfer performance. It also supports programming internal flash and external SPI flash memories. For more details, see the EZ-USB™ FX Control Center user guide in the Help section of the application.

1.5 Hardware dependencies

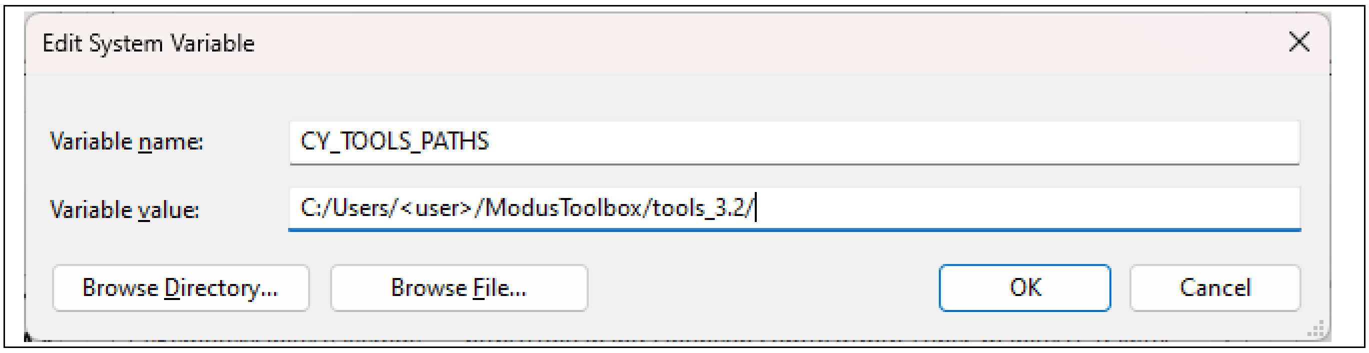
The KIT_FX2G3_104LGA DVK board can be used to evaluate the EZ-USB™ FX2G3 solutions. For more details on the kit, refer to KIT_FX2G3_104LGA DVK user guide.

2 Getting started with EZ-USB™ FX2G3 SDK

2.1 Environment updates

The ModusToolbox™ tools are located at *<install_path>/ModusToolbox/tools_MAJOR.MINOR*.

If multiple versions of ModusToolbox™ software are installed on the PC, multiple tools will be present in the *<install_path>/ModusToolbox/* location. Therefore, use the `CY_TOOLS_PATHS` system variable to have the required set of tools. On Windows, open the **Environment Variables** dialog, and create a new System/User Variable (see Figure 1).


Edit System Variable

Variable name: CY_TOOLS_PATHS

Variable value: C:/Users/<user>/ModusToolbox/tools_3.2/

Figure 1 Environment variable updates

*Note: For **Variable value**, use a Windows style path (that is, not like /cygdrive/c/). For example, C:/Users/<user>/ModusToolbox/tools_3.2/*

Use the correct method for setting variables in macOS and Linux for your system.

2.2 Using the reference projects

Each reference project provided with this SDK can be imported into Eclipse IDE for ModusToolbox™, customized as per the user requirements and compiled. This section describes steps for project creation, compiling, programming, and debugging for EZ-USB™ FX2G3 solution.

Note: These projects are designed to work with the specific FX2G3 devices only. Changing the target device using the library manager can fail the firmware build.

2.3 Create the project in Eclipse IDE for ModusToolbox™

1. Launch the Eclipse IDE for ModusToolbox™ software and select the workspace, as shown in Figure 2.

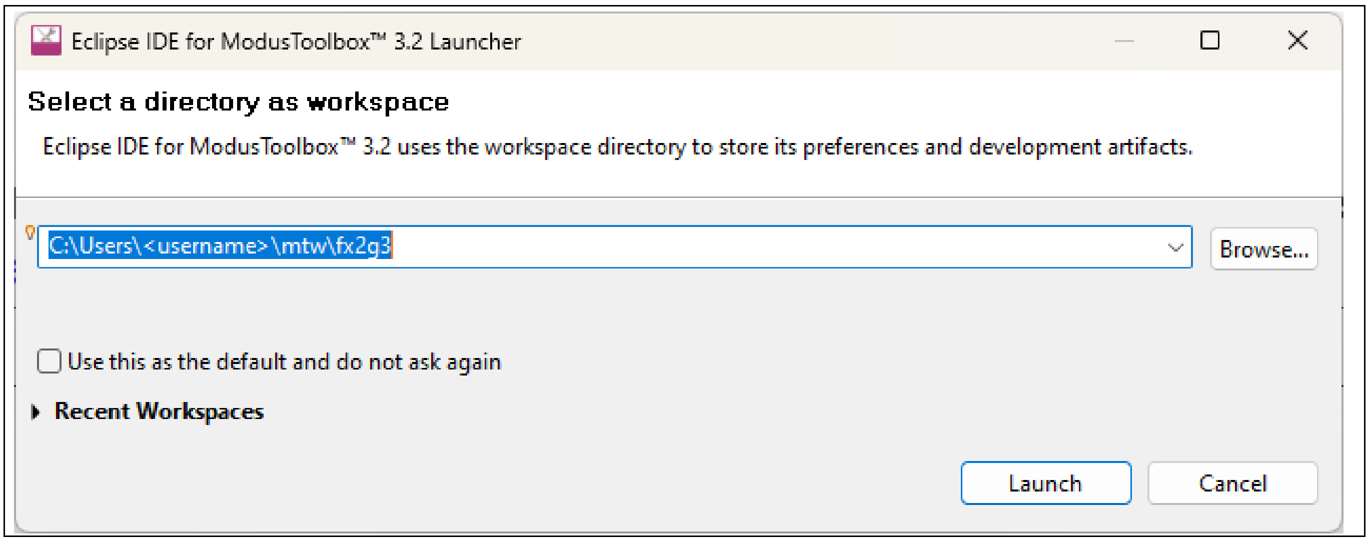

Eclipse IDE for ModusToolbox™ 3.2 Launcher
Select a directory as workspace
Eclipse IDE for ModusToolbox™ 3.2 uses the workspace directory to store its preferences and development artifacts.
C:\Users\<username>\mtw\fx2g3
Browse...
Use this as the default and do not ask again
Recent Workspaces
Launch
Cancel

Figure 2 **Launching Eclipse IDE for ModusToolbox™ software**

2. Click **New Application** in the **Quick Panel** (or use **File** > **New** > **ModusToolbox™ Application**), as shown in Figure 3. This launches the Project Creator tool.

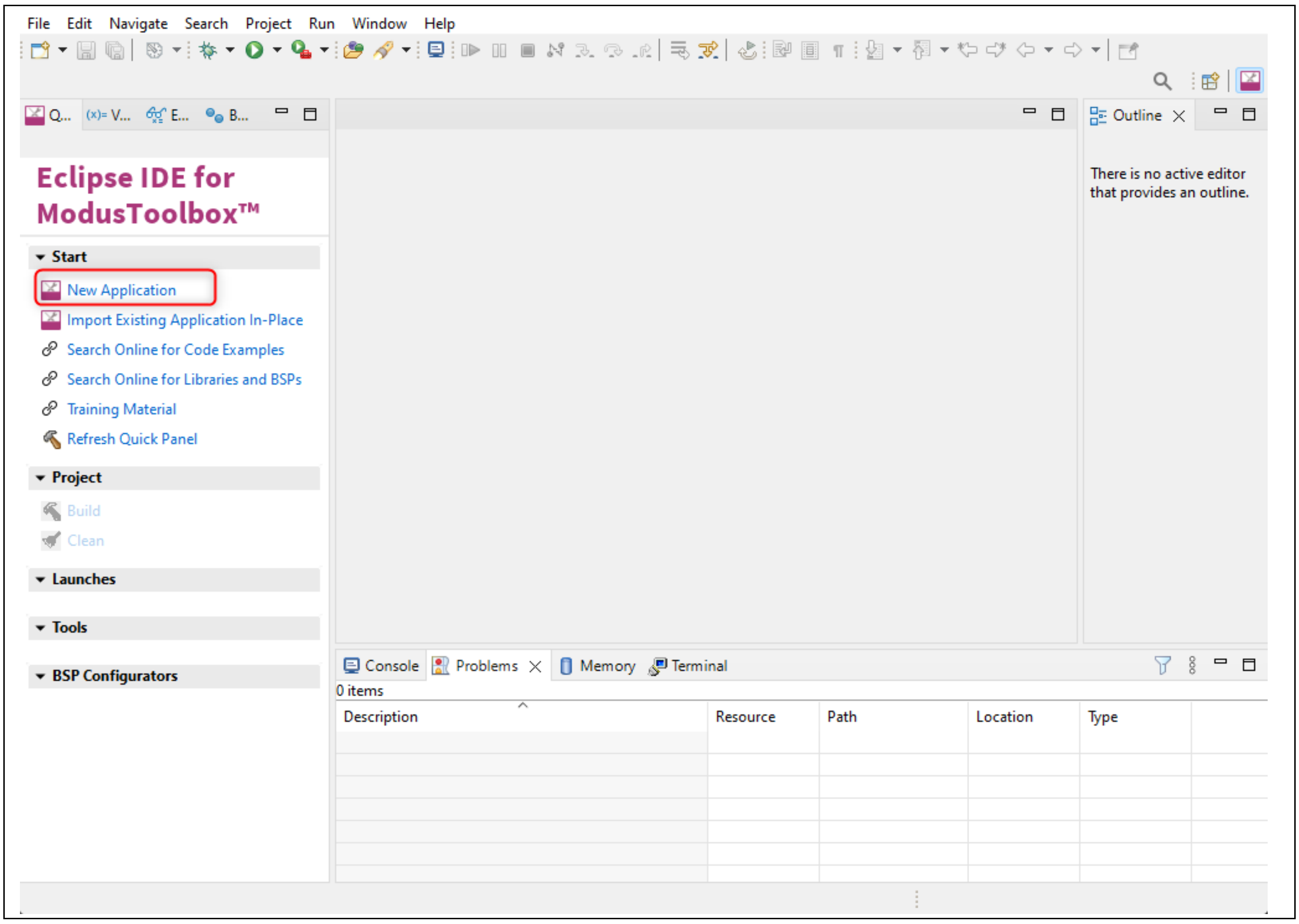
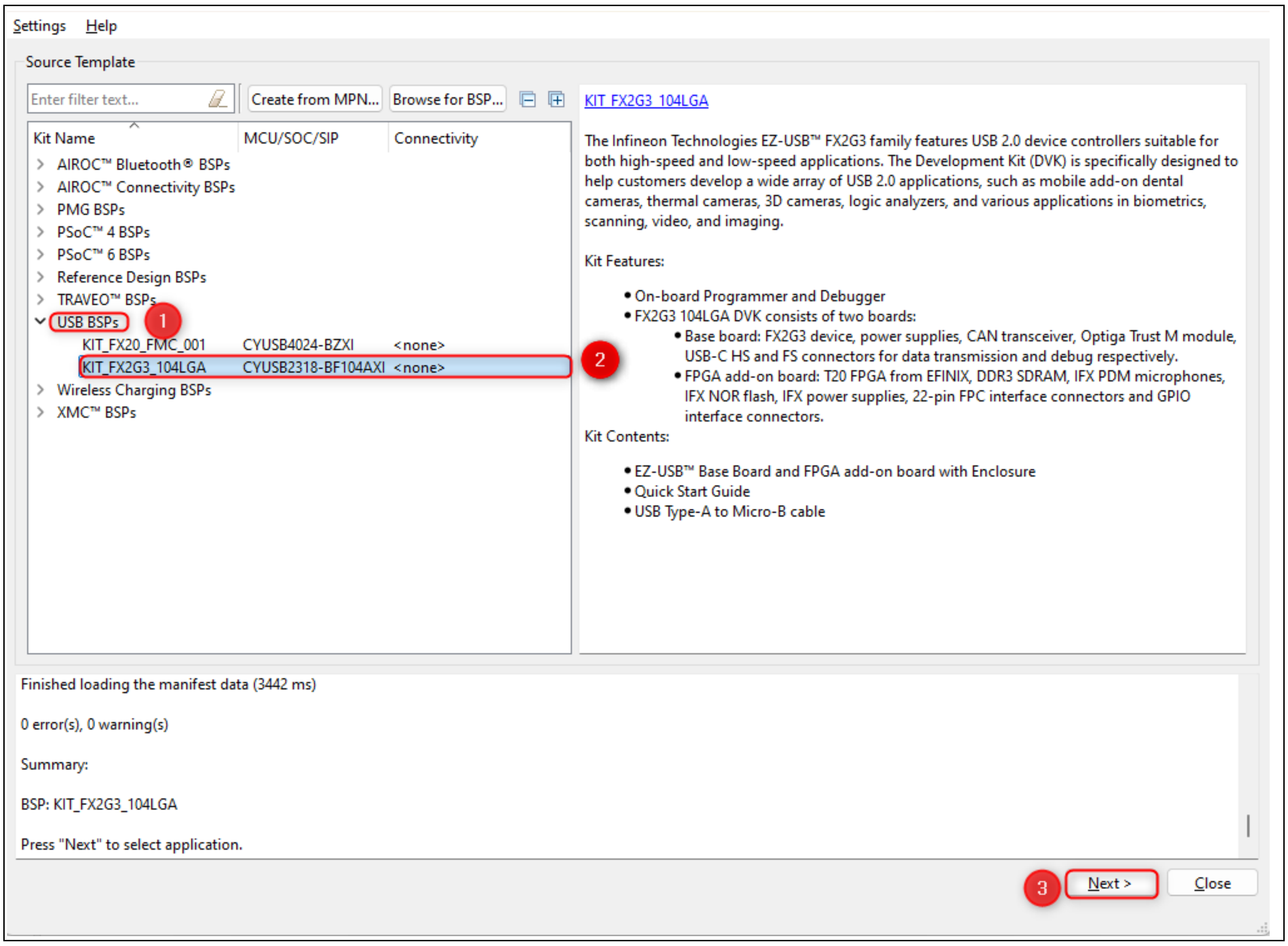

File Edit Navigate Search Project Run Window Help
Eclipse IDE for ModusToolbox™
Start
New Application
Import Existing Application In-Place
Search Online for Code Examples
Search Online for Libraries and BSPs
Training Material
Refresh Quick Panel
Project
Build
Clean
Launches
Tools
BSP Configurators
Outline
There is no active editor that provides an outline.
Console
Problems
Memory
Terminal
0 items
Description
Resource
Path
Location
Type

Figure 3 **Creating New Application**

3. In **Choose Board Support Package (BSP) – Project Creator 2.X** window, do the following:
   1) Expand the **USB BSPs** drop-down.
   2) Select **KIT_FX2G3_104LGA**.
   3) Click **Next**.



