

AC6102 开发板 USB3.0 开发教程

严正声明：

本文档由狂热的电子技术爱好者“小梅哥”根据 Cypress 官方套件提供的相关内容进行整理改版调试得到，适用于小梅哥团队最新推出的带 USB3.0 的 FPGA 开发板 AC6102，其他板卡商如需借鉴，请注明出处。各位用户如果在使用过程中发现任何疏漏或不严谨的地方，恳请通过各种方式告知我们，以便我们不断改进，不断进步。本文档当前为 1.0 版本，后续可能会有更新，如需及时获取最新消息，欢迎关注我们的微信公众号“小梅哥”。

关于本教程涉及到的相关设计文件，可以从我们提供的 AC6102 开发板配套资料中获取，也可通过邮件向我们索取。联系方式见本文末尾。

梅雪松

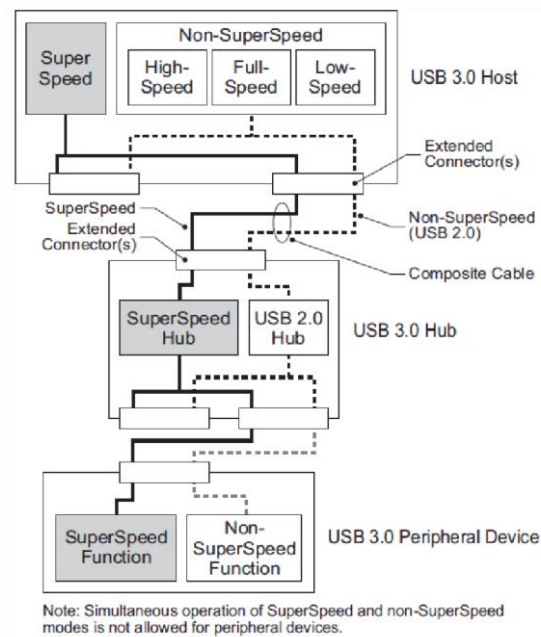
2016 年 12 月 22 日

关于 USB3.0

USB 3.0 概述

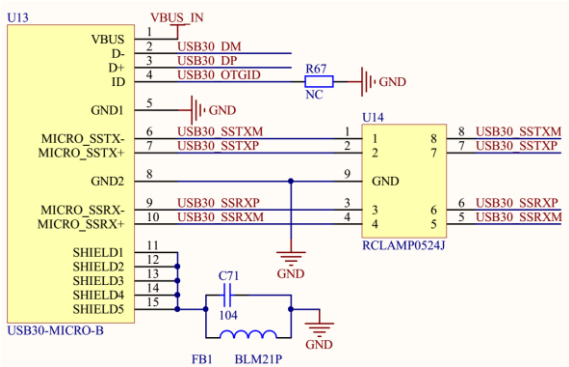
USB 3.0 支持高达 5 Gbps 的数据传输速率，降低功耗，并与 USB 2.0 向后兼容。由 USB-IF 发布的 USB 规范可在 <http://www.usb.org/developers/ssusb/> 找到。下图显示了 USB 3.0 的架构。

图 . USB 3.0 双总线架构



参考源: <http://usb.org>

除了 USB 2.0 接口外,USB 3.0 的物理接口增加了两对差分引脚和一个接地引脚,用于超高速传输。这样可以保证 USB 3.0 能够与 USB 2.0 向后兼容。



电气接口

与 USB 2.0 相比,USB 3.0 的引脚分布有所不同。除了具有 USB 2.0 的 VBUS、D-、D+和 GND 引脚外,USB 3.0 还有另外五个引脚,即两对差分引脚和一个接地引脚(GND_DRAIN)。两对差分引脚用于超高速数据传输,支持双单工超高速的信号传输。GND_DRAIN 引脚用于地线终端、信号完整性的管理以及抗 EMI 干扰性能。下表是对九个引脚的说明。

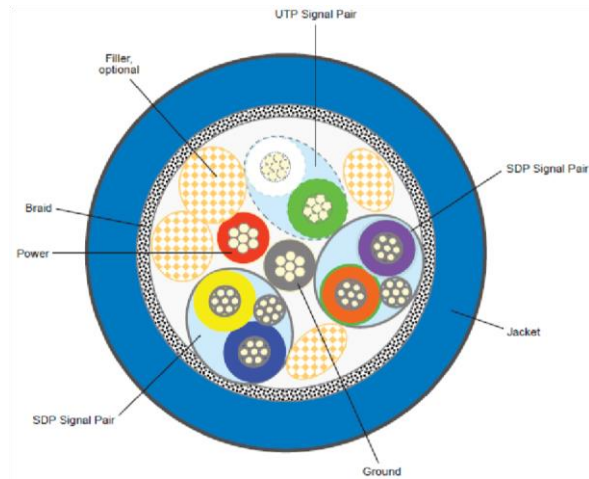
表 3. USB 3.0 引脚说明

引脚名称	说明
VBUS	电源
D-	USB 2.0 的差分对
D+	
GND	电源的接地端
SSRX-	超高速接收差分对
SSRX+	
SSTX-	超高速传送差分对
SSTX+	
GND_DRAIN	信号返回的接地端

线缆与连接器

与 USB2.0 相比,USB 3.0 添加了四个数据线(即 SSRX+、SSRX-、SSTX+和 SSTX-)用于数据传输,以及一个接地线用于地线终端、信号完整性管理以及抗 EMI 的干扰。下图显示了 USB 3.0 线缆的结构。

USB 3.0 线缆结构



参考源: <http://usb.org>

表 4. USB 3.0 线缆说明

名称	说明	颜色
电源	VBUS	红
UTP_D-	USB 2.0 D-	白
UTP_D+	USB 2.0 D+	绿
接地	电源的接地线	黑
SDP1-	屏蔽差分对 1	蓝
SDP1+		黄
SDP1_Drain	SDP1 的接地线	
SDP2-	屏蔽差分对 2	紫
SDP2+		橘
SDP2_Drain	SDP2 的接地线	

USB 3.0 规范中也定义了下面各个连接器:

■ USB 3.0 Standard-A 插头和插座

SuperSpeed 标准中将 USB 3.0 Standard-A 连接器定义为主连接器。该连接器是基于 USB 2.0 Standard-A 连接器的设计, 另外还添加了超高速信号。USB 3.0

Standard-A 插座可与 USB 3.0 Standard-A 插头或 USB 2.0 Standard-A 插头连接。USB 3.0 Standard-A 连接器使用唯一的颜色, 易于识别。图 33 显示了所推荐的颜色分配。



Standard-A型公口、母口实物模拟图

■ USB 3.0 Standard-B 插头和插座

USB 3.0 Standard-B 连接器用于比较大的固定设备，如外部硬盘驱动器和打印机。USB 3.0 Standard-B 插座可与 USB 3.0 Standard-B 插头或 USB 2.0 Standard-B 插头连接。不能将 USB 3.0 Standard-B 插头插入 USB 2.0 Standard-B 插座。



USB 3.0 Powered-B 插头与插座

通过 USB 3.0 Powered-B 连接器，USB 3.0 设备可向 USB 适配器供电，而不需使用外部电源。该连接器的尺寸与 USB 3.0 Standard-B 连接器相同，但增加了两个引脚：一个用于电源供电（DPWR），另一个用于接地（DGND）。

■ USB 3.0 Micro-B 插头与插座

USB 3.0 Micro-B 连接器用于体积较小的手提设备。

■ USB 3.0 Micro-AB 和 USB 3.0 Micro-A 连接器

USB 3.0 Micro-AB 插座类似于 USB 3.0 Micro-B 插座，只有外形设计是不同的。它可与 USB 3.0 Micro-A 插头、USB 3.0 Micro-B 插头、USB 2.0 Micro-A 插头以及 USB 2.0 Micro-B 插头连接。USB 3.0 Micro-AB 插座只能使用于 OTG 产品。该产品可作为主机或设备使用。USB 3.0 Micro-AB 插座不可用于其他用途。

USB 3.0 Micro-A 插头类似于 USB 3.0 Micro-B 插头，只有外形和 ID 引脚是不同的。USB 3.0 Micro-A 插头、USB 3.0 Micro-AB 插座、USB 3.0 Micro-B 插座和插头都属于 USB 3.0 Micro 连接器系列。这些接口只是外形不同。与 USB 2.0 Micro-A 插头相同，USB 3.0 Micro-A 插头只用于 OTG 应用。



USB 3.0 与 2.0

USB 3.0 采用双总线架构，支持 USB 2.0 和 3.0。下表显示了 USB 3.0 与 USB 2.0 的主要区别。表 5. USB 3.0 与 USB 2.0 的区别

特性	USB 2.0	USB 3.0
数据传输速率	480 Mbits/s（高速） 12 Mbits/s（全速） 1.5 Mbits/s（低速）	5.0 Gbits/s（超高速） 480 Mbits/s（高速） 12 Mbits/s（全速） 1.5 Mbits/s（低速）
数据传输	半双工 两线差分信号	双单工 四线差分信号
线缆信号数量	共有四个信号： - 两个用于 USB 2.0（D、D-） - 两个分别用于 VBUS 和 GND	共有九个信号： - 四个用于超高速数据 - 两个用于 USB 2.0（D、D-） - 三个分别用于 VBUS 和 GND
总线数据传输协议	直接连接主机协议 轮询数据流 数据包传输到所有下行设备 没有数据流复用	直接连接主机协议 异步通知 数据包只能传输到目的设备 可批量传输多个数据流
电源管理	两种模式 - 活动 - 挂起	四种模式 - 活动（U0） - 闲置、快速（U1） - 闲置、低速（U2） - 挂起、低速（U3）

总线供电	低功耗设备: 100 mA 高功耗设备: 500 mA	低功耗设备: 150 mA 高功耗设备: 900 mA
端口状态	端口硬件检测连接事件。 系统软件使用端口指令将端口切换到使能状态。	端口硬件检测连接事件,并将端口状态转换为操作状态,准备进行超高速数据通信。
线缆的最大长度	5 米	基于电气规格。实际上是 3 米长的 26 AWG 类铜线
数据传输类型	四种数据传输类型: 控制传输、批量传输、中断传输以及同步传输。	支持 USB 2.0 的四种数据传输类型,并具有超高速传输功能。 批量传输具有数据流传输功能。

AC6102 USB3.0 FPGA 开发板

概述

AC6102 上集成了一颗 Cypress 推出的高性能 USB3.0 传输芯片 CYUSB3014, Cypress 称之为 EZ-USBFX3。该芯片性能强劲, 功能强大, 接口简单, 非常适合用于各种需要高速数据传输的场合, 以下为摘自 EZ-USBFX3 数据手册中关于该芯片的介绍:

赛普拉斯 EZ-USBFX3 是新一代 USB3.0 外设控制器, 具有高度集成的灵活特性, 可帮助开发人员为任何系统添加 USB3.0 功能。EZ-USBFX3 具有一个可进行完全配置的并行通用可编程接口 GPIFII, 它可与任何处理器、ASIC 或 FPGA 连接。这个通用可编程接口 GPIFII 是赛普拉斯旗舰 USB2.0 产品 FX2LP 中的 GPIF 的增强版本。它可轻松无缝地连接至多种常用接口, 比如异步 SRAM、异步和同步地址数据复用式接口、并行 ATA 等等。

EZ-USBFX3 集成了 USB3.0 和 USB2.0 物理层(PHY)以及 32 位 ARM926EJ-S 微处理器, 具有强大的数据处理能力, 并可用于构建定制应用。该产品采用了一种巧妙的架构, 使从 GPIFII 到 USB 接口的数据传输速度可达 320Mbps[1]。通过集成的 USB2.0OTG 控制器, 可以实现需要双角色使用场合的应用。例如, EZ-USBFX3 可以作为 MSC 和 HID 级设备的 OTG 主机使用。EZ-USBFX3 配有用于存储代码和数据的 512KB 片上 SRAM。EZ-USBFX3 还具有可连接至 UART、SPI、I2C 和 I2S 等串行外设的接口。

EZ-USBFX3 附带易用的 EZ-USB 工具, 为快速应用开发提供完整的解决方案。软件开发工具包中附带应用示例, 可缩短上市时间。

EZ-USBFX3 完全符合 USB3.0 规范 1.0 版, 并可向下兼容 USB2.0。本设备还符合电池充电规范 v1.1 和 USB2.0OTG 规范 2.0 版。

从该介绍中, 摘选出我们所关心的以下信息:

- 1、USB3.0 外设控制器, 兼容 USB2.0 协议
- 2、对外是一个 GPIFII 接口, 可以灵活的与 FPGA 连接

- 3、片上集成 ARM926EJ-S 微处理器（ARM9 处理器）
- 4、用户接口（GPIFII）到 USB 接口的数据传输速度可达 320Mbps
- 5、支持 USB2.0 OTG 功能，可以作为海量存储器控制器（MSC，如 U 盘）或人机接口设备（HID，如鼠标、键盘）的主机
- 6、片上 512K 字节 SRAM 用于存储程序和数据
- 7、UART、SPI、I2C、I2S 串行外事接口。
- 8、提供方便易用的 EZ-USB 开发

通过以上介绍可知，EZ-USBFX3 的功能还是非常强大的。因此，如何高效快速的学会使用该芯片也就显得非常有意义了。

本文作为 AC6102 开发板的第一期配套资料，仅对 AC6102 上的 USB3.0 电路功能和性能测试进行简单的介绍。关于更加详细的开发教程，将在第二期或第三期资料推出。

本实验需要用到以下器件：

- 1、芯航线 AC6102 型 FPGA 开发板一块
- 2、带 USB3.0 接口的 PC 机一台
- 3、Micro USB3.0 数据线一根
- 4、5V/2A 电源适配器一个
- 5、USB Blaster 下载器一个



安装 SDK

开发 EZ-USBFX3，需要用到 Cypress 提供的开发工具，该软件已经经历了多次更新，目

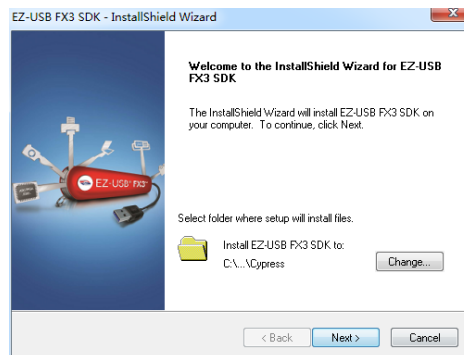
前最新版本的叫做 FX3_SDK_Windows_v1.3.3.exe。该软件以及 EZ-USBFX3 的其他所有相关文档都可以从以下地址下载得到。下载需要注册，注册非常简单，仅需要邮箱就能成功注册，这里就不给大家介绍注册方法了。

CYUSB3014 开发套件官方资料下载地址：

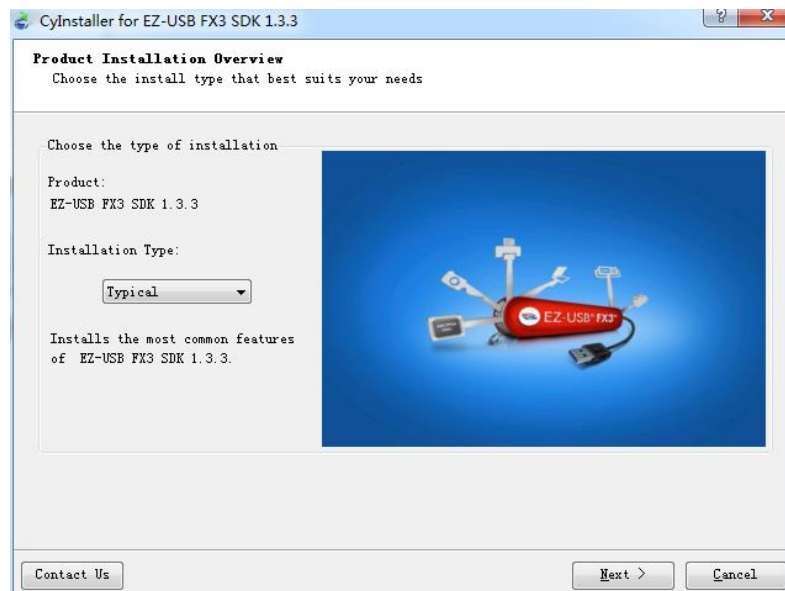
<http://www.cypress.com/documentation/software-and-drivers/ez-usb-fx3-software-development-kit>

下载好之后，即可安装该软件，该软件实际是一个集成包，里面包含了开发 EZ-USBFX3 的软硬件所需的全部内容，包括开发软件、DEMO、文档等。安装过程也非常简单，以下对其需要注意的地方进行简单说明。

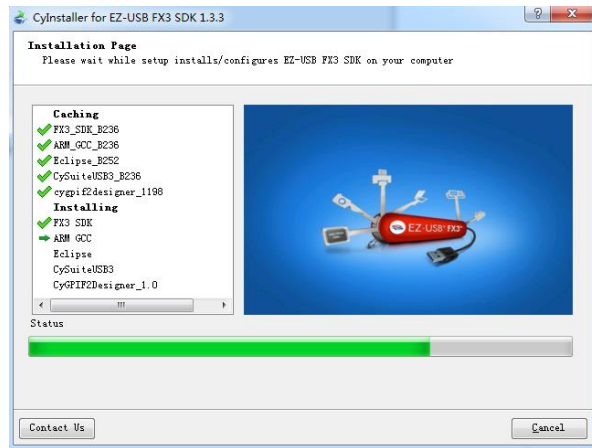
- 1、双击 FX3_SDK_Windows_v1.3.3.exe 以开设运行软件
- 2、安装路径选择，建议选择默认，这可以帮我们在开发初期避免路径不通带来的差异，可以省不少事。点击 Next。



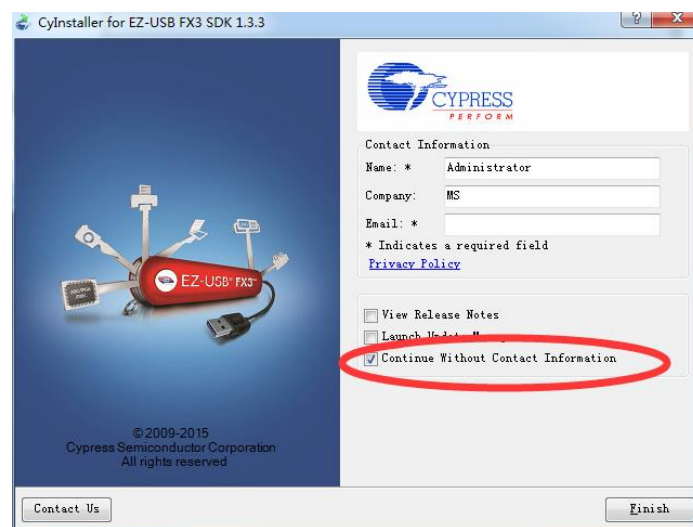
- 3、下一个界面要求选择安装的模式，典型的，还是用户自定义安装，第一次接触，建议按照典型方式安装。



- 4、接下来，就是各种接受协议，各种下一步。接受了所有的协议后，点击 next 就可以开始正式安装，以下为安装过程

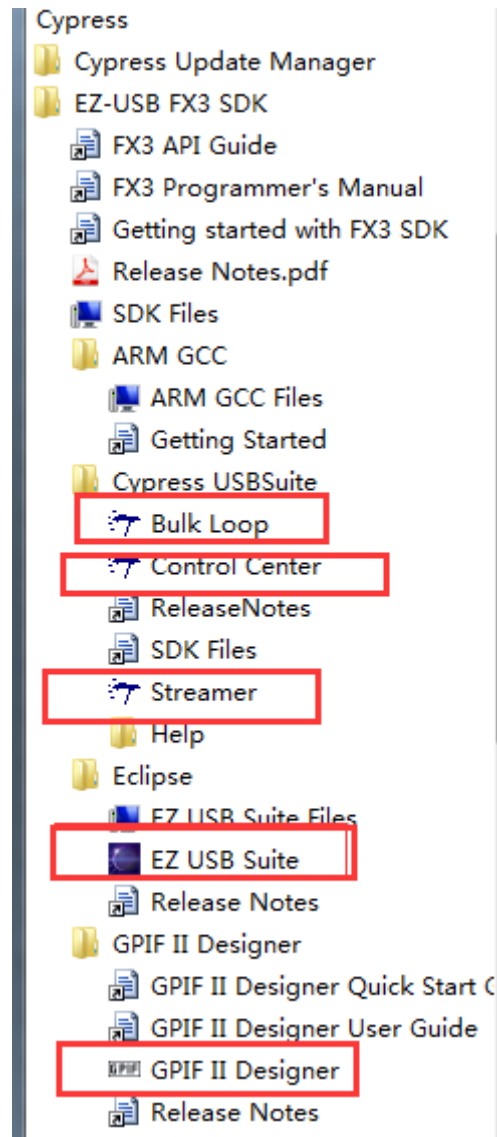


安装最后的一个界面是反馈信息，默认 finish 是灰色的无法选中，我们选择 Continue Without Contact Information 即可，然后点击 Finish。



