EZ-USB™ FX SDK user guide

About this document

Scope and purpose

The EZ-USB™ FX Software Development Kit (SDK) user guide provides detailed information, instructions and other guidelines for leveraging the FX family of USB device controllers. This includes support for the following devices: EZ-USB™ FX20, EZ-USB™ FX10, EZ-USB™ FX5N, and EZ-USB™ FX5.
The guide aims to facilitate seamless development and integration of USB-based applications while ensuring optimal utilization of the capabilities of the EZ-USB™ FX family controllers.

Intended audience

This guide is designed to assist software developers, firmware engineers, and system integrators in implementing high-bandwidth data transfer applications.

Table of contents

Table of contents

1 Introduction

The EZ-USB™ FX20 family of USB 20 Gbps peripheral controllers designed for next-generation applications in the camera, video, imaging, and data acquisition markets. The device features a dual-core architecture with Arm® Cortex®-M4 and Cortex®-M0+ CPUs, complemented by 512 KB flash memory, 128 KB SRAM, and 128 KB ROM. It integrates seven serial communication blocks (SCBs), a cryptography accelerator, and a high-bandwidth data subsystem that facilitates DMA data transfers between LVDS/LVCMOS interfaces and USB ports at speeds up to 20 Gbps.

The high-bandwidth data subsystem includes 1 MB of SRAM for USB data buffering, ensuring efficient and reliable data handling during high-speed transfers. Additionally, the EZ-USB™ FX20 supports USB Type-C plug orientation detection and flip-mux functionality, removing the requirement for external logic. These features make the EZ-USB™ FX20 a robust solution for high-performance applications requiring high-speed USB connectivity and advanced data processing capabilities.

Note: *EZ-USB™ FX SDK and the accompanying documentation are applicable to all EZ-USB™ FX20, FX10, FX5N, and FX5 devices, unless stated otherwise.*

1.1 Software prerequisites

The Software tools which are part of the EZ-USB™ FX SDK are listed in Table 1.

Table 1 Prerequisites

#	Software	Description	Version
1	ModusToolbox™	Eclipse IDE, configuration and build tools	3.2 or later
2	EZ-USB™ FX Control Center	Tools for device programming and testing	0.1 or later

1.2 Firmware support

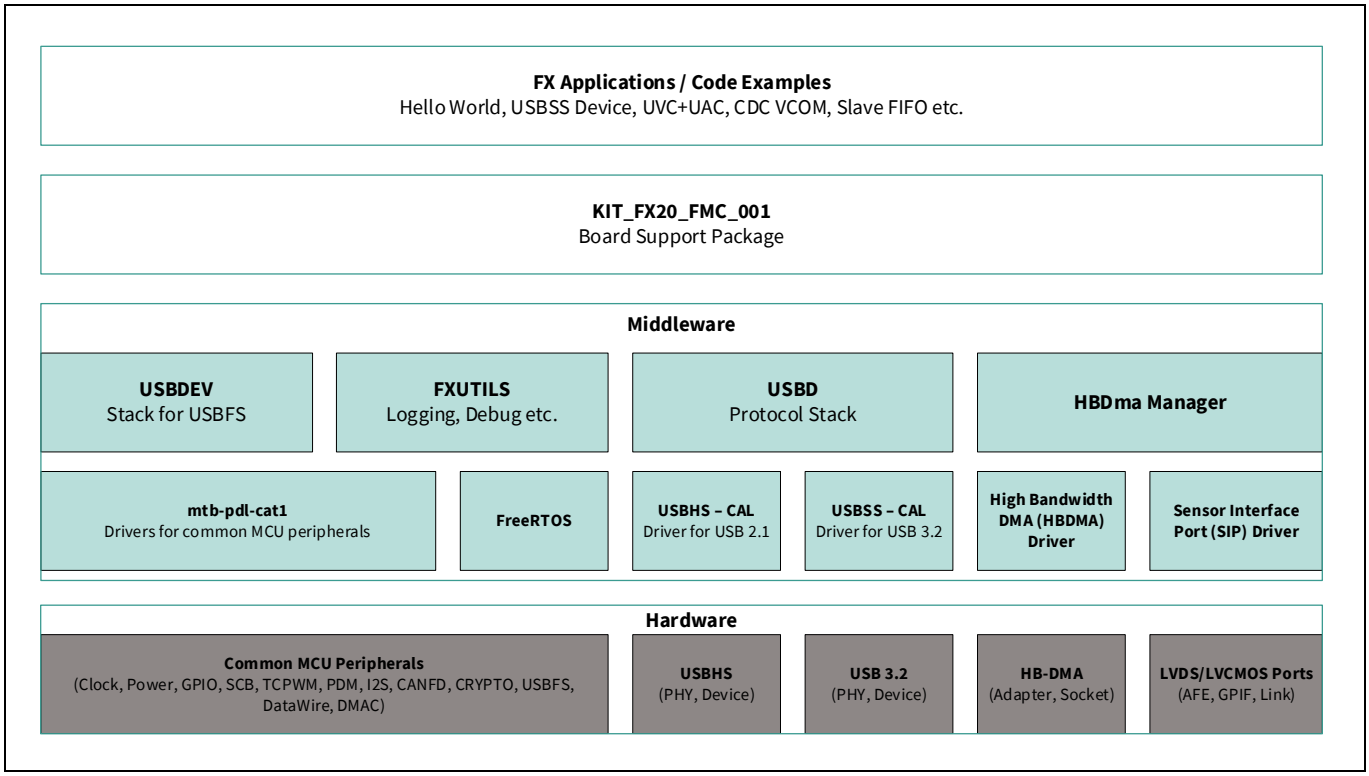

Figure 1 Firmware components of EZ-USB™ FX SDK

Firmware support for the EZ-USB™ FX devices is provided in the form of Peripheral Driver Libraries (PDL), FreeRTOS middleware library, usbfxstack middleware library and code examples which demonstrate typical use cases for these devices. All the firmware components are provided in the form of source and are hosted on GitHub.

The Eclipse IDE for ModusToolbox™ provides an automated mechanism for the desired code examples and all dependency libraries to be downloaded from GitHub, updated as desired and compiled into binary form.

The firmware components included in the EZ-USB™ FX SDK is shown in Figure 1.

1.2.1 EZ-USB™ FX20 code examples

The list of code examples available for the EZ-USB™ FX device families is shown below:

- *mtb-example-fx20-hello-world*: Simple application which toggle GPIOs and outputs log messages through the USBFS debug port or a UART interface.
- *mtb-example-fx20-usbss-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-fx20-usb-testapp*: Implements a vendor-specific USB device which demonstrates all USB data and DMA capabilities of the EZ-USB™ FX20 device. The application supports data loopback functionality for data integrity testing as well as source/sink functionality for throughput testing. An I2C control interface is provided so that the set of endpoints and functionality supported by the application can be modified dynamically.

Introduction

- *mtb-example-fx20-uvc-uac*: Implements USB Video Class (UVC) and integrated USB Audio Class (UAC) functions. The UVC function streams video data received over LVDS or LVCMOS interface from an FPGA. The UAC function streams audio data received from a pair of PDM stereo microphones.
- *mtb-example-fx20-cdc-vcom*: Implements a USB Communication Devices Class (CDC) Virtual COM port which can be used to send/receive UART data over USB endpoints.
- *mtb-example-fx20-smif-dma*: Implements a vendor specific interface to read/write data from/to a NOR flash device connected over a Quad SPI interface to the EZ-USB™ FX20 device. The read or write data is transferred between the USB and QSPI interfaces using DMA channels.
- *mtb-example-fx20-slave-fifo-2bit*: Implements a data acquisition application which receives up to 4 streams of data from an FPGA over an LVCMOS interface and sends them to the USB host controller. The LVCMOS interface is configured in Wide Link mode and uses 2 address pins to select the data stream.

See the *README.md* file in each of these application folders for a detailed description of the application functionality, design and implementation.

2 Installation

2.1 Dependencies

The SDK requires the user to install the following:

1. ModusToolbox™ to obtain the compiler and other build tools. Instructions for installing this dependency is provided in the following section.
2. EZ-USB™ FX Control Center to program the generated firmware binaries to a EZ-USB™ FX device and to perform WinUSB based data transfers.

2.1.1 ModusToolbox™

Download and install ModusToolbox™ tools package. The current version of ModusToolbox™ is 3.4. The rest of the instructions assume that the default options are used for ModusToolbox™ installation.

2.1.2 EZ-USB™ FX control center

Download and install EZ-USB™ FX Control Center from Infineon Developer Center

2.2 Components of the EZ-USB™ FX SDK

This version of the EZ-USB™ FX SDK contains:

- **Firmware files:** Contains the EZ-USB™ FX Firmware libraries, header files, code examples and build scripts.
- **Documentation:** This user guide as well as EZ-USB™ FX firmware API reference manuals.

Note: *This SDK and the accompanying documentation are applicable to all EZ-USB™ FX20, EZ-USB™ FX10, EZ-USB™ FX5N, and EZ-USB™ FX5 devices, unless stated otherwise.*

2.2.1 Firmware library modules

The firmware libraries provided in the EZ-USB™ FX SDK includes the following modules:

- **FreeRTOS operating system:** Sources files for FreeRTOS Kernel version 10.4.6
- **EZ-USB™ FX Peripheral Driver Library (PDL):** Driver library for the clock, GPIO, serial peripherals, TCPWM, USB Full-Speed, QSPI, PDM, and I2S interfaces
- **LVDS/LVCMOS IP driver:** Driver for the Sensor Interface Ports (SIP)[1] can be configured to function with either an LVDS interface or an LVCMOS interface
- **USB Stack:** Integrated USB stack which supports the Hi-Speed, SuperSpeed (USB 3.2 Gen1) and SuperSpeedPlus (USB 3.2 Gen2) functionality
- **DMA Manager:** Provides a generic DMA channel abstraction for the DMA resources that move data across the USB and LVDS/LVCMOS interfaces
- **Utility functions:** A set of utility functions which provide functionality including logging support, abstraction for USB data transfers and fault handlers

These functions are provided in C source form through GitHub based Middleware repositories.

[1] Sensor Interface Port (SIP) refers to the LVDS/LVCMOS interface for the FX device

2.2.2 Application examples

See EZ-USB™ FX20 code examples for more details on the included code examples.

3 EZ-USB™ FX Firmware Architecture

3.1 EZ-USB™ FX USB bootloader

A USB-based bootloader has been provided for the EZ-USB™ FX device to allow programming of custom firmware applications without having to make use of a dedicated SWD programmer. This bootloader is pre-programmed on the EZ-USB™ FX parts and the binary is provided in the *<Code example>\Bootloader\bin* folder.

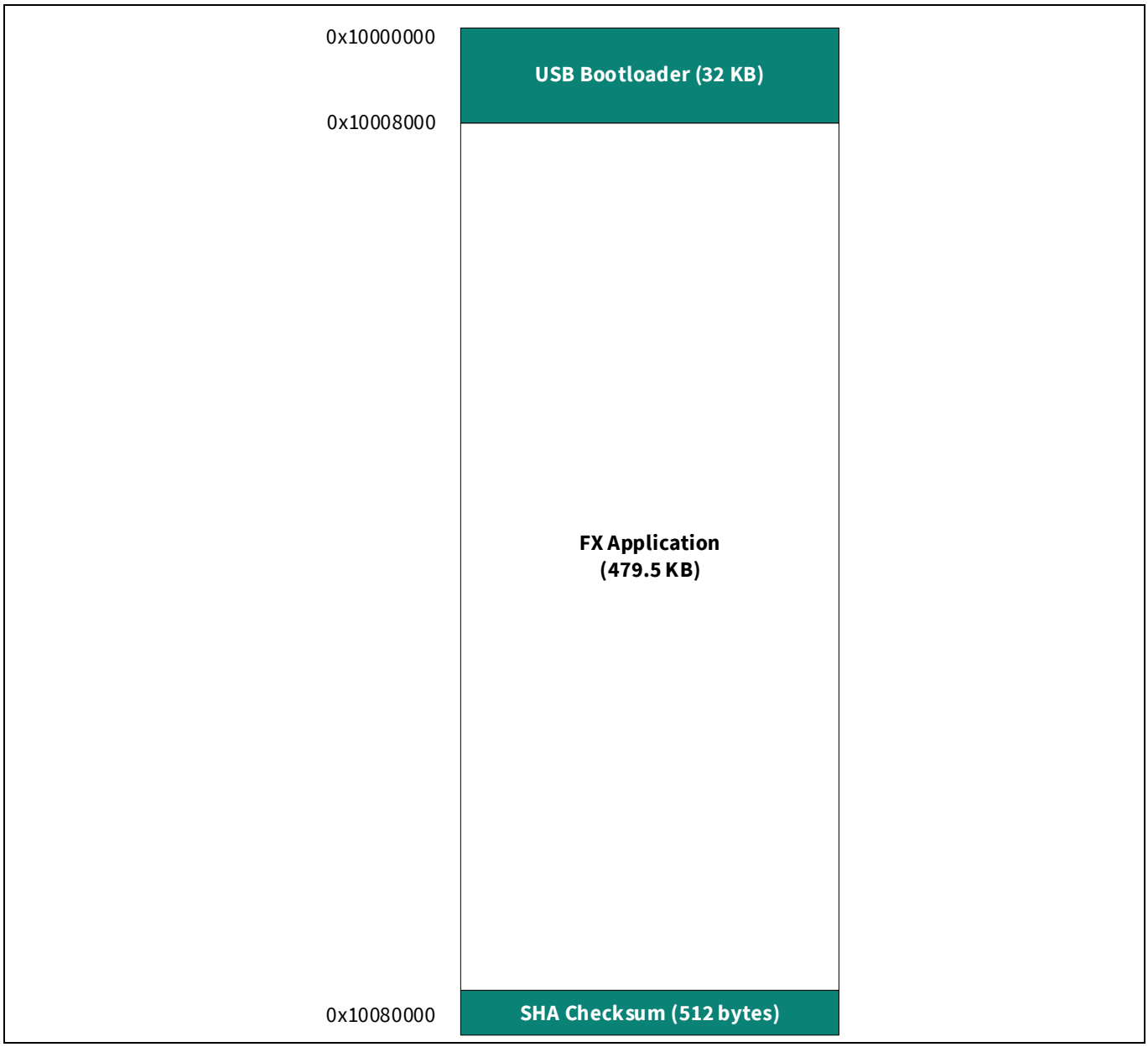

Figure 2 EZ-USB™ FX device flash memory layout

Figure 2 shows the flash memory layout on the EZ-USB™ FX 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 SHA256 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. The checksum is calculated and appended to the application HEX file by the post build commands provided in the code example.

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.

3.1.1 Building firmware for bootloader compatibility

By default, all code examples are configured to build firmware with bootloader compatibility.

The following changes are done in the firmware binary for being used along with the USB bootloader.

- The starting location of the firmware binary is moved to 0x10008000 instead of 0x10000000.
- SHA256 checksum is calculated over the firmware binary and placed in the last 512-byte flash row.

The makefile provided along with each code example support compiling the application for compatibility with the USB bootloader or as a standalone operation (requires SWD programming).

Use the `make` command to build the application and generate a binary which can be programmed through the USB bootloader.

Note: *The binaries generated through this command will not work if programmed directly through the SWD interface.*

Use `make BLENABLE=no` command to build a stand-alone binary which should be programmed onto the device through the SWD using OpenOCD as described in Program application directly from ModusToolbox.

3.1.2 Programming using the USB bootloader

The EZ-USB™ Control Center Utility can be used to program the binaries onto the EZ-USB™ FX device through the USB bootloader.