KIT_FX2G3_104LGA

The Infineon Technologies EZ-USB™ FX2G3 family features USB 2.0 device controllers suitable for both high-speed and low-speed applications. The Development Kit (DVK) is specifically designed to help customers develop a wide array of USB 2.0 applications, such as mobile add-on dental cameras, thermal cameras, 3D cameras, logic analyzers, and various applications in biometrics, scanning, video, and imaging.

Kit Features:

- On-board Programmer and Debugger
- FX2G3 104LGA DVK consists of two boards:
  - Base board: FX2G3 device, power supplies, CAN transceiver, Optiga Trust M module, USB-C HS and FS connectors for data transmission and debug respectively.
  - FPGA add-on board: T20 FPGA from EFINIX, DDR3 SDRAM, IFX PDM microphones, IFX NOR flash, IFX power supplies, 22-pin FPC interface connectors and GPIO interface connectors.

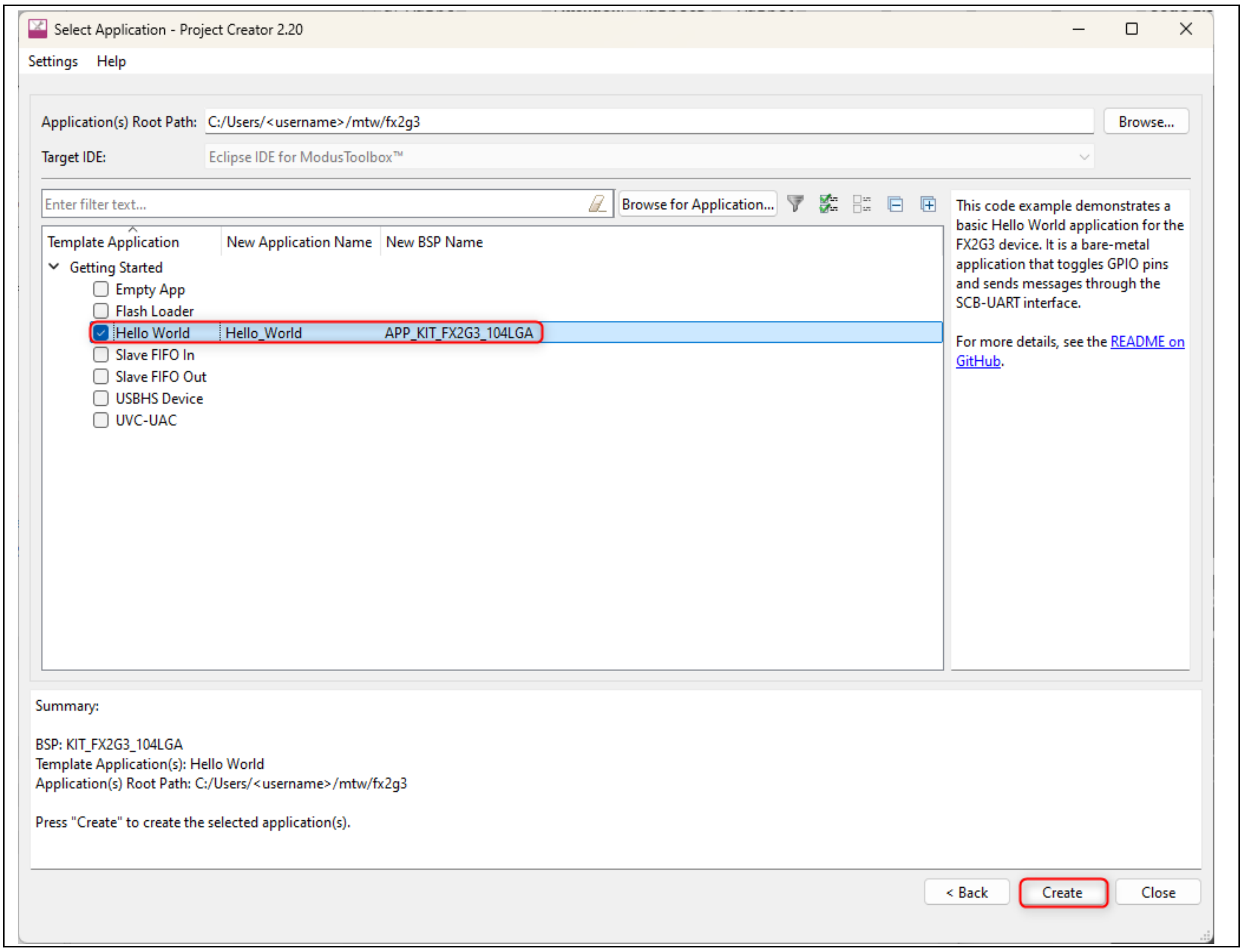
Kit Contents:

- EZ-USB™ Base Board and FPGA add-on board with Enclosure
- Quick Start Guide
- USB Type-A to Micro-B cable

```
Finished loading the manifest data (3442 ms)

0 error(s), 0 warning(s)

Summary:

BSP: KIT_FX2G3_104LGA

Press "Next" to select application.
```

Figure 4 Selection of BSP

4. To create an application, choose the desired template from the list of available applications shown in the Figure 5, and then click on **Create.**



Application(s) Root Path: C:/Users/<username>/mtw/fx2g3
Target IDE: Eclipse IDE for ModusToolbox™

Template Application	New Application Name	New BSP Name
✓ Hello World	Hello_World	APP_KIT_FX2G3_104LGA

This code example demonstrates a basic Hello World application for the FX2G3 device. It is a bare-metal application that toggles GPIO pins and sends messages through the SCB-UART interface.

For more details, see the README on GitHub.

Summary:

BSP: KIT_FX2G3_104LGA
Template Application(s): Hello World
Application(s) Root Path: C:/Users/<username>/mtw/fx2g3

Press "Create" to create the selected application(s).

Figure 5 Selection of application

5. The Project Creator GUI will close automatically if the project creation is successful or if it encountered an error in between creation of the project. If the project creation was successful, the project will be available in the Eclipse IDE for ModusToolbox™ to modify, compile, build, and debug.

2.4 Compile the reference project

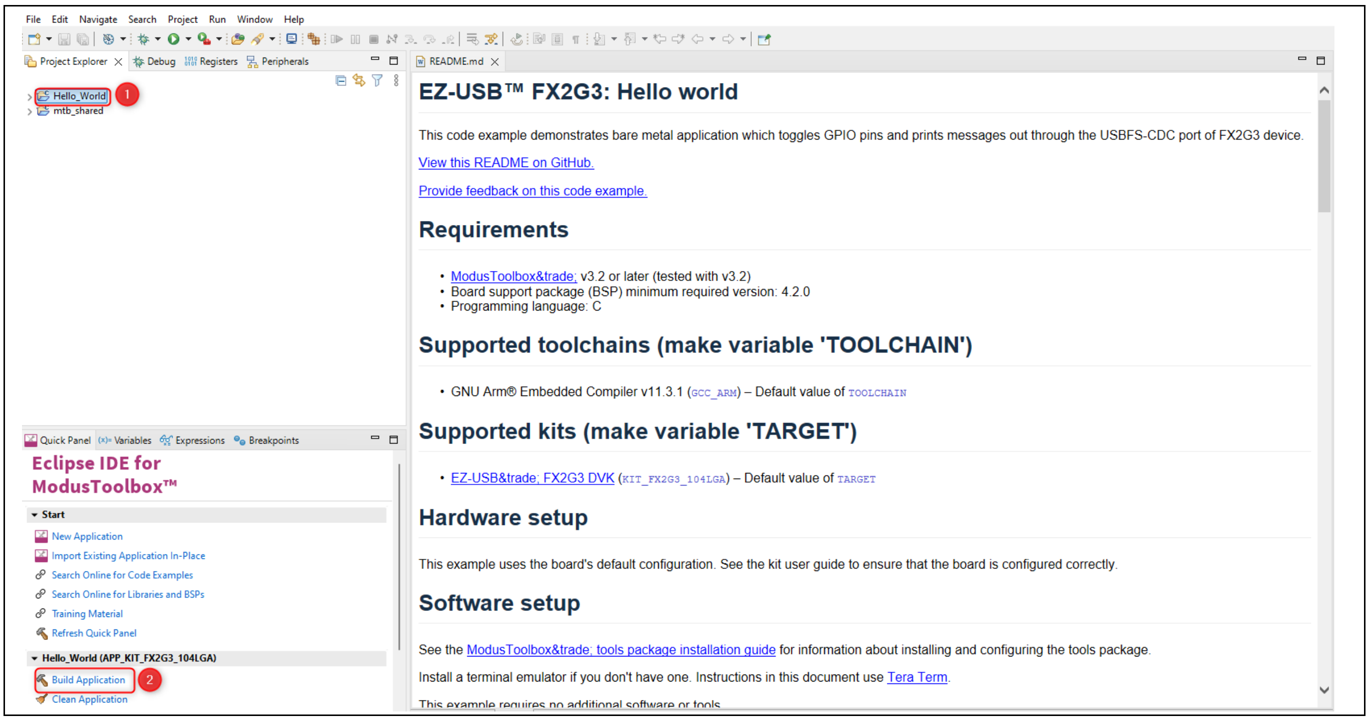
EZ-USB™ FX2G3 solutions are designed to work with a flash-based USB bootloader. For the flash memory map and for more information on firmware operation, see Figure 9.

The Makefile present in the application is designed to disable/enable the certain features during the build process.

To start the build operation, use either of the steps described in the Using Eclipse IDE for ModusToolbox™ software and Using CLI sections.

2.4.1 Using Eclipse IDE for ModusToolbox™ software

1. In **Project Explorer**, select the application (“Hello World” in this case).
2. In **Quick Panel**, select **Build Application,** as shown in Figure 6.


File Edit Navigate Search Project Run Window Help
Project Explorer
Debug
Registers
Peripherals
Hello_World
1
mtb_shared
README.md
Quick Panel
Variables
Expressions
Breakpoints
Eclipse IDE for ModusToolbox™
Start
New Application
Import Existing Application In-Place
Search Online for Code Examples
Search Online for Libraries and BSPs
Training Material
Refresh Quick Panel
Hello_World (APP_KIT_FX2G3_104LGA)
Build Application
2
Clean Application

EZ-USB™ FX2G3: Hello world

This code example demonstrates bare metal application which toggles GPIO pins and prints messages out through the USBFS-CDC port of FX2G3 device.

View this README on GitHub.

Provide feedback on this code example.

Requirements

- ModusToolbox&trade; v3.2 or later (tested with v3.2)
- Board support package (BSP) minimum required version: 4.2.0
- Programming language: C

Supported toolchains (make variable 'TOOLCHAIN')

- GNU Arm® Embedded Compiler v11.3.1 (`GCC_ARM`) – Default value of `TOOLCHAIN`

Supported kits (make variable 'TARGET')

- EZ-USB&trade; FX2G3 DVK (`KIT_FX2G3_104LGA`) – Default value of `TARGET`

Hardware setup

This example uses the board's default configuration. See the kit user guide to ensure that the board is configured correctly.

Software setup

See the ModusToolbox&trade; tools package installation guide for information about installing and configuring the tools package.

Install a terminal emulator if you don't have one. Instructions in this document use Tera Term.

This example requires no additional software or tools.