至此，EZ-USBFX3 开发套件就安装完成了

安装完成后，在系统的开始菜单下，会出现一个叫做 Cypress 的文件夹，里面包含以下内容：



Cypress Update Manager:

Cypress 的更新管理器，如果开发套件有更新版本，使用该管理器可以检查并更新到最新版本。

Eclipse IDE

为 C/C++ 开发者提供的 Eclipse IDE 是 FX3 SDK 的一部分。该集成开发环境（IDE）包括基础 Eclipse 平台和 CPP 功能。IDE 包含开发所需插件。

支持 GNU ARM C/C++ 开发

Zylin 嵌入式 CDT

它是 Eclipse IDE 中的一个通用插件，通过该插件可以使用 GNU 调试器调试 FX3 固件。

Java(TM) 平台、标准版实时环境版本 7 (JRE)

GNU 工具链

GNU 工具链作为 FX3 SDK 的部分，包括以下内容：

- GCC 编译器（gcc）
- GNU 链接器（ld）
- GNU 汇编器（as）
- GNU 调试器（gdb）

这些可执行文件都由 Eclipse IDE 调用。

GPIF II Designer（设计程序）

GPIF II 接口设计工具是一个 Windows 应用，并作为 FX3 SDK 的一部分。该工具提供了一个图形用户界面，用于为目标设备指定必要的接口。用户通过使用状态机设计了一个接口，工具将它翻译成一个 C 头文件，并包含到 Eclipse 项目。

Windows USB 设备驱动程序

“cyusb3.sys”是一个 USB 设备驱动程序，适用对象包括：32 位的 Windows XP、32/64 位的 Windows Vista、32/64 位的 Windows 7 以及 32/64 位的 Windows 8。该驱动程序能够同所有符合 USB 2.0 和 USB 3.0 规范的设备进行通信。它是通用的驱动程序，能够理解基本的 USB 指令。然而它没有支持 USB 设备类协议。例如，驱动程序不能直接将一个 USB 大容量存储设备连接到 Windows 文件系统。应用固件必须实现该逻辑。赛普拉斯通用驱动程序的 VID 为 0x04B4，PID 为 0x00F0、0x00F1 和 0x00F3。如果您想用自己的 VID/PID，您必须获得由微软 WHQL 签署的驱动程序。开发过程中，您可以通过禁用电脑上的软件签名强制来使用一个未签名的驱动程序。该驱动程序是自定义的 USB 应用同供货商特定的设备进行通信的理想选择。或者，对于实验或诊断应用，驱动程序可用于将低级 USB 请求发送给所有 USB 设备。为了使用驱动程序同某个设备通信，Windows 必须使该设备同驱动程序相匹配。类库（CyAPI.lib 和 Cyusb.dll）为驱动程序提供了一个高级的编程接口。

特性

- 与 Windows 驱动程序基础（WDF）相兼容
- 与所有 USB 2.0 兼容的设备相兼容
- 与赛普拉斯 USB 3.0 兼容的设备相兼容
- 支持 USB 3.0 的基本性能
- 支持 Windows 即插即用性能以及电源管理
- 支持 USB 远程唤醒
- 支持 CONTROL（控制）、BULK（批量）、INTERRUPT（中断）以及 ISOCHRONOUS（同步）等端点

支持多个 USB 设备同时连接

支持可自定义的驱动程序 GUID(而无需重建驱动程序) 可在每一帧传送多个数据包, 因此能够实现高带宽的数据传输

您不需要使用赛普拉斯所提供的驱动程序。可以编程 FX3, 以实现标准 USB 类设备。实现这样的设备时, 应该使用 USB 类驱动程序来代替。例如: 如果 FX3 正在执行 USB 视频类别 (UVC), 则不需要使用赛普拉斯通用驱动程序, 取而代之的是操作系统 OS 所提供的 UVC 驱动程序。这样会让 Windows, Linux 和 Mac 的支持变得简单, 因为操作系统 OS 已经提供了相应的标准驱动。

应用接口

CyAPI.lib

CyAPI.lib 为 USB 设备提供了一个既简单又强大的 C++编程接口。C++类库为 cyusb3.sys 驱动程序提供了一个高级编程接口。此库只能与该驱动程序所支持的 USB 设备通信。有关 CyAPI.lib 的详细信息, 请参考《C++库编程者参考手册》中介绍的内容。它位于 USB 套件帮助文档中, 并作为 SDK 的一部分。

CyUSB.dll

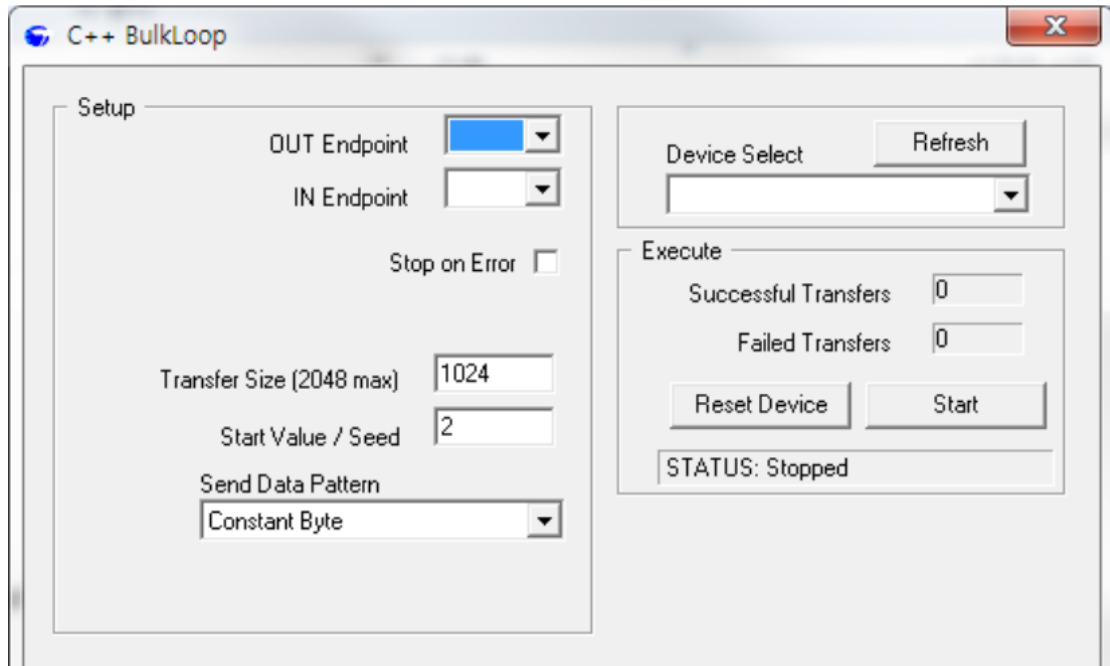
CyUSB.dll 是一个托管 Microsoft.NET 类库, 为 USB 设备提供了一个高级编程接口。因此应用可通过库方法和属性来访问 USB 设备, 而不用通过低级的 Win32 API 调用与 USB 设备驱动程序直接通信。由于 CyUSB.dll 是一个托管.NET 库, 因此用户可通过 Visual Basic.NET、C#、Visual J#和托管 C++等所有 Microsoft Visual Studio.NET 语言对它的类和方法进行访问。如要使用该库, 请将 CyUSB.dll 参考添加到项目的 References(参考) 文件夹中。然后, 访问 CyUSB 命名空间的任何源文件都必须拥有用于按照准确语法添加命名空间的一行。

有关 CyUSB.dll 的详细信息, 请参考《C#库编程者参考手册》中的内容。它位于 USB 套件帮助文档中, 并作为 SDK 的一部分。请参考赛普拉斯 USB 套件应用开发 — 快速入门指南, 了解有关使用 CyAPI.lib 和 CyUSB.dll 的详细信息。

Windows 软件示例

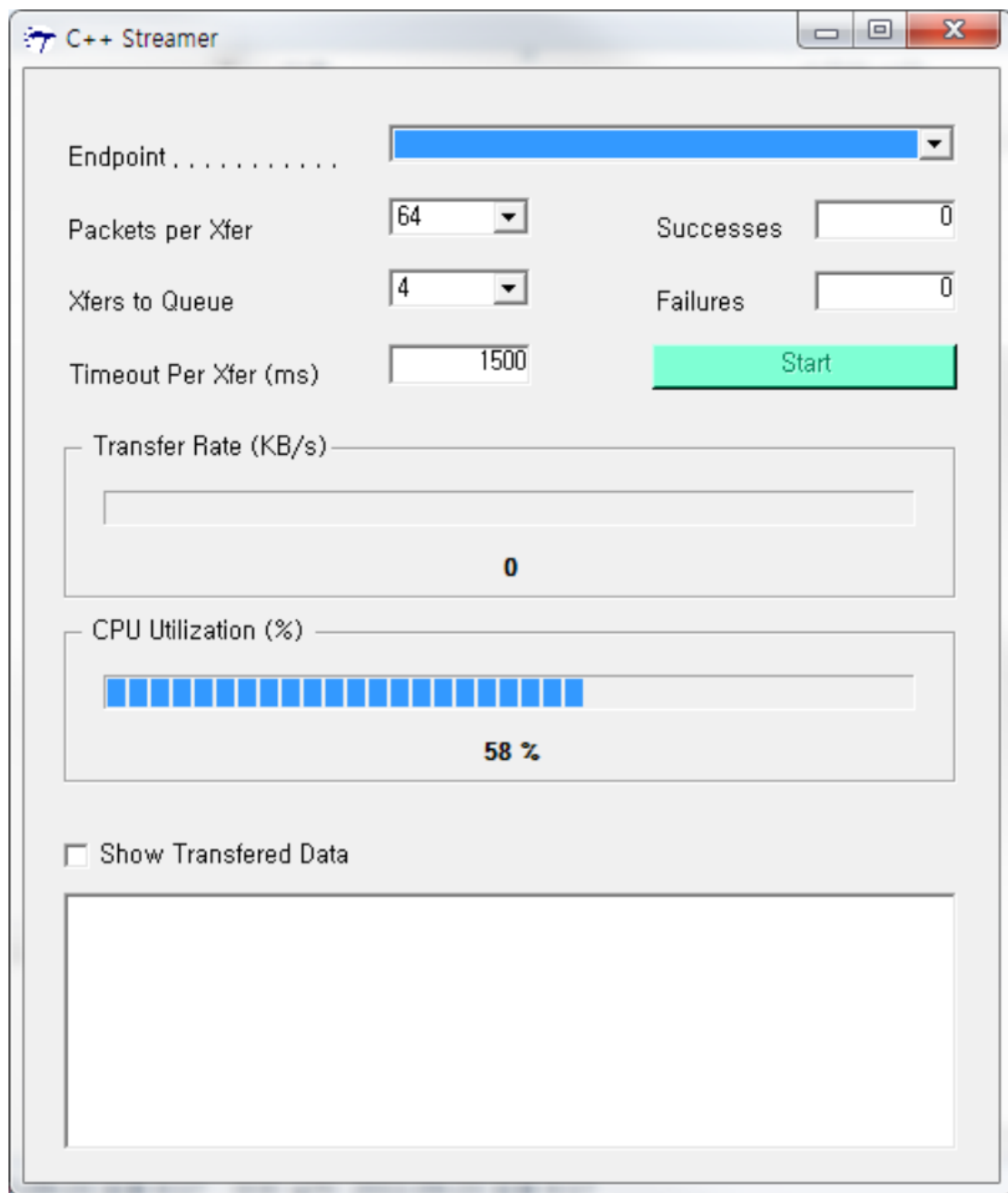
BULKLoop 示例

通过 BULKLoop 应用测试 BULK 端点上数据传输的回送。



Streamer 示例

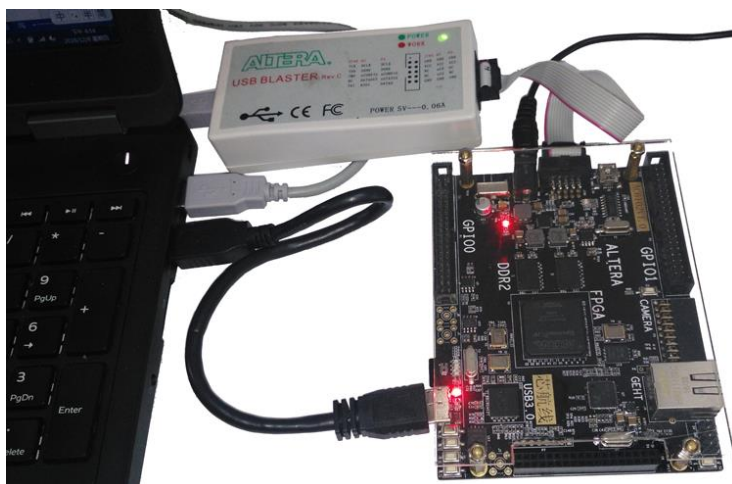
Streamer 应用用于测试 BULK/ISOCRONOUS 端点上的数据传输。



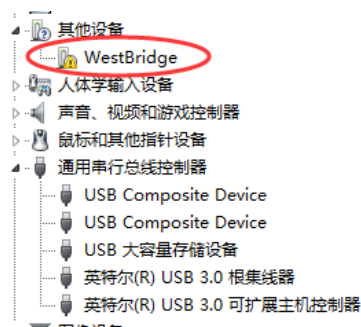
连接 USB3.0 到 PC

以上，我们完成了 EZ-USB FX3 的开发套件的安装，只有安装了该套件，才能在后续将 AC6102 的 USB3.0 接口连接到 PC 机时，成功的安装驱动。

- 1、使用配套的 5V 电源给开发板供电，打开电源开关，插上 USB Blaster 下载器。
- 2、使用配套的 USB3.0 数据线连接 PC 和 AC6102 的 USB3.0 接口。如下图所示：

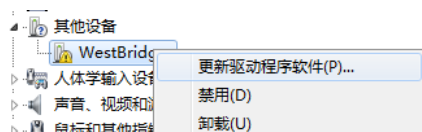


插上 USB 线并打开开发板电源以后，PC 会自动识别到设备，设备名称为“WestBridge”，如下图所示，由于此时设备没有安装驱动，因此还是带黄色“！”，表明设备不可用。



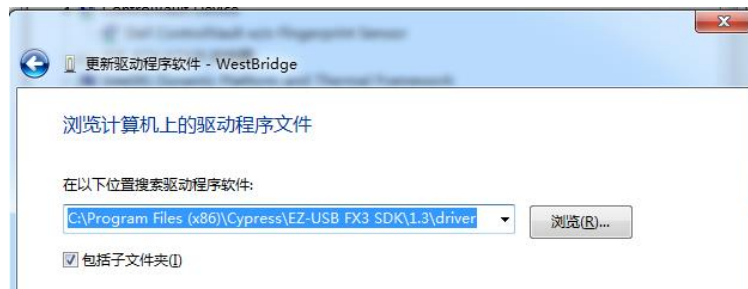
安装 EZ-USB 芯片驱动

当开发板和 PC 机连接完成后，我们就可以为设备安装驱动了。选中设备，单击鼠标右键，选中更新驱动程序，选则浏览计算机以查找驱动程序软件。



定位驱动路径到以下文件夹（以实际的 SDK 安装路径为准）。C:\Program Files

(x86)\Cypress\EZ-USB FX3 SDK\1.3\driver,并勾选上“包括子文件夹”选项。然后点击下一步。



如果弹出以下界面则表明驱动安装成功。



此时，回到设备管理器中，可以发现如下设备：



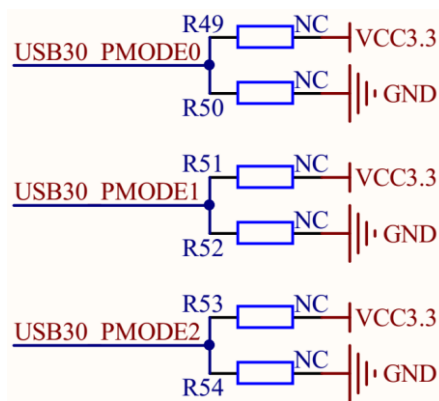
到此我们就完成了所有软硬件的连接，接下来，就可以对设备进行测试了。

关于 EZ-USB 的启动方式

在进行测试之前，需要补充介绍下 EZ-USBFX3 的启动方式，EZ-USBFX3 芯片总共支持 7 种引导方式，所谓引导，就是 EZ-USBFX3 芯片上电以后，片上的 ARM9 处理器从何处开始读取程序并运行。其支持的 7 种启动方式如下所示：

- ✧ 从 USB 引导
- ✧ 从 I2C 引导
- ✧ 从 SPI （支持的 SPI 器件为 M25P16 (16 Mbit)、M25P80(8 Mbit) 和 M25P40 (4 Mbit)）或同类器件引导
- ✧ 从 GPIF II 异步 ADMUX 模式引导
- ✧ 从 GPIF II 同步 ADMUX 模式引导
- ✧ 从 GPIF II 异步 SRAM 模式引导

那么究竟 EZ-USBFX3 芯片上电后从哪里开始启动呢？芯片提供了 3 个启动配置引脚，通过设置启动引脚不同的值，来选择不同的启动方式。这三个引脚名叫 PMODE 脚。



这三个配置引脚，每个引脚都有三种状态，1（接高电平）、0（接低电平）、F（浮空，既不接高电平，也不接低电平），三个配置引脚的不同状态与对应的启动方式如下表所示：

序号	PMODE[2:0]	引导位置
1	F00	同步 ADMUX （16 位）
2	F01	异步 ADMUX （16 位）
3	F11	USB 引导
4	F0F	异步 SRAM （16 位）
5	F1F	I2C，如失败，则启用 USB 引导
6	1FF	仅 I2C
7	0F1	SPI，如失败，则启用 USB 引导

其中，对于 FPGA 和 USB3.0 组成的系统，我们很少用到第 1、2、4 种启动方式。第 7 种启动方式，即使用 SPI 引导，主要用在 EZ-USBFX3 作为整个系统的主机层，片上的处理器需要运行复杂的程序的情况下，使用片外的大容量 SPI FLASH 存储器启动。FPGA + EZ-USBFX3，使用最多的是 SlaveFifo 接口，因此，可以选择 EEPROM 存储器，即 I2C 方式启动。同时，为了保证系统在 IIC 接口的 EEPROM 存储器失效的情况下，能够有其他方式可以恢复，所以，AC6102 上的启动方式设置为了方式 5，即默认从 I2C 引导，如果从 I2C 引导失败，则启用从 USB 引导。对应的 PMODE[2:0]的配置为 F1F，在原理图上，即电阻 R51 使用 0R 电阻，其他电阻全部不焊接即可。

EZ-USB 测试方案分析

由于 AC6102 上的 USB3.0 电路更像是一个桥，一端，使用 USB 总线连接到 PC 机的 USB 端口上，另一端，使用 GPIFII 连接到 FPGA 芯片上。所以，为了测试整个链路是否通畅，可以分成两步测试，第一步，测试 EZ-USBFX3 和 PC 间的 USB3.0 总线是否稳定，第二步，再测试 FPGA 通过 EZ-USBFX3 到 PC 的链路是否稳定。

为了测试 EZ-USBFX3 和 PC 间的 USB 总线是否可靠，可以采用内部回环传输的方式进行测试，所谓内部回环传输，就是 PC 发数据给 EZ-USBFX3，EZ-USBFX3 将接收到的数据再发回给 PC，以实现通信 USB 总线的测试。

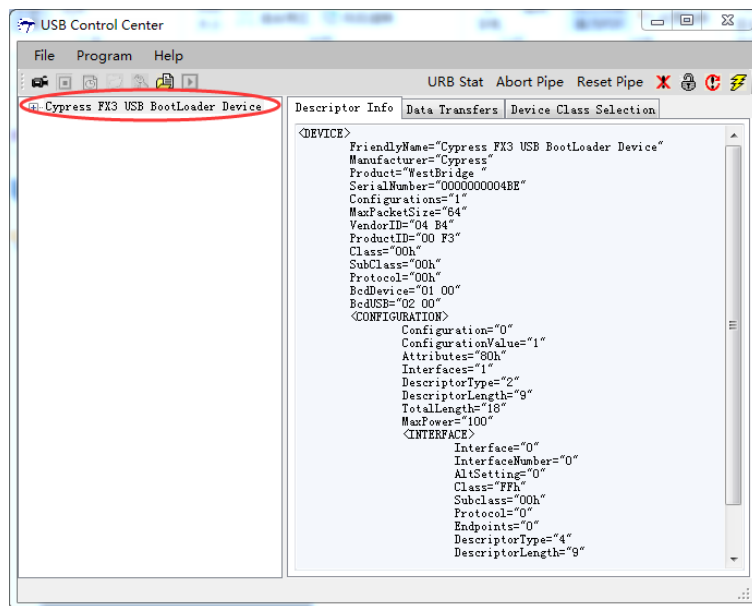
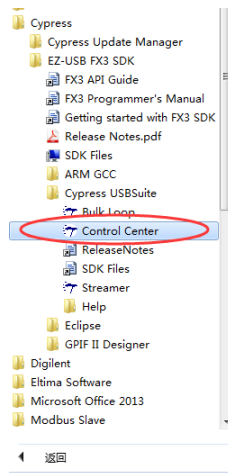
为了使 EZ-USB 能够工作在回环模式，需要有相应的固件支持。对于 EZ-USBFX3，由于我们将其引导方式设置为 I2C -> USB，即默认会从 EEPROM 上启动。但是我们的 EEPROM 中