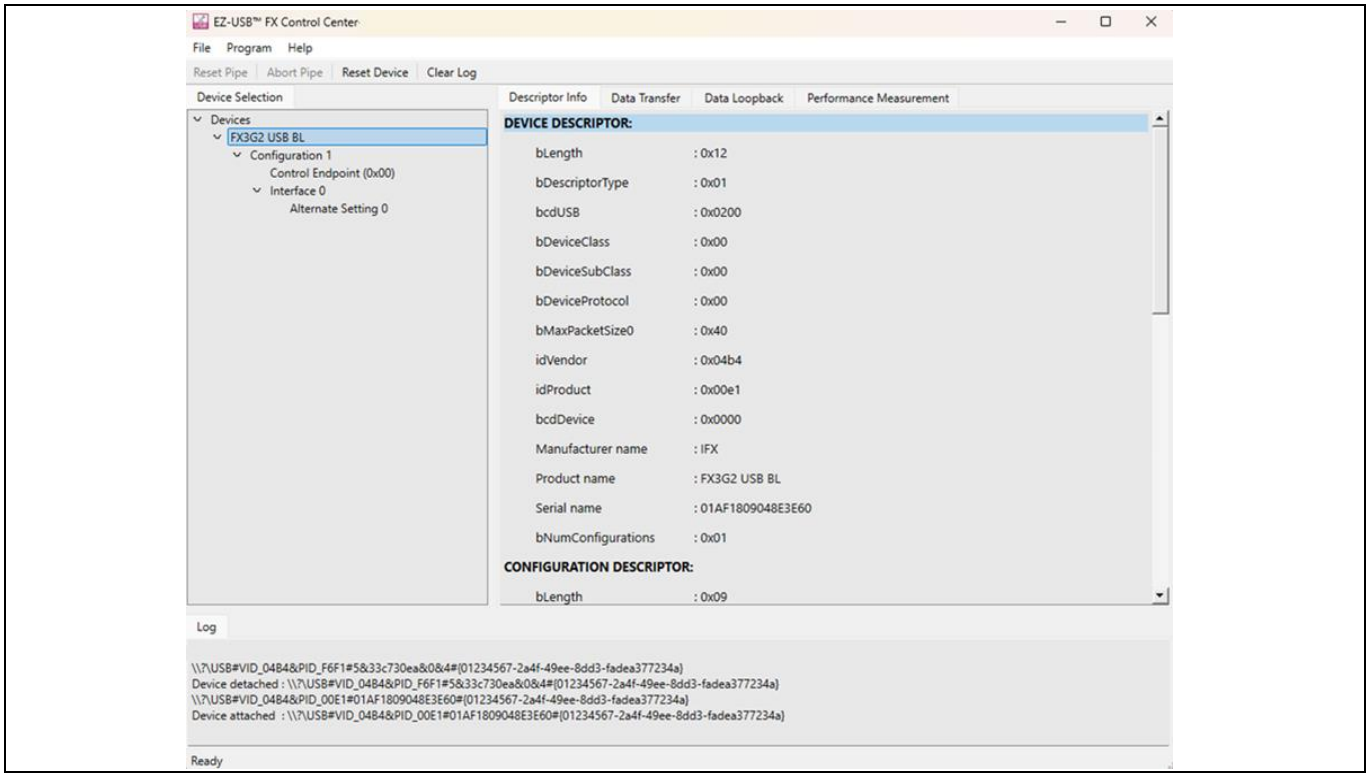

Figure 3 EZ-USB™ control center GUI

Select the **Program > Internal Flash** option from the Actions menu, then browse and select the firmware binary to be programmed to the EZ-USB™ FX device. As the programming operation erases and programs new content into 480 KB of the on-chip flash memory, the operation will take about 15 seconds to complete.

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

3.1.3 Returning control to the USB bootloader

Once the firmware binary has been programmed onto the EZ-USB™ FX 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_FX20_DVK_FMC_001 kit 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 this 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
0x080003C0	0x544F4F42	“BOOT”
0x080003C4	0x45444F4D	“MODE”

The following code snippet can be incorporated in user applications to initiate return to USB boot mode, when required.

Code listing 1 Code snippet to conditionally return to USB bootloader

```
if <condition> {
    /* Set the boot mode request signature in RAM and reset to return to BL. */
    DBG_APP_INFO("Return to boot-loader\r\n");
    *((volatile uint32_t *)0x080003C0UL) = 0x544F4F42UL;
    *((volatile uint32_t *)0x080003C4UL) = 0x45444F4DUL;
    NVIC_SystemReset();
}
```

3.2 EZ-USB™ FX dual-core boot flow

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

When the EZ-USB™ FX device powers up (or comes out of reset), only the Cortex®-M0+ core will be running and the Cortex®-M4 is held in reset. The Cortex®-M0+ core executes the ROM-based boot code before transferring control to the user application at the beginning of the on-chip flash memory (address 0x10000000). This can be the starting of the EZ-USB™ FX USB bootloader or that of the EZ-USB™ FX application if the USB bootloader is not being used.

This location is expected to hold a Cortex®-M0+ 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).

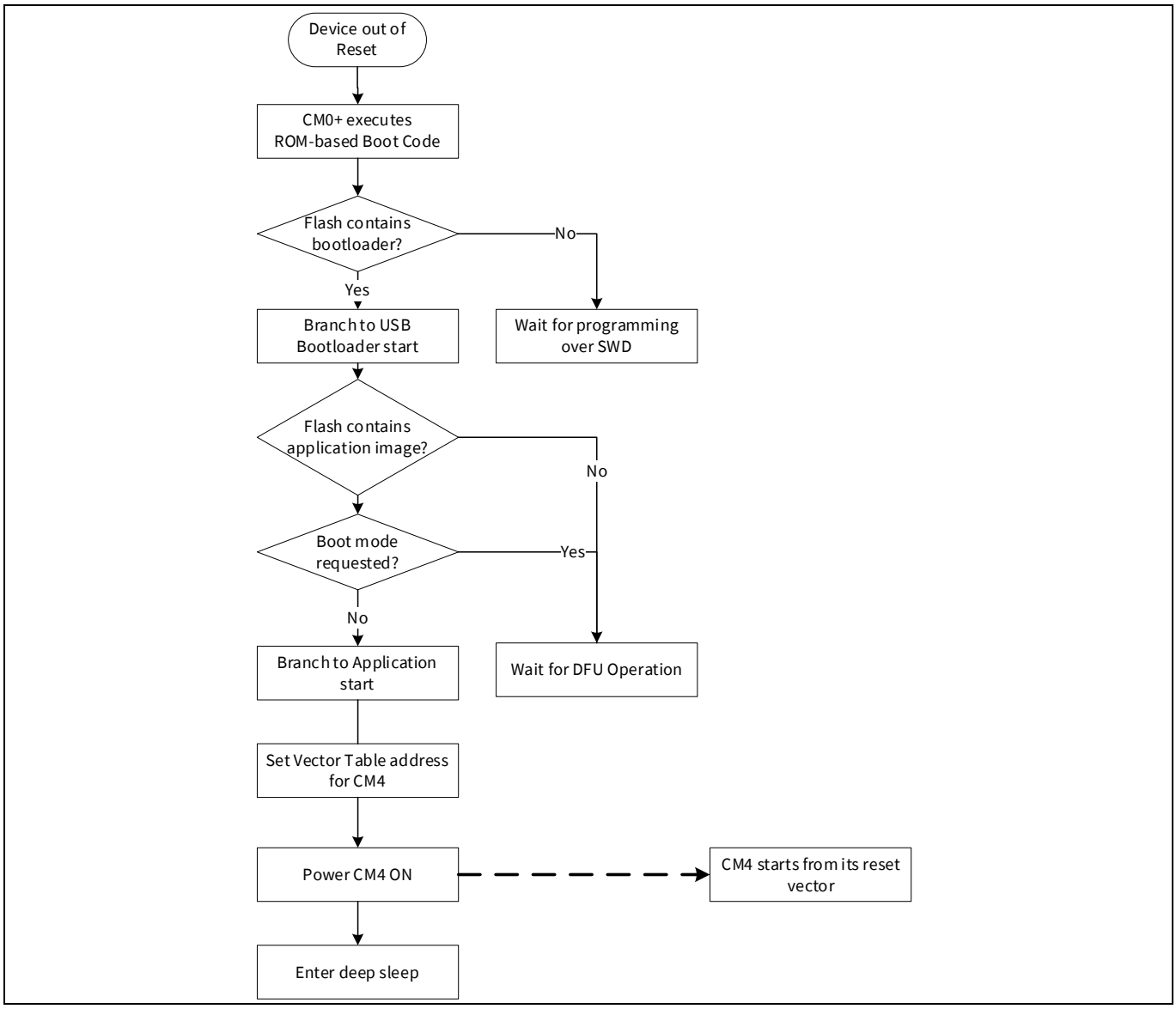

Figure 4 EZ-USB™ FX boot flowchart

The USB bootloader checks the flash region from address 0x10008000 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 0x10008000.

The Cortex®-M0+ core starts executing the firmware application. As the first step, it sets the vector table location for the Cortex®-M4 processor. This is typically set to the flash location 0x10008400 (1 KB offset from the start of the application). The bootloader then powers the Cortex®-M4 core on, at which point the core starts running from its reset vector location.

Once the Cortex®-M4 core has been powered on, the Cortex®-M0+ has no more work to do and enters Deep Sleep state.

The code to be executed on the Cortex®-M0+ core is pre-compiled into a binary blob and embedded into the firmware applications as part of the cm0_code.c file as part of each code example.

3.3 EZ-USB™ FX firmware libraries

As shown in Figure 1, EZ-USB™ FX20 device initialization and configuration is done by a set of drivers and middleware libraries. The libraries are available in source form on GitHub and will automatically be downloaded into the ***mtb_shared*** folder of the workspace when creating a new application using the Eclipse IDE for ModusToolBox.

3.3.1 RTOS usage in firmware libraries

In general, the USBD protocol stack and the High BandWidth DMA Manager libraries make use of FreeRTOS™ services for flexible operation. The use of FreeRTOS™ in these libraries can be disabled by compiling the entire application with the ***FREERTOS_ENABLE*** pre-processor definition set to a value of 0 in the application makefile. By default, usage of FreeRTOS™ services is enabled in all EZ-USB™ FX20 Code Examples except ***mtb-example-fx20-hello-world***.

The usage of FreeRTOS in the USBD and HBDma Manager middleware is summarized in the following sections.

3.3.1.1 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 *FreeRTOSConfig.h* used within the application.

Please note that this 32 KB memory region is placed in the bss segment which will be in the System RAM area of the EZ-USB™ FX20 device by default. If a larger heap region is to be allocated, it may be necessary to move the bss segment to the DMA buffer area by modifying the linker script used in the application.

3.3.1.2 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.

3.3.1.3 Message queues

The High BandWidth DMA manager and USB device stack 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 Interrupt Service Routine (ISR) and the bulk of the work is expected to be done by 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 as well as USBSS IP blocks are passed to the USB stack task through this message queue.

3.3.1.4 RTOS timers

The USB stack makes use of a pair RTOS timer instances to manage event sequences in a specification compliant manner.

- The usbdTimer instance is used 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. The period of this timer is set to 10 ms in USB 2.x connections and 50 ms in USB 3.x connections.
- The usb3ConnTimer instance is used to schedule the USB 3.x re-connection once a USB 2.x bus reset has been detected. The delay provided by this timer allows sufficient time for the USB Enhance SuperSpeed capability in the host controller to get enabled. The duration of this timer is set to 400 ms.

3.4 EZ-USB™ FX device initialization

See section
for details of how the EZ-USB™ FX device powers up and how the Cortex®-M0+ core enables the Cortex®-M4 core which implements the remaining functionality.

3.4.1 Clock initialization

Once the Cortex®-M4 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 EZ-USB™ FX internal clock tree is shown in Figure 5. The primary clock sources available on the EZ-USB™ FX device are:

- **Internal master 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+ 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 EZ-USB™ FX 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 EZ-USB™ FX 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.

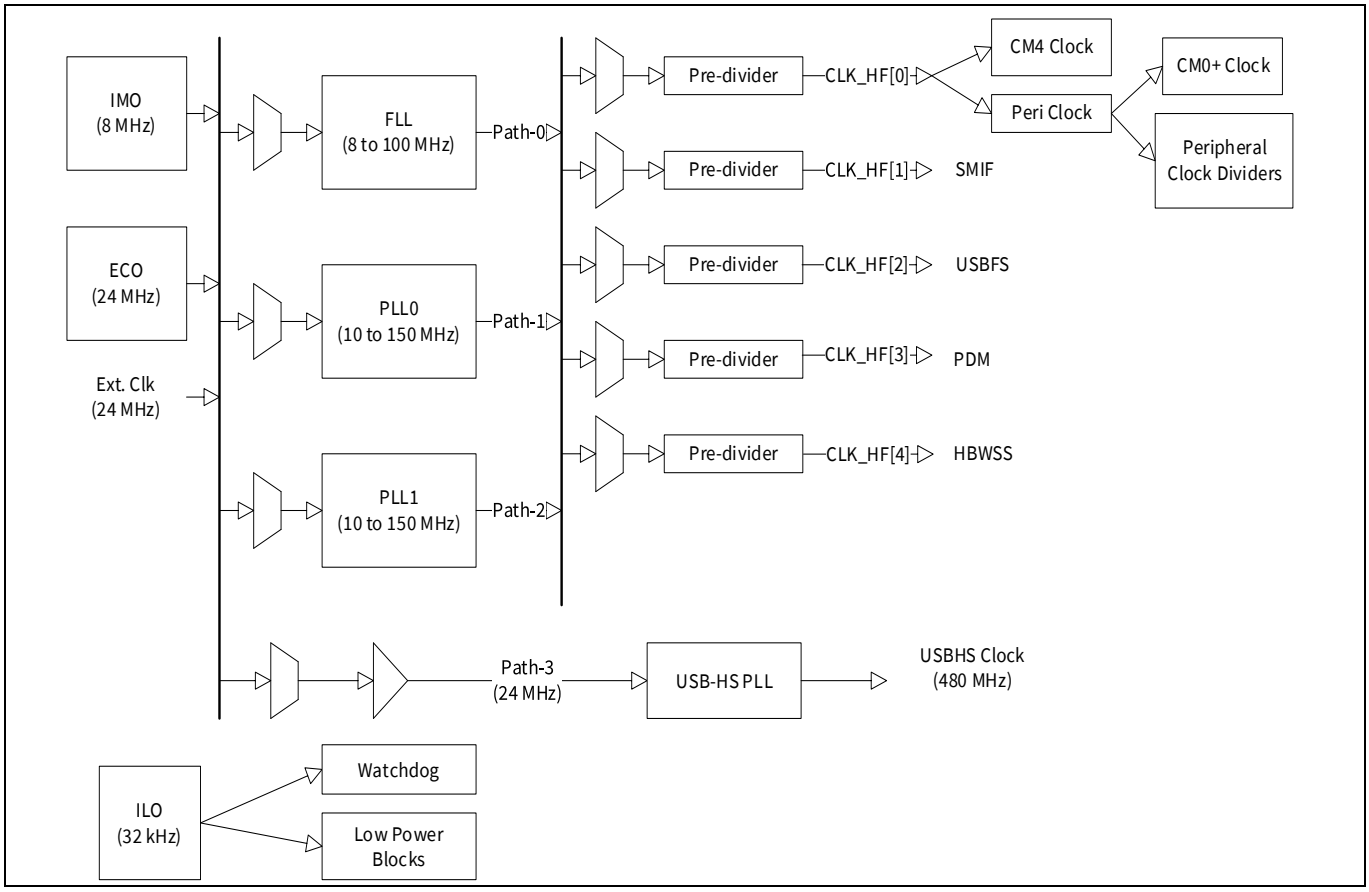

Figure 5 Internal clock tree for all EZ-USB™ FX devices

The high speed clocks required for EZ-USB™ FX operation are provided by:

- **Frequency locked loop (FLL):** FLL can generate frequencies up to 100 MHz.
- **Phase locked loop (PLL0 and PLL1):** EZ-USB™ FX 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 EZ-USB™ FX operation through a set of clock buffers and dividers. EZ-USB™ FX 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 CANFD.
   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 CANFD. 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 EZ-USB™ FX 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 and 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 EZ-USB™ FX device can only access the DMA buffer RAM when the clk_hf[4] clock is enabled. The maximum frequency supported is 150 MHz.