Figure 6 Building application using Eclipse IDE for ModusToolbox™ software

2.4.2 Using CLI

1. Open terminal and navigate to the application directory.
2. Run the following command:

```
make build -j8
```

2.5 Programming EZ-USB FX2G3

This section describes programming the HEX file generated by building the project to the target that is, EZ-USB™ FX Control Center. The HEX file generated will be in the following location within the application directory:

build/APP_KIT_FX2G3_104LGA/RELEASE/<APPNAME>.hex

where,

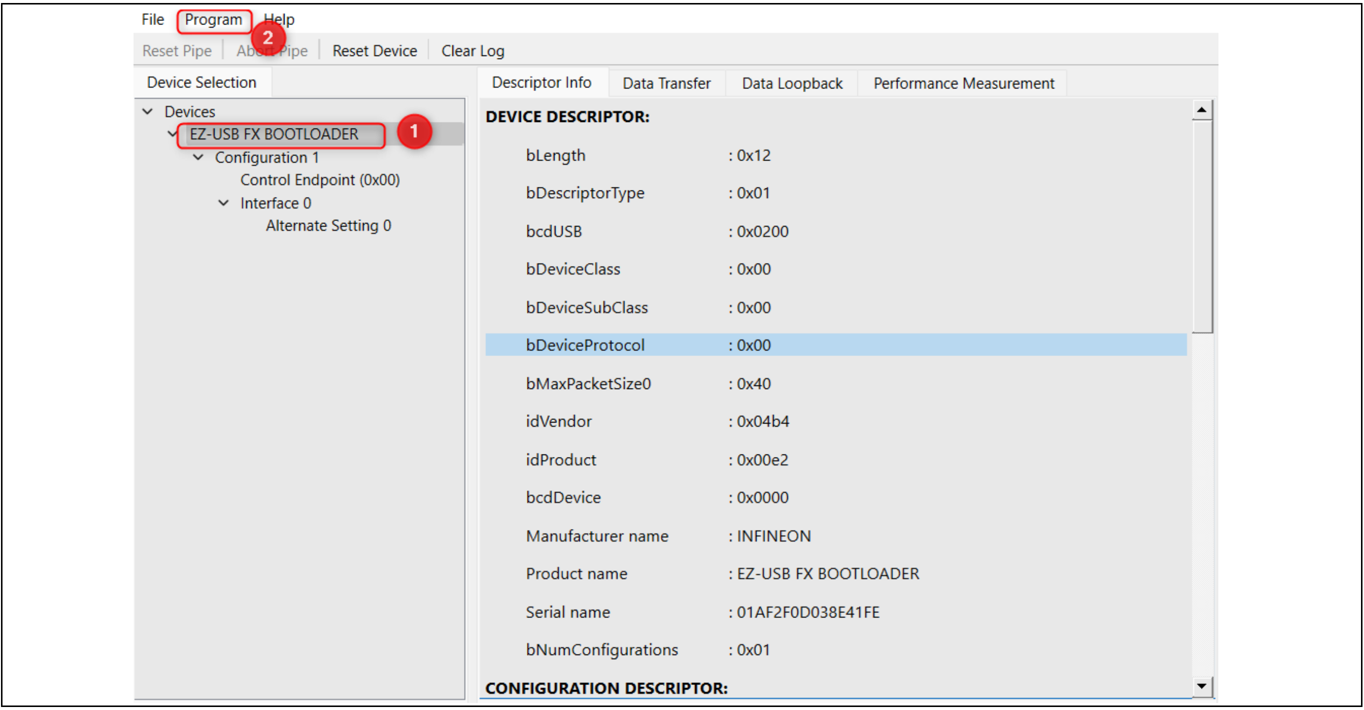
- "RELEASE" can be debug, release, or custom depending on the makefile setting.

"<APPNAME>" is the name of the application that was compiled. It is "mtb-example-fx2g3-hello-world" for this example.

2.5.1 Programming using EZ-USB™ FX Control Center

The EZ-USB™ FX2G3 device comes with a pre-programmed USB bootloader. Connect *KIT_FX2G3_104LGA DVK* to the USB Host or PC. The device enumerates as a WinUSB device.

1. EZ-USB™ FX2G3 device comes up as "EZ-USB FX bootloader".
2. Select the device and click **Program** > **Internal Flash** option.

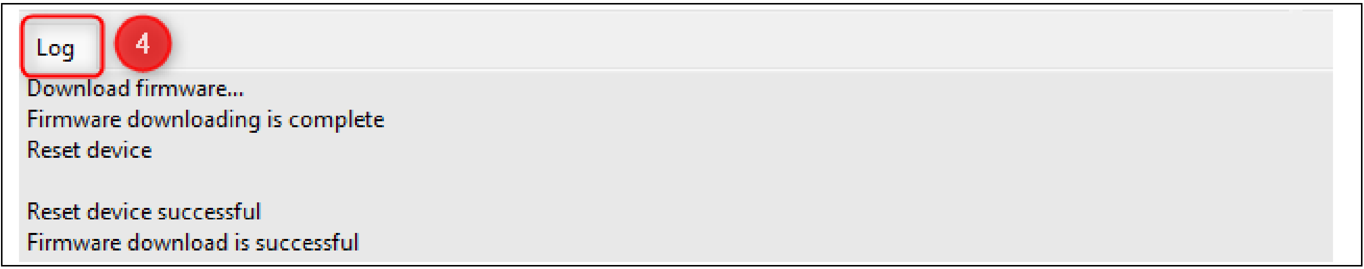


```
DEVICE DESCRIPTOR:
    bLength                 : 0x12
    bDescriptorType         : 0x01
    bcdUSB                  : 0x0200
    bDeviceClass            : 0x00
    bDeviceSubClass         : 0x00
    bDeviceProtocol         : 0x00
    bMaxPacketSize0         : 0x40
    idVendor                : 0x04b4
    idProduct               : 0x00e2
    bcdDevice               : 0x0000
    Manufacturer name       : INFINEON
    Product name            : EZ-USB FX BOOTLOADER
    Serial name             : 01AF2F0D038E41FE
    bNumConfigurations      : 0x01
CONFIGURATION DESCRIPTOR:
```

Figure 7 Programming EZ-USB™ FX2G3 using EZ-USB™ FX Control Center

1. Navigate to the application hex file folder.
2. Confirm if the programming is successful in the log window.

After the binary is programmed, the device gets reset and the new application shall start execution.



```
Download firmware...
Firmware downloading is complete
Reset device

Reset device successful
Firmware download is successful
```

Figure 8 Programming successful

3 USB bootloader

A USB-based bootloader is pre-programmed on the EZ-USB™ FX2G3 device to allow programming of custom firmware applications without having to make use of a dedicated SWD programmer.

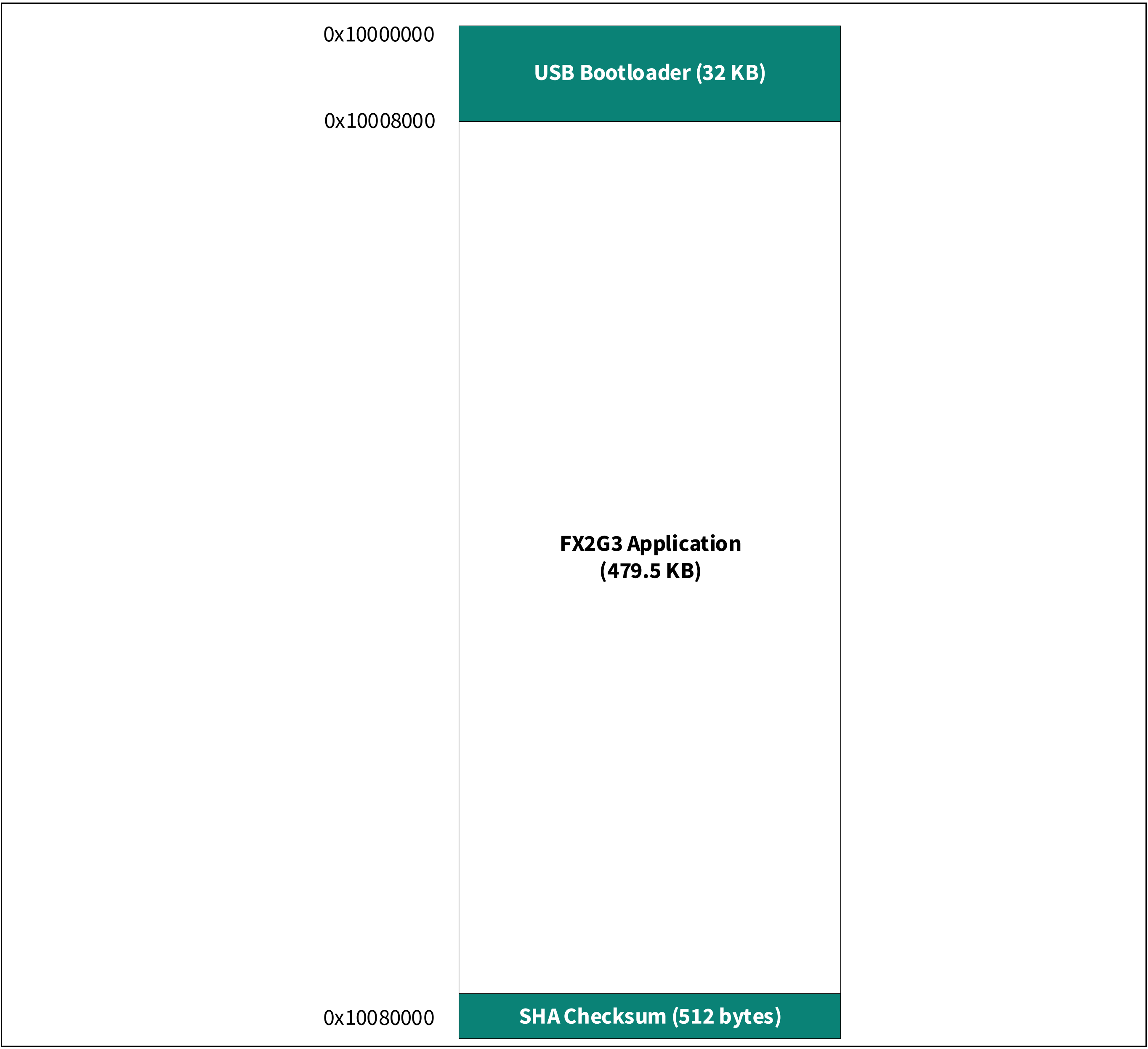


Figure 9 FX2G3 device flash memory layout

Figure 9 shows the flash memory layout on the FX2G3 device. The USB bootloader occupies the first 32 KB of the on-chip flash memory and leaves the remaining 480 KB for application-specific usage.

The last 512 bytes of the on-chip flash is expected to store an SHA-256 checksum calculated over the rest of the application-specific flash contents. The bootloader makes use of this checksum to verify the integrity of the application before transferring control to it.

If a valid application is not found in the flash, the bootloader proceeds to enumerate as a Hi-Speed USB device and enables programming the application binary onto the flash.

3.1 Firmware build for bootloader compatibility

The following changes are required in the firmware binary when it is being used along with the USB bootloader.

- Move the starting location of the firmware binary to 0x1000 8000 instead of 0x1000 0000.
- Calculate the SHA-256 checksum over the firmware binary and place it in the last 512-byte flash row.

The FX2G3 firmware build scripts support compiling each application for compatibility with the USB bootloader.

By default, the generated binary can be programmed through the USB bootloader.

3.2 Returning control to the USB bootloader

Once the firmware binary has been programmed onto the FX2G3 device flash, the bootloader will keep transferring control to the application on every subsequent reset.

If the BOOT MODE/PMODE (SW1) switch on the KIT_FX2G3_104LGA DVK is held pressed while the device is reset or power cycled, the device will stay in bootloader mode instead of booting into the application. This is done by sensing the P13.0 pin on the device and staying in Bootloader mode if the pin is HIGH.

If GPIO control is not possible, it is possible for the application to instruct the bootloader to stay in Boot mode and not transfer control on the next reset.

This is achieved by storing an 8-byte signature into a specific RAM region and initiating a soft (not power-on) reset. As the RAM content is retained across the soft reset, the bootloader will detect this signature in the RAM location and stay in Boot mode, thereby allowing a firmware update.

The Boot mode signature to be stored is shown in Table 2.

Table 2 Boot mode request passed through SRAM

SRAM location	Expected Value	String
0x0800 03C0	0x544F 4F42	“BOOT”
0x0800 03C4	0x4544 4F4D	“MODE”

3.3 FX2G3 dual-core boot flow

The EZ-USB™ FX2G3 controller incorporates a Cortex®-M0+ (CM0+) core as well as a Cortex®-M4 (CM4) core. All the applications in the SDK are designed to run on the CM4 core.

When the FX2G3 device powers up (or comes out of reset), only the CM0+ core will be running and the Cortex®-M4 is held in reset. The CM0+ core executes the ROM-based boot code before transferring control to the user application at the beginning of the on-chip flash memory (address 0x1000 0000). This can be the starting of the FX2G3 USB bootloader or that of the FX2G3 application if the USB bootloader is not being used.

This location is expected to hold a CM0+ vector table including the reset vector. The ROM-based boot code will identify the reset vector address and branch to that address to start running the bootloader (or the application).