目前没有烧写任何固件，因此一定会引导失败，引导失败后，则会选择从 USB 引导。所以，我们可以在系统上电后，通过 PC 直接将固件下载到 EZ-USB 的 RAM 中并运行。

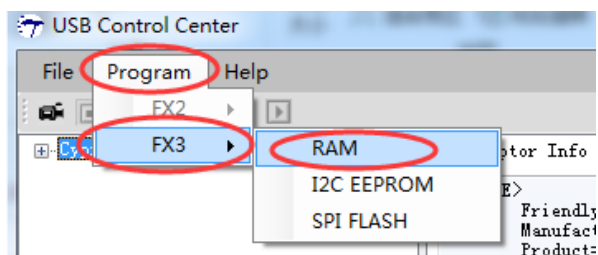
块传输模式测试

下载 EZ-USB 固件

在开始菜单中找到 Control Center，如下图 X1 所示，打开后软件会自动扫描到我们的 USB 设备，如下图 X2 所示。



这里，我们选中设备（Cypress FX3 USB BootLoader Device），然后点击菜单栏中的 Program，依次选择 FX3 -> RAM。



在打开的文件选择窗口中，我们定位到我们提供的 DEMO 目录下的..\AC6102\demo_release\ez_usb\01_USBBulkLoopAutoEnum 文件夹，选择 USBBulkLoopAutoEnum.img 文件，点击打开，如下图 X1 所示，则软件会自动读取该文件并通过 USB 总线下载到 EZ-USBFX3 芯片的 RAM 中，此时，我们再看 USB Control Center，发现此时的设备名称已变成了 Cpress USB BulkloopExample，左下角显示 Programming Succeeded。如下图 X2 所示。（关于 USBBulkLoopAutoEnum.img 文件的生成，将在第二期资料中提供详细的开发教程）。

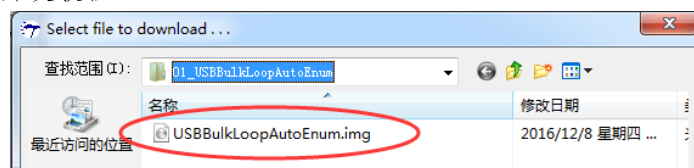


图 X1

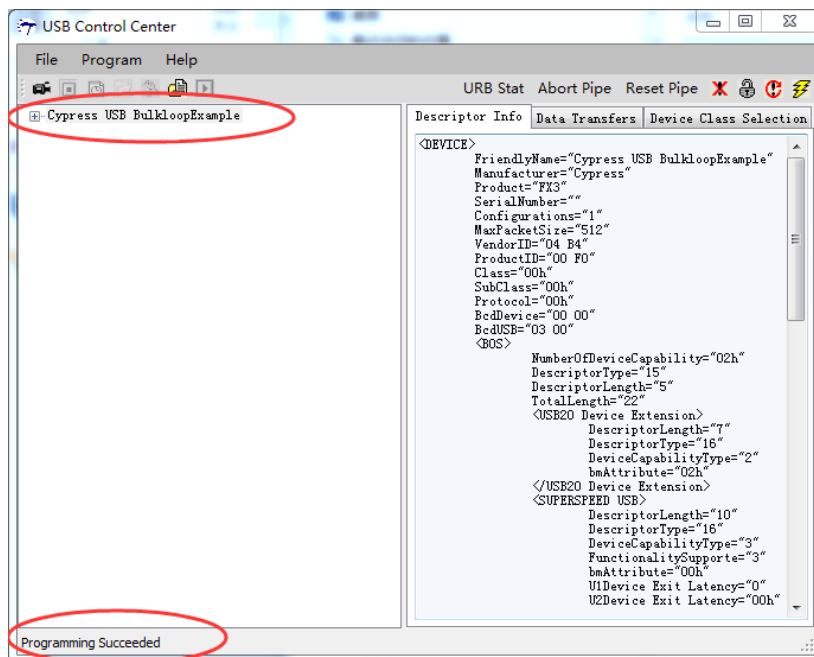
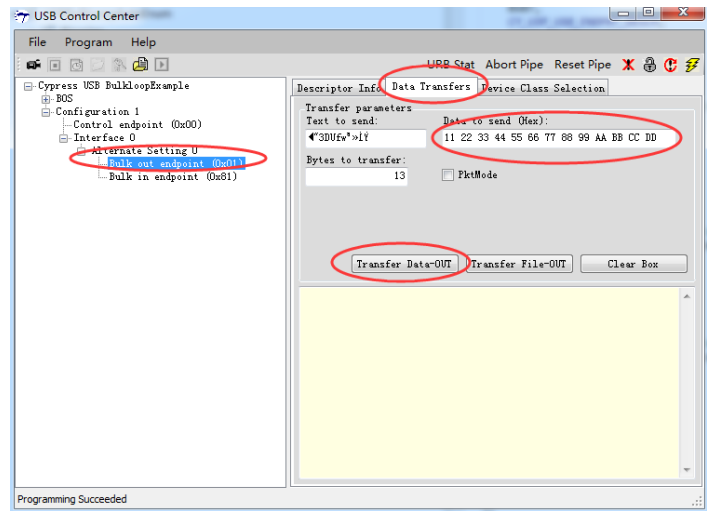


图 X2

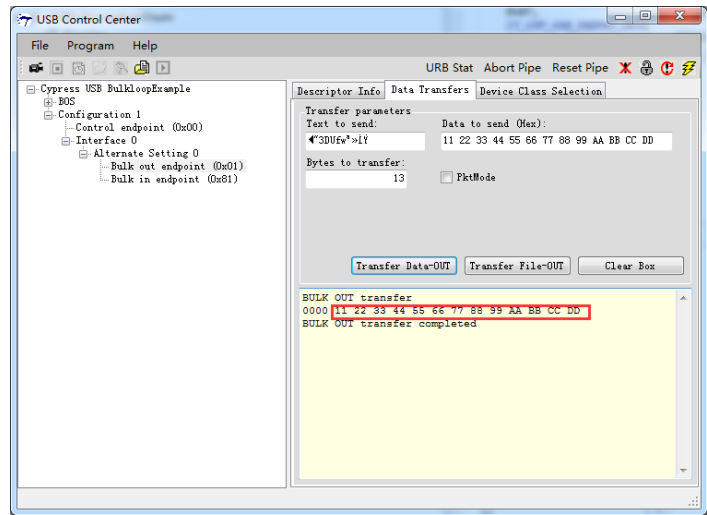
基本回环测试

USB Control Center 软件提供了一个基本的回环测试功能，这里，我们依次点开 Cpress USB BulkloopExample 前面的+号，找到 Bulk outendpoint (0x01)，选中该项，在右侧 “Data Transfers” 的 “Data to send (Hex)” 中输入需要从 PC 发送到 FX3

的数据。然后单击 Transfer Data-OUT 即可实现数据发送。如下图 X 所示。

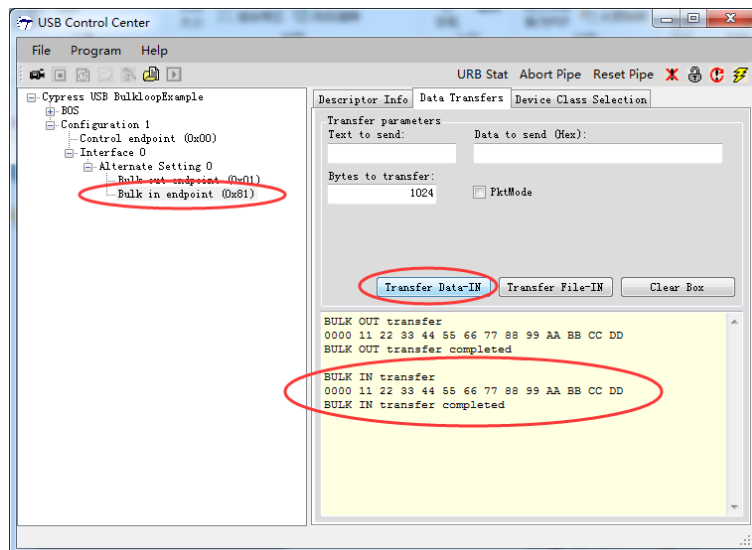


发送完成后，在下方信息窗口，可以看到提示信息，表明发送已经完成。



那么此时，这些发送出去的数据在哪里呢？本例烧写的固件是基于内部 DMA 的回环测试，所谓内部 DMA 回环，就是 EZ-USBFX3 会自动将 out 端点发送的数据搬运到 in 端点，所以，我们刚刚发出去的这 13 个数据，已经被 EZ-USBFX3 芯片搬运到了 in 端点了。这样，当电脑读取 in 端点，就能读取到刚刚发出去的这 13 个数据。

在左侧选中 Bulk in endpoint (0x81) 端点，然后右侧窗口中，单击 Transfer Data-IN 按钮，即可将 0x81 端点中的数据读出来，如下图所示。

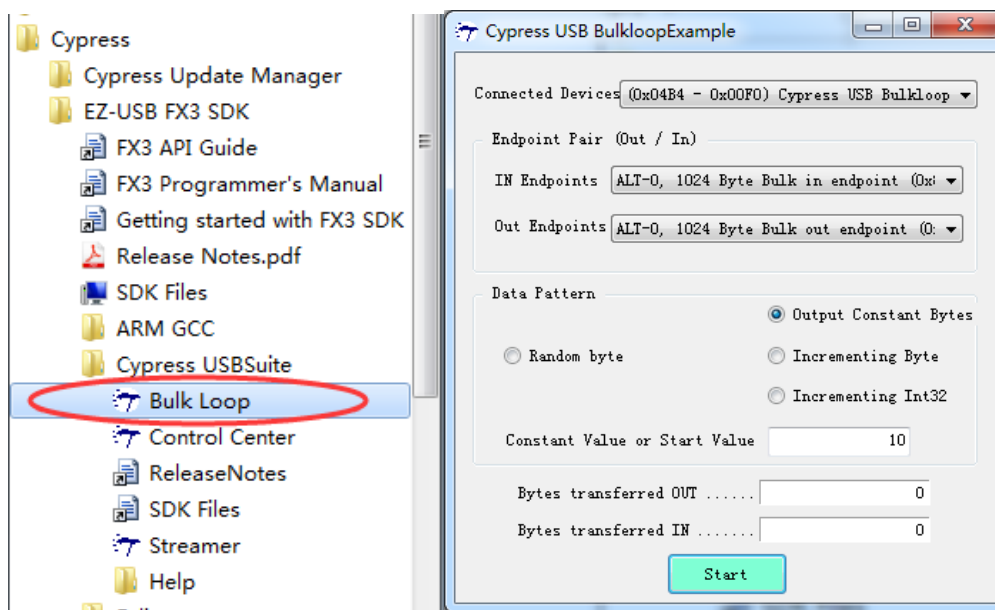


可以看到，读取回来的数据与我们发出去的数据完全一致。因此可知 USB 总线能够正常的收发数据。

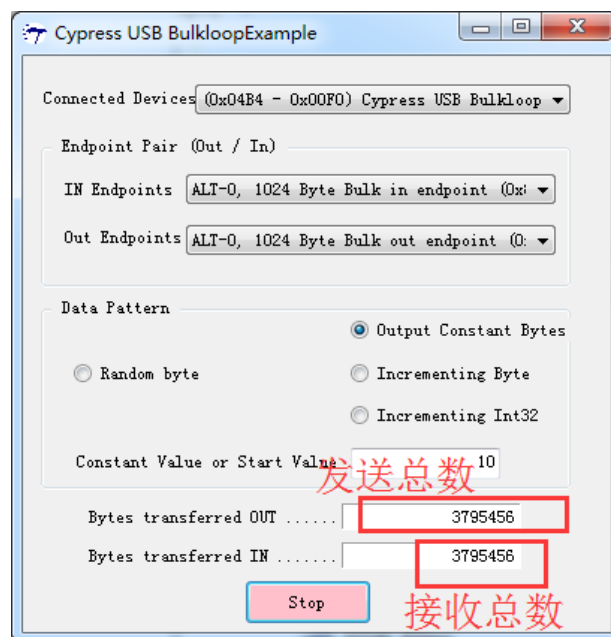
使用 Bulk Loop 工具测试

上述测试需要我们手工输入待测试数据并手动点击发送和读取按钮，因此效率十分低下，无法测试 USB3.0 在高强度传输时候的稳定性。接下来，我们就使用 EZ-USBFX3 的开发套件中提供的一个专用的回环测试软件来进行测试。

在开始菜单中找到 Bulk Loop 软件，如图 X1 所示，点击打开，打开后界面如图 X2 所示：



打开之后，直接默认设置即可，点击 Start 即可开始测试。发送和接受数据统计窗口会显示实时发送和接收的数据总数，通过比对发送和接收的数据是否一致，即可了解 EZ-USBFX3 和 PC 间的 USB 总线性能是否可靠。



至此，我们就完成了 USB 总线可靠性的测试。之所以要进行此测试，是因为 USB3.0 串行总线数据线的传输速度高达 Gbps，AC6102 这几根串行数据线在布线时需要完成阻抗匹配和差分布线，只有阻抗匹配和差分布线都满足 USB3.0 协议的要求，才能够稳定长期的传输数据不出错。

小梅哥

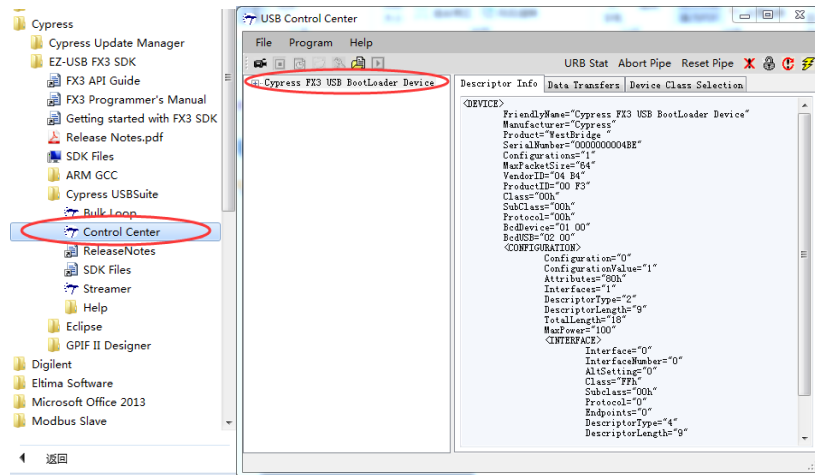
2016 年 12 月 8 日于成都市电子科技大学

流模式数据吞吐量测试

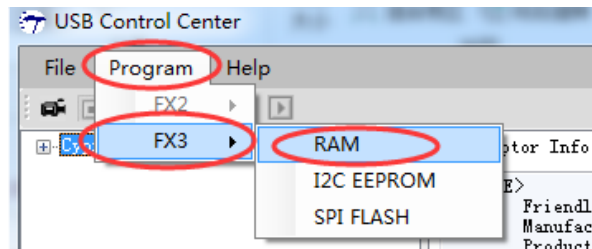
上一节，我们介绍了使用 EZ-USBFX3 的片上回环方式测试 USB 总线的可靠性。该种方式虽然能够测的 USB 总线的传输是否可靠，但是却无法体现 USB3.0 的高传输速率。本节，我们将通过另一种方式——流传输方式，来测试我们的 USB3.0 总线的传输速率。

下载 EZ-USB 固件

在开始菜单中找到 Control Center，如下图 X1 所示，打开后软件会自动扫描到我们的 USB 设备，如下图 X2 所示。



这里，我们选中设备（Cypress FX3 USB BootLoader Device），然后点击菜单栏中的 Program，依次选择 FX3 -> RAM。



在打开的文件选择窗口中，我们定位到我们提供的 DEMO 目录下的 AC6102\demo_release\ez_usb\02_USBBulkSourceSink，选择 USBBulkSourceSink.img 文件，点击打开，如下图 X1 所示，则软件会自动读取该文件并通过 USB 总线下载到 EZ-USBFX3 芯片的 RAM 中，此时，我们再看 USB Control Center，发现此时的设备名称已变成了 Cypress USB StreamerExample，左下角显示 Programming Succeeded。如下图 X2 所示。（关于 USBBulkSourceSink.img 文件的生成，将在第二期资料中提供详细的开发教程）。

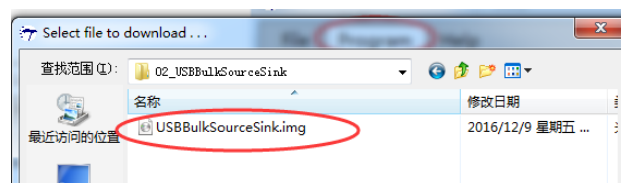


图 X1

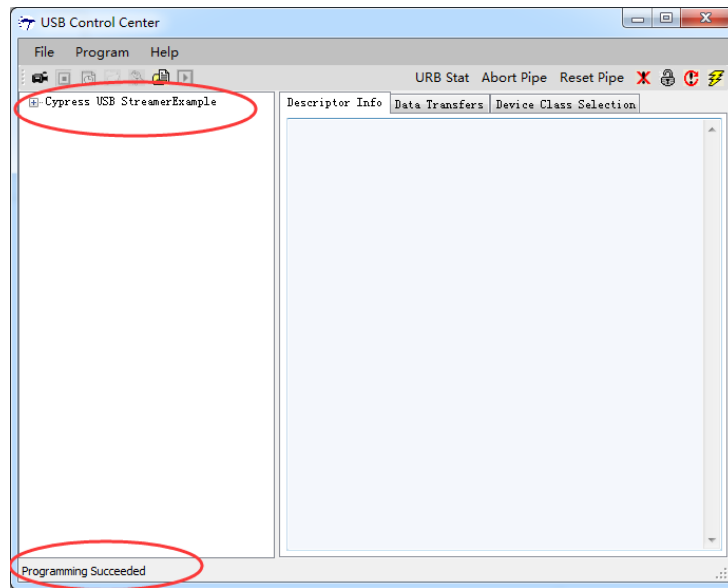
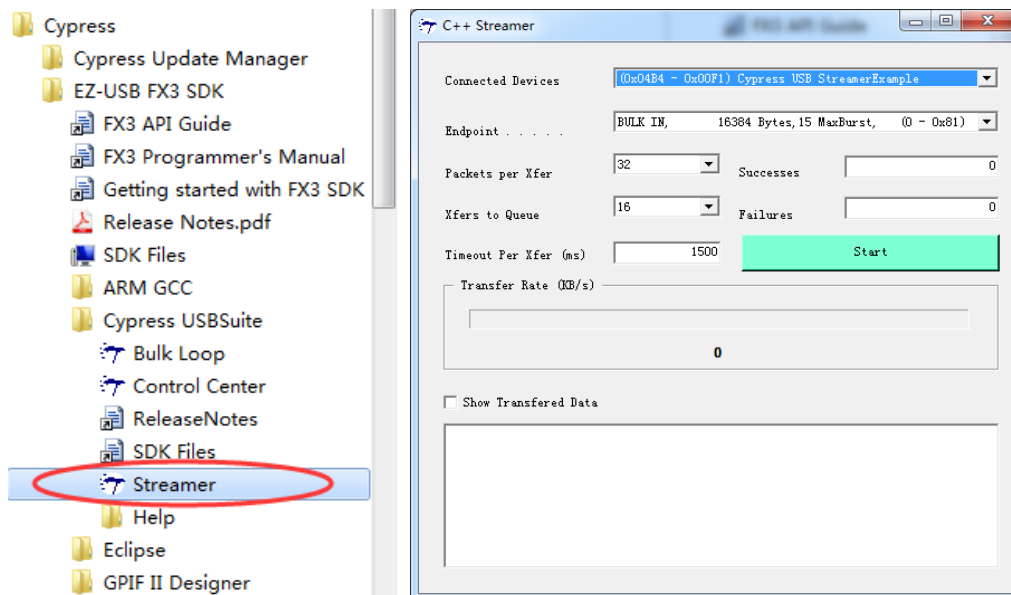