3.4.1.1 BSP specific clock configuration

The code to configure the clocks is generated by the ***Device Configurator*** tool of ModusToolbox based on the settings provided by the BSP. The settings used in each application are saved under the *templates/TARGET_KIT_FX20_FMC_001/config/design.modus* file and can be edited using the Device Configurator.

To open the Device Configurator, click on the "*BSP Configurators (APP_KIT_FX20_FMC_001) → Device Configurator*" option in the Quick Panel of the ModusToolBox Eclipse IDE.

The high frequency clocks at the device level are configured using the System tab of the Device Configurator as shown in Figure 6.

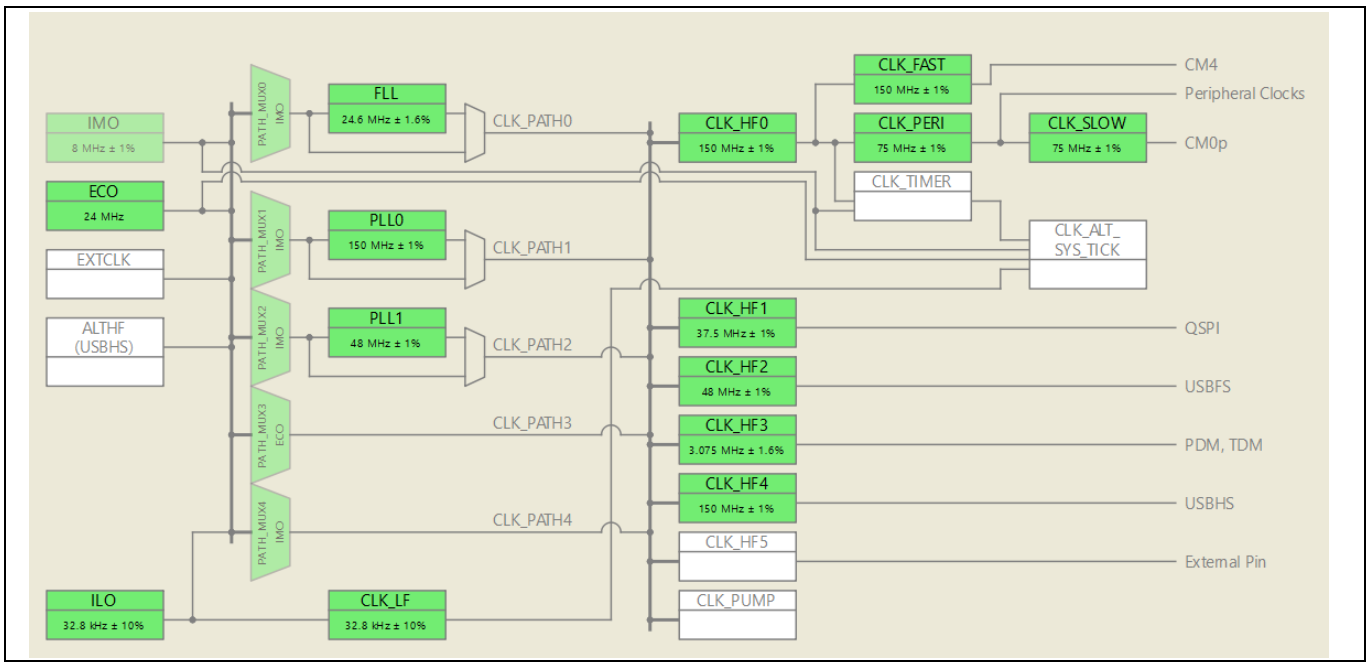

Figure 6 Screenshot of high frequency clock configuration in EZ-USB™ FX20 BSP

As shown in the figure, the default clock settings used are as follows:

- PLL0 uses IMO clock as reference and generates a 150 MHz clock which is used as CLK_HF0 and CLK_HF4. This clock is divided to 75 MHz frequency and used as CLK_PERI and CLK_SLOW.

- PLL1 uses IMO clock as reference and generates a 48 MHz clock which is used as CLK_HF2 which drives the Debug USB port on EZ-USB™ FX20.
- FLL uses IMO clock as reference and generates a 24.6 MHz clock which is divided down to 3.075 MHz and used as CLK_HF3 which drives the PDM and I2S audio peripherals.
- The ECO is enabled and expected to be connected to a 24 MHz crystal. The ECO clock output is connected as reference to the PLL instance in the USB IP blocks and is not shown in the diagram.
- ILO is enabled to generate a clock of approximately 32.8 KHz frequency to drive all the retention logic in the chip level deep sleep state.

3.4.1.2 On reset initialization

Most of EZ-USB™ FX reference applications make use of part of the DMA buffer RAM for storing data structures like the USB descriptors to be used during device enumeration. These array and structure variables will typically be part of the data section and are expected to be initialized by the compiler generated start-up code when the application starts running.

The DMA buffer RAM is not enabled by default when the EZ-USB™ FX device comes out of reset. The RAM needs to be enabled and provided with an operational clock input before any read/write accesses can be done to this area.

These steps must be completed before the start-up code attempts to initialize the arrays and structure placed in this memory region. Hence, it is necessary to enable the DMA buffer RAM much before the application `main()` function is executed.

The `Cy_FX_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.

3.4.2 I/O configuration

Most of the digital I/Os on the EZ-USB™ FX device support multiple functions and configure to select the desired function.

The common I/Os and functions used in the EZ-USB™ FX sample applications are summarized as follows:

Table 1 Common EZ-USB™ FX I/O Usage

EZ-USB™ FX pin	GPIO Id	Function
H5	P8.0	SCB1 UART Receive Pin: Input buffer enabled, Output drivers disabled for Hi-Z state
H6	P8.1	SCB1 UART Transmit Pin: Configured as strong drive output
H13	P0.0	Debug output pin: Configured as strong drive output
G13	P0.1	Debug output pin: Configured as strong drive output
K13	P1.0 (CC1)	USB-C CC1 pin used for orientation detection Note: This pin cannot be used for any output functions
J13	P1.1 (CC2)	USB-C CC2 pin used for orientation detection Note: This pin cannot be used for any output functions

EZ-USB™ FX pin	GPIO Id	Function
G5	P4.0	Vbus detect pin: When low, indicates that Vbus supply is present on the USB port
J7	P9.2	Debug output pin: Configured as strong drive output
J8	P9.3	Debug output pin: Configured as strong drive output
J9	P9.4	Debug output pin: Configured as strong drive output
J10	P9.5	Debug output pin: Configured as strong drive output
K7	P11.0	Debug output pin: Configured as strong drive output
K8	P11.1	Debug output pin: Configured as strong drive output
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

The P11.0, P11.1, P9.2, P9.3, P9.4 and P9.5 pins of the EZ-USB™ FX device can be used to bring out internal signals for debug monitoring. Several signals from the USB SuperSpeed PHY, USB controller and the LVDS controller can be routed to these pins.

The debug usage of these pins is enabled and the signals to be routed to these pins are selected using the `Cy_FX_SelectDFTFunctions` API. Any of the listed signals from the USB or LVDS/LVCMOS blocks can be routed to P11.0 and P11.1 pins. Only USB related signals can be routed to the P9.2 and P9.3 pins for monitoring. Only LVDS related signals can be routed to the P9.4 and P9.5 pins for monitoring.

3.4.3 Peripheral initialization

3.4.3.1 Watchdog reset disable

The watchdog reset module on the EZ-USB™ FX 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 disabled during start up. All the code examples provided in the EZ-USB™ FX SDK disable the watchdog reset by calling the `Cy_WDT_Unlock()` and `Cy_WDT_Disable()` functions during startup.

3.4.3.2 Debug logging enable

The EZ-USB™ FX 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 EZ-USB™ FX device (USB debug or J3 port on the KIT_FX20_FMC_001 DVK).

3.4.3.2.1 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 as 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 **921600 baud** with one stop bit and no parity bits.

3.4.3.2.2 USBFS (debug) initialization for logging

If the debug logs are to be output through the USBFS debug port, enable the USBFS block 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.

3.4.3.2.3 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 3 Logging configuration parameters

EZ-USB™ FX 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 EZ-USB™ FX. Currently supported values are USBFS debug port, CDC interface added to active USB configuration, SCB0-UART, SCB1-UART and SCB4-UART.
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. It is recommended that this flag be set to true.
recvEnabled	When a USBFS or USB CDC interface is used for logging, it is possible for the application to receive data from the host computer through the corresponding USB endpoint. recvEnabled flag should be set to true if the application needs to access the data received on these endpoints. Otherwise, the device will be configured to automatically discard any data received on the endpoint.
pCpuDw0Base	Pointer to DataWire-0 control register block. A valid pointer is required if the dbgIntfce selected is CY_DEBUG_INTFCE_USB_CDC.
pCpuDw1Base	Pointer to DataWire-1 control register block. A valid pointer is required if the dbgIntfce selected is CY_DEBUG_INTFCE_USB_CDC.
cdcEpIn	If the dbgIntfce is CY_DEBUG_INTFCE_USB_CDC, this field specifies the index of the IN endpoint used for sending the debug logs.
cdcEpOut	If the dbgIntfce is CY_DEBUG_INTFCE_USB_CDC, this field specifies the index of the OUT endpoint used in the interface.

EZ-USB™ FX pin	Function
bufCount	If the dbgIntfce is CY_DEBUG_INTFCE_USB_CDC, this field specifies the number of RAM buffers which are to be allocated for holding the debug log data.
pUsbCtxt	If the dbgIntfce is CY_DEBUG_INTFCE_USB_CDC, a valid pointer to the USBD stack context structure should be provided so that the logging module can access stack functionality.

The logger module is enabled by calling the `Cy_Debug_LogInit()` function. If the USBFS debug or USB 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.

3.5 USB block operation

For an introduction to the USB programming model on the EZ-USB™ FX device, see section 4.7: USB Programming Model of the EZ-USB™ FX Device Controller Architecture Technical Reference Manual (TRM)

As described in the Architecture manual, the USB High-Speed (2.x) and USB Super-Speed (3.x) functions are implemented by different hardware blocks on the EZ-USB™ FX device. Since the USB interface can only function in one of these modes at one time, a common USB stack is provided which manages the configuration, initialization and operation of the USB interface.

3.5.1 USB initialization

Steps for enabling and operating the EZ-USB™ FX USB interface:

1. Registration of Interrupt Service Routines (ISRs) for the 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.
   b) There are two interrupts associated with the USBSS block: ***lvds2usb32ss_usb32_int_o_IRQn*** and ***lvds2usb32ss_usb32_wakeup_int_o_IRQn***. The `Cy_USBSS_Cal_IntrHandler()` function provided in the firmware library needs to be called when either of these interrupts is asserted.
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 EZ-USB™ FX 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 EZ-USB™ FX 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_FX20_FMC_001 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.

3.5.2 USB enumeration

3.5.2.1 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.

In the default configuration, the USB stack assumes that the USB connection is made through a USB-C cable and that the EZ-USB™ FX device should identify the correct set of USB 3.2 TX/RX pins through CC pin based orientation detection. The USB-C cable plug orientation is done by measuring the voltage on the CC1 and CC2 pins. The voltage on the CC1 and CC2 pins are measured using ADC and the connection polarity is determined based on the pin on which a valid V_{Rd} voltage is detected. The orientation information is used to choose either the USB3TX1 and USB3RX1 pairs or the USB3TX2 and USB3RX2 pairs for operation in case of USB single-lane modes (USB Gen 1x1, USB Gen 2x1). In USB dual-lane modes (USB Gen1x2 and USB 2x2) orientation information is used to detect the configuration lane.

If the USB connection is being made through a USB Type-A or Type-B cable, or if the USB-C orientation is being detected externally by a USB Power Delivery controller, the `Cy_USBD_SelectConfigLane()` function can be used to skip the in-built orientation detection. This function allows the user to specify which USB lane should be used as the configuration lane (only active lane in the case of EZ-USB™ FX10 and EZ-USB™ FX5 devices). Once the USB configuration lane has been identified, the corresponding USB PHY instance is initialized as described in Section 4.7 of the TRM document.

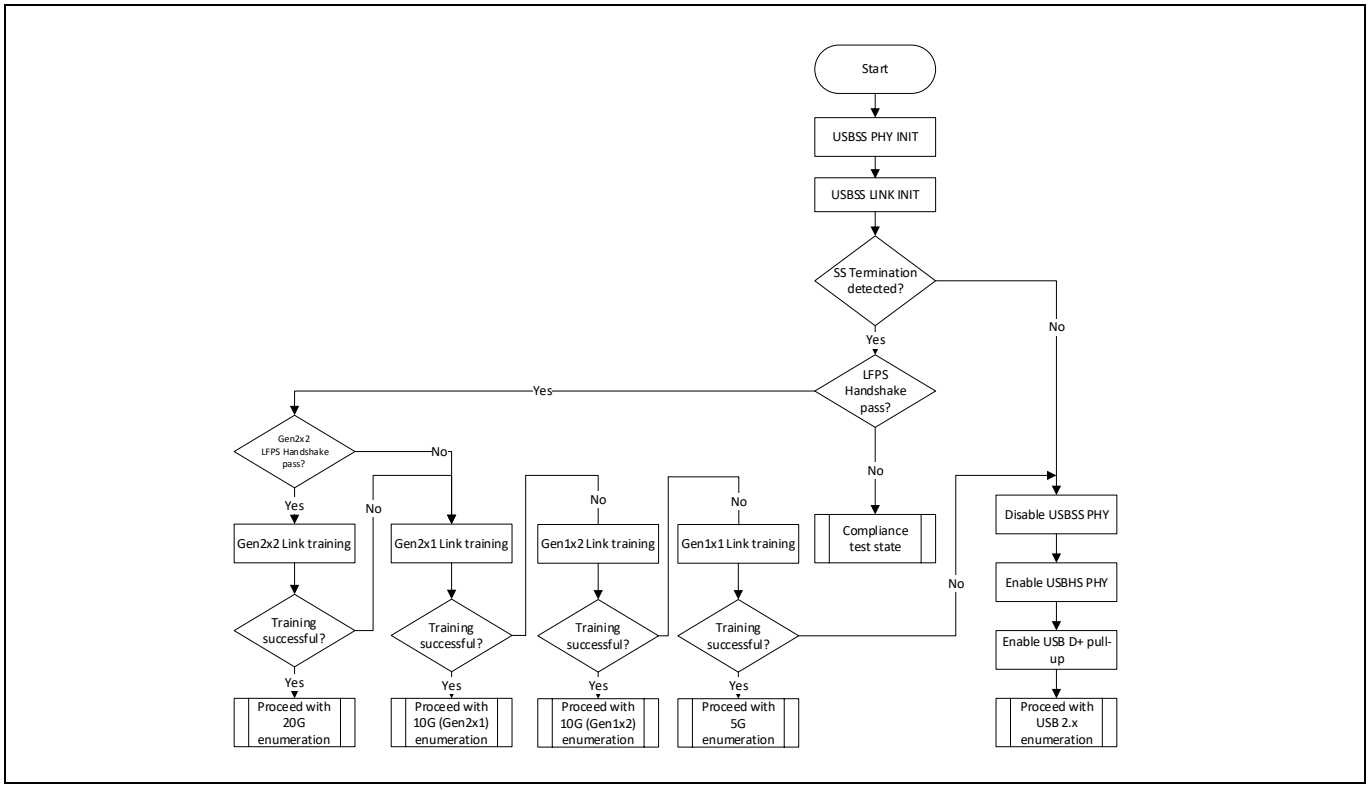

Figure 7 USB link initialization flow chart

3.5.2.2 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 only applicable 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 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. If the user wants to respond with a Zero Length Packet (ZLP) to an IN data transfer, `Cy_USBD_SendEgressZLP` () API should be used, followed by `Cy_USBD_SendAckSetupDataStatusStage` () to complete the status handshake.