This location is expected to hold a CM0+ vector table including the reset vector. The ROM-based boot code will identify the reset vector address and branch to that address to start running the bootloader (or the application).

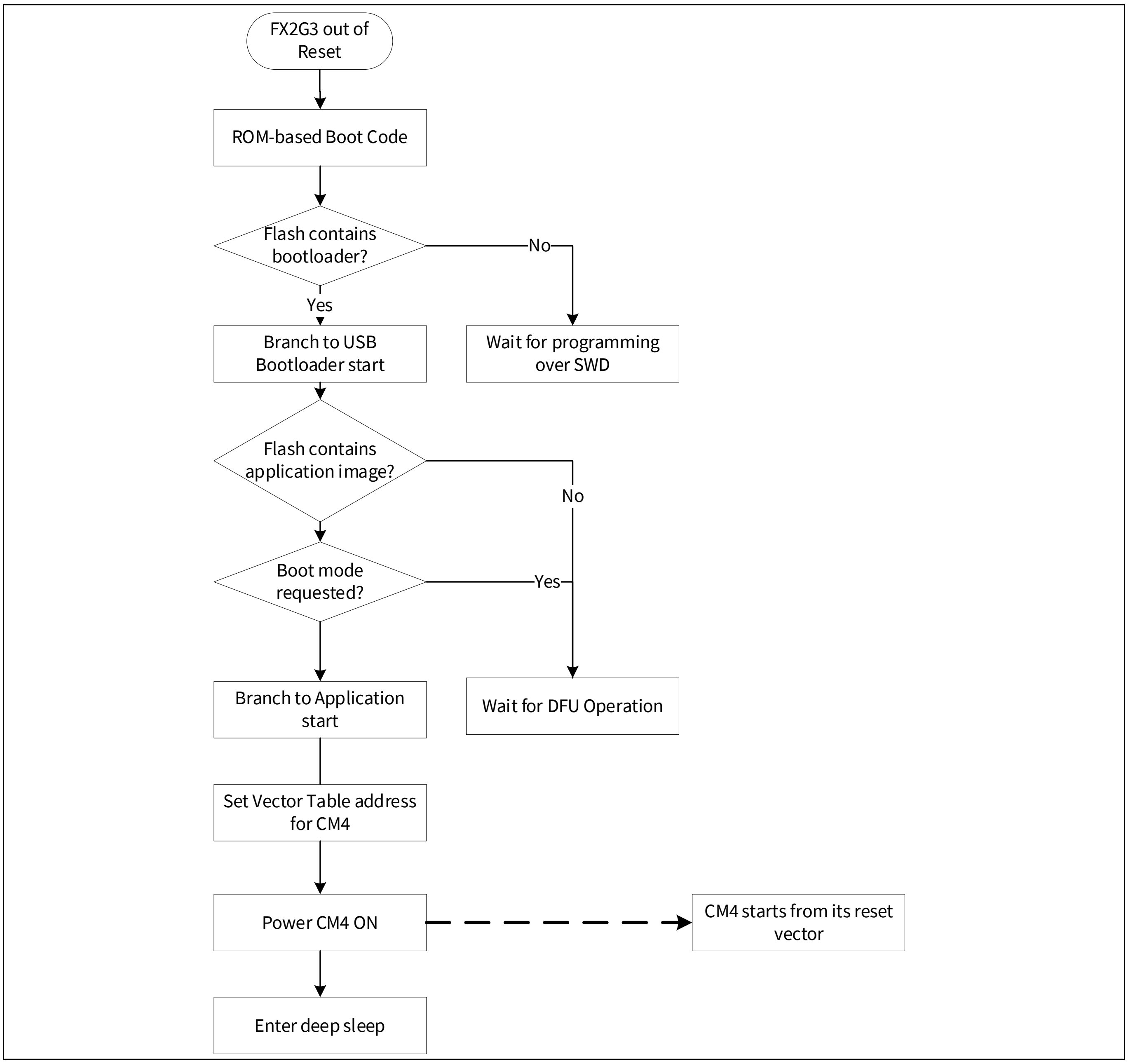

FX2G3 out of Reset
ROM-based Boot Code
Flash contains bootloader?
No
Yes
Branch to USB Bootloader start
Wait for programming over SWD
Flash contains application image?
No
Boot mode requested?
Yes
Branch to Application start
Wait for DFU Operation
Set Vector Table address for CM4
Power CM4 ON
CM4 starts from its reset vector
Enter deep sleep

Figure 10 FX2G3 boot flowchart

The USB bootloader checks the flash region from address 0x1000 8000 onwards for the presence of a valid application image. If it is found and a Boot mode request is not detected, the bootloader transfers control to the application image by setting the stack pointer and branching to the reset vector found in the vector table at address 0x1000 8000.

The CM0+ core starts executing the firmware application. As the first step, it sets the vector table location for the CM4 processor. This is typically set to the flash location 0x1000 8400 (1 KB offset from the start of the application). The bootloader then powers the CM4 core on, at which point the core starts running from its reset vector location.

Once the Cortex®-M4 core has been powered on, the CM0+ has no more work to do and enters Deep Sleep state.

The code to be executed on the CM0+ core is pre-compiled into a binary blob and embedded into the firmware applications as part of the `Cm0Code` array which is placed at the start of the flash region allocated using linker directives.

4 Firmware architecture

The following sections summarizes the firmware design flow using the SDK components like middleware, board support packages, and PDL.

A comprehensive overview of each code example is provided in the corresponding *README.md* file.

4.1 RTOS usage in firmware libraries

When the FX2G3 SDK components are built with the default options, they make use of FreeRTOS from *usbfxstack* as summarized in the following sections.

4.1.1 Memory configuration

The FreeRTOS configuration used enables dynamic memory allocation and makes use of the heap_4 allocation strategy. A memory block of 32 KB in the system SRAM region is reserved for the heap usage by default. This value can be changed (if required) by modifying the `configTOTAL_HEAP_SIZE` value in *FreeRTOS/FreeRTOSConfig.h* and recompiling the library.

4.1.2 RTOS tasks

FreeRTOS is configured to use task priorities in the range of 0 (lowest) to 15 (highest).

- **Idle task created by FreeRTOS Kernel:** Created with a priority level of 0 and stack allocation of 400 bytes.
- **Timer task created by FreeRTOS Kernel:** Created with a priority level of 14 and stack allocation of 2 KB.
- **High-Bandwidth DMA manager task:** Created with a priority level of 10 and with a stack allocation of 2 KB.
- **USB device stack task:** Created with a priority level of 10 and stack allocation of 2 KB.

4.1.3 Message queues

The High BandWidth DMA manager and USB device components of usbfxstack make use of message queues to pass information about events of interest to the respective tasks for processing. Only minimal handling of interrupts is done in the ISR and the bulk of the work is expected to be done in the task.

- **High BandWidth DMA message queue:** Interrupts received from all High BandWidth DMA sockets are passed to the HBDma manager task through this message queue after the interrupt has been masked out in the ISR.
- **USB device stack message queue:** Interrupts received from the USBHS IP blocks are passed to the USB stack task through this message queue.

4.1.4 RTOS timers

The USB stack makes use of an RTOS timer instance to schedule the re-enablement of USB low-power mode (LPM) transitions after the link has returned to an active state. This delay is used to ensure that any pending transactions on the link can be completed before the link enters a low-power state again.

On a USB 2.x connection, the transition to L1 state is disabled as soon as the link enters L1 and re-enabled 10 ms after the link returns to the L0 state.

4.2 FX2G3 device initialization

See Section 3.3 for details of how the FX2G3 device powers up and how the CM0+ core enables the CM4 core which implements all of the remaining functionality.

4.2.1 Clock initialization

Once the CM4 core starts running, the first task to be performed is to enable the various on-chip clocks required for operation.

A simplified view of the FX2G3 internal clock tree is shown in **Figure 5**. The primary clock sources available on the FX2G3 device are:

- **Internal main oscillator (IMO):** This generates a clock with a frequency of 8 MHz ± 2% and is automatically enabled when the device powers up. The relevant clock buffers are enabled such that the Cortex®-M0+ core can function based on this clock when the device is out of reset.
- **Integrated low-speed oscillator (ILO):** This generates a clock with a frequency of 32 kHz ± 10% and is used to clock the watchdog as well as other low-power circuits. This is the only clock source which will be active when the FX2G3 device is in Deep Sleep state.
- **External crystal oscillator (ECO):** Any high-speed clock derived from the IMO will have the same 2% error range of the IMO which is too high for use as a USB Hi-Speed clock reference. A more accurate (< 0.5%) clock reference is required for USB operation on the FX2G3 device. This is generally provided by a crystal oscillator circuit which uses a 24 MHz crystal connected between the XTALIN and XTALOUT pins. There is an option for this circuit to use a 24 MHz clock input connected on the XTALOUT pin instead of the crystal oscillator.

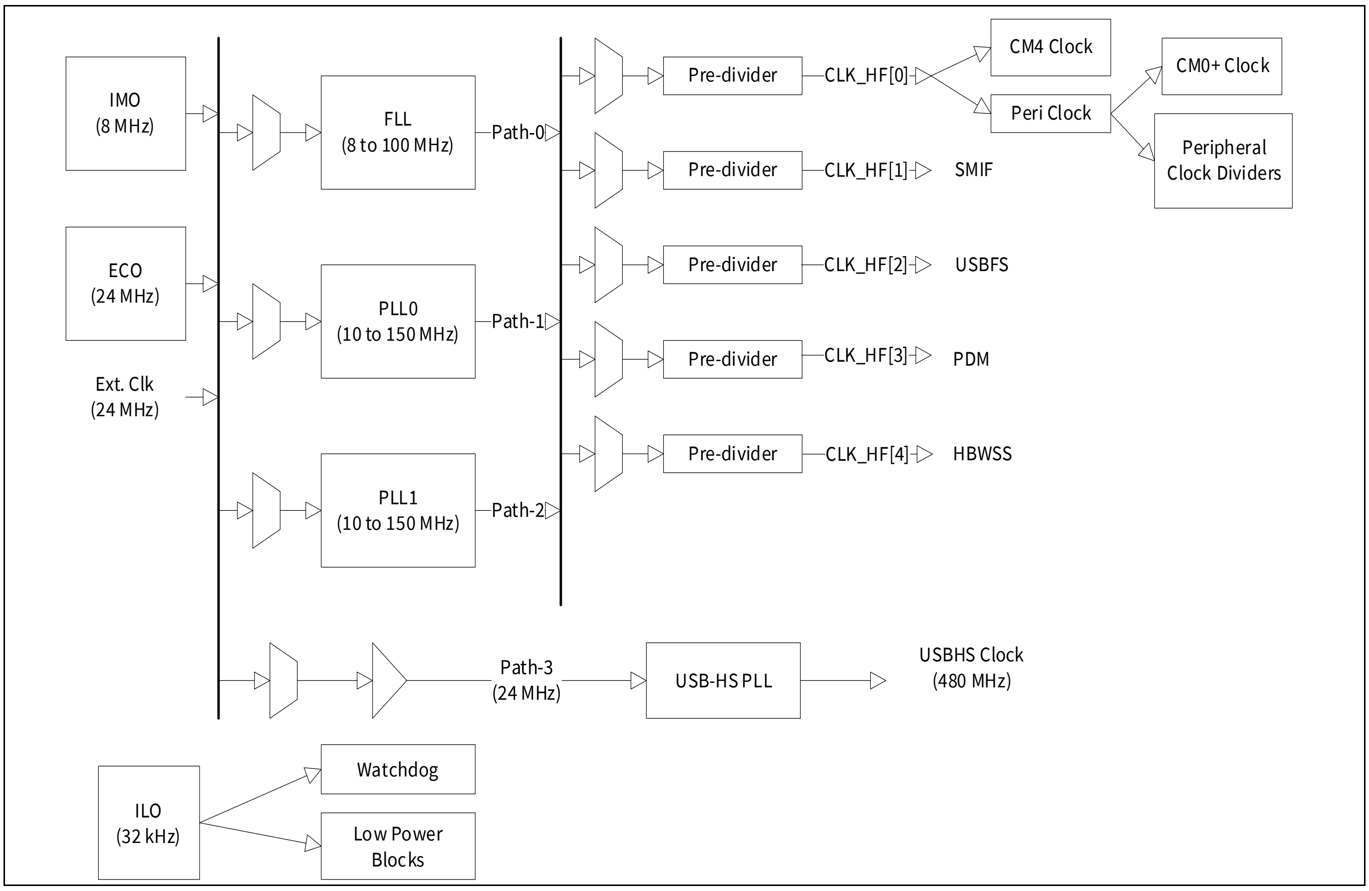


Figure 11 Internal clock tree of FX2G3

The high-speed clocks required for FX2G3 operation are provided by:

- **Frequency-locked loop (FLL):** FLL can generate frequencies up to 100 MHz.

- **Phase-locked loop (PLL0 and PLL1):** FX2G3 has two internal general-purpose PLLs, each of which can be independently configured to generate clocks up to 150 MHz.

The outputs from the FLL, PLLs, and ECO are then used to generate all other internal clocks required for FX2G3 operation through a set of clock buffers and dividers. FX2G3 has five root clocks which need to be connected to one of these high-speed clock sources.