图 X2

基本 Streamer 测试

接下来，我们就使用 EZ-USBFX3 的开发套件中提供的一个专用的 steamer 测试软件来进行测试。

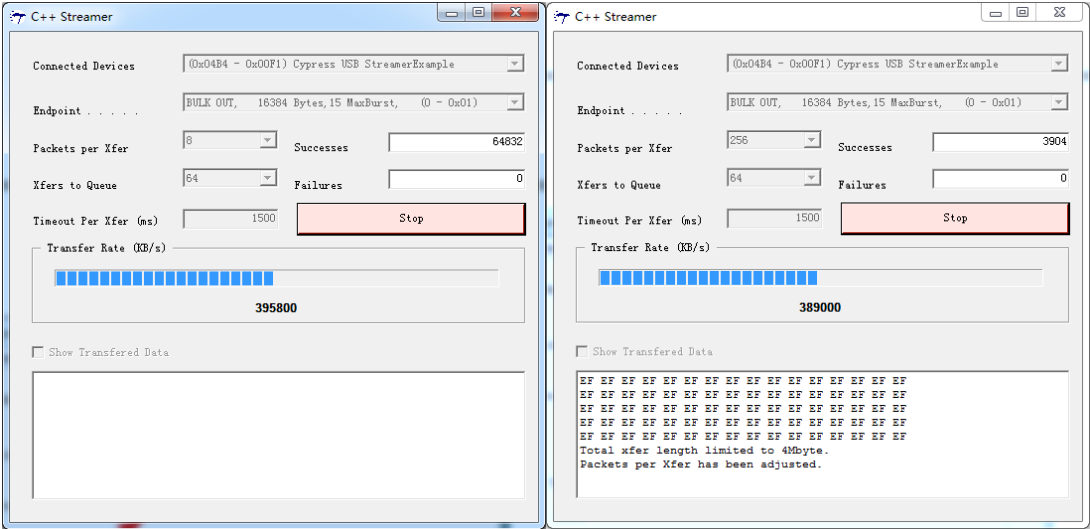
在开始菜单中找到 Bulk Loop 软件，如图 X1 所示，点击打开，打开后界面如图 X2 所示：



选择 Endpoint 为 0x01，Packets per Xfer 为 8、Xfers to Queue 为 64，然后点击 Start 开始测试，界面上的能量条会实时显示当前传输速度，也会同时显示成功

(Successes) 和失败 (Failures) 的数据包个数。可以看到，传输速度为 395800KB/S 左右，即 386MB/S。速度可谓相当的高。

点击 stop 停止传输，切换 Packets per Xfer 为 256、Xfers to Queue 为 64，然后再次点击 Start，测试速度如下所示。可以看到，传输速度稳定在 389000KB/S 左右，也就是 380MB/S，这个速度已经相当的高了。



SlaveFifo 模式应用实战

GPIO II 介绍

EZ-USBFX3 的套件提供了对 GPIFII 配置的一个快捷工具，叫做 GPIF II Designer，使用该工具，可以快速创建 GPIF II 的配置信息，然后提交给 Eclipse 编译以得到对应的固件。这里，先对 GPIF II 进行简单的介绍。然后介绍 GPIF II Designer 的使用。

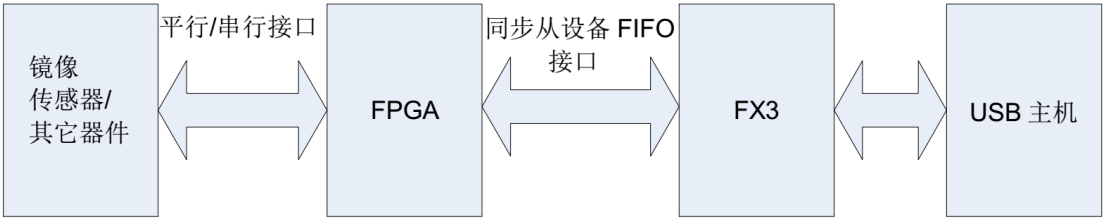
GPIF II 是一个可编程状态机，可以灵活地实现一个工业标准或专用接口。它能够作为主设备，也可以作为从设备运行。

GPIF II 具有以下特性：

- 可用作主设备或从设备
- 提供 256 种固件可编程状态
- 支持 8 位、16 位和 32 位并行数据总线
- 接口频率可高达 100 MHz
- 使用 32 位数据总线时，可支持 14 个可配置的控制引脚；所有的控制引脚可作为输入/输出或双向引脚使用
- 使用 16 位或 8 位数据总线时，可支持 16 个可配置的控制引脚；所有的控制引脚可作为输入/输出或双向引脚使用

GPIF II 状态切换根据控制输入信号发生，而控制输出信号则由 GPIF II 状态转换所驱动。状态机的运行方式由一个描述符定义。该描述符为满足所需接口规格而设计。本质上，GPIF II 描述符是一组可编程寄存器配置。EZ-USB FX3 寄存器空间中的 8 KB 专门作为 GPIF II 波形存储空间，用于存储 GPIF II 描述符。

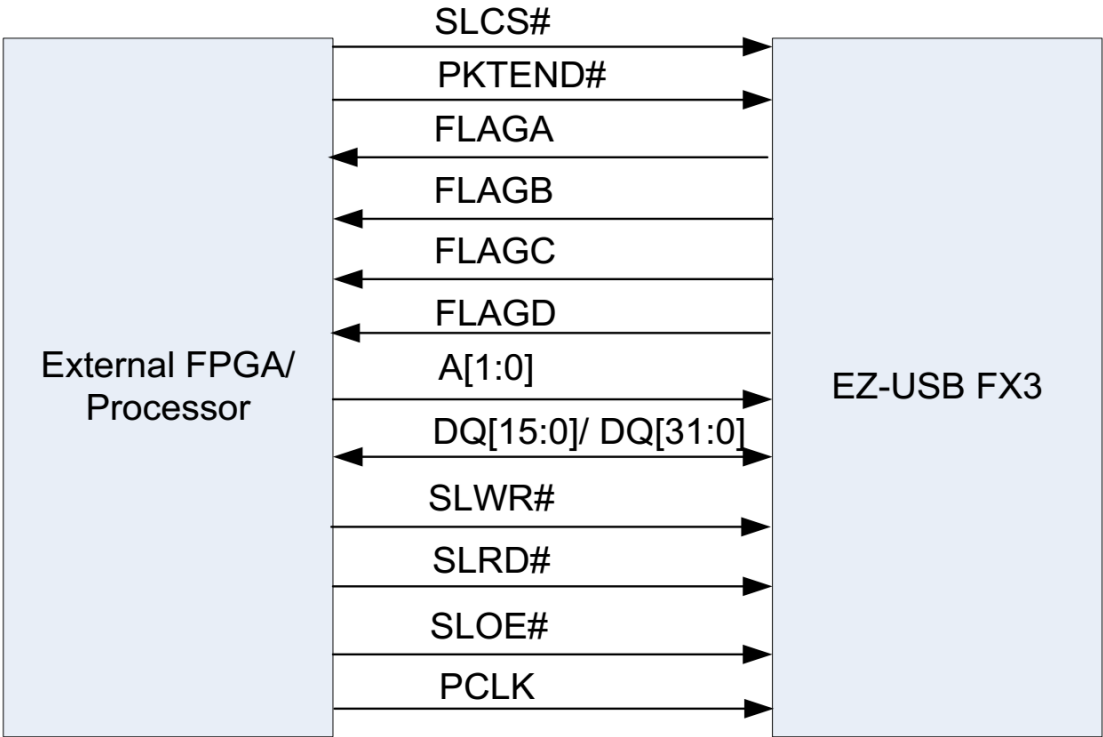
GPIO 的常见实现是同步从设备 FIFO 接口。下面章节将详尽介绍该内容。图 1 显示了使用同步从设备 FIFO 接口的示例应用框图。



同步从设备 FIFO 接口

同步从设备 FIFO 接口非常合适外部处理器或器件需要对 EZ-USB FX3 的内部 FIFO 缓冲区进行数据读/写访问的应用。寄存器访问不可通过从设备 FIFO 接口执行。同步从设备 FIFO 接口通常是 USB 应用的首选接口，可以满足高吞吐量的要求。

下图显示了同步从设备 FIFO 接口的接口框图。下表说明了图 1 中所显示的信号的功能



同步从设备 FIFO 接口信号

SLCS#	这是从设备 FIFO 接口的芯片选择信号。必须激活它，才可以访问从设备 FIFO。
SLWR#	这是从设备 FIFO 接口的写入选通信号。必须激活它，才可以对从设备 FIFO 执行写入操作。
SLRD#	这是从设备 FIFO 接口的读取选通信号。必须激活它，才可以对从设备 FIFO 执行读取操作。

SLOE#	这是输出使能信号。该信号激活后，FX3 就可驱动从设备 FIFO 接口的数据总线。必须激活它，才可以对从设备 FIFO 执行读取操作。
FLAGA/FLAGB/FLAGC/ FLAGD	这些信号均是 FX3 的标志输出。它们表示 FX3 套接字是否可用。在示例工程中，FLAGA、FLAGB 使用于从设备 FIFO 写入操作，而 FLAGC、FLAGD 使用于从设备 FIFO 读取操作。
A[1:0]	这是从设备 FIFO 的 2 位地址总线。以实现 4 个 fifo 的选择
DQ[15:0]/DQ[31:0]	这是从设备 FIFO 的 16 位或 32 位数据总线。
PKTEND#	激活该信号后，可将短数据包或零长度数据包写入从设备 FIFO 内。
PCLK	这是从设备 FIFO 接口的时钟。

带两个地址线和带五个地址线的从设备 FIFO 之间的差别

带两个地址线的同步从设备 FIFO 接口支持多达四个套接字。如要访问四个套接字以上，需要使用带五个地址线的同步从设备 FIFO 接口。除了额外的地址线之外，该接口还配有一个称为 EPSWITCH#的信号。引脚数量的增加导致用作标志的引脚数量降低，因此，需要将标志配置为一个 current_thread 标志。

使用带五个地址线的同步从设备 FIFO 接口时会产生额外的延迟：

- ◆ 每次开始传输时，在从地址得以读取到标志有效的期间将发生两个周期的延迟。
- ◆ 每当套接字地址切换时，将要经过多个周期的延迟来完成套接字的切换。

由于额外的延迟和其他接口协议要求，建议您只在应用需要访问四个 GPIF II 套接字以上时才使用带五个地址线的同步从设备 FIFO 接口。

本应用笔记中的下面部分介绍了带两个地址线的同步从设备 FIFO 接口。

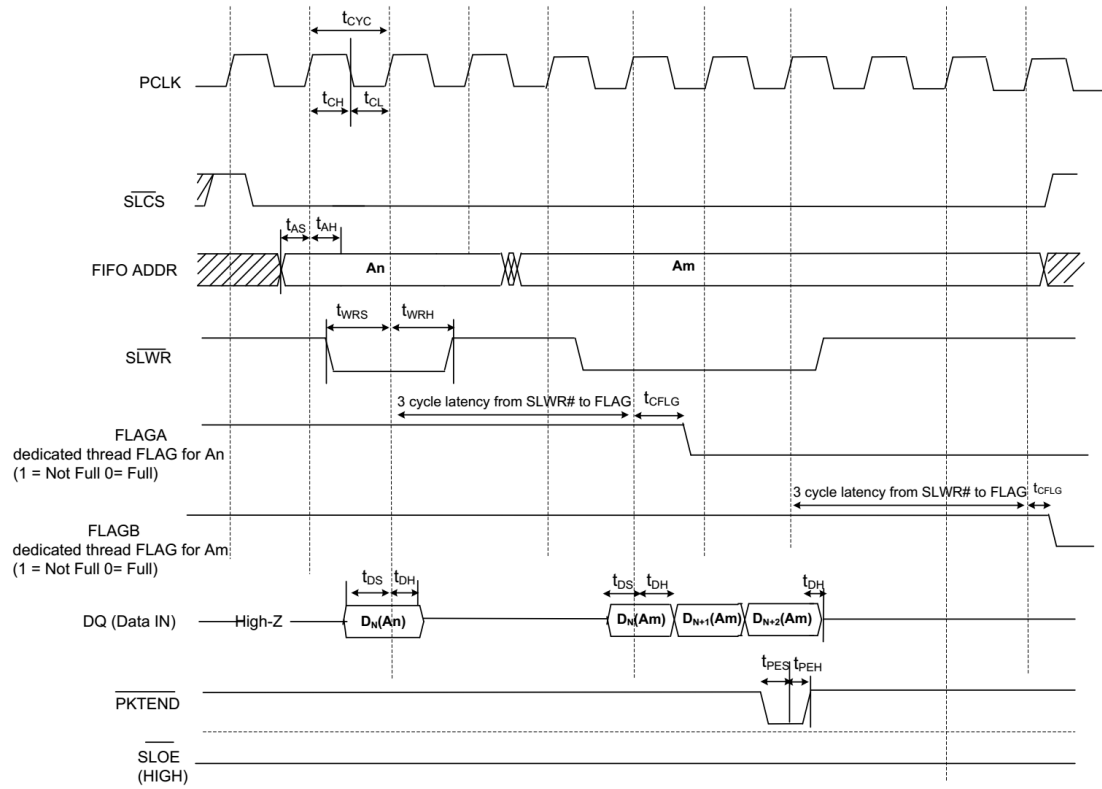
从设备 FIFO 访问序列和接口时序

本节介绍了同步从设备 FIFO 接口的访问序列和时序。

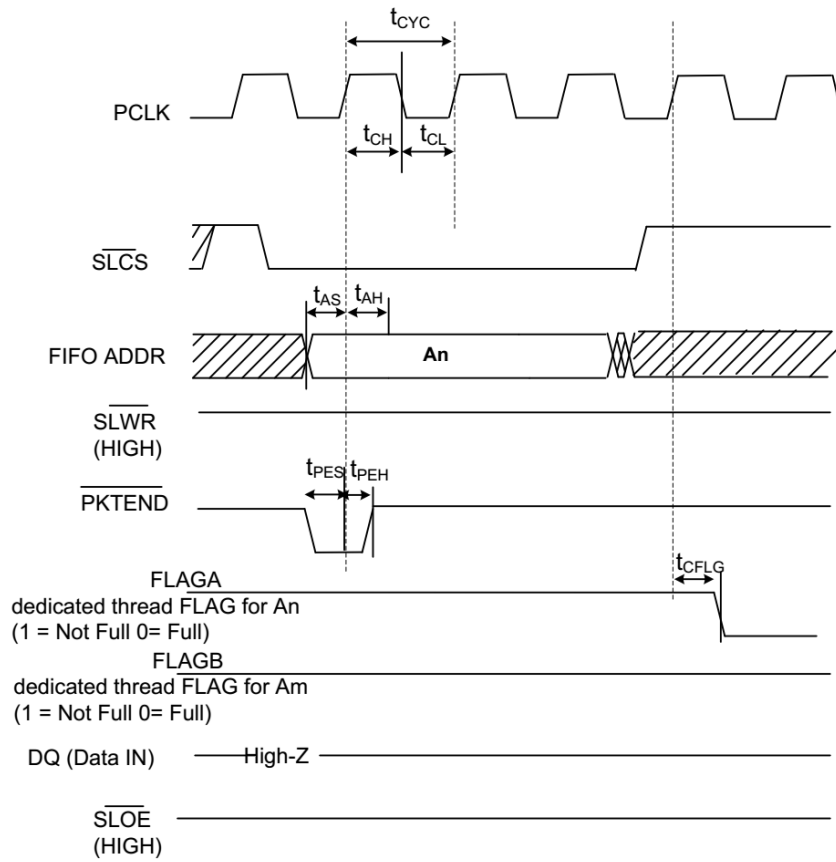
作为接口主设备的外部处理器或器件，例如 FPGA，可对 EZ-USB FX3 的内部 FIFO 缓冲区进行单周期或突发数据访问。外部主设备在 ADDR 线上输出 2 位地址，并激活读选通或写选通信号。EZ-USB FX3 激活标志信号，以表示缓冲区的空/满状态。

同步从设备 FIFO 接口时序

同步从设备 FIFO 读序列



同步 ZLP 写周期时序



同步从设备 FIFO 写序列

对同步从设备 FIFO 接口进行写操作的序列如下：

1. FIFO 地址稳定且 SLCS#信号被激活。
2. 外部主设备/外设将数据输出到数据总线上。
3. SLWR#被激活。
4. 当 SLWR#被激活时，数据将被写入到 FIFO 内，并且，FIFO 指针将在 PCLK 的上升沿上递增。
5. 从时钟的上升沿起，经过 t_{CFLG} 的延迟后，FIFO 标志将被更新。

突发写入时将发生相同的事件序列。

注意：在突发模式下，SLWR#和 SLCS#在整个突发写入过程中保持激活状态。在突发写入模式下，SLWR#被激活后，每当 PCLK 的上升沿到来时，都会将数据总线上的值写入到 FIFO 内。此外，FIFO 指针也在 PCLK 的每个上升沿上得到更新。

短数据包：通过 PKTEND#信号可将某个短数据包发送到 USB 主机。需要设计外部器件/处理器，使之在传输最后数据字时同时激活与该字相应的 SLWR#脉冲和 PKTEND#。PKTEND#激活期间，FIFOADDR 需要保持不变。同时激活 PKTEND#和 SLWR#时，GPIF II 状态机会将数据包视为短数据包，并将其发送到 USB 接口。如果协议不要求传输任何短数据包，则 PKTEND#信号可被置高。

请注意，执行读操作时，没有任何具体信号表示已从 USB 获取了短数据包。空标志必须由外部主设备监控，以确定读取完所有数据的时间。

零长度数据包：外部器件/处理器可仅通过激活 PKTEND#，而没有激活 SLWR#来传输一个零长度数据包（ZLP）。必须驱动 SLCS#和地址，如图 5 所示。

标志使用情况：外部处理器监控标志信号来控制流量。标志信号由 EZ-USB FX3 器件输出。通过配置各标志，可显示专用线程或当前寻址线程的空/满/局部状态。

同步从设备 FIFO 时序参数

参数	说明	最小值	最大值	单位
FREQ	接口时钟频率	-	100	MHz
TCYC	时钟周期	10	-	ns
tCH	时钟为高电平的时间	4	-	ns
tCL	时钟为低电平的时间	4	-	ns
TRDS	SLRD#至 CLK 的建立时间	2	-	ns
TRDH	SLRD#至 CLK 的数据保持时间	0.5	-	ns
TWRS	SLWR#至 CLK 的建立时间	2	-	ns
TWRH	SLWR#至 CLK 的数据保持时间	0.5	-	ns
TCO	时钟至数据有效的时间	-	7	ns
TDS	数据输入的建立时间	2	-	ns
TDH	时钟上升沿后数据输入保持的时间	0	-	ns

tAS	地址到时钟的建立时间	2	-	ns
tAH	CLK 至地址的地址保持时间	0.5	-	ns
tOELZ	SLOE#到数据为低阻态的时间	0	-	ns
tCFLG	时钟到标志输出的传输延迟	-	8	ns
tOEZ	SLOE#取消激活到数据为高阻态的时间	-	8	ns
tPES	PKTEND#到 CLK 的建立时间	2	-	ns
tPEH	CLK 上升沿后 PKTEND#保持的时间	0.5	-	ns
tCDH	CLK 上升沿后数据输出保持的时间	2	-	ns
注意：从 ADDR 到 DATA 的三个周期延迟				

下面各部分介绍了使用 GPIF II Designer 和 EZ-USB FX3 SDK 对标志信号进行的配置。描述各种标志配置之前，首先需要理解线程、套接字以及 DMA 通道的概念。

线程和套接字

本节简要说明了对 FX3 进行数据传入/传出所需的概念。

- 套接字
- DMA 描述符
- DMA 缓冲区
- GPIF 线程

套接字是外设硬件模块和 FX3 RAM 之间的连接点。FX3 上的每个外设硬件模块（如 USB、GPIF、UART 和 SPI）具有与其相关的固定套接字数量。流出外设的独立数据数量等于此外设上的套接字数量。使用一组寄存器来实现套接字，这些寄存器指向有效的 DMA 描述符，并使能或标记与套接字相关的中断。

DMA 描述符是一组位于 FX3 RAM 中的寄存器。它存储了 DMA 缓冲区的地址和大小数据，以及指向下一个 DMA 描述符的指针。这些指针构建成 DMA 描述符链。

DMA 缓冲区是 RAM 的一部，用于暂时存储通过 FX3 器件传输的数据。通过 FX3 固件，可将部分 RAM 空间作为 DMA 缓冲区使用。这些缓冲区的地址被存储为 DMA 描述符的一部分。

GPIF 线程是 GPIF II 模块内的专用数据路径，用来将外部数据引脚连接至套接字。套接字可通过某些事件直接互相发出信号，或者通过中断向 FX3 CPU 发出信号。该操作由固件配置的。例如，将数据流从 GPIF II 模块传输给 USB 模块。GPIF 套接字可以通知 USB 套接字它已经向 DMA 缓冲区写满了数据，或者 USB 套接字可以通知 GPIF 套接字该 DMA 缓冲区目前为空。该操作被称为自动 DMA 通道。当 FX3 CPU 不用修改数据流中的任何数据时，将实现自动 DMA 通道。

另外，GPIF 套接字可向 FX3 CPU 发送一个中断，来通知 GPIF 套接字已经写满了 DMA 缓冲区。FX3 CPU 可将该信息传递给 USB 套接字。USB 套接字可向 FX3 CPU 发送一个中断，来通知 USB 套接字已经读空了 DMA 缓冲区。此时，FX3 CPU 可将该信