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.

3.5.3 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.

3.5.3.1 USB 2.x power states

The USB 2.0 specification along with ECN updates define 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.

In a bus-powered system, the application firmware can take actions to save power by getting the device into the power-saving deep sleep mode once the link has entered the L2 state. This can be done by calling the `Cy_SysPm_CpuEnterDeepSleep()` API.

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 EZ-USB™ FX 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.

3.5.3.2 USB 3.x power states

As in the case of USB 2.x connection, the USB 3.2 specification defines a set of power states which apply in the case of 20Gbps, 10 Gbps and 5 Gbps USB connections.

- **U0 or Active State:** Data transfers between the host and USB device are only permitted in this state. While in the U0 state, the host is required is to send periodic ITP packets to the device and link keep-alive packets (LDN and LUP) are supposed to be exchanged periodically between each pair of link partners.
- **U3 or suspend state:** The U3 or suspend state is entered when the upstream host/hub sends a LGO_U3 request to the device and the device acknowledges the same with an LAU command. USB devices are not allowed to reject any entry into the U3 state. Once in the U3 state, the link is completely idle. To resume the link into the U0 state, either the host/hub or the device can initiate a U3 exit LFPS signal followed by a link recovery phase where both link partners exchange training sequences.
  The current version of the usbfxstack does not provide any callback notifications when the USB 3.x link enters the U3 state. It is possible to determine the current link state using the `Cy_USBSS_Cal_GetLinkPowerState()` API and identify whether the U3 state has been entered. No specific actions are required from the application for either entry into U3 state or exit back into the U0 state.
- **U1 State:** This is an intermediate power state which is close to the U0 state. Transitioning into the U1 state occurs when one link partner requests entry by sending an LGO_U1 command and the other link partner acknowledges by sending an LAU command. Both host/hub and device ports are allowed to reject entry into the U1 state by sending an LXU command. U1 state is exited when either of the ports sends a U1 exit LFPS burst followed by a link recovery phase where training sequences are exchanged.
  By default, all USB applications in the EZ-USB™ FX SDK make use of the ***Cy_USBD_LpmDisable()*** API to systematically reject all transitions into the U1 state. The EZ-USB™ FX device or firmware never initiates

entry into the U1 state either. With this configuration, the LPM related tests in the USB Chapter 9 compliance test suite are expected to fail.
The ***Cy_USBD_LpmDisable()*** call made in the SET_CONFIG callback function can be commented out to leave transitions into the U1 state enabled. The *mtb-example-fx20-usb-testapp* application supports a build-time switch to enable the USB low power mode handling which can be used to verify proper link power state transitions.

- **U2 State:** This is another intermediate power state which is close to the U3 state. All of the description provided for the U1 state is also applicable for the U2 state. The only difference is that the USB specification defines an implicit transition from the U1 state into the U2 state when the link has remained idle for a pre-defined timeout period.

3.5.4 USB disconnection and re-connection

If the EZ-USB™ FX 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 3.5.1.

3.6 Sensor Interface Port (SIP)[1] operation

The Sensor Interface Port (SIP) on the EZ-USB™ FX 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.

3.6.1 Sensor interface initialization

Configuring the SIP on the EZ-USB™ FX device involves multiple stages as follows:

1. **Dependency with USBHS block**: The SIP block uses some voltage and current references as well as the 480 MHz clock which is generated by the USBHS block on the EZ-USB™ FX Device. Hence, it is required that the USBHS block be initialized by calling `Cy_USB_USBD_Init()` function before any SIP initialization is performed.
2. **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_Ext()` or `Cy_LVDS_Init()` API. The parameters provided through the **lvdsConfig** >**phyConfig** structure are used for the PHY initialization.

Table 4 Sensor interface parameters

Parameter	Type	Description
`wideLink`	Boolean	Set to true if the SIP is being used in WideLink (single link with 32 LVCMOS lanes or 16 LVDS lanes) mode. If Narrow Link operation is selected, the two links need to be configured through separate API calls. The same lvdsContext structure should be passed when initializing both of the links.
`modeSelect`	Enumeration	Selects between LVDS and LVCMOS operation.
`dataBusWidth`	Enumeration	Selects the number of data lanes: 1, 2, 4, 8,12 or 16 for LVDS; 8, 16, 24 or 32 for LVCMOS

[1] Sensor Interface Port (SIP) refers to the LVDS/LVCMOS interface for the FX device

Parameter	Type	Description
`interfaceClock`	Enumeration	Expected interface clock frequency. Only a set of pre-defined discrete values are supported (74.25 MHz, 148.5 MHz, 156.25 MHz and 625 MHz for LVDS and 80 MHz, 100 MHz and 160 MHz for LVCMOS).
`interfaceClock_kHz`	uint32_t	Expected interface clock frequency in KHz units. When the Cy_LVDS_Init_Ext() function is used, this field can be used to specify any interface clock frequency within the valid range defined in the datasheet. Please note that this parameter is ignored and interfaceClock field will be used if the Cy_LVDS_Init() function is used to initialize the block.
`gearingRatio`	Enumeration	Gearing ratio for data transfers
`clkSrc`	Enumeration	Select clock used for the data link and GPIF operation: select from between interface clock, EZ-USB™ FX system level clk_hf[4] clock and 480 MHz clock derived from USB2 (USB-HS) block.
`clkDivider`	Enumeration	Select division factor used to reduce 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	Only applicable for LVDS: Specifies the data byte value which will be sent on the control and data lanes for PHY training purposes. Using the default value of 0x9C is recommended.
`linkTrainingPattern`	uint32_t	Specifies the data that will be sent on the data lanes to enable the link to adjust for any inter-lane skew and align the incoming data. Link training is required in all LVDS modes and in the LVCMOS DDR mode of operation.
`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	Whether the two SIP links should be mutually connected to enable link level data loopback function. This capability is used so that an internal generated data pattern can be applied on one of the RX data links to test the data streaming function.
`isPutLoopbackMode`	Bool	When loopbackModeEn is set to true, selects the SIP link which serves as the data receive port. The other link will be configured as the loopback data source. It is recommended that link #0 is used as data receive port and link #1 is used as the loopback data source.
`slaveFifoMode`	Enumeration	Used in LVCMOS mode to select between 2-bit Slave FIFO operation and 5-bit Slave FIFO operation.

Parameter	Type	Description
`dataBusDirection`	Enumeration	Specifies whether the data bus which is part of the link is being used in input only, output only or bi-directional modes.
`lvcmosClkMode`	Enumeration	Specifies whether the interface clock is input to the EZ-USB™ FX device or output from it. Output mode is only supported in LVCMOS configuration.
`lvcmosMasterClkSrc`	Enumeration	Specifies the source of the interface clock in cases where it is being output by the EZ-USB™ FX device.

3. **Perform PHY training**: PHY training is required in the LVDS mode of operation to ensure that the LVDS receiver is properly sampling the data sent by the master. The master is expected to continuously send a known data pattern (matching the `phyTrainingPattern` config parameter) on the control and all data lanes for at least 50 us. The SIP on EZ-USB™ FX needs to be prepared to go through PHY training by calling the ***Cy_LVDS_PhyTrainingStart()*** function. The `phyTrainingPattern` should be sent by the LVDS master for at least 50 us after this API has been called on the firmware side.

   When the window of time available for PHY training is short, it is recommended that a signal be sent from the EZ-USB™ FX20 device to the FPGA indicating that PHY training pattern should be started. This signal can be a GPIO asserted or an I2C/UART transfer to the FPGA, and should be triggered just before the Cy_LVDS_PhyTrainingStart() function is called.

   Note: When the LVDS interface is being used in WideLink mode, the PHY training pattern needs to be sent on both P0CTL and P1CTL lanes, even though P1CTL lane will not be used for data transfers.
4. **Perform LINK training:** Link training is used to ensure that data received by EZ-USB™ FX20 from each of the lanes is properly aligned. This process is required when the Sensor Interface Port is used with LVDS interface, as well as when a WideLink LVCMOS Port (24 or 32 data lanes) is used in LVCMOS DDR mode. Link training is not required if the LVCMOS port is used with SDR clock or when the LVCMOS ports are configured as separate NarrowLink ports (8 or 16 data lanes).

   Link training is performed by sending a 4-byte data sequence matching the `linkTrainingPattern` config parameter. If LVDS interface is used, the training sequence needs to be sent on each of the control and data lanes. If LVCMOS DDR interface is used, the training sequence should be sent in parallel across all the data lanes. The link training pattern can be sent once any time after PHY training has been completed.

   A PHY event callback with event type of CY_LVDS_PHY_LNK_TRAIN_BLK_DET will be received once the link training process has been completed without error. If this event is not being received within the expected time, the `Cy_LVDS_GetLinkTrainingStatus()` API can be used to check the status of link training.
5. **GPIF state machine initialization:** The SIP data link operation on the EZ-USB™ FX 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.

3.6.1.1 GPIF state machines

Since the LVDS/LVCMOS interfaces of the EZ-USB™ FX device are expected to be driven by an FPGA, we only make use of 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 EZ-USB™ FX code examples.

3.6.1.1.1 GPIF config for LVDS interface

When the SIP is used in LVDS mode, the control information is exchanged through the dedicated LVDS control lane and there is no protocol level need for any LVCMOS control signals. Hence, a simplified GPIF state machine is used to control the link operation in this case.
See section 25.4 of the EZ-USB™ FX Device Controller Architecture Manual for details of the LVDS command protocol.
The state machine and configuration used for this interface can be found in the cy_gpif_header_lvds.h file in the code examples like mtb-example-fx20-uvc-uac and mtb-example-fx20-usb3vision.

3.6.1.1.1.1 Interface signals

This configuration uses the Link #0 of the SIP in either Narrow Link or Wide Link mode. Selection between Narrow Link and Wide Link can be done through the WL_EN pre-processor macro setting.

No LVCMOS control signals are necessary for the operation of the LVDS interface. In the provided code examples, P0CTL6 pin from EZ-USB™ FX20 is used as PHYRDY (or Link Ready) trigger signal to the FPGA requesting that PHY training sequence should be started. This pin is over-ridden as a GPIO and is asserted high by the firmware just before the Cy_LVDS_PhyTrainingStart() function is called. The FPGA is expected to drive the PHY training sequence for at least 50 us and then send the link training sequence when it see the pin going high.

In the provided code examples, P0CTL5 pin is used as a DMA ready flag (FLAGA) from EZ-USB™ FX20 to the FPGA. This pin will be asserted high to indicate that device is ready to receive data from the FPGA and will be driven low at times when the interface is not ready to receive data. The LVCMOS control signal configuration used in the state machine is summarized in Table 2.

Table 2 LVCMOS control signal usage in LVDS Receiver State Machine

EZ-USB™ FX pin	Function	Description
P0CTL6	PHYRDY0	This signal is driven high when the LVDS block has been powered up and the device is ready to start the PHY training process. This signal is used when Sensor Interface Port-0 is being used in NarrowLink or WideLink modes.
P1CTL6	PHYRDY1	This signal is driven high when the LVDS block has been powered up and the device is ready to start the PHY training process. This signal is used when Sensor Interface Port-1 is being used in NarrowLink mode.
P0CTL5	DMA_RDY_CUR0	Active low DMA ready signal which indicates whether EZ-USB™ FX20 is ready to receive data on the currently selected thread under Port-0.
P1CTL5	DMA_RDY_CUR1	Active low DMA ready signal which indicates whether EZ-USB™ FX20 is ready to receive data on the currently selected thread under Port-1. This is only applicable when Port-1 is being used in NarrowLink mode.
P0CTL0	INIT_DONE	This is a signal which is driven high by the GPIF state machine to indicate that the link initialization has been completed and device is ready for data transfers from the FPGA side.

3.6.1.1.1.2 GPIF state machine

Figure 8 shows the state machine used by this LVDS receiver implementation. The state machine is a simple one which will sample the data on the LVDS data lanes and write into the DMA buffer whenever the opcode sent on the control lane matches a Data command and the DMA buffers are ready to accept data.
The thread and socket selection in this application are done through the STAD and SSAD commands received on the control lane. Before the master sends data into any thread, it is expected to verify that the corresponding DMA_RDY signal is being asserted (low) by the EZ-USB™ FX device.

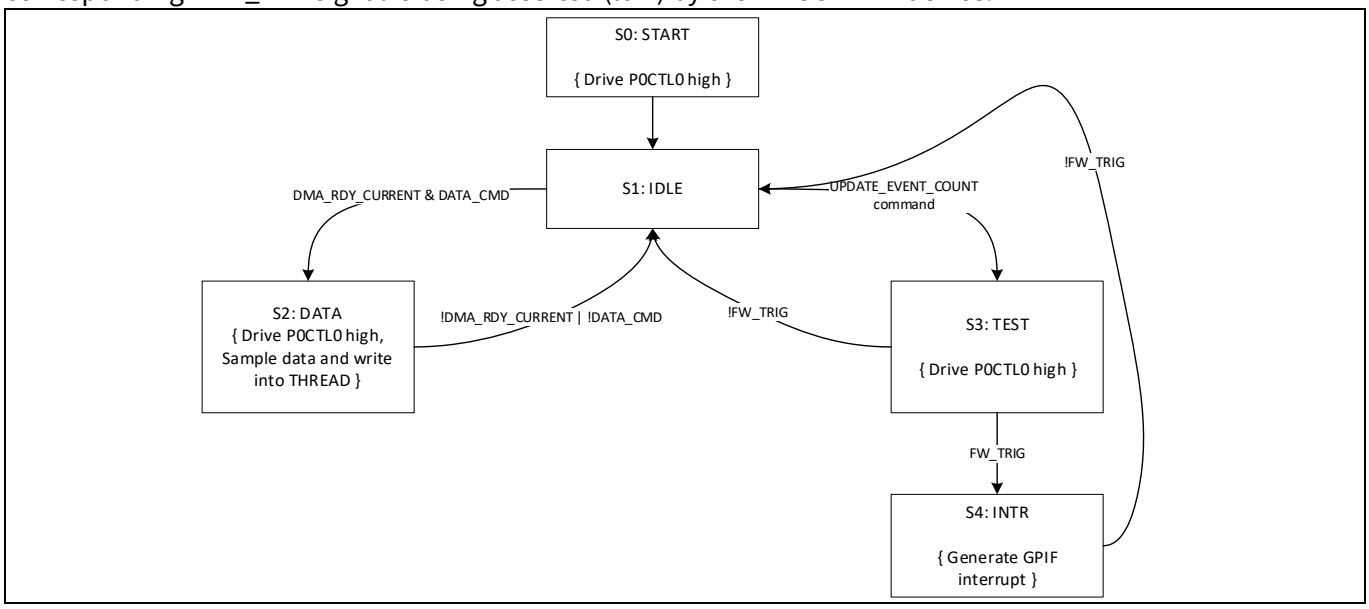

Figure 8 GPIF state machine used by LVDS receiver application

3.6.1.1.2 GPIF config for LVCMOS receiver application

The *mtb-example-fx20-slave-fifo-2bit* and *mtb-example-fx20-uvc-uac* applications use the 2-bit Slave FIFO interface when the sensor interface ports are used with LVCMOS interface. This is a standard LVCMOS configuration which allows the FPGA master to select and perform data transfers to one out of four 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 makes use of only two sockets to receive data from the FPGA and hence the most significant bit of the address (A1) must be set to 0. The lower address bit (A0) can be used to select between the two sockets when transferring data.