1. clk_hf[0] clock tree provides the clock input to the Cortex®-M4 and Cortex®-M0+ cores, the AHB-based DMA engines, and the programmable peripherals such as SCB, PWM, and CAN FD.
   a) clk_hf[0] is divided by an integer factor between 1 and 256 to derive the clk_fast clock which is used as the clock for the Cortex®-M4 core. The maximum clk_fast frequency supported is 150 MHz which is the same as the maximum clk_hf[0] frequency supported.
   b) clk_hf[0] is divided by an integer factor between 1 and 256 to derive the clk_peri clock. As the maximum clk_peri frequency supported is 100 MHz, use a divider of 2 and reduce clk_peri to 75 MHz when clk_hf[0] frequency is set to 150 MHz. If clk_hf[0] frequency is restricted to 100 MHz, It is possible to configure both clk_fast and clk_peri to operate at 100 MHz.
   c) clk_peri is divided by an integer factor between 1 and 256 to derive the clk_slow clock which is used by the Cortex®-M0+ core as well as the AHB-based DMA engines. The maximum clk_slow frequency supported is 100 MHz.
   d) clk_peri is divided by a set of programmable dividers to provide clocks to other on-chip peripheral blocks such as SCB, PWM, and CAN FD. The USB, SMIF, and PDM blocks have their own dedicated clock paths, which need to be configured separately.
2. clk_hf[1] is used as the clock for the Serial Memory Interface (SMIF) block which connects FX2G3 device to external serial memory devices. The maximum clk_hf[1] frequency supported is 75 MHz.
3. clk_hf[2] is used as the clock for the USB Full-Speed block and set to a frequency of 48 MHz whenever this needs to be used.
4. clk_hf[3] is used as the root clock for the PDM and I2S-based audio interfaces. The maximum frequency supported is 75 MHz; the frequency can be set based on the audio interface requirements.
5. clk_hf[4] is used as the clock for the High BandWidth subsystem including the DMA buffer RAM. The MCU cores on the FX2G3 device can only access the DMA buffer RAM when the clk_hf[4] clock is enabled. The maximum frequency supported is 150 MHz.

The `cybsp_init()` function provided in the FX2G3 BSP configures all the clocks for a typical FX2G3 use case:

- PLL0 is used to generate a clock of 150 MHz which will drive clk_hf[0], clk_hf[1], and clk_hf[4].
- PLL1 is used to generate a clock of 48 MHz which will drive clk_hf[2].
- FLL is used to generate a frequency of 24.576 MHz which will drive clk_hf[3].
- clk_fast is set to 150 MHz so that the Cortex®-M4 CPU is running at the maximum supported rate.
- clk_peri and clk_slow are set to 75 MHz, which is the maximum possible frequency when clk_hf[0] is set to 150 MHz.
- clk_hf[1] for the SMIF interface is set to a frequency of 75 MHz.
- clk_hf[4] for the High BandWidth subsystem is set to a frequency of 150 MHz.

4.2.2 On-reset initialization

Most of the sample FX2G3 applications make use of part of the DMA buffer RAM for storing data structures like the USB descriptors to be used during device enumeration. When linker directives have been used to place read/write data sections in the DMA buffer RAM, the CPU should access the buffer RAM before the application initialization is complete and the `main()` function is executed.

The clk_hf[4] root, which provides the clock input to the High BandWidth subsystem is disabled during FX2G3 power-up and is enabled explicitly before this memory region can be accessed by either of the Cortex®-M4 or M0+ cores.

The `Cy_Fx2g3_OnResetInit()` function is used to enable the clk_hf[4] clock (and connect it to the IMO output at 8 MHz) and to enable the High BandWidth subsystem completely. It is mandatory for this function to be called from the `Cy_OnResetUser()` function if the application intends to place any read/write data directly in the DMA buffer RAM region.

4.2.3 I/O configuration

Most of the digital I/Os on the FX2G3 device support multiple functions and configure to select the desired function.

The common I/Os and functions used in the FX2G3 sample applications are summarized as follows:

Table 3 Common FX2G3 I/O usage

FX2G3 pin	GPIO ID	Function
B16	P11.1	**SCB4 UART receive pin:** Input buffer enabled, output drivers disabled for High-Z state
B15	P11.0	**SCB4 UART transmit pin:** Configured as strong drive output
H12	P10.0	**SCB0 I2C data pin:** Configured as open-drain output
G12	P10.1	**SCB0 I2C clock pin:** Configured as open-drain output
G5	P4.0	**Vbus detect pin:** When LOW, it indicates that the Vbus supply is present on the USB port
K9	P11.2	**SWD data pin for CPU debug:** Configured with internal resistive pull-up enabled
K10	P11.3	**SWD clock pin for CPU debug:** Configured with internal resistive pull-down enabled

4.2.4 Peripheral initialization

FX2G3's peripherals are initialized using the APIs from the mtb-pdl-cat1 PDL.

4.2.4.1 Watchdog reset disable

The watchdog reset module on the FX2G3 device is enabled by default and will result in resetting the device after about 4.3 seconds of operation. The watchdog either needs to be cleared periodically or the reset function is disabled during start-up. All the code examples provided in the FX2G3 SDK disable the watchdog reset by calling the `Cy_WDT_Unlock()` and `Cy_WDT_Disable()` functions during start-up.

4.2.4.2 Debug logging enable

The FX2G3 firmware library and applications make use of a configurable logging infrastructure which is capable of routing messages to either a UART console or to a virtual COM port implemented using the USBFS port of the FX2G3 device (USB debug or J3 port on the KIT_FX2G3_104LGA DVK).

UART initialization for logging

If the debug logs are to be output through a UART interface, enable the corresponding SCB block and configure for this purpose. This is done by calling the `InitUart()` function with the SCB index (4 by default) as a parameter. As the SCB is only used for logging (output) purposes, the application does not prepare to receive any data on the corresponding UART_RX pin.

The UART used for logging is configured for operation at 921,600 baud with one stop bit and no parity bits.

USBFS (debug) initialization for logging

If the debug logs are to be output through the USBFS debug port, enable the USBFS block from the utilities component of the usbfxstack middleware and configure for enumeration as a Communications Device Class – Abstract Control Model device which provides a virtual COM port functionality.

The USBFS block initialization is performed implicitly when the debug logger module is enabled with the output interface selected as `CY_DEBUG_INTFCE_USBFS_CDC`. It is expected that a physical USB cable connection from the USB debug port to the host controller is present when this interface is being used for logging.

Logging configuration

The debug logging is implemented using a RAM-based buffer. The data to be logged is formatted and stored into this buffer and then output through the selected interface. Select the following configuration parameters before enabling the debug logging module.

Table 4 Logging configuration parameters

FX2G3 pin	Function
pBuffer	Pointer to a RAM buffer where the log messages are stored temporarily.
bufSize	Size of the RAM buffer in bytes. Recommended to be 1 KB or more.
dbfIntfce	Selects the interface through which logs are output from FX2G3. Currently supported values are all SCB UART modules and the USBFS debug port.
traceLvl	Selects the verbosity of output messages. Log messages with lower priority will be filtered out by the debug module. The levels are: • Level 1: Error messages • Level 2: Warning messages • Level 3: Info messages • Level 4: Trace messages. It is not recommended to enable the output of trace messages.
printNow	Boolean flag indicating whether debug logs should be output as soon as possible. Recommend setting this value to true.

The logger module is enabled by calling the `Cy_Debug_LogInit()` function. If the USBFS debug port is used for logging output, it is recommended that a delay be provided after calling this function to allow the CDC device enumeration and driver binding to be completed before any other operations are performed.

4.3 USB block operation

USB HS block is initialized and configured using the USB stack component of the usbfxstack middleware library.

4.3.1 USB initialization

Steps for enabling and operating the FX2G3 USB interface:

1. Registration of ISRs for USBHS interrupts so that the respective driver functions are called to handle the interrupts.
   a) The `usbhsdev_interrupt_u2d_active_o_IRQn` and `usbhsdev_interrupt_u2d_dpslp_o_IRQn` interrupts are associated with the USBHS block. Call the `Cy_USBHS_Cal_IntrHandler()` function provided in the firmware library when either of these interrupts is asserted. In an RTOS-enabled configuration, this function will return 'true' if a context switch is to be triggered before the ISR exits.
2. **Powering on and configuring the USB blocks for operation:** During this operation, the internal power supplies in the USBHS block are enabled and the PLL is configured to generate the 480 MHz clock. As this clock is used by multiple other blocks on the FX2G3 device, this needs to be enabled independent of the type of USB connection to be established. The `Cy_USB_USBD_Init()` function performs this initialization and also creates the USB stack task and associated message queues.
3. Register the descriptors (device descriptor, configuration descriptor, string descriptors, device qualifier descriptor, binary object store descriptor, etc.), which are to be returned by the device during device enumeration. These descriptors are expected to be stored in dedicated memory buffers and the corresponding pointers are registered with the USB device stack using the `Cy_USBD_SetDscr()` function. It is recommended (though it is not mandatory) that each of the descriptors be placed at 16-byte aligned addresses in the DMA buffer RAM.
4. Register callback functions to handle important USB connection-related events using the `Cy_USBD_RegisterCallback()` API. At a minimum, callbacks for the `CY_USB_USBD_CB_RESET`, `CY_USB_USBD_CB_SETUP`, and `CY_USB_USBD_CB_SET_CONFIG` events need to be registered.
5. If the system that uses the FX2G3 device is self-powered (not powered through the USB port), firmware needs to wait until the Vbus supply is present on the USB port before attempting to enable the connection. The mechanism used to do this is system/circuit specific. On the KIT_FX2G3_104LGA DVK, this is done using the VBUSDETECT (P4.0) GPIO. This signal will be HIGH (pulled up to V3P3_1P8) when Vbus supply is not present and LOW (connected to GND) when Vbus supply is present.
6. Enable the USB connection by calling the `Cy_USBD_ConnectDevice()` API. The desired USB speed of operation can be passed as a parameter. The USB stack will automatically initialize the link to the best possible speed less than the requested value based on the upstream port's capabilities.

4.3.2 USB enumeration

4.3.2.1 USB link initialization

When `Cy_USBD_ConnectDevice()` is called to enable the USB connection, the USB stack attempts to initialize the USB link as defined in the USB specification.

4.3.2.2 USB control request handling

The initial part of USB device enumeration involves the host querying the device capabilities through the set of USB descriptors. The USB stack automatically handles most of these requests by returning the pre-registered set of descriptors.

When the stack receives any control request which it is unable to handle, the request is passed to the application through the callback function registered for the `CY_USB_USBD_CB_SETUP` event. The 8-byte data associated with the control request can be retrieved from the setupReq field in the USBD stack context structure.

Note that the callback for the control requests is issued from the USB driver task context and it is recommended to not block this function for an extended period of time. Where a delay is required as part of the control request handling, the application can defer the handling to its own task using inter-process communication methods such as shared variables, message queues, or event groups.

The application has four options for handling a control request:

1. **Acknowledge the request and complete the status handshake:** This option is applicable only if there is no data phase associated with the request (wLength = 0). The `Cy_USBD_SendAckSetupDataStatusStage()` function can be called to acknowledge the control transfer with no errors.
2. **Send data to the host and then complete the status handshake:** This option is applicable when there is an IN data transfer associated with the request (wLength ≠ 0 and MSB of bmRequestType is set to '1'). The `Cy_USB_USBD_SendEndp0Data()` function can be called to initiate the data transfer. It is recommended that the data to be sent be placed in a 16-byte aligned buffer located in the DMA buffer RAM. If the data is not placed in the DMA buffer RAM, the API makes a copy of the data in the buffer RAM and then initiates the data transfer from there. Note that all the data to be sent in response for the control request is expected to be sent through a single call of the `Cy_USB_USBD_SendEndp0Data()` API. The API waits until the DMA transfer to the host has been completed or a timeout period of 2.5 seconds has elapsed.
3. **Receive data from the host and then complete the status handshake:** This option is applicable when there is an OUT data transfer associated with the request (wLength ≠ 0 and MSB of bmRequestType is '0'). The transfer can be initiated using the `Cy_USB_USBD_RecvEndp0Data()` API. It is required that the data buffer passed to receive the data should be placed at a 16-byte aligned location in the DMA buffer RAM. The API will return an error if the buffer placement is not as expected.
4. **Send a STALL handshake indicating request handling error:** This option is allowed for all control transfers and is performed by calling the `Cy_USB_USBD_EndpSetClearStall()` API to set the STALL status bit in Endpoint-0.

Special control requests

Special callbacks are associated with the `SET_ADDRESS`, `SET_CONFIGURATION` and `SET_INTERFACE` control requests and the `CY_USB_USBD_CB_SETUP` callback is not used for these.