息反馈给 GPIF 套接字。该操作被称为手动 DMA 通道实现。如果 FX3 CPU 需要添加、删除或修改数据流中的数据，将需要执行该操作。

发送套接字是指用于将数据写入 DMA 缓冲区内的套接字。接收套接字是指从 DMA 缓冲区内读取数据的套接字。套接字使用存储于 DMA 描述符上的 DMA 缓冲区地址、DMA 缓冲区大小和 DMA 描述符链的值来管理数据。套接字写满或读空 DMA 缓冲区后，需要经过一段时间（几微秒）从一个 DMA 描述符转移到另一个描述符。转换过程中，套接字不能传输数据。

EZ-USB FX3 可提供多达四个物理硬件线程，以用于通过 GPIF II 传输数据。每次可将任何一个套接字映射到一个物理线程上。默认情况下，PIB 套接字 0 被映射到线程 0，PIB 套接字 1 被映射到线程 1，PIB 套接字 2 被映射到线程 2 以及 PIB 套接字 3 被映射到线程 3。请注意，接口上的地址信号 A1:A0 表示需要访问的线程。FX3 的 DMA 运行机制将数据传输到映射到该线程上的套接字。因此，当 A1:A0=0 时，将访问线程 0，并且，通过线程 0 传输的所有数据将被传输到套接字 0。同样地，当 A1:A0=1 时，将数据传入/传出套接字 1。

注意：从设备 FIFO 接口仅包含两个地址线，所以最多只能访问四个套接字。如要访问四个套接字以上，必须使用带五个地址线的从设备 FIFO 接口。

DMA 通道配置

固件必须使用所需的发送套接字和接收套接字来配置 DMA 通道。

如果将数据从从设备 FIFO 接口传输到 USB 接口，那么“P”端口将为发送套接字，USB 为接收套接字，反之亦然。因此，若要通过从设备 FIFO 接口实现双向数据传输，则需配置两个 DMA 通道，其中一个带有作为发送套接字的“P”端口，另一个带有作为接收套接字的“P”端口。

作为发送套接字的“P”端口是外部器件通过从设备 FIFO 接口对其进行写入的套接字；作为接收套接字的“P”端口是外部器件通过从设备 FIFO 接口对其进行读取的套接字。

DMA 通道上的“P”端口套接字编号要与 A1:A0 上所寻址的套接字编号相同。

配置 DMA 通道时，可将多个缓冲区分配给一个特定的 DMA 通道。请注意，标志将分别表示每个缓冲区的满/空状态（单个缓冲区的最大尺寸为 64 KB-16）。

例如，如果已将两个大小均为 1024 字节的缓冲区分配给 DMA 通道，则当将 1024 字节写入到第一个缓冲区后，满标志将表示“满”状态。它持续显示满状态，直至 DMA 通道转到第二个缓冲区为止。虽然 DMA 通道转到下一个缓冲区所需的时长通常为几微秒，但未能确定该时长。外部主设备必须监控标志来确定完成切换操作以及可用下一个缓冲区来访问数据的时间。下一节将说明如何配置标志，以表示不同线程的状态。

标志配置

标志可被配置为空、满、一部分空或一部分满信号。这些信号并非由 GPIF II 状态机控制，而是由 EZ-USB FX3 内部的 DMA 硬件引擎控制。标志与特定线程或当前寻址的线程相关联，用于表示映射到该线程的套接字状态。

标志将根据（初始化套接字过程中所配置的）套接字方向来指示空或满状态。因此，如果从套接字读取数据，标志将表示空/未空状态；如果对套接字写入数据，标志将表示满/未满状态。

可用的标志类型分别为：

- 专用线程标志（空/满或一部分空/一部分满）
- 当前线程标志（空/满或一部分空/一部分满）

下面各节将介绍不同的标志类型。不同的标志配置会导致不同的延迟，如表 4 中所述。

专用线程标志

可对标志进行配置，以指示特定线程的状态。在这种情况下，无论在地址总线上寻址哪个线程，该标志将专门用于该线程，且始终只表示映射到该特定线程的套接字的状态。此时，外部处理器/器件必须跟踪专用于各线程的标志，并且每当寻址到不同线程时需要保持监控正确标志。

例如，如果 FLAGA 专用于线程 0，FLAGB 专用于线程 1，当外部处理器对线程 0 进行访问时，必须监控 FLAGA。当外部处理器访问线程 1 时，则必须监控 FLAGB。

标志可专用于将要访问的某个线程。如果应用要求访问四个线程，将会使用四个相应的标志。

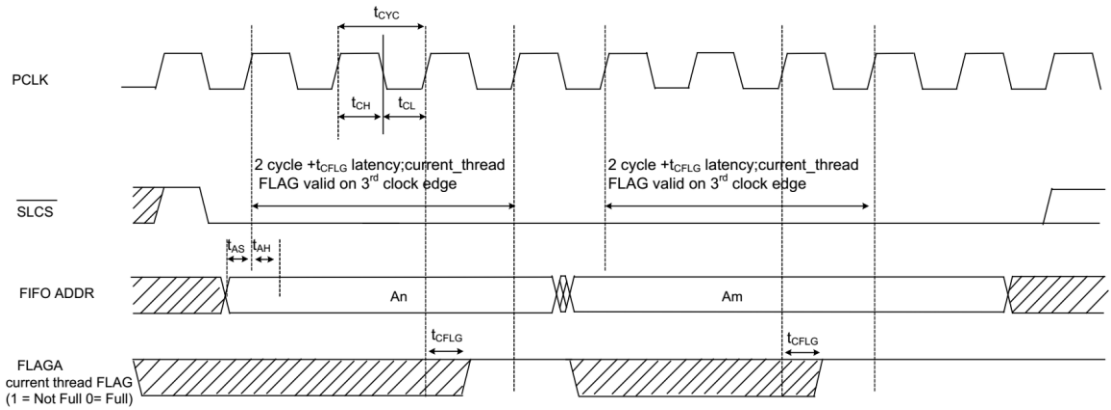
请注意，写传输过程结束时，标志始终经过三个周期的延迟。这三个周期的延迟是从使缓冲区变为满的写周期到标志被置低的时间。在第四个时钟沿上，外部主设备可以对处于低电平的标志进行采样，如图 4 所示。

读传输过程结束时，标志始终经过两个周期的延迟。两个周期的延迟是从使缓冲区变为空的读周期（最后一次激活 SLRD#）到标志被置低的时间。在第三个时钟沿上，外部主设备可以对处于低电平的标志进行采样，如图 3 所示。

当前线程标志

可对标志进行配置，来指示当前寻址的线程状态。在这种情况下，GPIF II 状态机对地址总线上的地址进行采样，然后更新相应标志，用于指示该线程的状态。这种配置要求的引脚数量更少，因为单个“current_thread”标志可用来表示四个线程的状态。但当同步从设备 FIFO 接口使用 current_thread 标志时，将发生两个周期的延迟，因为 GPIF II 先要对地址

进行采样，然后再更新该标志。两个周期的延迟是从有效地址被传输到接口算起。该事件发生后，在第三个时钟沿上，可以对刚寻址线程的标志的有效状态进行采样（请注意， SDK 中的从设备 FIFO 描述符使用 “current_thread”标志配置）。



注意：写传输过程结束时，标志始终经过三个周期的延迟。这三个周期的延迟是从使缓冲区变为满的写周期到标志被置低的时间。在第四个时钟沿上，外部主设备可以对处于低电平的标志进行采样，如图 4 所示。

读传输过程结束时，标志始终经过两个周期的延迟。两个周期的延迟是从使缓冲区变为空的读周期（最后一次激活 SLRD#）到标志被置低的时间。在第三个时钟沿上，外部主设备可以对处于低电平的标志进行采样，如图 3 所示。

局部标志

可对标志进行配置，以指示套接字的一部分空/一部分满状态。必须选择满足下面条件的水印值：当将要读/写的 32 位字数低于或等于该水印值时，将激活标志。

注意：局部标志的延迟取决于该标志的特定水印值。

表 4 汇总了各种标志配置的相应延迟。该表还显示了 GPIF II Designer 中必须为特定标志选择的设置。有关标志的 GPIFII Designer 设置的示例和截图，请查看标志配置一节。

表 4. 不同标志配置的相关延迟

标志配置	GPIF II Designer 标志设置选择	传输开始时从地址传输到标志的延迟	传输结束时的标志延迟		所需的附加 API 调用
			对从设备 FIFO 进行的写操作（从最后一次激活 SLWR# 到激活满标志的延迟）	对从设备 FIFO 进行的读操作（从最后一次激活 SLRD# 到激活空标志的延迟）	

专用于特定线程“n”的满/空标志	Thread_n_DMA_Ready	0 周期	3 个周期 + tCFLG (外部器件可在第 4 个时钟沿上对有效标志进行采样)	2 个周期 + tCFLG (外部器件可在第 3 个时钟沿上对有效标志进行采样)	N/A
用于当前寻址线程的满/空标志	Current_thread_DMA_Ready	2 个周期 +tCFLG (外部器件可在第 3 个时钟沿上对有效标志进行采样)	3 个周期 + tCFLG (外部器件可在第 4 个时钟沿上对有效标志进行采样)	2 个周期 + tCFLG (外部器件可在第 3 个时钟沿上对有效标志进行采样)	N/A
专用于特定线程“n”的一部分满/一部分空标志	Thread_n_DMA_Watermark	0 周期	取决于水印等级	取决于水印等级	通过调用 CyU3PGpifSocketConfigure() API 来设置水印等级。注意：水印的单位为 32 位数据字。示例： CyU3PGpifSocketConfigure(0,PIB_SOCKET_0,4,CyFalse,1)设置了线程 0 到 4 的水印值 CyU3PGpifSocketConfigure(3,PIB_SOCKET_3,4,CyFalse,1)设置了线程 3 到 4 的水印值
用于当前寻址线程的一部分满/一部分空标志	Current_thread_DMA_Watermark	2 个周期 +tCFLG (外部器件可在第 3 个时钟沿上对有效标志进行采样)	取决于水印等级	取决于水印等级	通过调用 CyU3PGpifSocketConfigure() API 来设置水印等级。注意：水印的单位为 32 位数据字。示例： CyU3PGpifSocketConfigure(0,PIB_SOCKET_0,4,CyFalse,1)设置了线程 0 到 4 的水印值

					CyU3PGpifSocketConfigure(3,PIB_SOCKET_3,4,CyFalse,1)设置了线程 3 到 4 的水印值
--	--	--	--	--	--

下一节将介绍如何使用 GPIF II Designer 工具和 EZ-USB FX3 SDK 来配置标志。

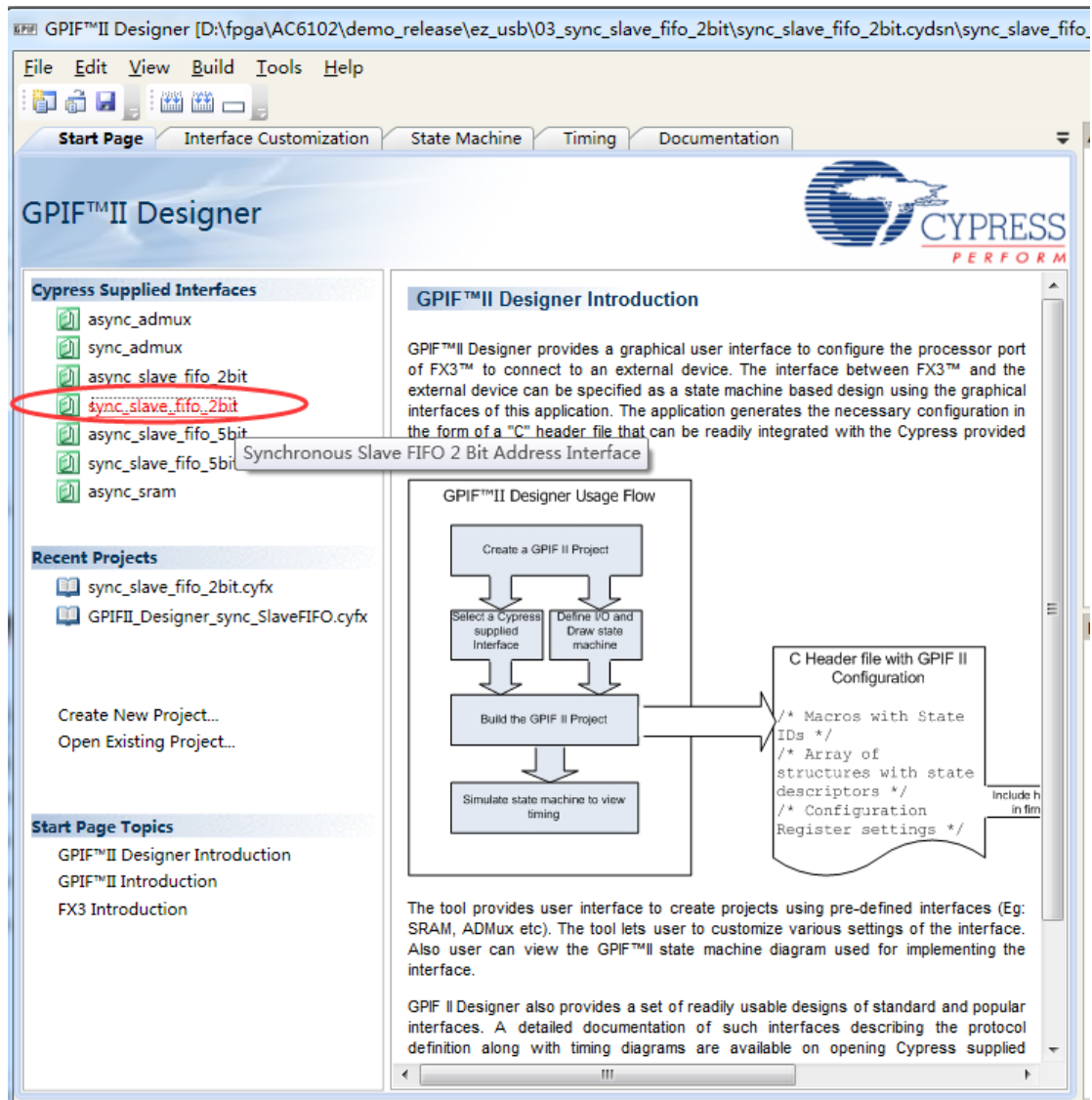
GPIF II Designer

实现同步从设备 FIFO 接口

通过安装 GPIF II Designer 工具， 用户可以看到从设备 FIFO 接口的 GPIF II 实现指导。从赛普拉斯网站 www.cypress.com 上安装 GPIF II Designer 工具。当启动 GPIF II Designer 时，可以在起始页上寻找 **Cypress Supplied Interfaces**（赛普拉斯提供的接口） 项。

GPIF II Designer 中提供了对 GPIF II 的若干示例配置，其中就包括与 FPGA 接口时最常用的同步从 fifo 接口，sync_slave_fifo_2bit 项目是使用 2 位地址进行同步从设备 FIFO 接口的 GPIF II 实现。下一节将说明如何使用 GPIF II Designer 配置局部标志。

GPIF II Designer 工具内的从设备 FIFO 项目 — Cypress Supplied Interfaces

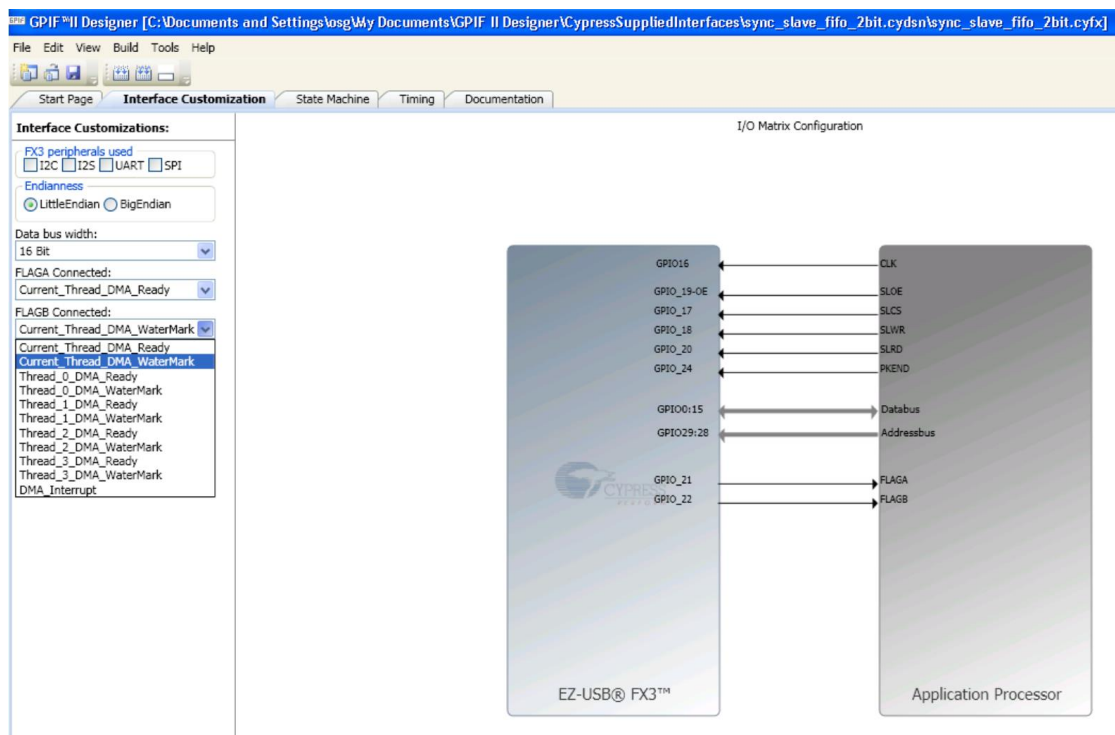
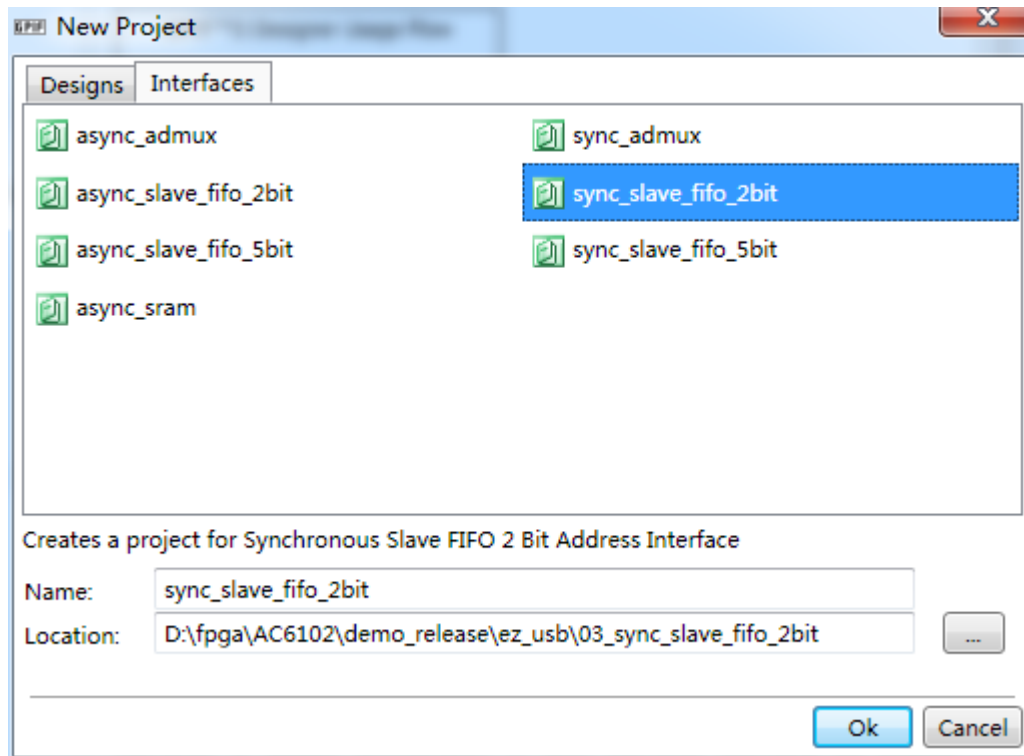