3.6.1.1.2.1 GPIF state machine

Figure 9 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 EZ-USB™ FX device.

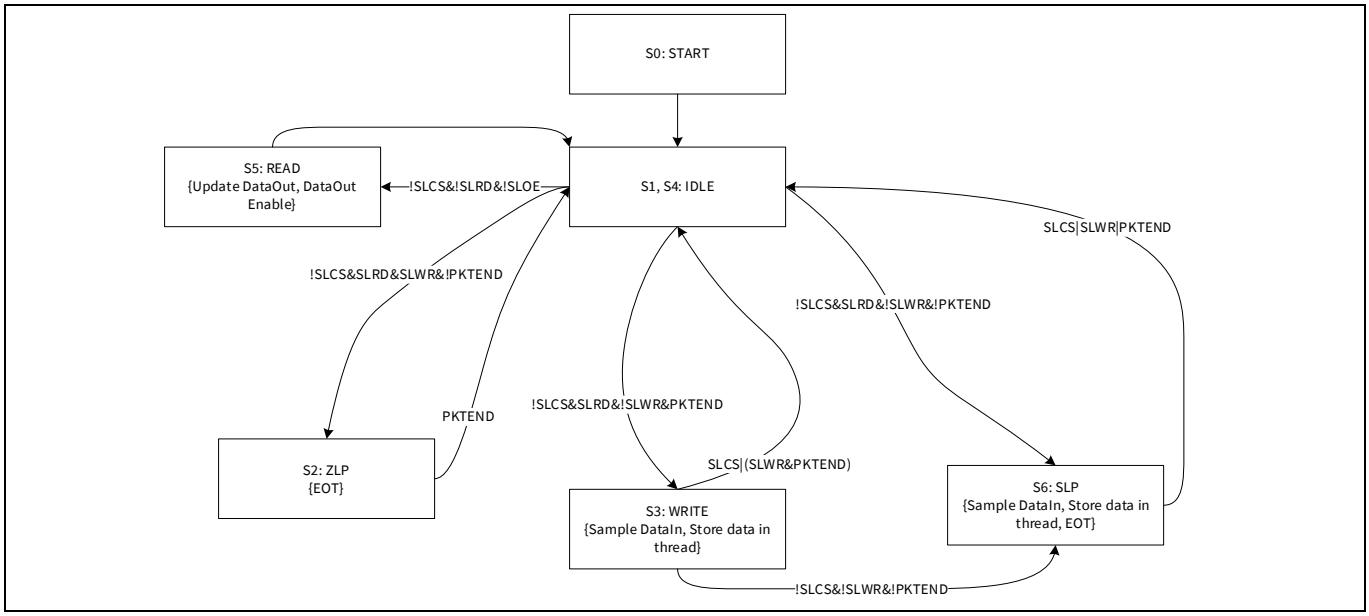

Figure 9 GPIF state machine used by 2-bit slave FIFO application

3.6.1.1.2.2 Interface signals

Table 5 Control signal usage in LVCMOS 2-bit Slave FIFO state machine

EZ-USB™ FX pin	Function	Description
P0CTL0	SLCS#	Active-low Chip Select signal. Should be asserted low by the master FPGA when communicating with EZ-USB™ FX.
P0CTL1	SLWR#	Active-low Write Enable signal. Should be asserted low by the master FPGA when sending any data to the EZ-USB™ FX. 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 PKTEND# signal to indicate that this data ends the transfer, and the DMA operation should be terminated.
P0CTL2	SLOE#	Active-low Output Enable signal.
P0CTL3	SLRD#	Active-low Read Enable signal. Not used in this application as data is only being received by EZ-USB™ FX
P0CTL4	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 with only SLCS# low and SLWR# high to complete the DMA transfer with zero bytes of data.
P0CTL5	Flag A	Active-low DMA ready indication for currently addressed/ active thread.
P1CTL9	A0	LS bit of 2-bit address bus used to select thread
P1CTL8	A1	MS bit of 2-bit address bus used to select thread

3.7 DMA datapath operation

Once all the required hardware blocks on EZ-USB™ FX 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.

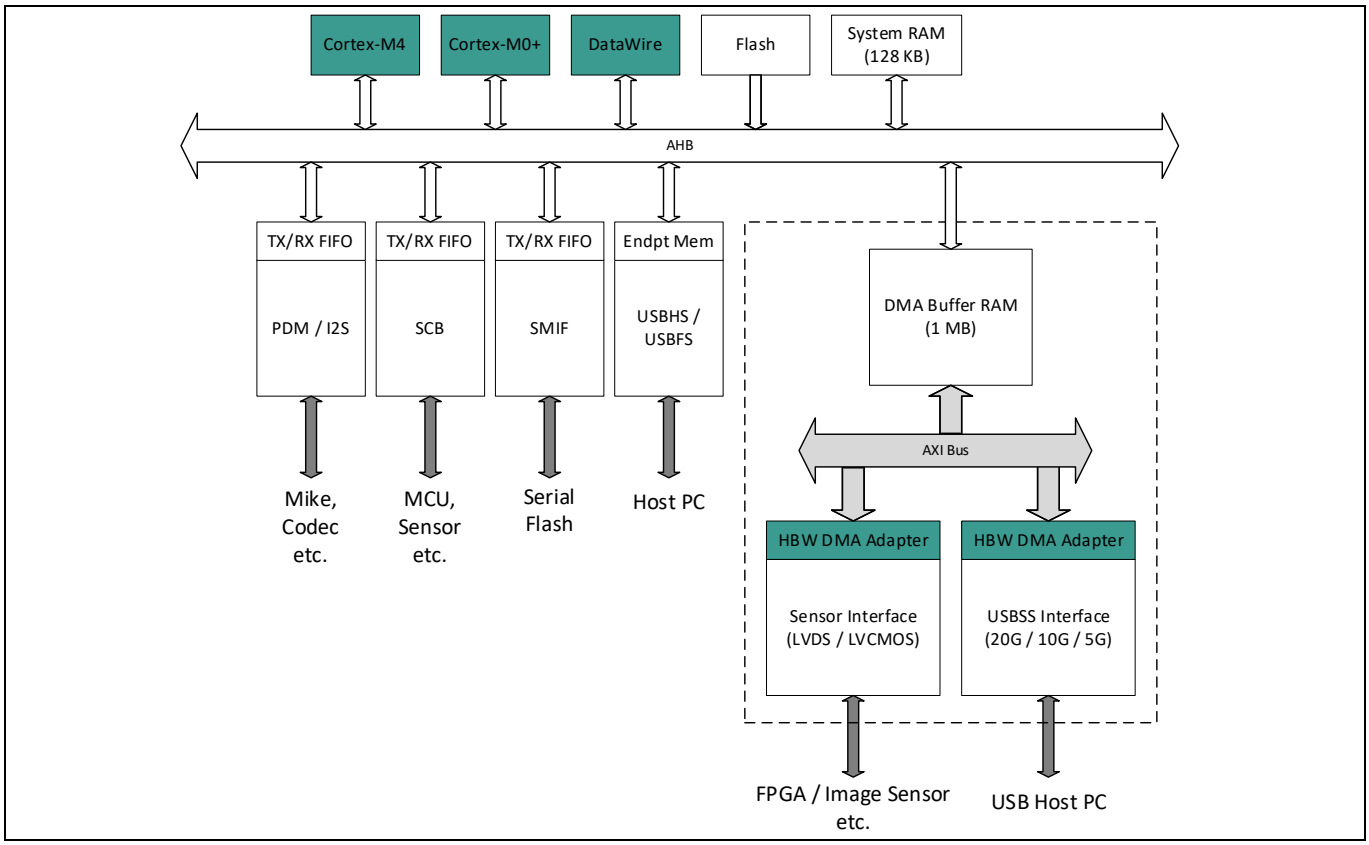

Figure 10 Datapaths within EZ-USB™ FX device

The EZ-USB™ FX 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 and the USBSS (3.x) port. 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, Serial Controller Block (SCB), Serial Memory Interface (SMIF), PDM block, and I2S block.

  The scope of each of these DMA data paths is shown in Figure 9.

  The High BandWidth DMA is dedicated to moving data between DMA Buffer RAM and the two interfaces: Sensor Interface (LVDS/LVCMOS) and USBSS. Any data to and from these interfaces goes through the DMA Buffer RAM, which serves an elastic buffer. This means that any data to be sent through the LVCMOS or USBSS interfaces first needs to be copied into the DMA buffer RAM and then the DMA should be initiated. Similarly, any data received through the LVDS, LVCMOS or USBSS interfaces must 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 High-Speed (2.x) connection, the High BandWidth DMA should be used to copy the data from the Sensor Interface Port into the Buffer RAM and DataWire should be used to send the received data to the USBHS block.

3.7.1.1 High BandWidth DMA (HBDma) programming

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.

3.7.1.1.1 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 are four DMA adapter interrupt sources for which interrupt handlers 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 Sensor Interface Port (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**.
   b) **lvds2usb32ss_lvds_dma_adap1_int_o_IRQn:** Corresponds to interrupts associated with any of the second 16 DMA sockets in the Sensor Interface Port (effectively DMA interrupts associated with LVDS link #1). 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_1**.
   c) **lvds2usb32ss_usb32_ingrs_dma_int_o_IRQn:** Corresponds to interrupts associated with USB sockets through which EZ-USB™ FX20 receives data associated with OUT endpoints. 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_USB_IN**.
   d) **lvds2usb32ss_usb32_egrs_dma_int_o_IRQn:** Corresponds to interrupts associated with USB sockets through which EZ-USB™ FX20 sends data associated with IN endpoints. 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_USB_EG**.

3.7.1.1.2 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 EZ-USB™ FX 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.

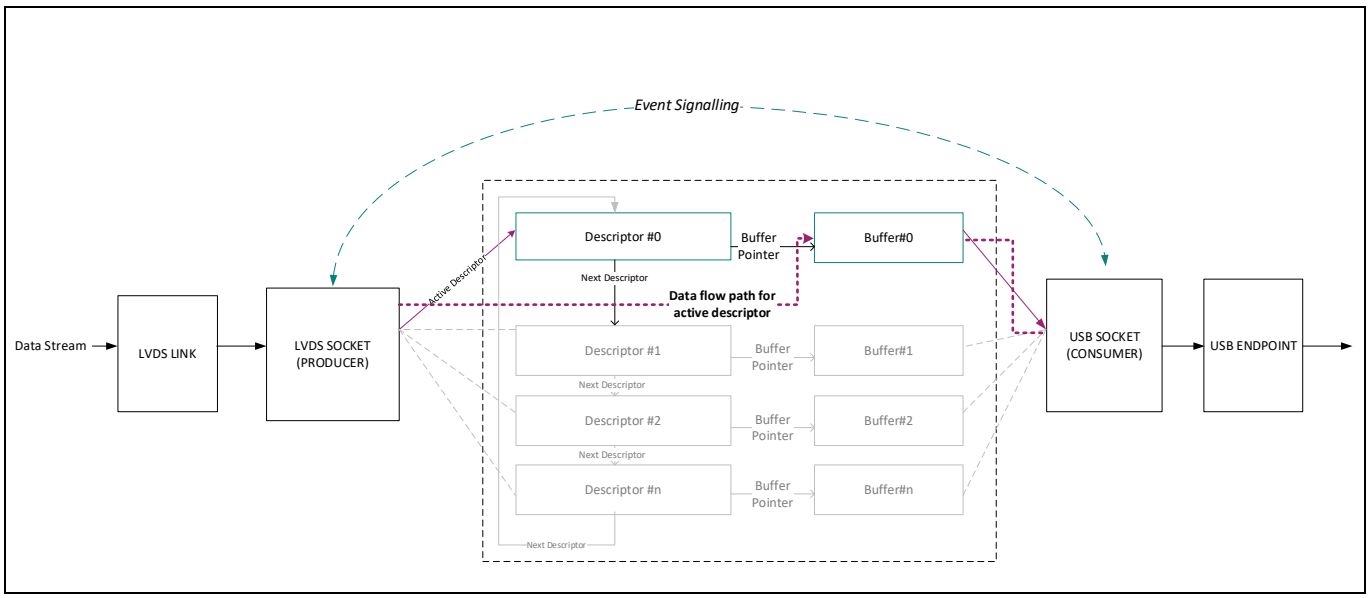

Figure 11 HBWSS DMA channel construct

A socket represents one end of a data stream which flows through the AXI DMA framework. The Sensor Interface Port on EZ-USB™ FX supports 32 sockets which means that 32 independent data streams can be managed across this port. The USBSS port on EZ-USB™ FX supports 16 sockets in each direction which means that 16 independent input data streams and 16 independent output data streams can be supported. 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 EZ-USB™ FX device are called Producer or Ingress sockets. Sockets through which data exits the EZ-USB™ FX device are called Consumer or Egress sockets.

Three types of channels can be created:

1. IP to IP channels which move data between the LVDS/LVCMOS and USB Interfaces. In this case, the channel will have valid producer and consumer sockets.
2. IP to Memory channels which receive data through LVDS/LVCMOS or USB interface and store it in memory. In this case, the channel will only have valid producer sockets.
3. Memory to IP channels which take data from memory and send it out through LVCMOS or USB interface. In this case the channel will only have valid consumer sockets.

A set of convenience API is provided in the SDK for creation, management, and operation of these DMA channels. A summary of the APIs is provided in Table 6.

Table 6 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 EZ-USB™ FX 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).
`Cy_HBDma_Channel_SendData`	This is a one-shot transfer function which allows firmware to initiate output of data which is currently present in the RAM buffer pointer provided. This function is expected to be used without enabling the DMA channel previously and leaves the channel in the idle state after the requested transfer has been completed.
`Cy_HBDma_Channel_WaitForSendCplt`	This function waits until a previously requested one-shot data transmission operation has been completed.
`Cy_HBDma_Channel_ReceiveData`	This is a one-shot transfer function which allows firmware to initiate reception of data into the RAM

DMA manager API	Description
	buffer pointer provided. This function is expected to be used without enabling the DMA channel previously and leaves the channel in the idle state after the requested transfer has been completed.
`Cy_HBDma_Channel_WaitForReceiveCplt`	This function waits until a previously requested one-shot data receive operation has been completed.
`Cy_HBDma_Channel_GetBufferInfo`	This function returns the list of RAM buffers which have been allocated for the specified DMA channel. This list of buffer pointers can be used in cases where the same RAM buffers are being set up for other operations such as DataWire based data transfers.

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 indicates 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.

IP to IP channels can either work under continuous firmware monitoring through the callback notifications and API calls. This method can be used if the application requires the ability to add, modify or delete the data which is being transferred through the channel. For example, this method can be used to receive plain video data from an image sensor through the LVDS socket, add USB Video Class specific headers into the data stream and forward it on the USB IN endpoint.