- **SET_ADDRESS:** The **CY_USB_USBD_CB_SETADDR** callback type is used to send notification of the SET_ADDRESS request. No application handling of this request is required as the request would be automatically acknowledged by the USB hardware block. The callback is only provided for information and can be used by the application to identify the type of USB connection which has been established.
- **SET_CONFIGURATION:** The **CY_USB_USBD_CB_SET_CONFIG** callback type is used to send notification of the SET_CONFIGURATION request. The application is expected to configure all the endpoints and associated DMA resources before returning from this callback function.
- **SET_INTERFACE:** The **CY_USB_USBD_CB_SET_INTF** callback type is used to send notification of a SET_INTERFACE request addressed to any of the interfaces. The interface and alternate setting selected can be retrieved from the setupReq structure in the USB stack context structure. In an application that supports multiple alternate settings for one or more interfaces, this callback is expected to:
  - Disable all endpoints used by the previous alternate setting and free up the corresponding DMA resources.
  - Configure and enable all endpoints used by the newly selected alternate setting and configure the corresponding DMA resources.

4.3.3 USB operation

Once the USB connection is made and enumeration is complete, the data transfers are mostly handled through the DMA infrastructure and there is no active role for the USB driver stack or API as long as the USB link is in the active state. When there are any USB link power state changes, these get handled by the USB stack and callbacks are raised where applicable to notify the application.

4.3.3.1 USB 2.x power states

The USB 2.0 specification along with updates defined in Engineering Change Notices (ECNs) ,outlines three power states for the USB link.

- **L0 or Active state:** This is the state in which the USB link remains active and data transfers are supported on the endpoints implemented by the device.
- **L2 or Suspend state:** Suspend state is used in the USB 2.x link for power saving. This state is entered when no data (including SOF packets) has been received on the link for more than 3 ms. The state can be exited by >15 ms of resume signaling initiated by the upstream host/hub or by the device (only when remote wakeup is allowed).

  The callback registered for the `CY_USB_USBD_CB_SUSPEND` event is called when the link enters the L2 state and the callback registered for the `CY_USB_USBD_CB_RESUME` event is called when the link resumes into the L0 state.

  The `Cy_USBD_SignalRemoteWakeup()` API can be used by the application if there is a need to initiate remote wake from the L2 state. Note that this function will only trigger the remote wake signaling if the operation is permitted by the host controller.
- **L1 state:** This is an intermediate low-power state defined through ECN extensions to the USB 2.0 specification. Unlike the L2 state, the entry into the L1 state involves a request from the upstream host/hub which can be acknowledged or rejected by the device.

  In the default configuration, the FX2G3 USB stack allows acceptance of transition into the L1 state whenever requested by the host/hub. The only exception is that entry into L1 is disabled for a period of 10 ms after the link resumes into L0 state from L1 state. This exception ensures that any pending transfers can be completed before the link returns into the L1 state. The `Cy_USBD_LpmDisable()` API can be used if all transitions into the L1 state are to be unconditionally rejected.

  The `CY_USB_USBD_CB_L1_SLEEP` callback is used to notify the application that the USB 2.x link has entered L1 state and the `CY_USB_USBD_CB_L1_RESUME` callback for notification of the link resuming into L0 state. As in the case of the L2 state, the `Cy_USBD_SignalRemoteWakeup()` API can be used by the application to initiate exit from the L1 state into the L0 state.

4.3.4 USB disconnection and reconnection

If the FX2G3 application has a requirement to break the active USB connection, call the `Cy_USBD_DisconnectDevice()` API. It is recommended that the USB PHY and controller blocks are disabled after the disconnection. When a new connection is desired, call the `Cy_USBD_ConnectDevice()` API again as described in Section 4.3.1.

4.4 Sensor Interface Port (SIP) operation

The Sensor Interface Port (SIP) on the FX2G3 device is used to connect to an external data source or sink such as an FPGA or image sensor. The SIP supports an LVCMOS interface.

4.4.1 Sensor interface initialization

The LVCMOS_LVDS block is initialized and configured using the LVCMOS/LVDS library which is a part of the usbfxstack middleware library.

Configuring the SIP/LVCMOS_LVDS on the FX2G3 device involves multiple stages as follows:

1. **Controller and PHY configuration:** This is where the operating mode for the SIP controller is selected and the respective PHY blocks are enabled. This stage is performed by calling the `Cy_LVDS_Init()` API. The parameters provided through the **lvdsConfig** > **phyConfig** structure are used for the PHY initialization.

Table 5 Sensor interface parameters

Parameter	Type	Description
`wideLink`	Boolean	Set to false for FX2G3 as FX2G3 supports only 16-bit bus width.
`modeSelect`	Enumeration	Selects between LVCMOS operation.
`dataBusWidth`	Enumeration	Selects the number of data lanes: 8 or 16 for LVCMOS
`interfaceClock`	Enumeration	Expected interface clock frequency
`gearingRatio`	Enumeration	Selects as 1:1 as FX2G3 supports SDR clock
`clkSrc`	Enumeration	Select the clock used for the data link and GPIF operation: Select from between interface clock, 480 MHz clock derived from USB2 (USB-HS) block.
`clkDivider`	Enumeration	Select the division factor that is used to reduce the frequency of the clock used for data link and GPIF operation. The maximum clock frequency supported for link and GPIF operation is 240 MHz. When the clkSrc is selected as USB2, it needs to be divided at least by 2 to reduce the frequency to a valid range.
`phyTrainingPattern`	uint8_t	Not applicable for FX2G3.
`linkTrainingPattern`	uint32_t	Not applicable for FX2G3.
`ctrlBusBitMap`	uint32_t	Specifies the LVCMOS control signals associated with the link which should be enabled as input pins. The init code will only enable the input buffers for the pins which have been selected here.
`loopbackModeEn`	bool	Not applicable for FX2G3. Must be set as false.
`isPutLoopbackMode`	bool	Not applicable for FX2G3. Must be set as false.
`slaveFifoMode`	Enumeration	FX2G3 supports 2-bit Slave FIFO operation.

2. **GPIF state machine initialization:** The SIP data link operation on the FX2G3 device is governed by the General Programmable Interface (GPIF) block. The GPIF state machine specifies the sequence based on which the data is either sampled from the interface or driven on the interface. The GPIF state machine is configured by the `Cy_LVDS_Init()` function and then enabled using the `Cy_LVDS_GpifSMStart()` function.

4.5 GPIF state machines

As the LVCMOS interface of the FX2G3 device are expected to be driven by an FPGA, use a set of standard pre-defined configurations for the GPIF state machines. The following sub-sections document the standard GPIF configurations and interfaces used in the various FX2G3 code examples.

4.5.1 GPIF config for LVCMOS receiver application

The *mtb-example-fx2g3-slave-fifo-in* and *mtb-example-fx2g3-uvc-uac* applications use the 2-bit Slave FIFO interface. This is a standard LVCMOS configuration which allows the FPGA master to select and perform data transfers to one out of two different pipes. A two-bit address bus is used to select the active pipe for transfer and control signals are used to enable sending of full-length packets, short-length packets, and zero-length packets. The code examples support two LVCMOS sockets using 2-bit address input. As only two threads/pipes are accessible in FX2G3, the MSB of a 2-bit address should always be '0'.

GPIF state machine

Figure 12 shows the state machine used by this LVCMOS receiver and transmitter implementation. The state machine is a simple one which will sample/driver the data on the LVCMOS data lanes based on the control signals and write/read into/from the DMA buffer when the DMA buffers are ready to accept/send the data.

The thread and socket selection in this application are done through the A[0:1] address lines driven by the master/FPGA. Before the master sends data into any thread, it is expected to verify that the corresponding DMA_RDY (Flag A) signal is being asserted (low) by the FX2G3 device.

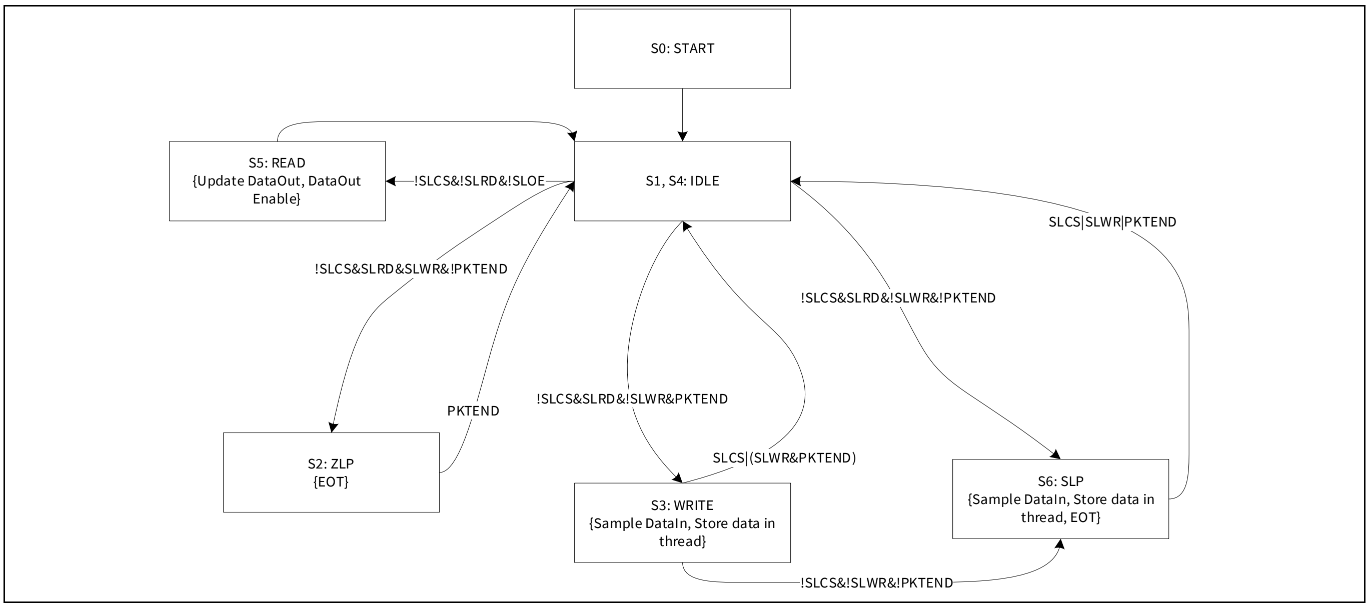


Figure 12 GPIF state machine used by 2-bit Slave FIFO application

Interface signals

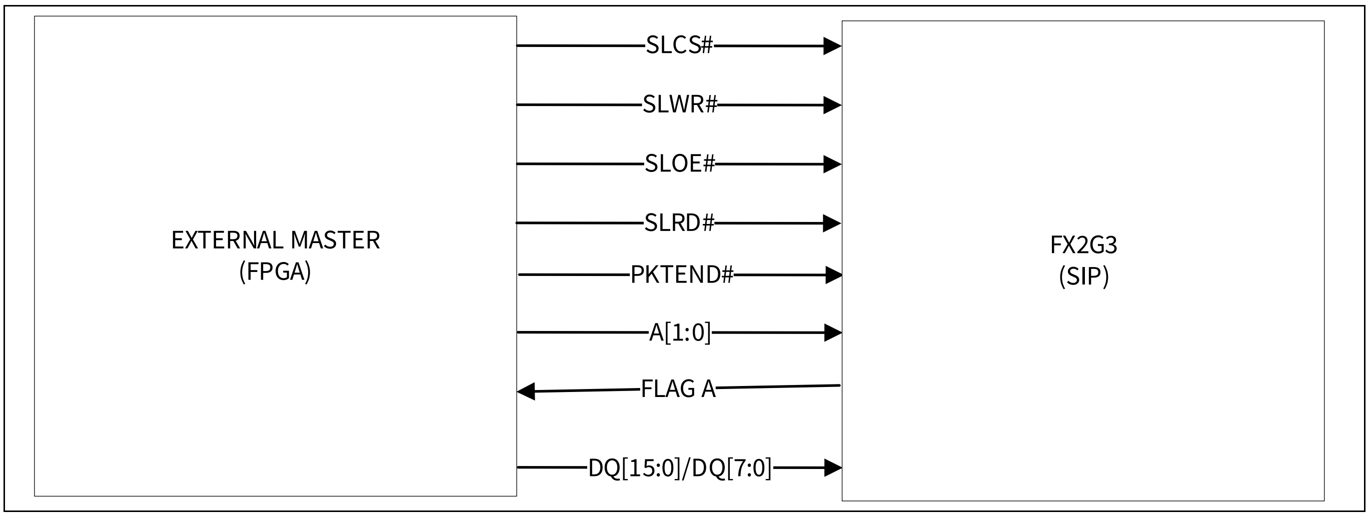


Figure 13 Control signal usage in LVCMOS receiver state machine

Table 6 Control signal usage in LVCMOS 2-bit Slave FIFO state machine

FX2G3 pin	Function	Description
P0CTL0 (A35)	SLCS#	Active LOW Chip Select signal. Should be asserted (LOW) by the master FPGA when communicating with FX2G3.
P0CTL1 (A2)	SLWR#	Active LOW Write Enable signal. Should be asserted (LOW) by the master FPGA when sending any data to the FX2G3. The data present on the data lanes will be sampled and stored into the DMA buffer when this signal is asserted along with SLCS#. Can be combined with the PKTEND# signal to indicate that this data ends the transfer, and the DMA operation should be terminated.
P0CTL2 (A1)	SLOE#	Active LOW Output Enable signal.
P0CTL3 (A37)	SLRD#	Active LOW Read Enable signal. Not used in this application as data is only being received by FX2G3
P0CTL4 (A38)	PKTEND#	Active LOW Packet End signal. Should be asserted (LOW) when the FPGA master wants to terminate the ongoing DMA transfer. Should be asserted along with SLWR# and the last cycle of data to complete transfers with non-empty data. Can be asserted only when SLCS# is LOW and SLWR# is HIGH to complete the DMA transfer with zero bytes of data.
P0CTL5 (A29)	Flag A	Active LOW DMA ready indication for currently addressed/ active thread.
P0D8 (A45)	A0	LS bit of 2-bit address bus used to select the thread. (applicable for 8-bit LVCMOS bus width)
P0D9 (A46)	A1	MS bit of 2-bit address bus used to select thread (applicable for 8-bit LVCMOS bus width)
P1CTL9 (A31)	A0	LS bit of 2-bit address bus used to select thread (applicable for 16-bit LVCMOS bus width)
P1CTL8 (A30)	A1	MS bit of 2-bit address bus used to select thread (applicable for 16-bit LVCMOS bus width)

Table 7 GPIOs for configuring FPGA on KIT_FX2G3_104LGA DVK

FX2G3 Pin	Function	Description
GPIO5 (B38)	CDONE#	Active HIGH signal. FPGA asserts when FPGA configuration is completed.
GPIO6 (B39)	INIT#_RESET	Active LOW signal. FX2G3 asserts to reset the FPGA.
GPIO7 (B41)	PROG#	Active LOW FPGA program signal.