配置局部标志

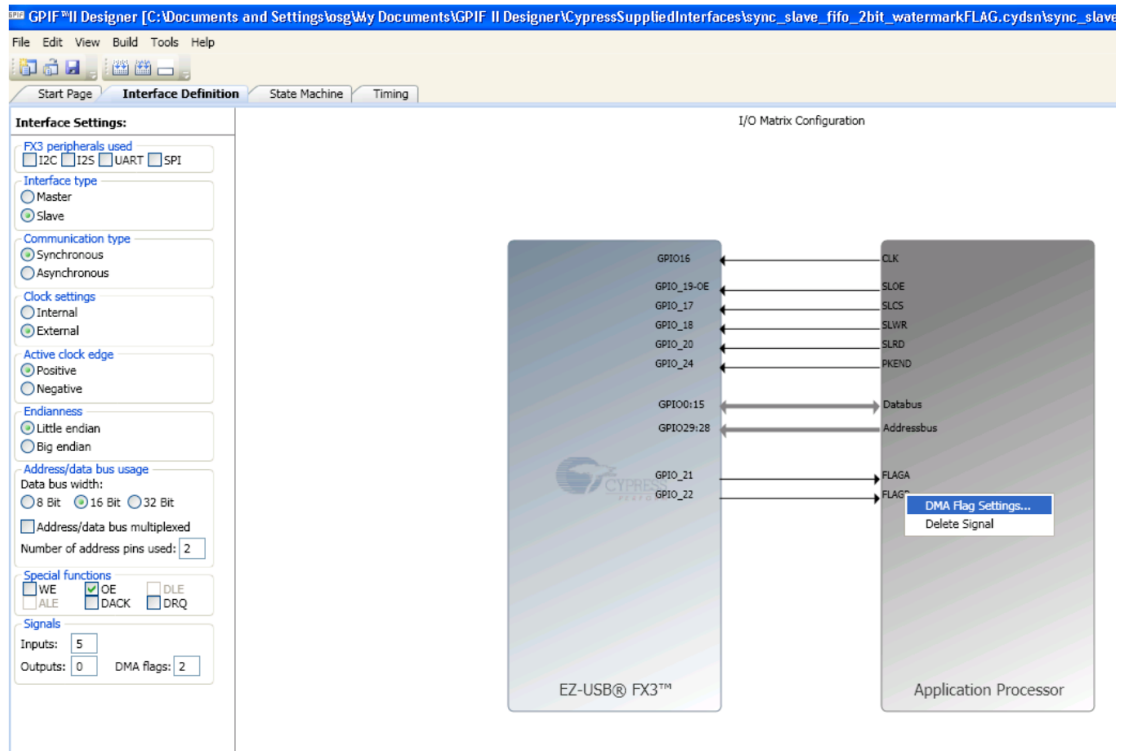
按照下面各个步骤对局部标志进行配置：

- 必须在 GPIF II Designer 工具中选中局部标志设置。
- 必须使用固件中的 CyU3PGpifSocketConfigure() API 设置局部标志的水印等级。

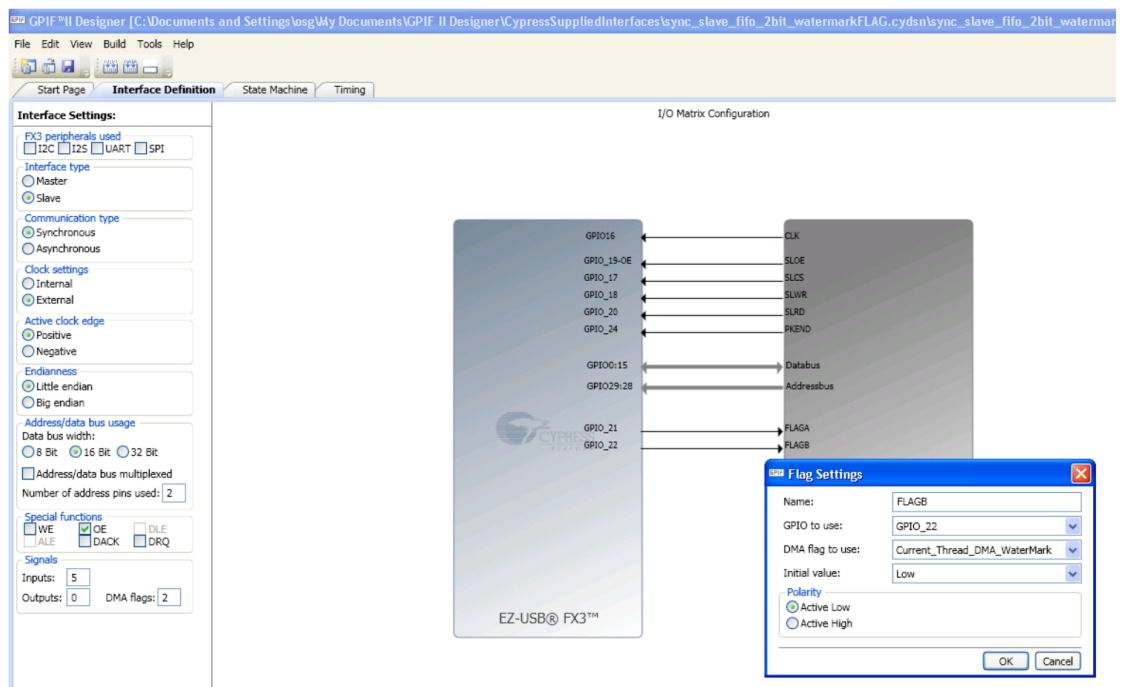
在 GPIF II Designer 工具中，打开 Cypress Supplied Interfaces 下方的 **sync_slave_fifo_2bit** 项目，然后选中“FLAGA Connected”或“FLAGB Connected”下的 **Current_Thread_DMA_Watermark**，以将标志配置为当前线程的局部标志。或者，选中 **Thread_n_DMA_Watermark**，以将标志配置为线程“n”的专用局部标志（n 等于 0、1、2、3）。



除了 *sync_slave_fifo_2bit.cyfx* 项目中所提供的设置之外， 若想要添加更多标志或进行更多修改， 请依次选择 **File > Save Project as Editable**。这样， 可以将该项目保存为一个新的名称， 从而对新保存的项目进行更改。在这种情况下，如要将某个标志配置为局部标志，请右击 I/O 矩阵配置框图中的标志。点击 **DMA Flag Settings**， 如下图所示。



下图显示了如何选择标志配置：



配置局部标志的第二步是指定固件项目中的标志水印值。在 `cyfxslfifo.c` 文件中，添加一个 `CyU3PGpifSocketConfigure()` API 调用，以指定水印值。在调用 `CyU3PGPIFLoad()` API 后可立即调用该 API。有关 `CyU3PGpifSocketConfigure()` API 的完整说明，请参阅 EZ-USB FX3 SDK API 指南。水印值是输入该 API 的参数之一。

水印值决定何时激活局部标志。下面一节介绍了用于计算激活局部标志后将要读/写的数字数量的公式。

使用局部标志情况下的通用公式

前一节描述了标志的可能配置以及配置局部标志的各步骤。本节说明如何确定某个局部标志的水印值。激活局部标志后，应使用下面公式来计算将要读/写的数字字数量。

注意： 在 `CyU3PgpifSocketConfigure()` API 中指定的水印值的单位为 32 位数据字。

1. 当外部主设备对同步从设备 **FIFO** 进行写入时:
 - (a) 时钟沿到来（这时，对处于低电平的局部标志进行采样）后将要写入的数据字的数量 = 水印值 $\times$ （ 32/总线宽度） - 4
2. 当外部主设备对同步从设备 **FIFO** 进行读取时:
 - (a) 时钟沿到来（这时，对处于激活状态的局部标志进行采样）后可用于读取的数据字的数量（ **SLOE#**维持激活时） = 水印值 $\times$ （ 32/总线宽度） - 1
 - (b) 已包含从 **SLRD#**置低到数据在总线上有效的两个周期的延迟。 因此，时钟沿到来（这时，对处于激活状态的局部标志进行采样）后维持激活 **SLRD#**的周期数 = 水印值 $\times$ （ 32/总线宽度） - 3。

CyU3PGpifSocketConfigure() API 使用示例

本节介绍了使用 `CyU3PGpifSocketConfigure()` API 指定的水印值的影响示例。还提供了截图，以明确显示使用不同水印值时的局部标志性能。

注意：在这些示例中，标志的极性被设为低电平有效。因此，当标志为低电平时，它们指示满/空或一部分满/一部分空状态。

示例 1

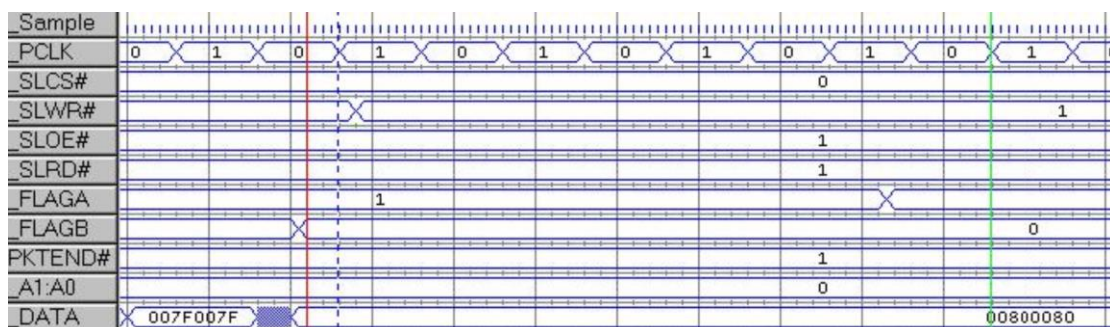
使用 32 位数据总线的从设备 FIFO:

- 在 GPIF II Designer 中，FLAGA 被配置为 Current_thread_DMA_RDY，FLAGB 被配置为 Current_thread_DMA_watermark。
- CyU3PGpifSocketConfigure (0,PIB_SOCKET_0,4,CyFalse,1)
- 外部 FPGA 通过从设备 FIFO 对 EZ-USB FX3 进行突发写入传输（最后写入的数据为 0x00800080）。

下面显示了标志在传输结束时变为 0 的逻辑分析器截图。可见，FLAGB（局部标志）在写入最后数据字的同一个周期内转为低电平。

使用 32 位数据总线宽度的突发写入传输

(FLAGA= Current_thread_DMA_RDY, FLAGB= Current_thread_DMA_watermark, watermark = 4)



示例 2

使用 32 位数据总线的从设备 FIFO:

- 在 GPIF II Designer 中, FLAGA 被配置为 Current_thread_DMA_RDY, FLAGB 被配置为 Current_thread_DMA_watermark。
- CyU3PGpifSocketConfigure (3, PIB_SOCKET_3, 4, CyFalse, 1)
- 外部 FPGA 通过从设备 FIFO 对 EZ-USB FX3 进行突发读取传输 (将要读取的最后数据为 0x00000080)。

下面显示了标志在传输结束时变为 0 的逻辑分析器截图。可见, 在读取最后数据前四个周期, FLAGB (局部标志) 转为低电平。这样, 在 FLAGB 转为低电平的周期后, 可以读取三个数据字。

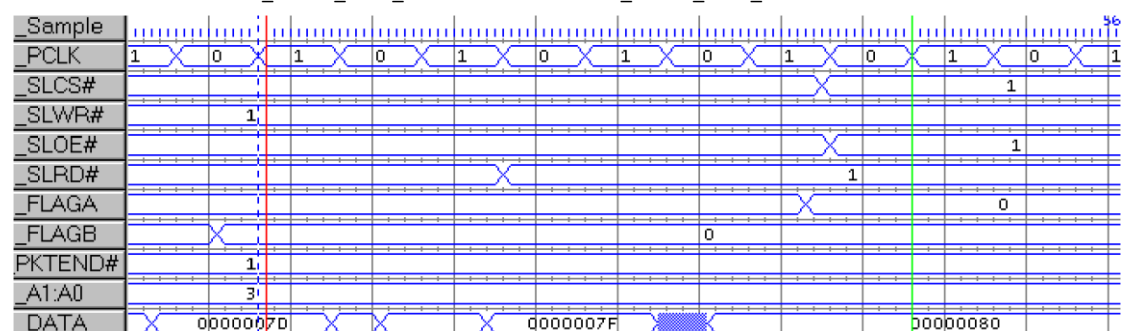
在该示例中, 可使用使用局部标志情况下的通用公式部分中的公式 2(a), 具体如下:

水印值 = 4, 总线宽度 = 32

因此, 时钟沿到来 (这时, 对处于激活状态的局部标志进行采样) 后可用于读取的 32 位数据字的数量 = $4 \times (32/32) - 1 = 3$ 。

使用 32 位数据总线宽度的突发读取传输

(FLAGA= Current_thread_DMA_RDY, FLAGB= Current_thread_DMA_watermark, watermark = 4)



示例 3

使用 16 位数据总线的从设备 FIFO:

- 在 GPIF II Designer 中, FLAGA 被配置为 Current_thread_DMA_RDY, FLAGB 被配置为 Current_thread_DMA_watermark。
- CyU3PGpifSocketConfigure (0, PIB_SOCKET_3, 3, CyFalse, 1)
- 外部 FPGA 通过从设备 FIFO 对 EZ-USB FX3 进行突发写入传输 (最后写入的数据为 0x0200)。

下面显示了标志在传输结束时变为 0 的逻辑分析器截图。可见, 在写入最后数据前的三个周期内, FLAGB (局部标志) 处于低电平。这样, 在 FLAGB 转为低电平的周期后, 可以读取两个数据字。

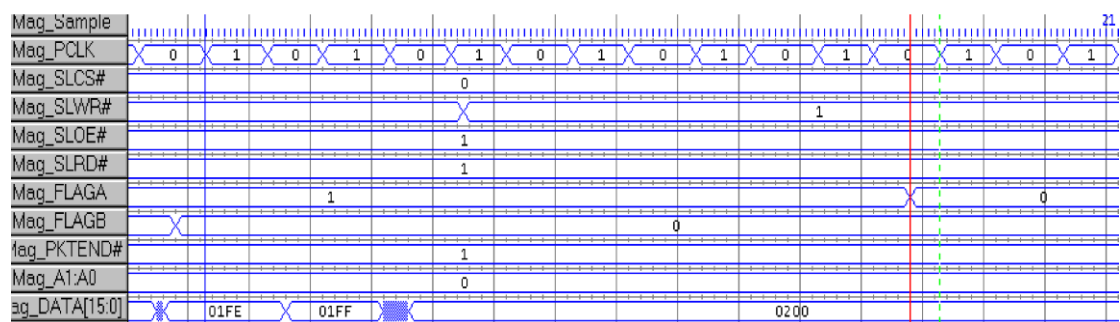
在该示例中, 可以使用使用局部标志情况下的通用公式部分中的公式 1, 具体如下:

水印值 = 3, 总线宽度 = 16

因此, 时钟沿 (所采样的局部标志为低电平时) 后可写入的 16 位数据字数量 = $3 \times (32/16) - 4 = 2$

使用 16 位数据总线宽度的突发写入传输

(FLAGA= Current_thread_DMA_RDY, FLAGB= Current_thread_DMA_watermark, watermark = 3)



示例 4

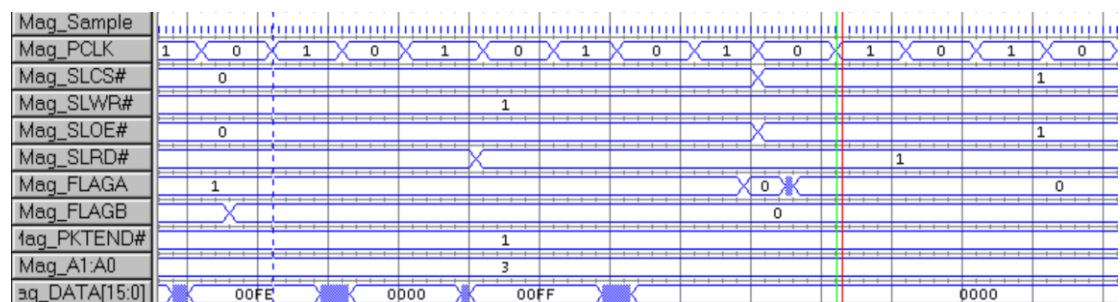
使用 16 位数据总线的从设备 FIFO:

- 在 GPIF II Designer 中，FLAGA 被配置为 Current_thread_DMA_RDY，FLAGB 被配置为 Current_thread_DMA_watermark。
- CyU3PGpifSocketConfigure (3, PIB_SOCKET_3, 2, CyFalse, 1)
- 外部 FPGA 通过从设备 FIFO 对 EZ-USB FX3 进行突发读取传输（最后写入的数据为 0x0000）。

下面显示了标志在传输结束时变为 0 的逻辑分析器截图。可见，在读取最后数据前的四个周期内，FLAGB（局部标志）处于低电平。这样，在 FLAGB 转为低电平的周期后，可以读取三个数据字。

使用 16 位数据总线宽度的突发读取传输

（FLAGA= Current_thread_DMA_RDY，FLAGB= Current_thread_DMA_watermark，watermark = 2）



使用局部标志时的其他注意事项:

- 局部标志仅用于决定传输结束的时间。必须在开始传输时监控满/空标志，以确保套接字的可用性。这意味着，不能单独使用局部标志，而必须和满/空标志结合使用。
- 如果外部主设备能够执行一个计数机制，并始终写入与 EZ-USB FX3 的 DMA 缓冲区大小相同的数据量，就能完全避免使用局部标志。外部主设备应计算将要写入或读取的数据量，并保证该计数值不超过创建 DMA 通道时所设置的缓冲区大小。在这种情况下，应通过监控满/空标志来决定开始传输的时间。
- 如果尚未执行上面介绍的计数机制，并且使用了一个局部标志，请按照下面各步骤的其中一个进行操作：

- 如果外部主设备始终使用固定的数据量进行突发传输，那么，必须在选择水印值时

考虑到该突发的大小。在 `CyU3PGpifSocketConfigure()` API 中设置突发值，该值为此 API 中的最后参数。突发值被设置后，DMA 硬件将在每个突发传输后（而不是每个字传输后）检测水印值。例如，如果外部主设备始终进行 8 个字的突发写入，则必须设置水印值，使之满足下面条件：当 EZ-USB FX3 的 DMA 缓冲区中有 8 字突发的空间，标志将转为低电平。在局部标志转为低电平后，外部主设备可写入一个完整的 8 字突发。对于在 16 位模式下（对从设备 FIFO 进行）的写操作，请根据使用局部标志情况下的通用公式一节所介绍的公式，将 `CyU3PGpifSocketConfigure()` API 中的水印值设为 ‘6’。

- 前一步骤的备用步骤是，在局部标志变为零后，外部主设备不会执行突发访问，而切换到单周期访问模式。然后，在每个周期，外部主设备将在写入前先检查满/空标志，以确保缓冲区是可用的。

由标志违规引起的错误条件

当局部标志或满/空标志指示缓冲区尚未可用时，不能对从设备 FIFO 接口进行数据读/写访问。

如果读取空白的缓冲区，将发生缓冲区欠载错误。如果写入到已满的缓冲区，则发生缓冲区过载错误。这些错误可导致缓冲区边界的数据被损坏。由于从设备 FIFO 接口位于 FX3 的 PIB 模块域，因此，与接口相关的所有错误将被表示为 PIB 错误。

如果发生某个 PIB 错误，将触发 PIB 中断。FX3 SDK 允许寄存一个用于 PIB 中断的回调函数。该回调函数用于检查 PIB 错误代码。下面代码示例介绍了如何寄存一个回调函数以及用于检查欠载/过载错误并弹出一个调试信息的实际回调函数。

错误代码指示发生错误的线程，如表 5 所示。如果发生这些错误中的任意一个，赛普拉斯官方建议您使用一个逻辑分析器谨慎地分析接口时序，并特别注意局部标志变为零后所执行的读/写周期数量。有关的示例，请参考图 11 到图 14。

```
/* Register a callback for notification of PIB interrupts*/
CyU3PpibRegisterCallback(gpif_error_cb,intMask);
/* Callback function to check for PIB ERROR*/
void gpif_error_cb(CyU3PpibIntrType cbType, uint16_t cbArg)
{
    if(cbType==CYU3P_PIB_INTR_ERROR)
    {
        switch(CYU3P_GET_PIB_ERROR_TYPE(cbArg))
        {
            case CYU3P_PIB_ERR_THR0_WR_OVERRUN://线程 0 缓冲区中的写入过载
                CyU3PdebugPrint (4, "CYU3P_PIB_ERR_THR0_WR_OVERRUN");
                break;
            case CYU3P_PIB_ERR_THR1_WR_OVERRUN://线程 1 缓冲区中的写入过载
                CyU3PdebugPrint (4, "CYU3P_PIB_ERR_THR1_WR_OVERRUN");
```

```

        break;
    case CYU3P_PIB_ERR_THR2_WR_OVERRUN://线程 2 缓冲区中的写入过载
        CyU3PdebugPrint (4, "CYU3P_PIB_ERR_THR2_WR_OVERRUN");
        break;
    case CYU3P_PIB_ERR_THR3_WR_OVERRUN://线程 3 缓冲区中的写入过载
        CyU3PdebugPrint (4, "CYU3P_PIB_ERR_THR3_WR_OVERRUN");
        break;
    case CYU3P_PIB_ERR_THR0_RD_UNDERRUN://线程 0 缓冲区中的读取欠载
        CyU3PdebugPrint (4, "CYU3P_PIB_ERR_THR0_RD_UNDERRUN");
        break;
    case CYU3P_PIB_ERR_THR1_RD_UNDERRUN://线程 1 缓冲区中的读取欠载
        CyU3PdebugPrint (4, "CYU3P_PIB_ERR_THR1_RD_UNDERRUN");
        break;
    case CYU3P_PIB_ERR_THR2_RD_UNDERRUN://线程 2 缓冲区中的读取欠载
        CyU3PdebugPrint (4, "CYU3P_PIB_ERR_THR2_RD_UNDERRUN");
        break;
    case CYU3P_PIB_ERR_THR3_RD_UNDERRUN://线程 3 缓冲区中的读取欠载
        CyU3PdebugPrint (4, "CYU3P_PIB_ERR_THR3_RD_UNDERRUN");
        break;
    default:
        CyU3PdebugPrint (4, "No Underrun/Overrun Error");
        break;
    }
}
}
}