If no data modification is required on an IP to IP data path, all interrupts and callbacks can be disabled and the producer socket and consumer socket connected to each other for automatic synchronization. This method of transfer will yield the highest data throughput through the EZ-USB™ FX device.

3.7.1.2 DataWire DMA programming

EZ-USB™ FX 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).

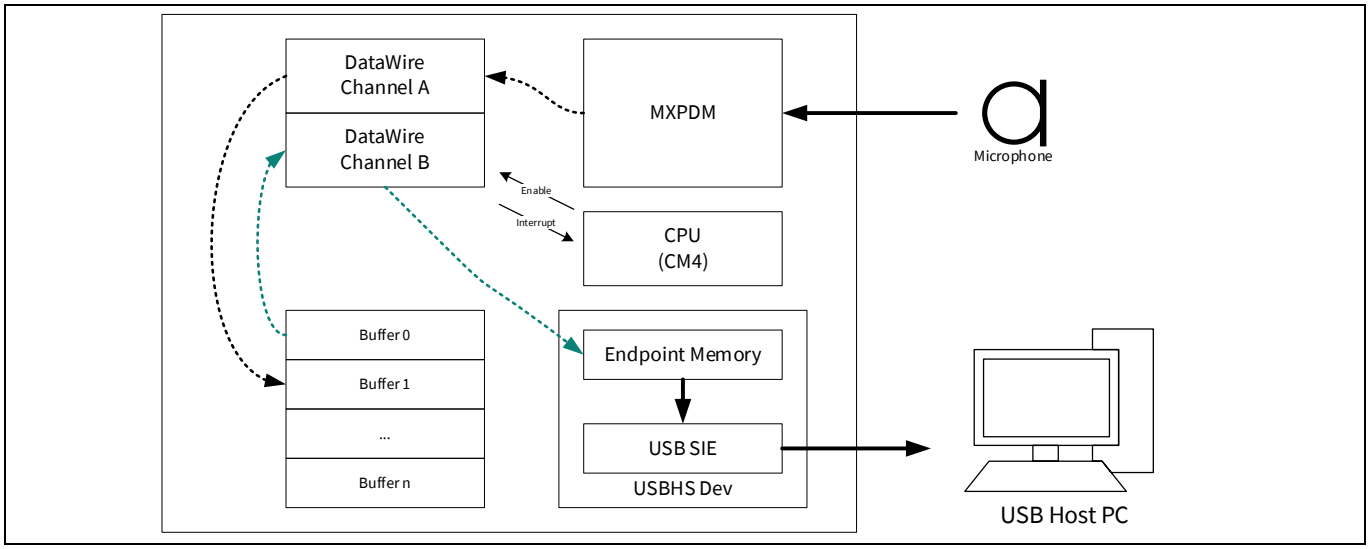

Figure 12 PDM microphone to USBHS transfer using DataWire

Figure 12 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.

3.7.1.2.1 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 EZ-USB™ FX as shown in Table 7. If any of the recommended transfers are not in use in a given application, the respective channel can be used for other purposes.

Table 7 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

3.7.1.2.2 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 EZ-USB™ FX 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. Trigger connections between the DataWire channels and other peripherals are established using the `Cy_TrigMux_Connect()` API.

3.7.1.2.3 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.

3.7.2 DMA transfer use cases

This section provides details on the DMA data flow in some of the typical EZ-USB™ FX use cases.

3.7.2.1 LVDS to USBSS data transfer

In this case, the entire end-to-end data transfer can be performed using a single High BandWidth DMA channel. CPU/Firmware intervention is not required in this case if no data modifications are to be performed.

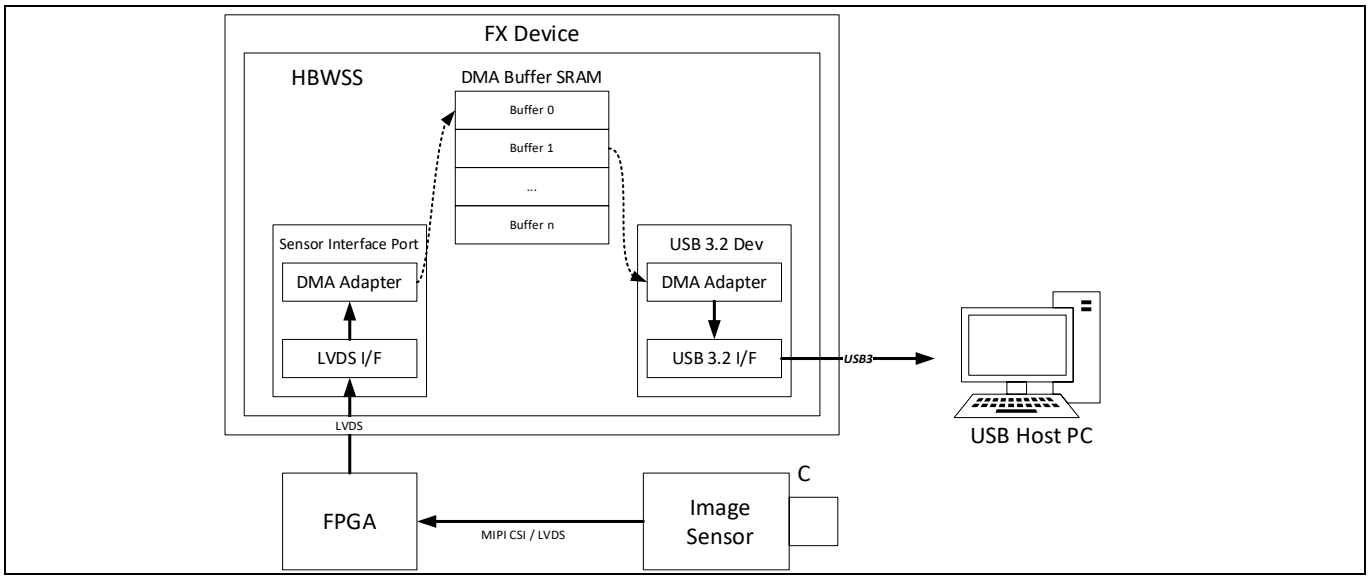

Figure 13 LVDS to USBSS data transfer using high bandwidth DMA

3.7.2.2 LVCMOS to USBHS data transfer

In this case, we 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.

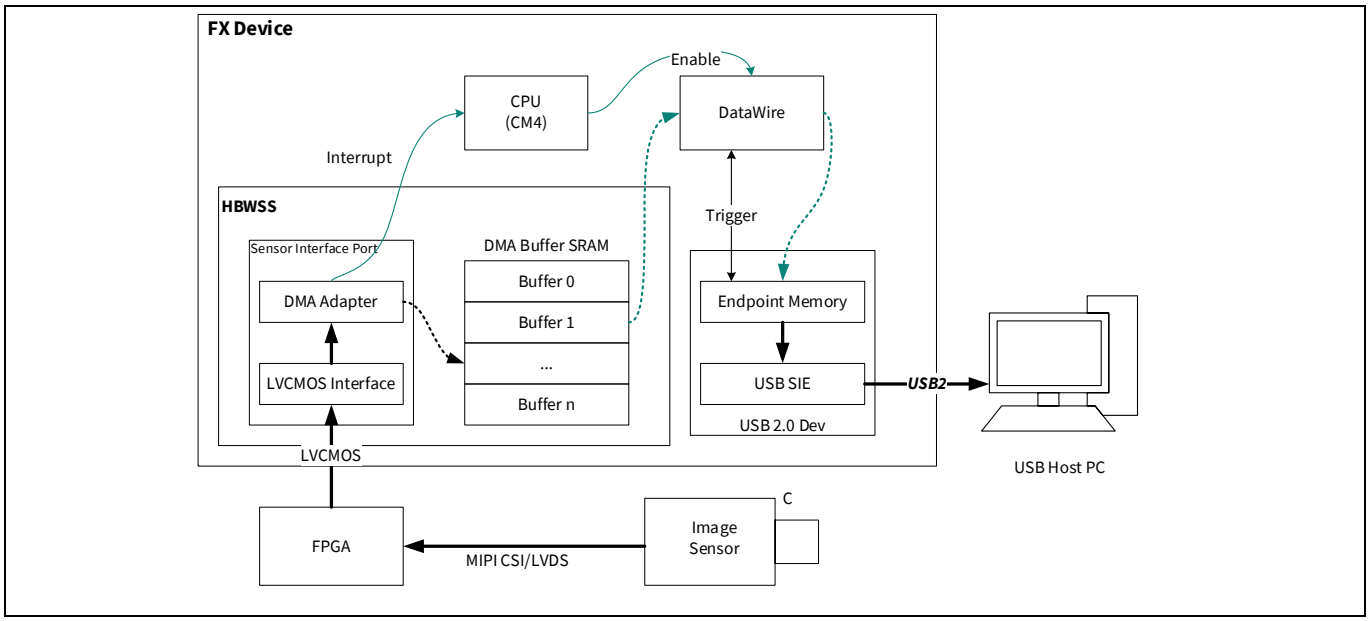

Figure 14 LVCMOS to USBHS data transfer

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.

3.7.2.3 PDM to USBHS data transfer

In this case, the transfer is done using only DataWire channels. The data flow is shown in Figure 12 and the transfer handling is described in Section 3.7.1.2.

4 Getting started with Modus ToolBox for EZ-USB™ FX devices

4.1 Installation

See Dependencies section for instructions on how to install Modus ToolBox for EZ-USB™ FX devices.

Beginning with the ModusToolbox™ tools package version 3.4.0, the Eclipse IDE for ModusToolbox™ is no longer included by default. Instead, it is a separate package that you can install via the ModusToolbox™ Setup program. The Setup program is located on our website.

Steps on how to open, build and program an EZ-USB™ FX device code example using the Eclipse IDE for ModusToolbox™ is shown are shown in the subsequent sections.

4.2 Launching Eclipse IDE

To launch the Eclipse IDE:

- On Windows, select the Eclipse IDE for ModusToolbox™ <version> item from the Start menu.
- For other operating systems, run the "modustoolbox" executable file.
- You can also launch Eclipse IDE from the ModusToolbox dashboard for all operating systems.

When launching the Eclipse IDE, it provides an option to select or create the workspace location on your machine. This location is used by the IDE for creating and storing the files as part of application creation for a particular platform. The default workspace location is a folder called "mtw" in your home directory. You may add additional folders under the "mtw" folder or to choose any other location for each workspace.

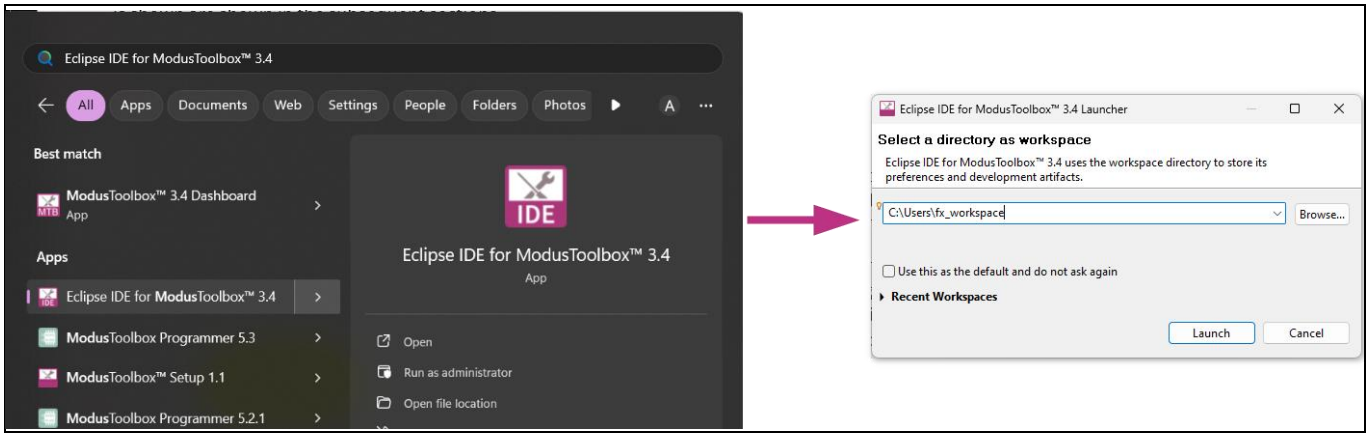

Figure 15 Launch and open a workspace

4.3 Opening an existing EZ-USB™ FX device code example

Click the New Application link in the Eclipse IDE Quick Panel. (or use File > New > ModusToolbox™ Application).

These commands launch the Project Creator tool, which provides several applications for use with different development kits or board support packages (BSPs).

Choose **USB BSP** > **KIT_FX20_FMC_001** and click on **Next** to open the Select Application page. See Figure 16 for more details.

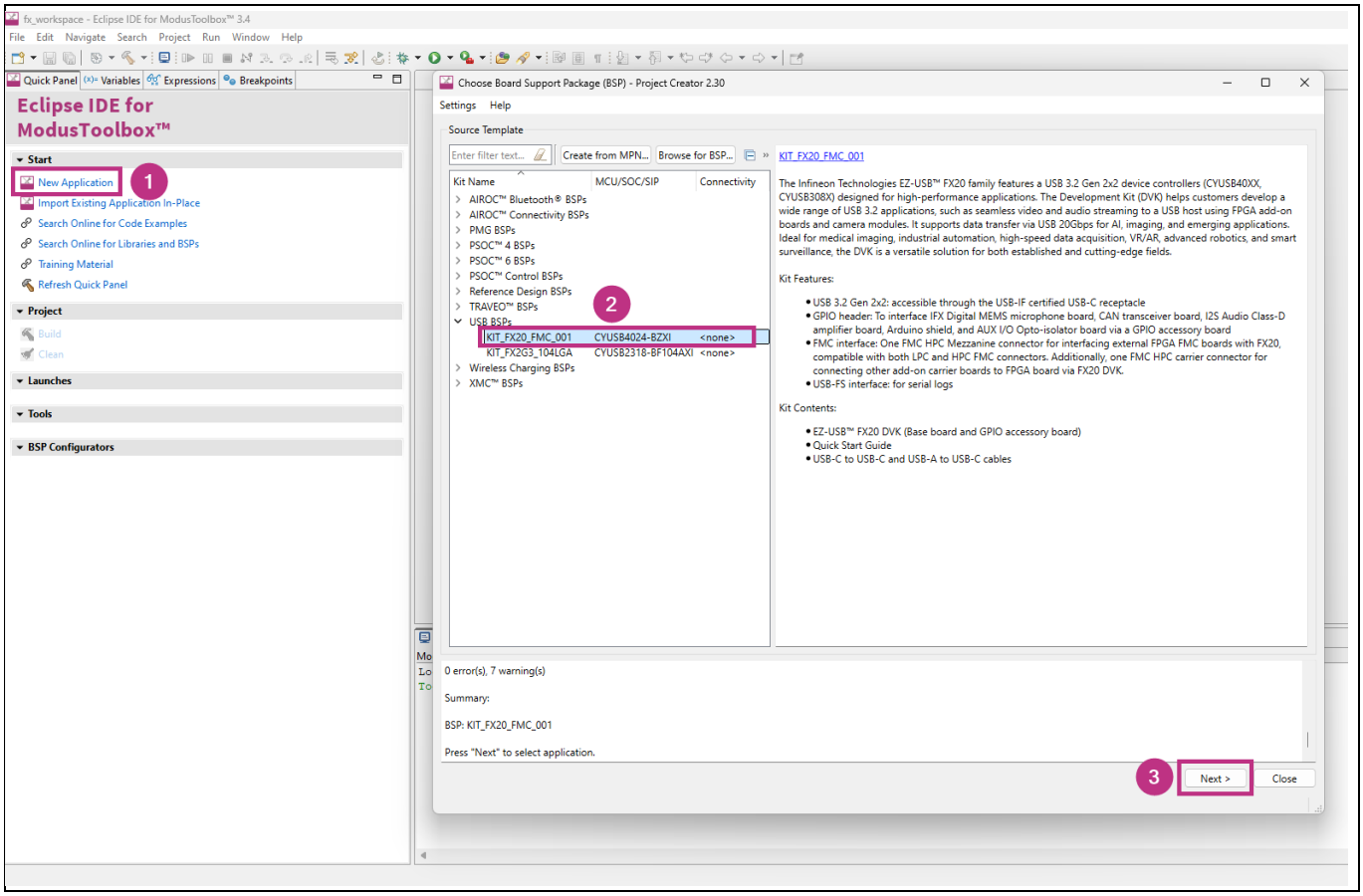

Figure 16 Choose the EZ-USB™ FX20 FMC kit Board Support Package (BSP)

This page lists various applications available for the selected EZ-USB™ FX kit. As an application is selected, a description displays on the right. Multiple applications for the selected BSP can be chosen by enabling the check box next to the applicable applications.