4.6 DMA datapath operation

Once all the required hardware blocks on FX2G3 have been initialized and the USB connection has been enabled, the data flow through the USB endpoints is handled through DMA acceleration with minimal firmware intervention.

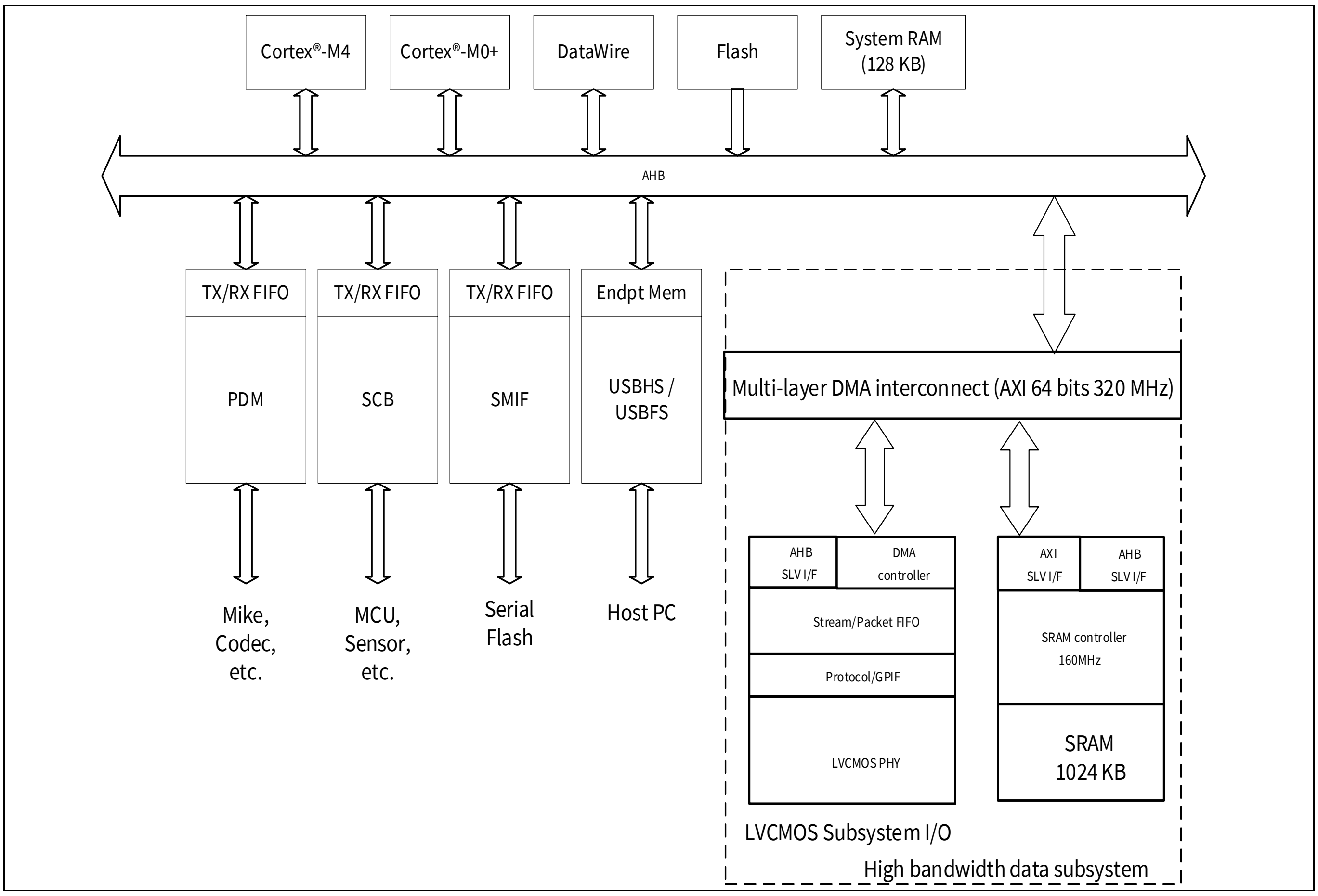


Figure 14 **Datapaths within FX2G3 device**

The FX2G3 device supports two different DMA frameworks:

- AXI-based high-speed DMA framework which is part of the High BandWidth subsystem which moves data between the Sensor Interface Port (SIP) and the SRAM. This DMA framework is similar to the one used on the EZ-USB™ FX3 device and the SDK provides a similar set of API to configure and manage these data transfers.
- AHB-based DataWire DMA framework which can move data between all on-chip memories as well as the legacy peripherals such as the USBHS block, USBFS block, SCB, SMIF, PDM block, and I2S block.

  The scope of each of these DMA data paths is shown in Figure 14.

The High BandWidth DMA framework can only move data in and out of the SIP and can only access the dedicated DMA buffer RAM. This means that any data to be sent through the LVCMOS first needs to be copied into the DMA buffer RAM and then the DMA initiated. Similarly, any data received through the LVCMOS interface has to be collected in the DMA buffer RAM and then copied elsewhere as needed.

The DataWire DMA framework can access all memories and peripherals which are connected to the system-wide Advanced High-performance Bus (AHB) including the flash memory, system as well as DMA buffer RAM and other peripherals.

Some use cases can require the use of both types of DMA for a single stream of data. For example, if data received through the LVDS interface needs to be sent to the USB host PC through a USB Hi-Speed (2.x) connection, the High BandWidth DMA should be used to copy the data from the SIP into the buffer RAM and DataWire should be used to send the received data to the USBHS block.

4.6.1 High BandWidth DMA (HBDma) programming

HBDma is initialized and configured using the DMA manger usbfxstack middleware library.

All data movement within the High BandWidth subsystem happens through temporary buffers located in the buffer RAM. When a DMA datapath is being setup between the LVCMOS and USBHS interfaces, the firmware needs to prepare RAM buffers which will be used for the transfers and configure a set of descriptors which track the state of these RAM buffers. This is performed using a set of convenience API provided as part of the High BandWidth DMA manager.

4.6.1.1 HBW DMA manager initialization

Before the High BandWidth DMA manager can be used, perform the following for initialization:

1. Initialize the DMA adapter blocks that are associated with the Sensor Interface Port. This is done by calling the `Cy_HBDma_Init()` API.
2. Initialize a free pool of DMA descriptors for use by the DMA manager. The maximum number of DMA descriptors which can be used at any one time needs to be specified as a parameter and it is recommended that this value be set to at least 512 descriptors. As each descriptor occupies 16 bytes of memory, enabling the free pool to use 512 descriptors reserves the first 8 KB of the buffer RAM for DMA descriptor usage. The `Cy_HBDma_DscrList_Create()` API is used to initialize this descriptor free pool.
3. Initialize the DMA buffer manager that performs the dynamic memory allocation of all RAM buffers required for the various High BandWidth DMA datapaths. The allocation scheme used is a custom one which ensures that all buffers allocated are placed at 64-byte aligned RAM locations and is designed to have a fixed memory allocation overhead. The `Cy_HBDma_BufMgr_Create()` API is used to initialize the buffer manager and the start address and size of the RAM based heap region need to be passed as parameters.
4. Initialize the HBDma manager module itself by calling the `Cy_HBDma_Mgr_Init()` API.
5. There is one DMA adapter interrupt source for which an interrupt handler need to be registered:
   a) **lvds2usb32ss_lvds_dma_adap0_int_o_IRQn:** Corresponds to interrupts associated with any of the first 16 DMA sockets in the SIP (effectively DMA interrupts associated with LVDS link #0). The ISR for this interrupt needs to call the `Cy_HBDma_HandleInterrupts()` function with the adapter parameter set to a value of **CY_HBDMA_ADAP_LVDS_0**.

4.6.1.2 HBW DMA channel functions

A High BandWidth DMA channel is a virtual construct which binds all resources associated with a single data path through the AXI bus. The resources include the sockets (pipes) on the LVCMOS side where the data enters or exits the FX2G3 DMA framework, the RAM buffers which are used for temporary storage of the data and the corresponding descriptors which store the location and state information about the RAM buffers.

A socket represents one end of a data stream which flows through the AXI DMA framework. There is dedicated logic associated with each socket which keeps track of the state of the ongoing data transfer. Sockets through which data enters the FX2G3 device are called “Producer” or “Ingress” sockets. Sockets through which data exits the FX2G3 device are called “Consumer” or “Egress sockets”.

The following channel can be created:

- IP to memory channels, which receive data through the LVDS/LVCMOS interface and store it in memory. In this case, the channel will only have valid producer sockets.
- Memory to IP channels which take data from memory and send it out through the LVCMOS interface. In this case, the channel will only have valid consumer sockets.

A set of convenience APIs is provided in the SDK for the creation, management, and operation of these DMA channels. A summary of the APIs is provided in Table 8.

Table 8 DMA channel APIs

DMA manager API	Description
`Cy_HBDma_Channel_Create`	Creates a DMA channel structure based on the parameters specified in the configuration structure. The channel create API allocates the required number of RAM buffers and descriptors, initializes them and configures the sockets through which data enters/exits the FX2G3 device. The channel is not ready for data transfer when this function returns and explicitly needs to be enabled.
`Cy_HBDma_Channel_Destroy`	Destroys the DMA channel structure and frees up all resources used for its operation. The sockets used by this DMA channel will be left in the disabled state.
`Cy_HBDma_Channel_Enable`	Enables a DMA channel to move the specified amount of data (can be infinite data as well). The sockets associated with the channel are only enabled when this API is called.
`Cy_HBDma_Channel_Disable`	Disables a DMA channel and restores it into the idle state which is equivalent to the state immediately after channel creation.
`Cy_HBDma_Channel_Reset`	This function is equivalent to `Cy_HBDma_Channel_Disable`.
`Cy_HBDma_Channel_GetBuffer`	This function is used to retrieve information about the active RAM buffer associated with the DMA channel. In the case of IP to IP and IP to Memory DMA channels, the API will only return a valid buffer if data is present in the buffer and yet to be processed. In the case of a Memory to IP channel, the API will return a valid buffer if it is empty and firmware can fill it with new data.
`Cy_HBDma_Channel_CommitBuffer`	This function is used in the case of IP to IP and Memory to IP DMA channels to enable sending of data in a RAM buffer through the output path (consumer socket).
`Cy_HBDma_Channel_DiscardBuffer`	This function is used in the case of IP to IP and IP to Memory DMA channels to discard data which has been received through the input path (producer socket).

The HBDma manager can generate callback notifications when events of interest occur on either the producer or consumer sockets associated with a channel. The `CY_HBDMA_CB_PROD_EVENT` notification indicates that a RAM buffer has been filled with data received through the producer socket and is ready for firmware processing or forwarding to the consumer socket. The `CY_HBDMA_CB_CONS_EVENT` notification that a RAM buffer has been freed up because the data present in it has been sent out through the consumer socket. It is possible for firmware to return this buffer to the use of the producer socket so that new data can be written into it.

IP to memory and memory to IP channels require firmware intervention at the granularity of each DMA buffer. The callback notifications listed above can be used to trigger the appropriate API calls to forward or discard the data associated with the channel.

4.6.2 DataWire DMA programming

Datawire DMA controllers are initialized & configured using the mtb-pdl-cat1 PDL.

FX2G3 device supports two instances of DataWire DMA controller each of which supports 24 channels. Each DataWire DMA channel is capable of moving data from a peripheral block or memory region to another peripheral block or memory region.

Unlike the High BandWidth DMA, the DataWire DMA does not understand USB packet boundaries and moves the user-specified amount of data. The DataWire DMA operation can be started/continued automatically based on trigger signals generated from peripheral blocks and connected to the DataWire DMA controller through a set of Trigger Multiplexers (TrigMux).

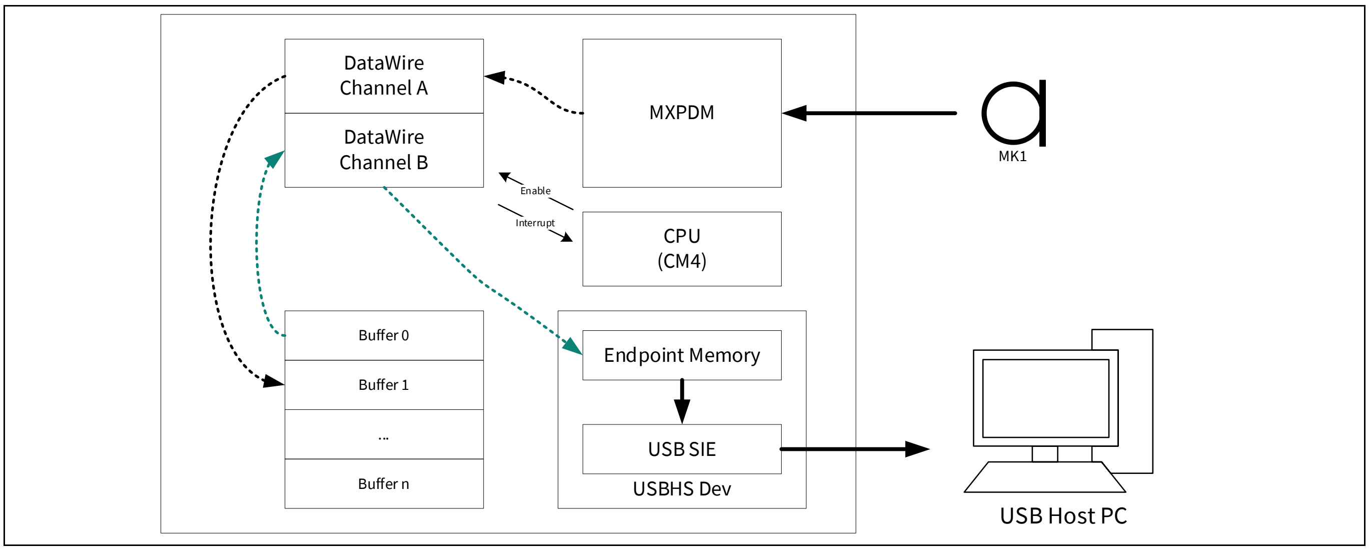


Figure 15 PDM microphone to USBHS transfer using DataWire

Figure 15 shows the data flow in a case where data received from a PDM microphone are transferred to USBHS IN endpoint as part of a USB Audio implementation.

While DataWire supports direct transfer from PDM FIFOs to the USB endpoint memory, this is not recommended for use because each DataWire channel can only take a single trigger input. Because of this constraint, it is not possible to configure the channel to perform a transfer only in cases where the PDM FIFO has data and the endpoint memory has free space.

Two DataWire channels are used to address this restriction:

- Channel A is used to transfer data from the received FIFOs in the PDM IP block to RAM buffers. This transfer is triggered whenever the FIFO has a specific amount of data available.
- Channel B is used to transfer data from the RAM buffers into the USB IN endpoint memory region for transfer to the USB host. This transfer is triggered whenever the endpoint memory has free space to hold one data packet.

Firmware-based control is required to manage the sequencing of the two DataWire channel operations. For example, Channel A should only be enabled for transfer if a free RAM buffer is available to hold the received data. Similarly, channel B should only be enabled when a RAM buffer has been filled with data received from the microphone.

4.6.2.1 DataWire channel mapping

DataWire channels do not have any restrictions with respect to the source and destination data regions used. This means that any DataWire channel can use any memory region, register set, or hardware FIFO which is accessible through the AHB as source or destination for the DMA transfers (subject to target level read/write capabilities; for example, flash memory cannot be used as destination as the memory is not directly writeable).

However, there are restrictions with respect to the triggers which can be connected to/from each of the DataWire channels. Due to these constraints, there is a recommended usage mapping for the DataWire channels supported on FX2G3 as shown in Table 9. If any of the recommended transfers are not in use in a given application, the respective channel can be used for other purposes.

Table 9 Recommended usage mapping for DataWire channels

DataWire-0 channel	Function	DataWire-1 channel	Function
DW0: Channel 0	Read from USBHS OUT Endpoint #0	DW1: Channel 0	Write to USBHS OUT Endpoint #0
DW0: Channel 1	Read from USBHS OUT Endpoint #1	DW1: Channel 1	Write to USBHS OUT Endpoint #1
DW0: Channel 2	Read from USBHS OUT Endpoint #2	DW1: Channel 2	Write to USBHS OUT Endpoint #2
…	–	…	–
DW0: Channel 14	Read from USBHS OUT Endpoint #14	DW1: Channel 14	Write to USBHS OUT Endpoint #14
DW0: Channel 15	Read from USBHS OUT Endpoint #15	DW1: Channel 15	Write to USBHS OUT Endpoint #15
DW0: Channel 16	Write to SCB0 TX FIFO	DW1: Channel 16	Write to SCB2 TX FIFO
DW0: Channel 17	Read from SCB0 RX FIFO	DW1: Channel 17	Read from SCB2 RX FIFO
DW0: Channel 18	Write to SCB1 TX FIFO	DW1: Channel 18	Write to SCB3 TX FIFO
DW0: Channel 19	Read from SCB1 RX FIFO	DW1: Channel 19	Read from SCB3 RX FIFO
DW0: Channel 20	Write to SMIF TX FIFO	DW1: Channel 20	Read from PDM RX FIFO #0
DW0: Channel 21	Read from SMIF RX FIFO	DW1: Channel 21	Read from PDM RX FIFO #1
DW0: Channel 22	Write to I2S TX FIFO	DW1: Channel 22	Read from CANFD RX FIFO #0
DW0: Channel 23	Read from I2S RX FIFO	DW1: Channel 23	Read from CANFD RX FIFO #1

4.6.2.2 DataWire transfers

Typical DataWire transfers can be configured as 1-D transfers or 2-D transfers. In both cases, the channel can transfer a specific amount of data on receiving a trigger signal. The channel can also generate an output trigger or raise an interrupt to the FX2G3 MCU core when a specific amount of data has been transferred.

- DataWire transfers are configured using the `Cy_DMA_Channel_Init()` API and enabled using the `Cy_DMA_Channel_Enable()` API.
- Details of the transfer to be performed are to be set in a RAM-based descriptor structure using the `Cy_DMA_Descriptor_Init()` API.

Separate interrupt vectors are associated with each DataWire channel and corresponding ISRs can be registered and used. There is no recommended handling required for any of the DataWire channel interrupts.

4.6.2.3 DataWire wrapper functions for USBHS

A set of convenience wrapper APIs have been provided to configure and manage DataWire-based transfers from/to the Endpoint Memory block in the USB Hi-Speed peripheral block.

- The `Cy_USBHS_App_EnableEpDmaSet()` API is used to set up the DataWire DMA resources required to read/write data to the USBHS endpoints to/from RAM-based buffers. This API also sets up the relevant trigger connections for these transfers.
- The `Cy_USBHS_App_DisableEpDmaSet()` is used to free up these DMA resources and break the previously made trigger connections.
- The `Cy_USBHS_App_QueueRead()` function is used to queue a DataWire based read operation to read one or more packets of data from the Endpoint Memory region corresponding to an OUT endpoint. It is expected that the data size specified is an integral multiple of the maximum packet size for the endpoint. Typically, the read operation will be queued even before any data has been received from the host controller so that the shared endpoint memory buffers are drained as soon as they are filled.
- DataWire channels move data based on a pre-configured transfer size. When a short packet is received on the OUT endpoint, the channel needs to be re-configured based on the actual size of data in the packet. Otherwise, there will be an underrun condition when the DataWire tries to read more data than is actually available in the Endpoint Memory.
- The `Cy_USBHS_App_ReadShortPacket()` API is used to re-configure the DataWire channel to read the actual amount of data present in the short packet which has been received. The function will return the complete transfer size including any full data packets which were previously received and transferred using the DataWire channel.
- The `Cy_USBHS_App_QueueWrite()` API is used to start writing one or more packets of data to the Endpoint Memory region corresponding to an IN endpoint. There are no restrictions on the data size and the transfer will be completed with a short packet where applicable.

4.7 DMA transfer use cases

4.7.1 LVCMOS to USBHS data transfer

In this case, you need to use a High BandWidth DMA channel to copy the data from the LVDS IP into the DMA buffer RAM and a DataWire channel to move the data to the USBHS endpoint memory.

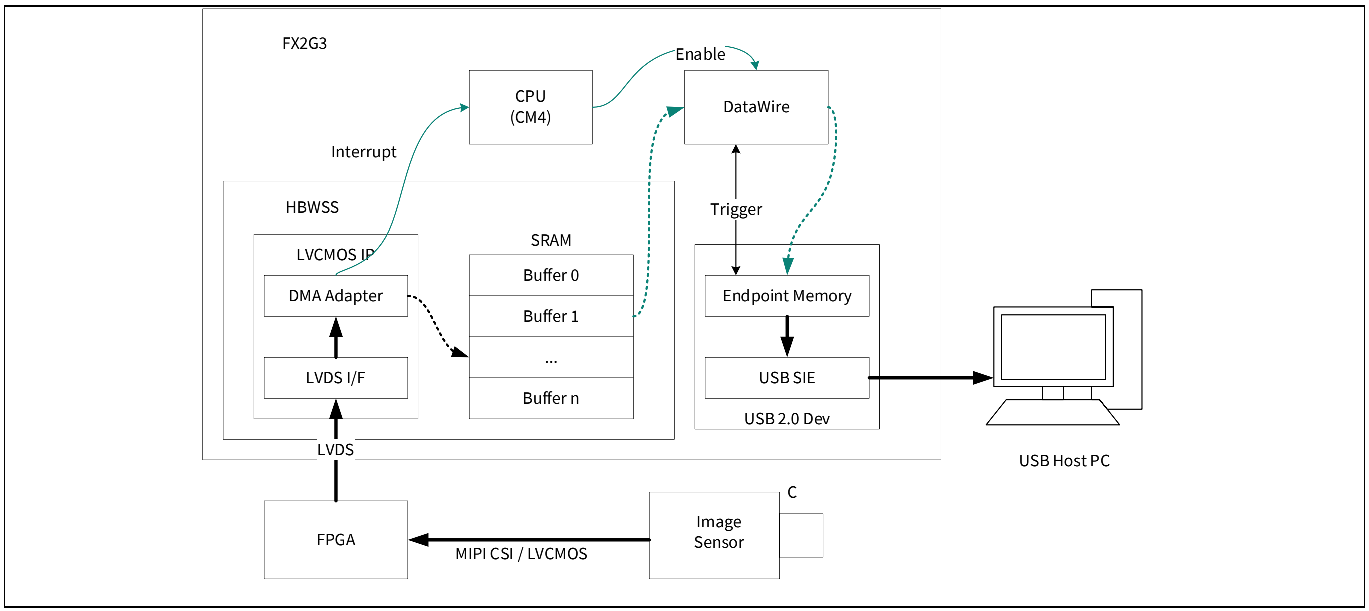


Figure 16 **LVCMOS to USBHS data transfer**

The produce event callbacks from the High BandWidth DMA channel are used to enable DataWire transfers to the USBHS endpoints. Once the DataWire transfer completion interrupt is received, the High BandWidth DMA buffers are marked free by calling the `Cy_HBDma_Channel_DiscardBuffer()` API.

4.7.2 PDM to USBHS data transfer

In this case, the transfer is done by using only DataWire channels. The data flow is shown in Figure 16; the transfer handling is described in Section 4.6.2.

5 Debugging the code examples

This section describes how to debug using the USBFS CDC port of KIT_FX2G3_104LGA DVK. By default, the code examples enable debug logs on the USB-FS CDC port (J7) of KIT_FX2G3_104LGA DVK.

The code examples can be debugged by setting debug levels for UART logs. The `DEBUG_LEVEL` macro in *main.c* can be set to the following values for debugging:

Table 10 Debug levels

#	Macro value	Purpose
1	1u	Enable error messages
2	2u	Enable warning messages
3	3u	Enable info messages
4	4u	Enable all messages

Revision history

Document revision	Date	Description of changes
**	2024-09-26	Initial release
*A	2024-10-09	Minor updates
*B	2024-10-23	Added information on using the ModusToolbox™ IDE.

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.
The Bluetooth® word mark and logos are registered trademarks owned by Bluetooth SIG, Inc., and any use of such marks by Infineon is under license.

Edition 2024-10-23

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2024 Infineon Technologies AG.
All Rights Reserved.

Do you have a question about this document?

Email:

erratum@infineon.com

Document reference

002-40622 Rev. *B

Important notice

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics (“Beschaffenheitsgarantie”).

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer’s compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer’s products and any use of the product of Infineon Technologies in customer’s applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer’s technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies’ products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.