As an example, UVC-UAC code example has been selected in Figure 17.

If desired, type a name for the selected application and/or BSP. Do not use spaces.

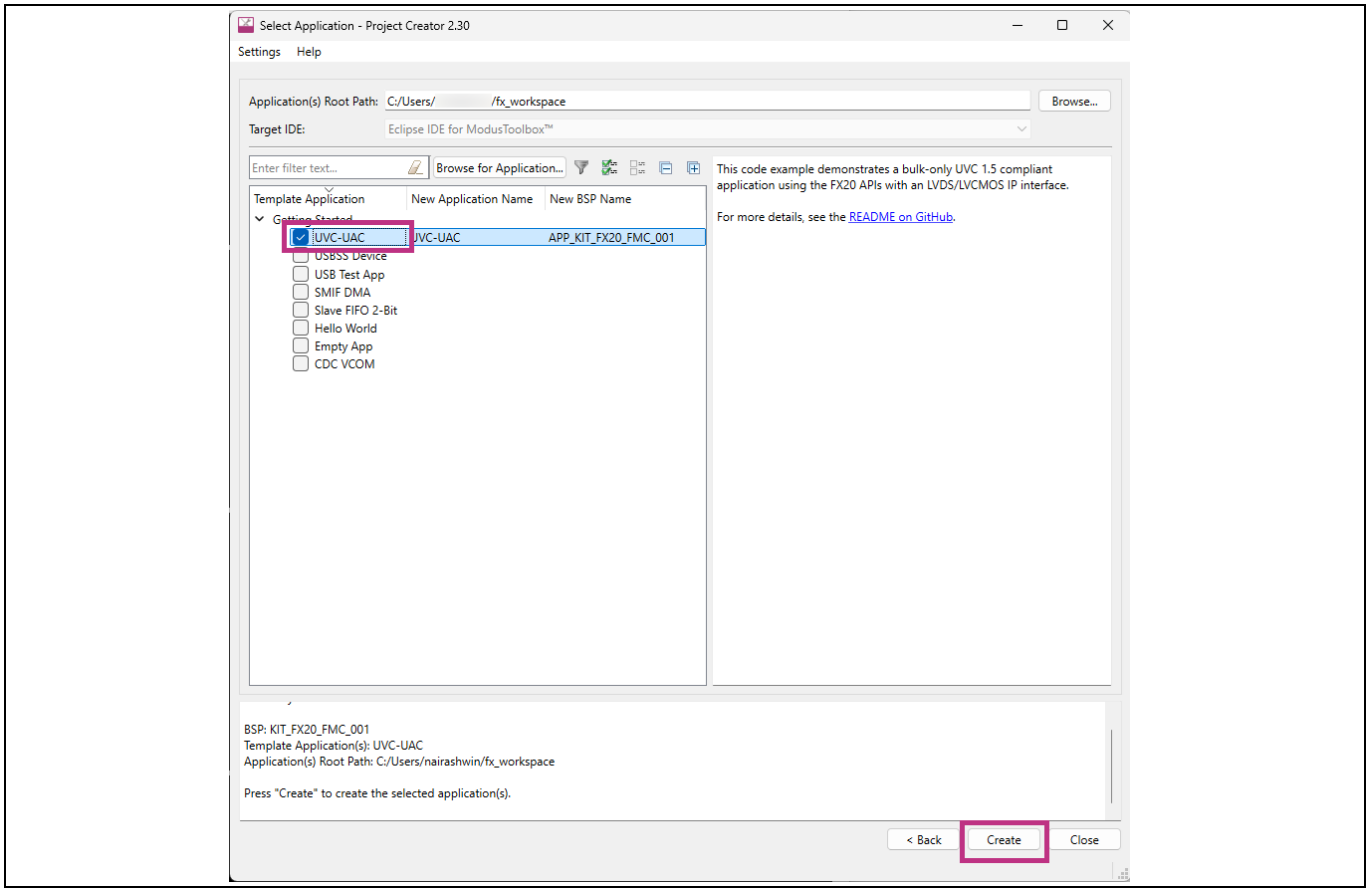

Figure 17 Select code example and click on create

Click **Create** to begin the application creation process.

The application creation process performs a **git clone** operation and downloads the selected application from the code example's GitHub page. Depending on the selected application, this process can take several minutes.

When complete, the Project Creator tool closes automatically if there are no errors or warnings. If there are warnings only, click **Close** and the application will open in the IDE.

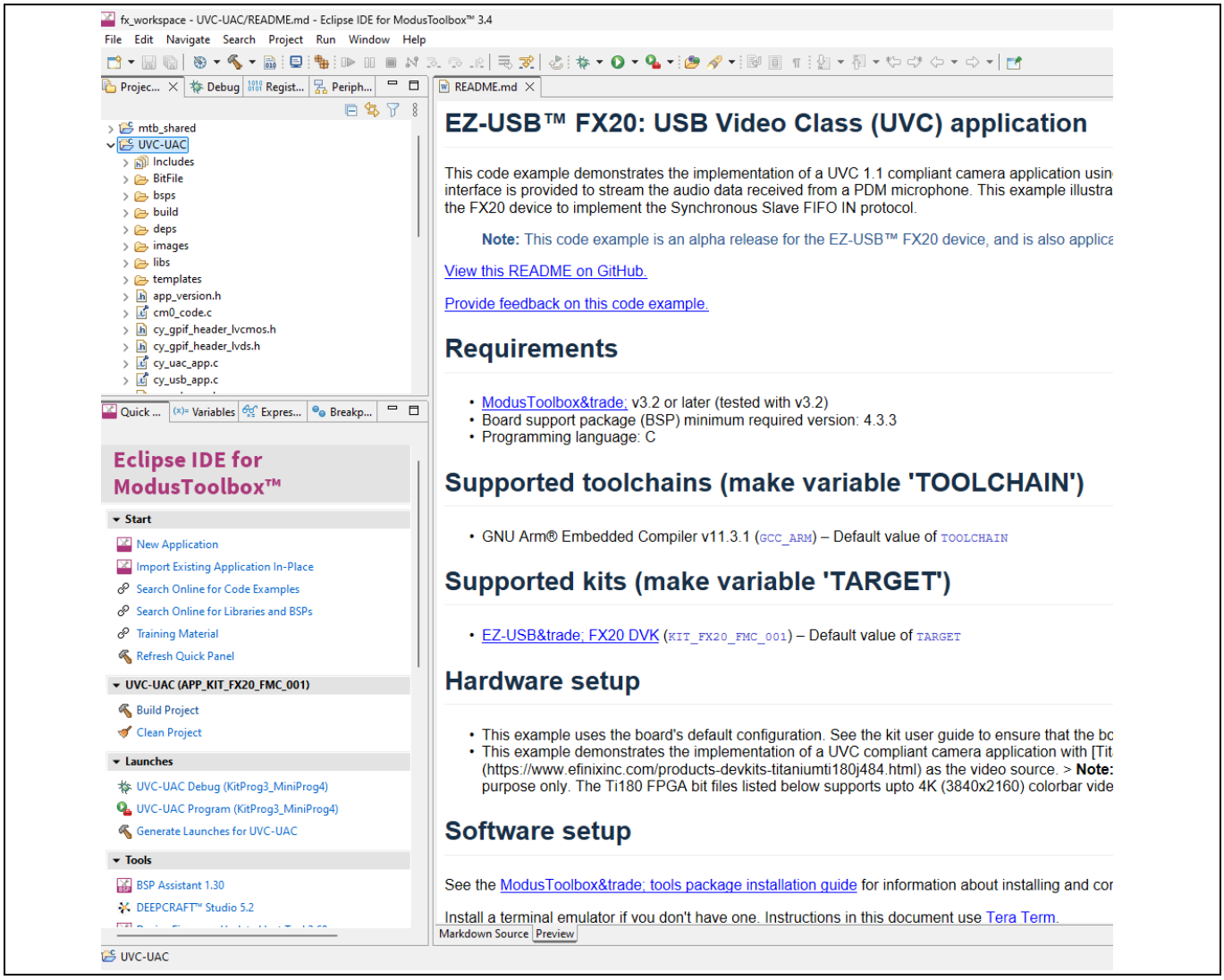

Figure 18 Code example created in Eclipse IDE

4.4 Build application

After loading an application, build it to generate the necessary target binary files.

Select the project. Then, in the **Quick Panel**, click the **Build Application** link.

The project console displays the build progress messages as shown in Figure 19.

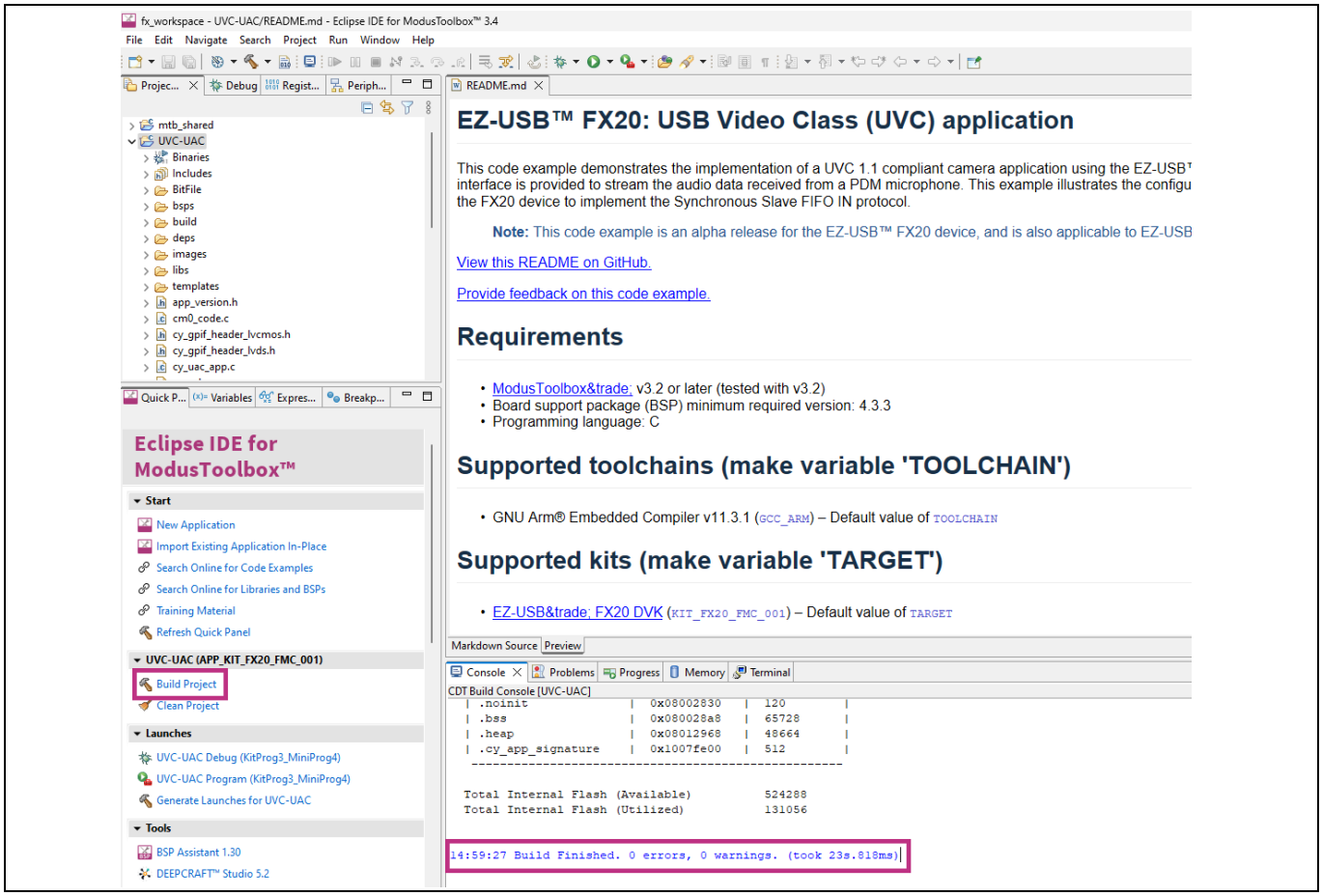

Figure 19 Build project

4.5 Program application

The compiled binary files can be programmed in two ways to the EZ-USB™ FX device.

4.5.1 Program application using EZ-USB™ FX control center

The compiled binary will be generated in ***<Code example name>/build/APP_KIT_FX20_FMC_001/Release.*** Figure 20 shows the generated hex file for the UVC-UAC code example.

EZ-USB™ FX Control Center tool can be used to program this hex file to the EZ-USB™ FX bootloader device.

See Programming using the USB bootloader section for more details.

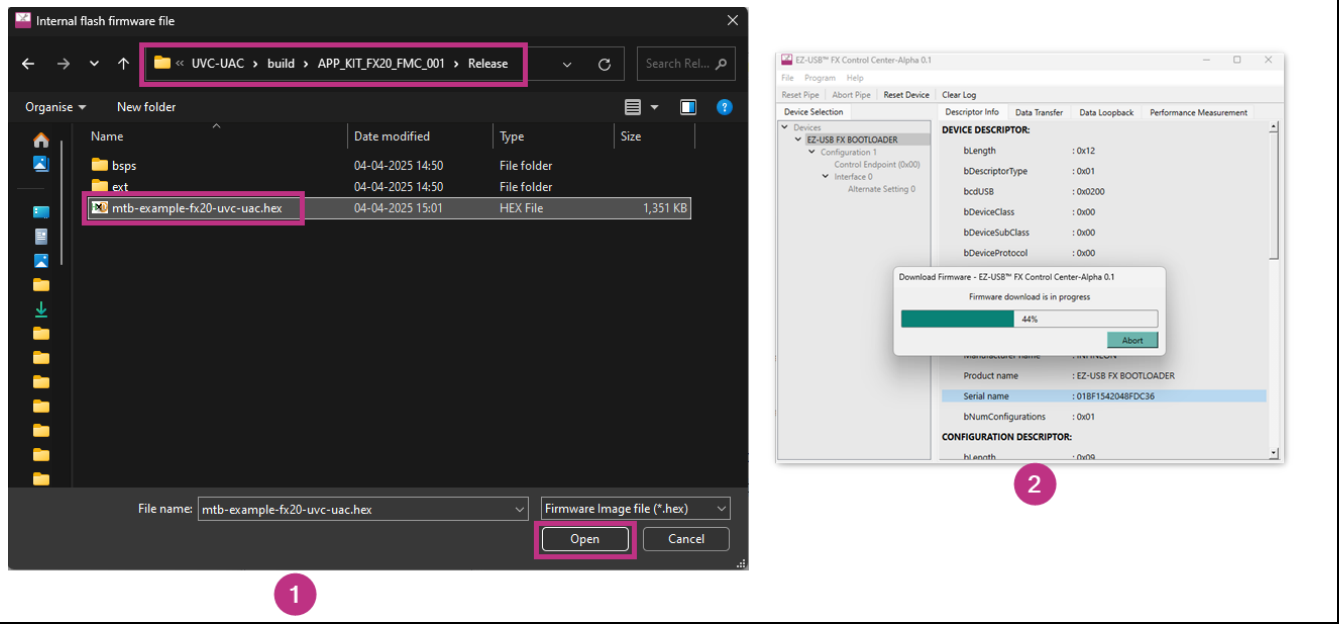

Figure 20 Generated hex file for UVC-UAC code example

4.5.2 Program application directly from ModusToolbox

Click on **UVC-UAC Program** link in the **Quick Panel** to program the code example to the kit directly from Modus Toolbox using OpenOCD.

Note: KitProg3 or Miniprog4 should be connected to the kit's SWD pins for directly programming the kit from ModusToolbox.

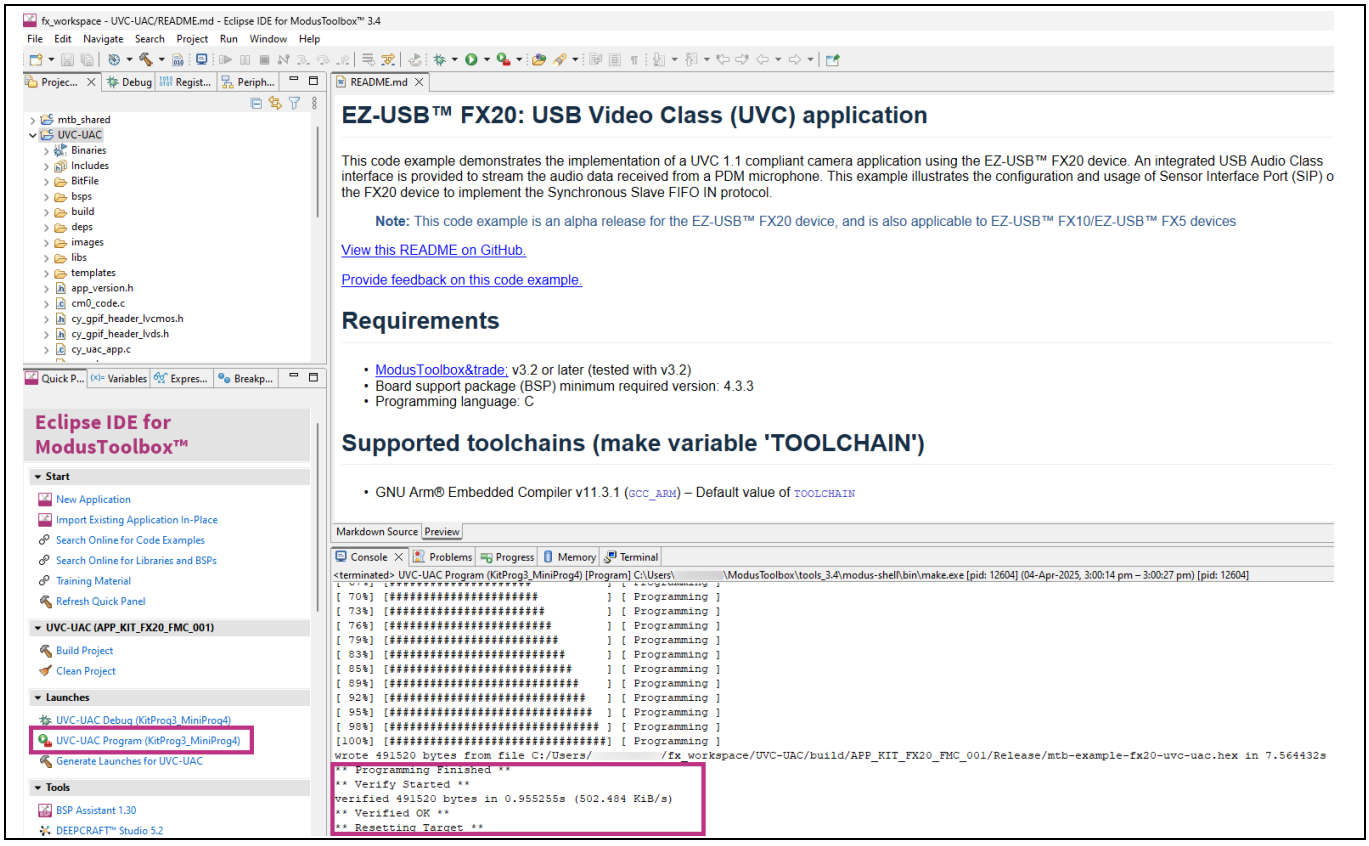

Figure 21 Program directly from ModusToolbox using Kitprog3 or Miniprog

4.6 Debugging EZ-USB™ FX code examples

Code examples can be debugged directly from ModusToolbox using a Kitprog3 or Miniprog using OpenOCD.

Click on **UVC-UAC Debug** link in the **Quick Panel** to start a debug session for the UVC-UAC code example.

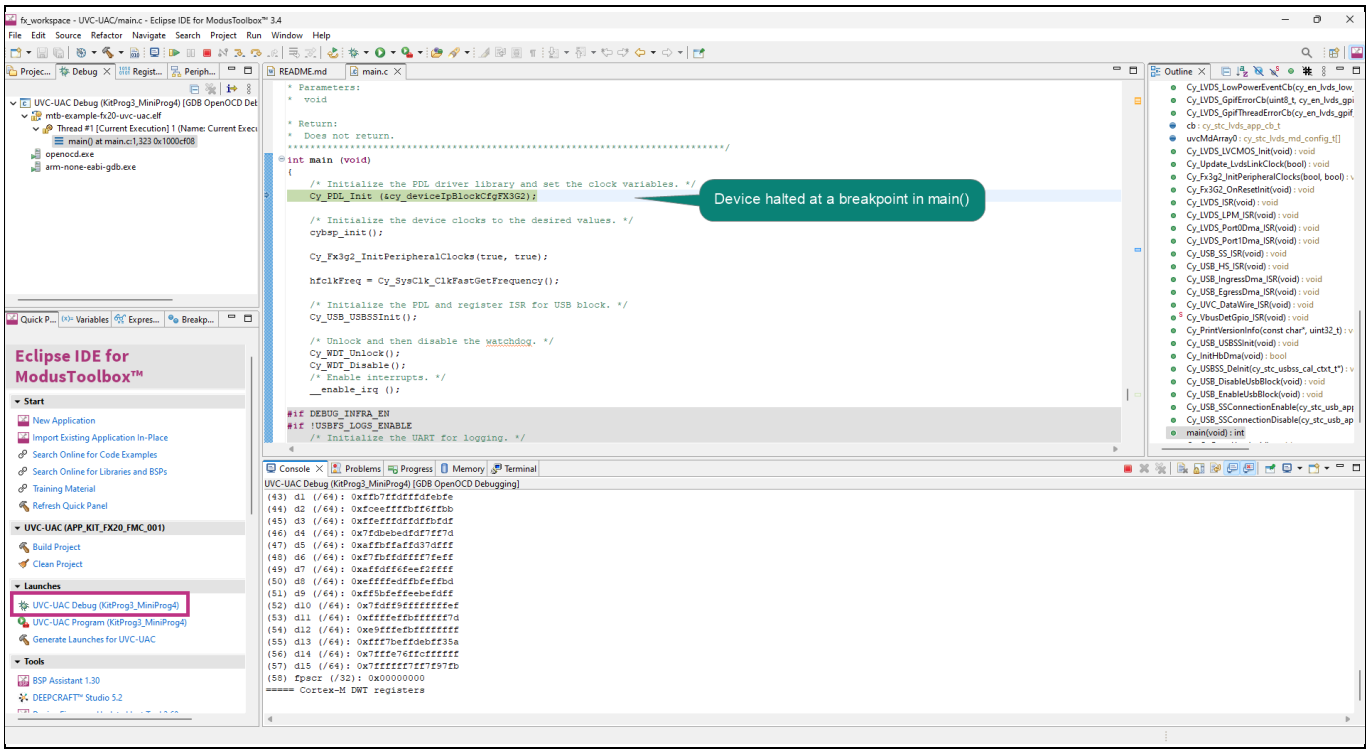

Figure 22 Debug session in progress for UVC-UAC code example

5 EZ-USB™ FX firmware applications

A comprehensive overview of each code examples is available in the corresponding README.md file.

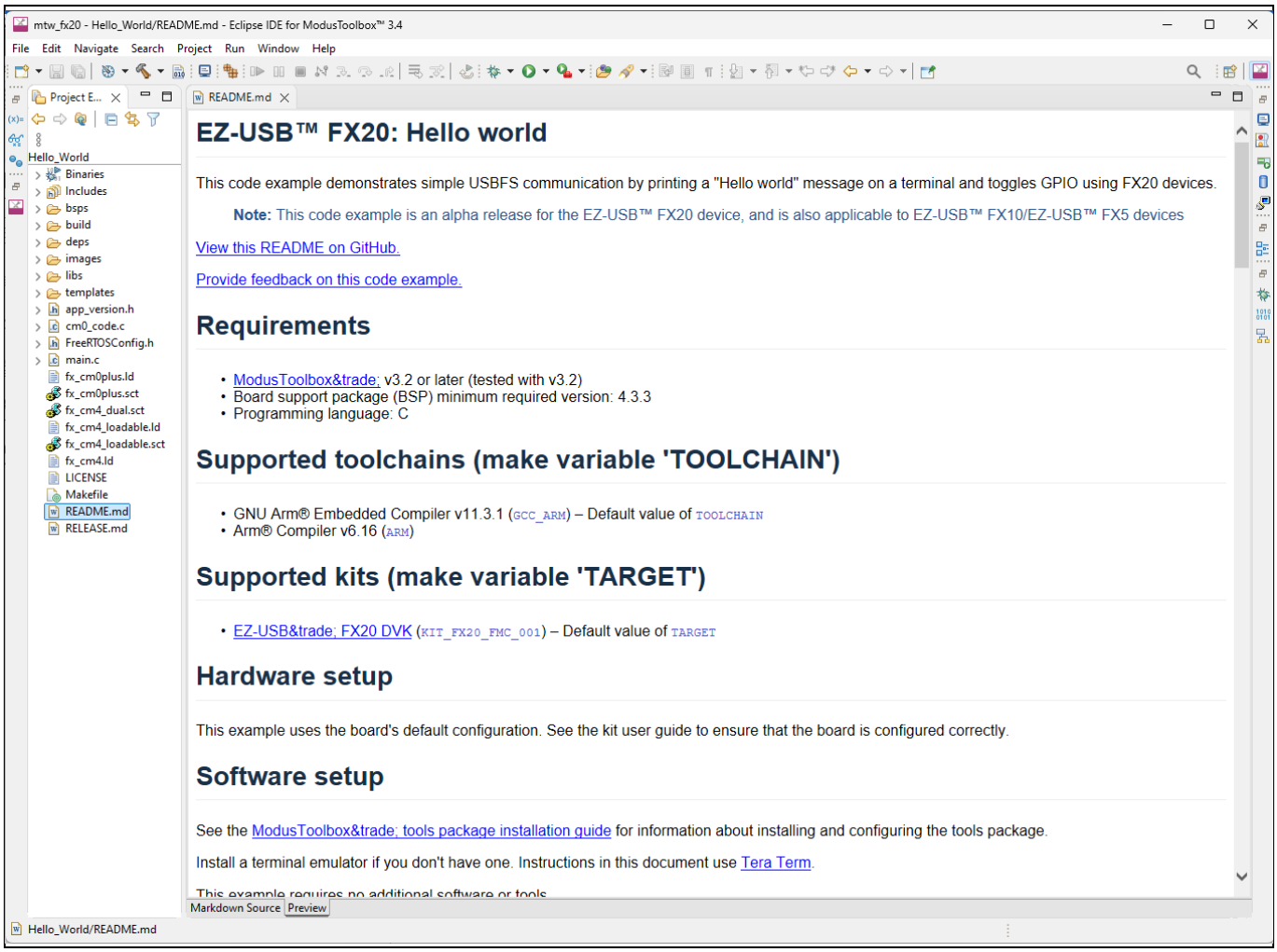

Figure 23 Readme file for Hello world code example

6 Troubleshooting

EZ-USB™ FX device enumeration	
Feature	EZ-USB™ FX device Enumeration
Error message	WinUSB device is enumerated in the device manager, but not listed in the Control Center device tree list in the left pane
Reason	This error is observed if: Missing DeviceInterfaceGUID {01234567-2A4F-49EE-8DD3-FADEA377234A} entry in Windows OS registry editor
Solution	Verify that the **DeviceInterfaceGUID** key has the value {01234567-2A4F-49EE-8DD3-FADEA377234A} at the following path: *Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB\VID_04B4&PID_XXXX<serial_string>\Device Parameters* To find the **serial_string** value: 1. Open **Device Manager** 2. Right-click on the USB device and select **Properties** 3. Go to the **Details** tab 4. Choose **Device Instance Path** from the **Property** drop down The value displayed will include the vendor ID, product ID, and serial string. For example, in USB\VID_04B4&PID_XXXX\5&33C730EA&0&4, the serial string is 5&33C730EA&0&4.

References

The following documents are available to support developers in the process of creating and testing solutions using the EZ-USB™ FX family of devices.

[1] Infineon Technologies AG: EZ-USB™ FX SDK User Guide; This document

[2] Infineon Technologies AG: Getting started with EZ-USB™ FX20/FX10/FX5N/FX5; Available online

[3] Infineon Technologies AG: EZ-USB™ FX5/FX5N/FX10/FX20 hardware design guidelines and schematic checklist; Available online

[4] Infineon Technologies AG: usbfxstack API documentation; Available online

[5] Infineon Technologies AG: mtb-pdl-cat1 API documentation; Available online

[6] Infineon Technologies AG: EZ-USB™ FX20 (USB 20 Gbps) device controller architecture; Available online

[7] Infineon Technologies AG: EZ-USB™ FX10 (USB 10 Gbps) device controller architecture; Available online

[8] Infineon Technologies AG: EZ-USB™ FX5N (USB 10 Gbps) device controller architecture; Available online

[9] Infineon Technologies AG: EZ-USB™ FX5 (USB 5 Gbps) device controller architecture; Available online

[10] Infineon Technologies AG: FX20 Hello World application description; Available online

[11] Infineon Technologies AG: FX20 USBSS device application description; Available online

[12] Infineon Technologies AG: FX20 USB Video Class application description; Available online

[13] Infineon Technologies AG: FX20 USB Test application description; Available online

[14] Infineon Technologies AG: FX20 Slave FIFO 2-bit application description; Available online

[15] Infineon Technologies AG: FX20 SMIF DMA application description; Available online

[16] Infineon Technologies AG: FX20 CDC VCOM application description; Available online

Revision history

Document revision	Date	Description of changes
**	2025-05-07	Initial release

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2025-05-07

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email:

erratum@infineon.com

Document reference

002-41342 Rev. **

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.