```

SDK 内的从设备 FIFO 固件示例

本应用笔记介绍了同步从设备 FIFO 接口以及配置标志的方法。实现 GPIF II Designer 工具中的所需配置后，需要将已更新的配置集成到固件内。完成编译 GPIF II Designer 中的项目后，将生成一个头文件 *cyfxgpiifconfig.h*。该头文件必须包含在固件项目中。EZ-USB FX3 SDK 包含一个集成了从设备 FIFO 接口的固件示例。

安装 EZ-USB FX3 SDK 后，可在下面目录中找到集成了同步从设备 FIFO 接口的固件示例：
[FX3 SDK Install Path]\EZ-USB FX3 SDK\1.3\firmware\slavefifo_examples\slfifosync

该固件示例支持 16 位和 32 位数据总线宽度。头文件 *cyfxslfifosync.h* 定义了常量 *CY_FX_SLFIFO_GPIF_16_32BIT_CONF_SELECT*。想要选择 32 位数据总线宽度，请将该常量设为 1；若想选择 16 位数据总线宽度，则将该常量设为 0。

注意：如果从设备 FIFO 在 100 MHz 的工作频率下使用 32 位数据总线运行，那么，必须将 PLL 频率配置为 400 MHz。为实现该操作，要将 *setSysClk400* 参数设置为 *CyU3PDeviceInit()* 函数的输入。更多信息，请参考 FX3 SDK 中的 API 指南。

设计示例：使用 AC6102 上 FPGA 与 FX3 的同步从设备 FIFO 接口

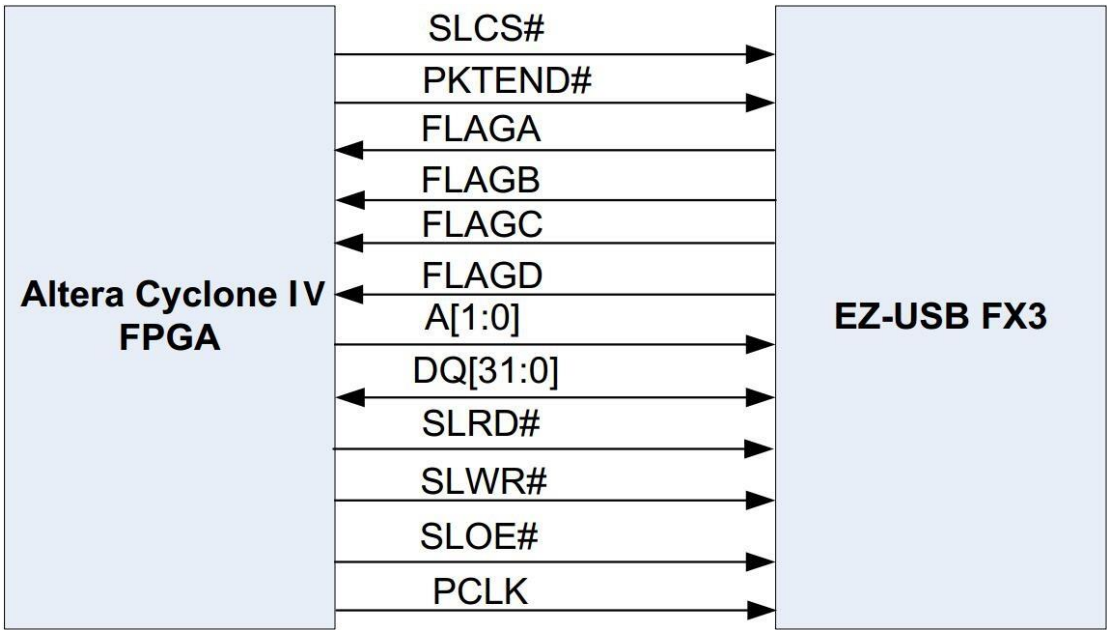
本节将提供一个基于 AC6102 开发板进行 USB 传输完整的设计示例，其中的 AC6102 开发板上的 FPGA 芯片通过同步从设备 FIFO 接口连接至 USB3.0 芯片 CYUSB3014。本节还描述用于实现该设计的硬件、固件和软件组件。

固件和软件组件

本实例用的的固件和软件组件主要包括以下内容：

- FX3 同步从设备 FIFO 固件。
- Control Center 和 Streamer 软件工具。

下图显示了 FPGA 和 FX3 之间的互连图。



该示例包括以下成分：

- 回送传输：FPGA 先从 FX3 读取整个缓冲区的内容，然后将其回写到 FX3 内。USB 主机应发送 OUT/IN 令牌数据包，用于发送和接收该数据。 您可使用 EZ-USB FX3 SDK 中提供的 Control Center 工具实现该操作。
- 短数据包： FPGA 先将一个完整的数据包传输到 FX3，然后再发送一个短数据包。USB 主机应发送 IN 令牌数据包，用于接收该数据。
- 零长度数据包（ZLP）传输： FPGA 先将一个完整的数据包传输到 FX3，然后再发送一个零长度数据包。USB 主机应发送 IN 令牌数据包，用于接收该数据。
- 串流（IN）数据传输： FPGA 实现单向传输，即是通过同步从设备 FIFO 连续将数据

写入到 FX3。USB 主机应发送 IN 令牌数据包，用于接收该数据。可使用 EZ-USB FX3 SDK 中的 Control Center 或 Streamer 工具实现该操作。

- 串流（OUT）数据传输：FPGA 实现单向传输，即是通过同步从设备 FIFO 从 FX3 连续读取数据。USB 主机应发送 OUT 令牌数据包，用于发送该数据。可使用 EZ-USB FX3 SDK 中的 Control Center 或 Streamer 工具实现该操作。

FX3 固件的详细信息

本设计实例的 FX3 固件是基于 FX3 SDK 中的示例项目。

该固件的主要特性包括：

- 启用 USB 3.0 和 USB 2.0。
- 使用赛普拉斯 VID/PID（即 0x04B4/0x00F1）进行枚举。这样，可以使用赛普拉斯的 Control Center 和 Streamer 工具启动 USB 传输。
- 集成具有下面特性的同步从设备 FIFO 描述符：
 - 支持访问多达四个套接字
 - 将数据总线的宽度配置为 32 位
 - 在 100 MHz 的 PCLK 输入时钟频率下工作
 - 能够配置四个标志：
 - i. FLAGA：线程 0 专用的满标志
 - ii. FLAGB：线程 0 水印值为 6 的专用局部标志
 - iii. FLAGC：线程 3 专用的空标志
 - iv. FLAGD：线程 3 水印值为 6 的专用局部标志

注意： 本应用笔记中提供的 GPIF II Designer 项目介绍了这些设置。此外，该固件项目还介绍了如何使用 CyU3PGpifSocketConfigure() API 来配置水印值。

- 将 PLL 频率配置为 400 MHz。为了实现该操作，要将 setSysClk400 参数设置为 CyU3PDeviceInit()函数的输入。

```
/*本段代码位于 cyfxslfifosync.c 文件中的 main 函数中*/

/* setSysClk400 clock configurations */
/*注意，器件时钟不等于 CPU 时钟，器件时钟是整个芯片的时钟频率，CPU 时钟由该时钟分频得到*/
clkCfg.setSysClk400 = CyTrue; /* 设置 FX3 器件主时钟频率 > 400 MHz */
clkCfg.cpuClkDiv = 2; /* 设置 CPU 时钟对主时钟分频系数，即设置 CPU 主频 200M */
clkCfg.dmaClkDiv = 2; /* 设置 DMA 工作时钟对主时钟分频系数，即设置 DMA 主频 200M */
clkCfg.mmioClkDiv = 2; /* 设置 DMMIO 工作时钟对主时钟分频系数，即设置 MMIO 主频 200M */
clkCfg.useStandbyClk = CyFalse; /* 不使用 32K 时钟 */
clkCfg.clkSrc = CY_U3P_SYS_CLK; /* 为外设指定时钟源 */

/* Initialize the device */
status = CyU3PDeviceInit (&clkCfg);
```

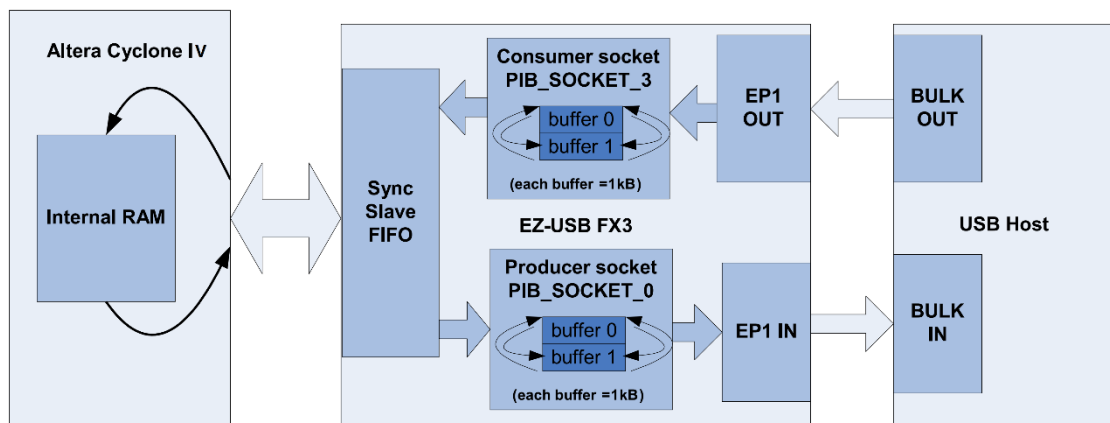
注意： 在工作频率为 100 MHz 且数据总线宽度为 32 位的条件下，该设置对从设备 FIFO

的运行起着关键作用。

■ 设置 DMA 通道：

- 对于回送传输、短数据包和 ZLP 传输，需要创建两个 DMA 通道：
 - ◆ 一个是 P2U 通道，其中 PIB_SOCKET_0 作为发送套接字，UIB_SOCKET_1 作为接收套接字。DMA 缓冲区大小为 512 还是 1024 取决于 USB 连接是 USB2.0 还是 USB3.0。DMA 缓冲区的数量为 2。
 - ◆ 另一个是 U2P 通道，其中 PIB_SOCKET_3 作为接收套接字，UIB_SOCKET_1 作为发送套接字。DMA 缓冲区大小为 512 还是 1024 取决于 USB 连接是 USB2.0 还是 USB3.0。DMA 缓冲区的数量为 2。

回送传输设置

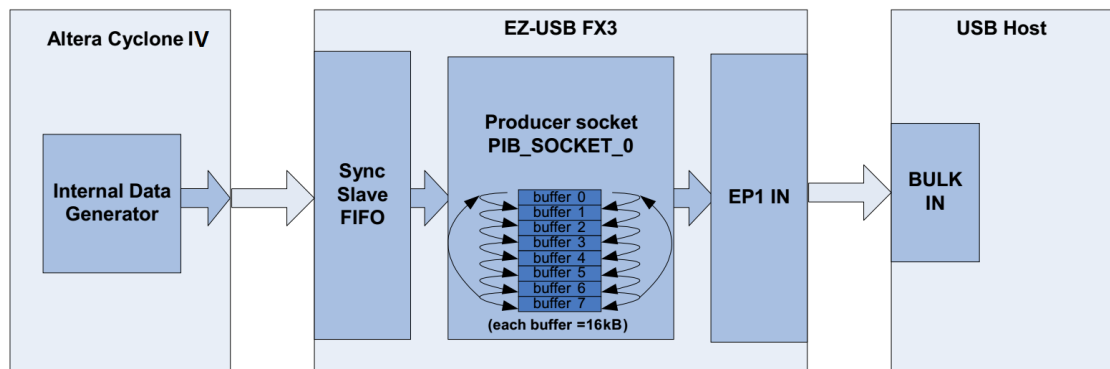


注意，只有 P2U 通道用于传输短数据包和 ZLP 数据包。

对于串流传输，需要创建两个 DMA 通道：

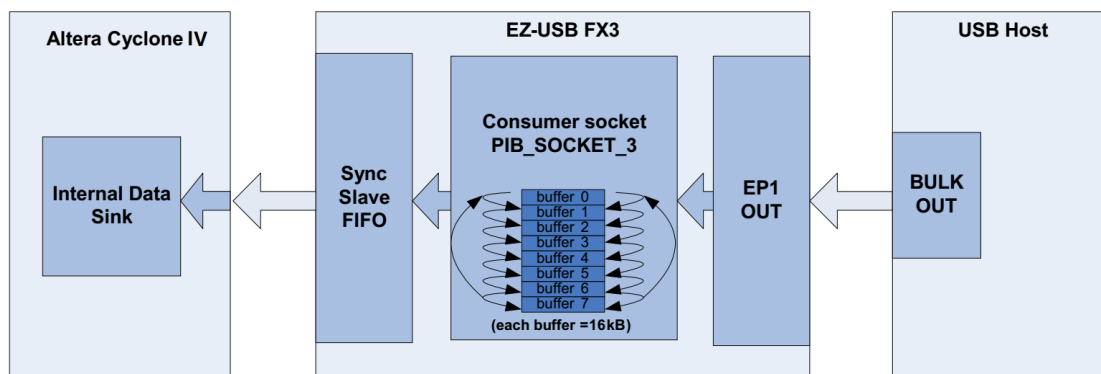
- 一个是 P2U 通道，其中 PIB_SOCKET_0 作为发送套接字，UIB_SOCKET_1 作为接收套接字。根据所使用的 USB 连接类型，DMA 缓冲区大小将为 16×1024（USB 3.0 连接方式）或为 16×512（USB 2.0 连接方式）。
- DMA 缓冲区的数量为 8。选择该缓冲区大小和数量，以得到高吞吐量性能。

图 42. 串流 IN 传输设置 — 优化缓冲区数量和大小，以得到更好的性能



另一个是 U2P 通道，其中 PIB_SOCKET_3 作为接收套接字，UIB_SOCKET_1 作为发送套接字。DMA 缓冲区大小为 16×1024（USB 3.0 连接方式）或为 16×512（USB 2.0 连接方式）。DMA 缓冲区的数量为 4。请注意，可通过增大缓冲区数量来增强性能，但需要降低 P2U 通道的缓冲区数量。其原因是 FX3 SDK 所提供的缓冲区存储空间不足以支持两个通道同时具有 8 个大小为 16×1024 的缓冲区。

图 43. 串流 OUT 传输设置 — 优化缓冲区数量和大小，以得到更好的性能



前面描述的 DMA 通道仅在本应用笔记中提供的 FX3 固件项目内 *cyfxslfifosync.h* 文件中的下列定义被使能时设置。

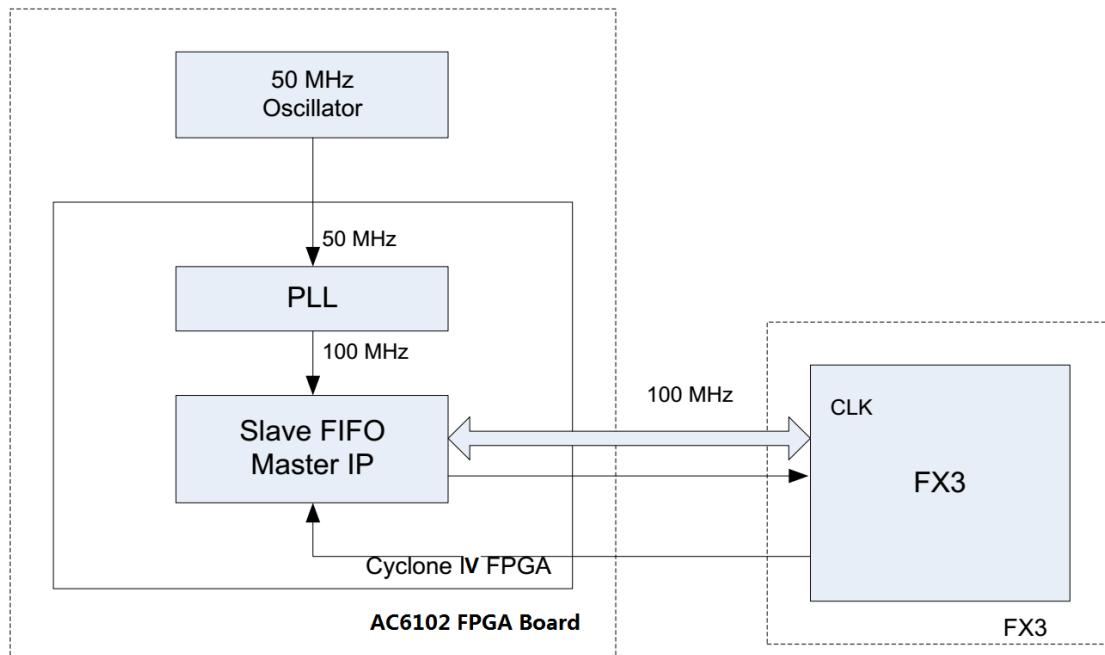
```
/* set up DMA channel for stream IN/OUT transfers */
#define STREAM_IN_OUT
```

通过使用下面各定义(*cyfxslfifosync.h* 文件中也包含这些定义)，可以控制分配给 P2U 和 U2P DMA 通道的缓冲区数量。

```
#ifdef STREAM_IN_OUT
#define BURST_LEN 16
#define DMA_BUF_SIZE (16)
/* Slave FIFO P_2_U channel buffer count */
#define CY_FX_SLFIFO_DMA_BUF_COUNT_P_2_U (8)
/* Slave FIFO U_2_P channel buffer count */
#define CY_FX_SLFIFO_DMA_BUF_COUNT_U_2_P (4)
#endif
```

FPGA 实现的详细信息

使用 AC6102 开发板实现的 Altera Cyclone IV (EP4CE30F23C6) USB3.0 评估电路模型。

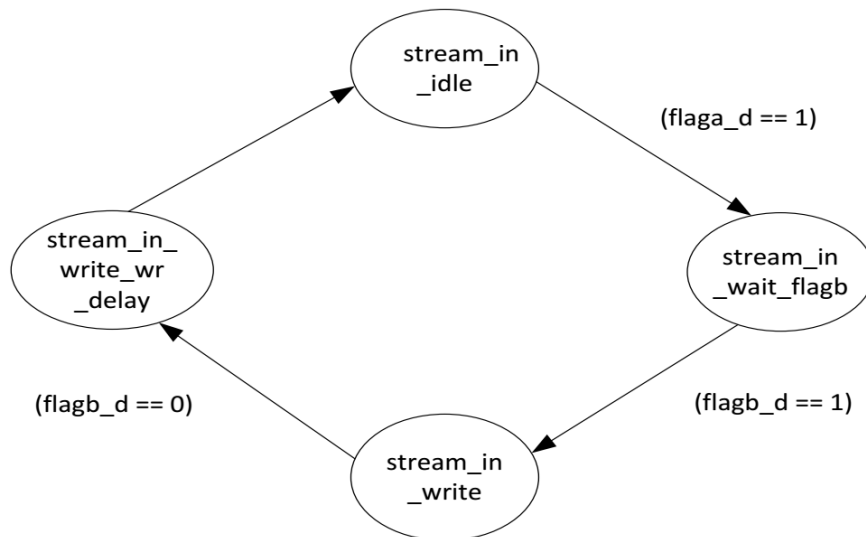


为了得到 FX3 的最大性能， GPIF 接口将以 100 MHz 的频率工作。AC6102 开发板配有一个 50 MHz 和一个 27MHz 的板上单端振荡器。FPGA 使用一个 PLL，以从 50 MHz 的时钟生成一个 100 MHz 的时钟。

串流 IN 示例[FPGA 对从设备 FIFO 进行写操作]

下图显示了 Verilog RTL 中针对串流 IN 传输执行的状态机。

串流 IN 的 FPGA 状态机



stream_in_idle 状态:

该状态初始化状态机中所使用的所有寄存器和信号。从设备 FIFO 控制线的状态为:

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] = 0

stream_in_wait_flagb 状态:

每当 flaga_d = 1 时，状态机将进入该状态，并等待 flagb_d。

stream_in_write 状态:

每当 flagb_d = 1 时，状态机将进入该状态，并开始写入从设备 FIFO 接口。从设备 FIFO 控制线的状态为：

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 0; A[1:0] = 0

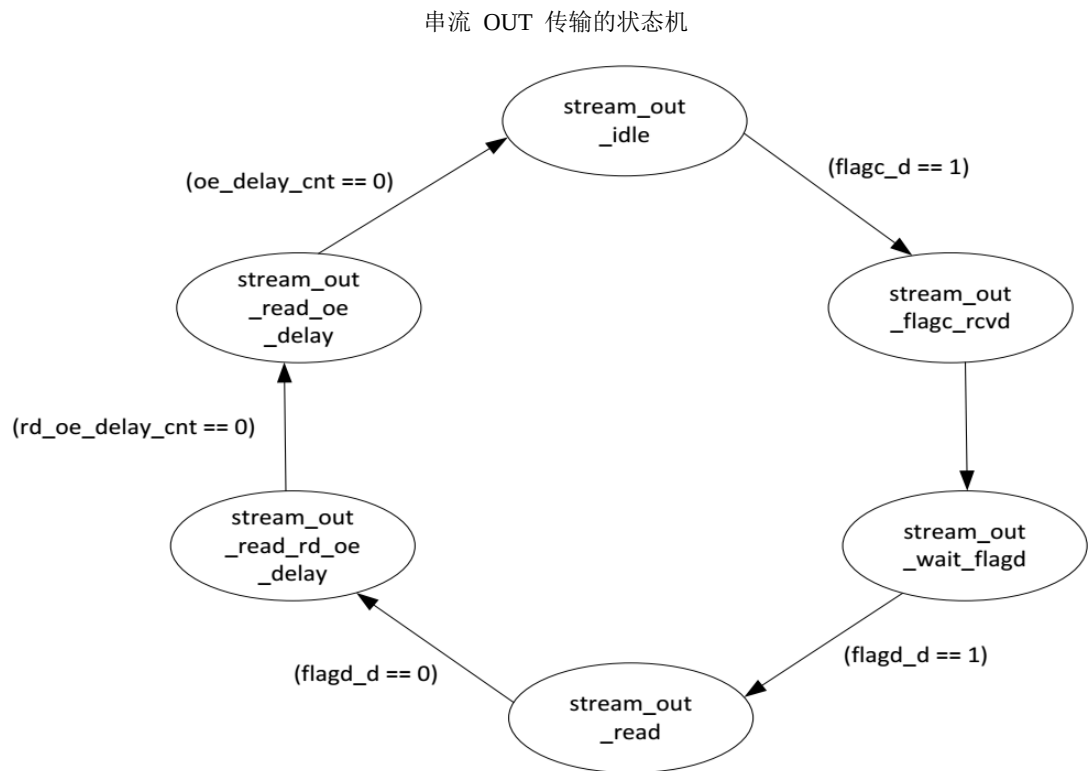
stream_in_write_wr_delay 状态：

每当 flagb_d = 0 时，状态机将进入该状态。从设备 FIFO 控制线的状态为：

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] = 0

经过一个时钟周期后，状态机将进入 stream_in_idle 状态，根据[使用局部标志情况下的通用公式](#)一节中的公式（1），局部标志（flagb）变为 0 后，FX3 需要对处于激活状态的 SLWR# 进行采样两个周期。由于考虑到 FPGA 至接口的一个周期的传输延迟，FPGA 在对被置为 0 的 flagb_d（flagb 的触发输出）进行采样后激活 SLWR# 一个周期。

在 Verilog RTL 中针对串流 OUT 示例执行的状态机



stream_out_idle 状态：

该状态初始化状态机中所使用的所有寄存器和信号。从设备 FIFO 控制线的状态为：

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] = 3

stream_out_flagc_rcvd 状态：

每当 flagc_d = 1 时，状态机将进入该状态。

stream_out_wait_flagd 状态：

经过一个时钟周期后，状态机将进入该状态。

stream_out_read 状态：

每当 flagc_d = 1 时，状态机将进入该状态。状态机将激活读控制信号如下：

PKTEND# = 1; SLOE# = 0; SLRD# = 0; SLCS# = 0; SLWR# = 1; A[1:0] = 3

stream_out_read_rd_oe_delay 状态:

每当 flagc_d = 0 时，状态机将进入该状态。从设备 FIFO 控制线的状态为:

PKTEND# = 1; SLOE# = 0; SLRD# = 0; SLCS# = 0; SLWR# = 1; A[1:0] = 3

根据使用局部标志情况下的通用公式一节中的公式 (2b)，局部标志 (flagd) 变为 0 后，FX3 需要对处于激活状态的 SLRD# 进行采样三个周期。由于考虑到 FPGA 至接口的一个周期的传输延迟，FPGA 对被置为 0 的 flagd_d (flagd 的触发输出) 进行采样后，激活 SLRD# 一个周期。

stream_out_read_oe_delay 状态:

每当 rd_oe_delay_cnt = 0 时，状态机将进入该状态。从设备 FIFO 控制线的状态为:

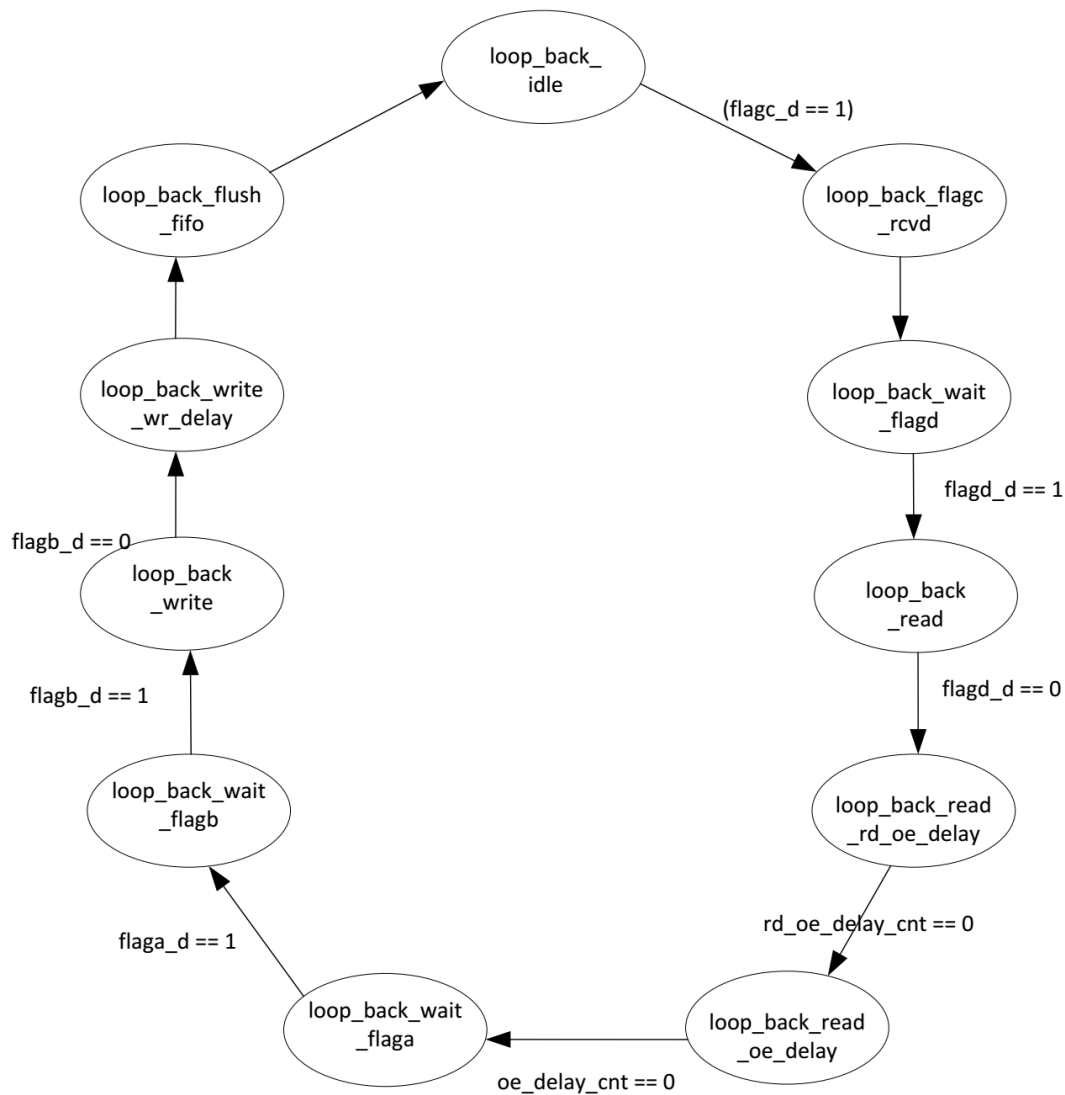
PKTEND# = 1; SLOE# = 0; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] = 3

如果 oe_delay_cnt = 0，状态机将从该状态切换到 stream_out_idle 状态。

回送示例[FPGA 从从设备 FIFO 读取数据，并将数据回写到从设备 FIFO]:

状态机需要经过六个状态后才能完成一个回送循环。下图显示了状态机及其相应操作。

回送传输的状态机



loop_back_idle 状态:

该状态初始化状态机中所使用的所有寄存器和信号。从设备 FIFO 控制线的状态为:

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] = 3

loop_back_flagc_rcvd 状态:

每当 flagc_d = 1 时, 状态机将进入该状态。

loop_back_wait_flagd 状态:

经过一个时钟周期后, 状态机将进入该状态并等待 flagd。

loop_back_read 状态:

如果 flagd_d = 1, 状态机将进入该状态。状态机将激活读控制信号如下:

PKTEND# = 1; SLOE# = 0; SLRD# = 0; SLCS# = 0; SLWR# = 1; A[1:0] = 3

loop_back_read_rd_oe_delay 状态:

每当 flagc_d = 0 时, 状态机将进入该状态。从设备 FIFO 控制线的状态为:

PKTEND# = 1; SLOE# = 0; SLRD# = 0;

SLCS# = 0; SLWR# = 1; A[1:0] = 3

根据[使用局部标志情况下的通用公式](#)一节中的公式 (2b), 局部标志 (flagd) 变为 0 后, FX3 需要对处于激活状态的 SLRD# 进行采样三个周期。由于考虑到 FPGA 至接口的一个周期的传输延迟, FPGA 对被置为 0 的 flagd_d (flagd 的触发输出) 进行采样后, 激活 SLRD# 一个周期。

loop_back_read_oe_delay 状态:

每当 rd_oe_delay_cnt = 0 时, 状态机将进入该状态。从设备 FIFO 控制线的状态为:

PKTEND# = 1; SLOE# = 0; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] = 3

loop_back_wait_flaga 状态:

如果 oe_delay_cnt = 0, 状态机将进入 loop_back_wait_flaga 状态。从设备 FIFO 控制线的状态为:

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] = 0

loop_back_wait_flagb 状态:

如果 flaga_d = 1, 状态机将进入 loop_back_wait_flaga 状态。从设备 FIFO 控制线的状态为:

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] = 0

loop_back_write 状态:

如果 flagb_d = 1, 状态机将进入 loop_back_wait_flaga 状态。从设备 FIFO 控制线的状态为:

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 0; A[1:0] = 0

loop_back_write_wr_delay 状态:

如果 flagb_d = 0, 状态机将进入 loop_back_wait_flaga 状态。从设备 FIFO 控制线的状态为:

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] = 0

根据[使用局部标志情况下的通用公式](#)一节中的公式 (1), 局部标志 (flagb) 变为 0 后, FX3 需要对处于激活状态的 SLWR# 进行采样两个周期。由于考虑到 FPGA 至接口的一个周期的传输延迟, FPGA 在对被置为 0 的局部 flagb_d (flagb 的触发输出) 进行采样后激活 SLWR# 一个周期。

loop_back_flush_fifo 状态:

经过一个时钟周期后, 状态机将进入该状态, 并清理内部 FIFO。从设备 FIFO 控制线的状态为:

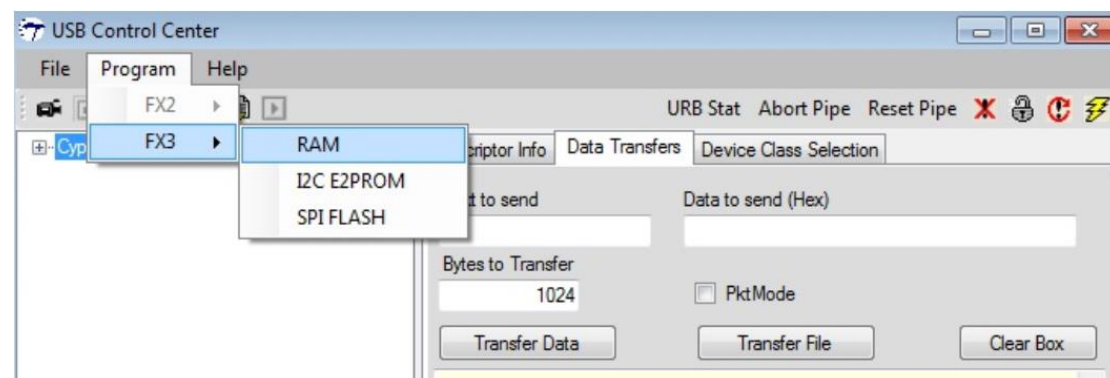
PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] = 0

项目操作

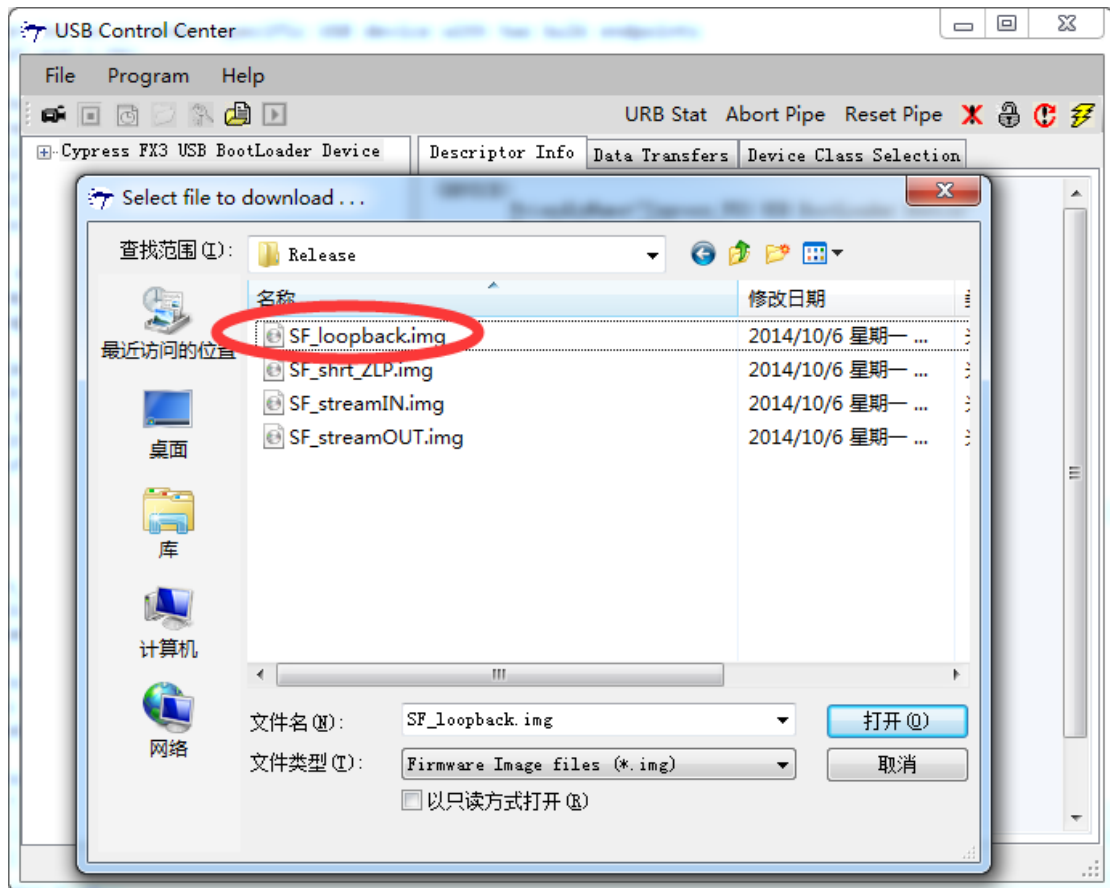
回送传输的测试步骤

使用 Control Center 工具通过 USB 主机给 FX3 下载 *SF_loopback.img* 固件文件。下载固件后, FX3 器件会作为一个 SuperSpeed 器件进行枚举 (前提是设备连接到 USB 3.0 端口)。

使用 Control Center 对 FX3 固件进行编程

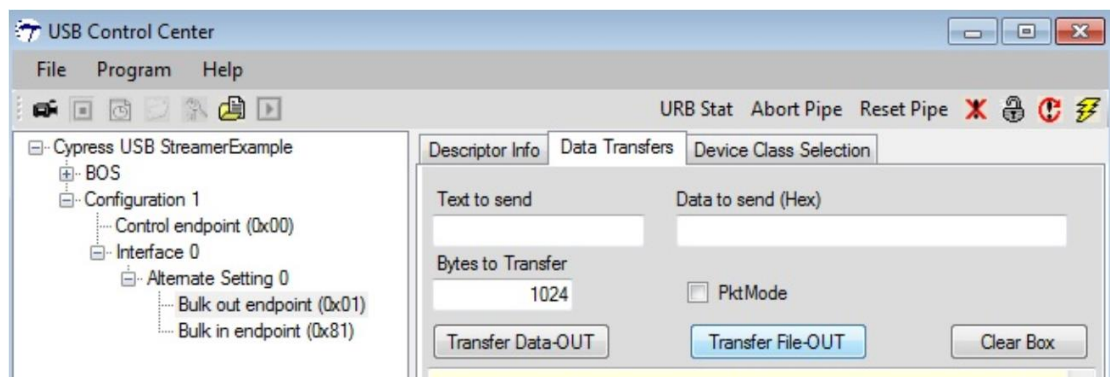


使用 Control Center 编程 FX3 固件, 用于回送测试



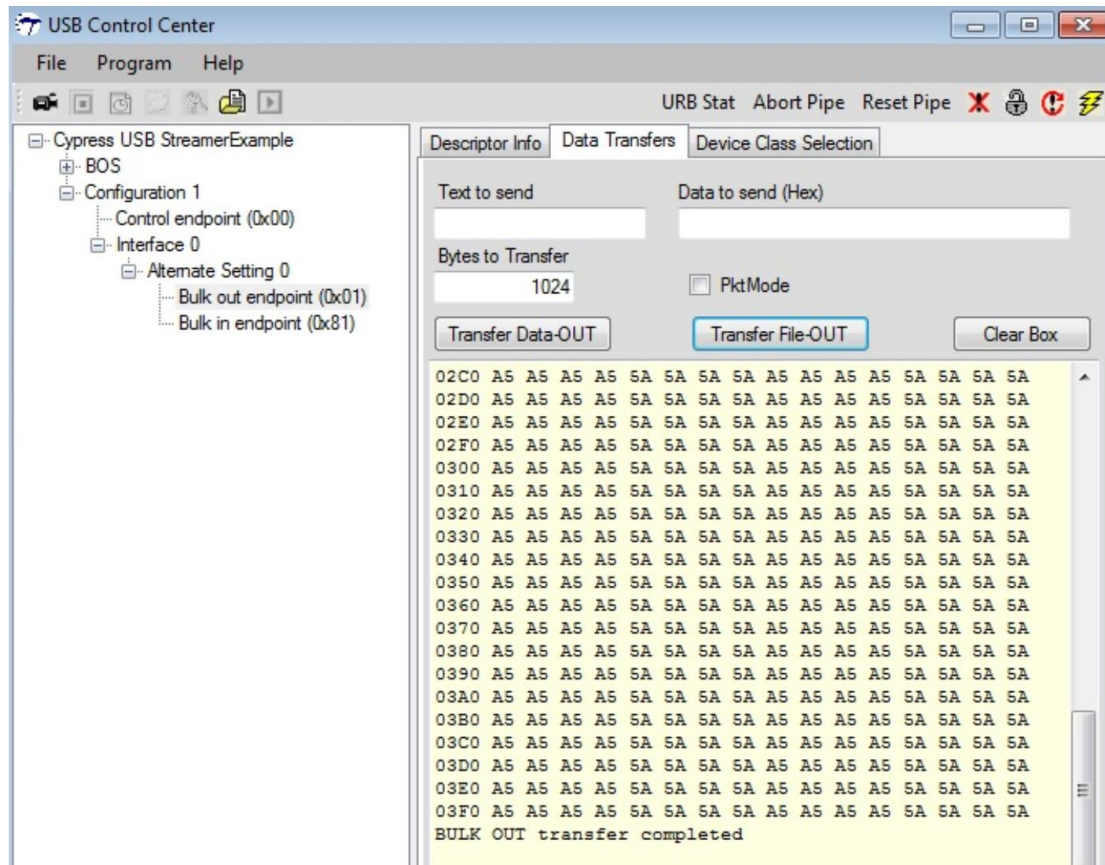
3. 使用 `usb30_loopback.sof` 文件编程 Altera Cyclone IV FPGA。
4. 现在，可使用 Control Center 工具启动传输过程。首先，从 USB 主机启动一个 BULK OUT 传输。在 ControlCenter 中选择 BULK OUT endpoint 项，然后点击 **Transfer File-OUT** 按钮。

使用 Transfer File-OUT 选项启动 BULK OUT 传输



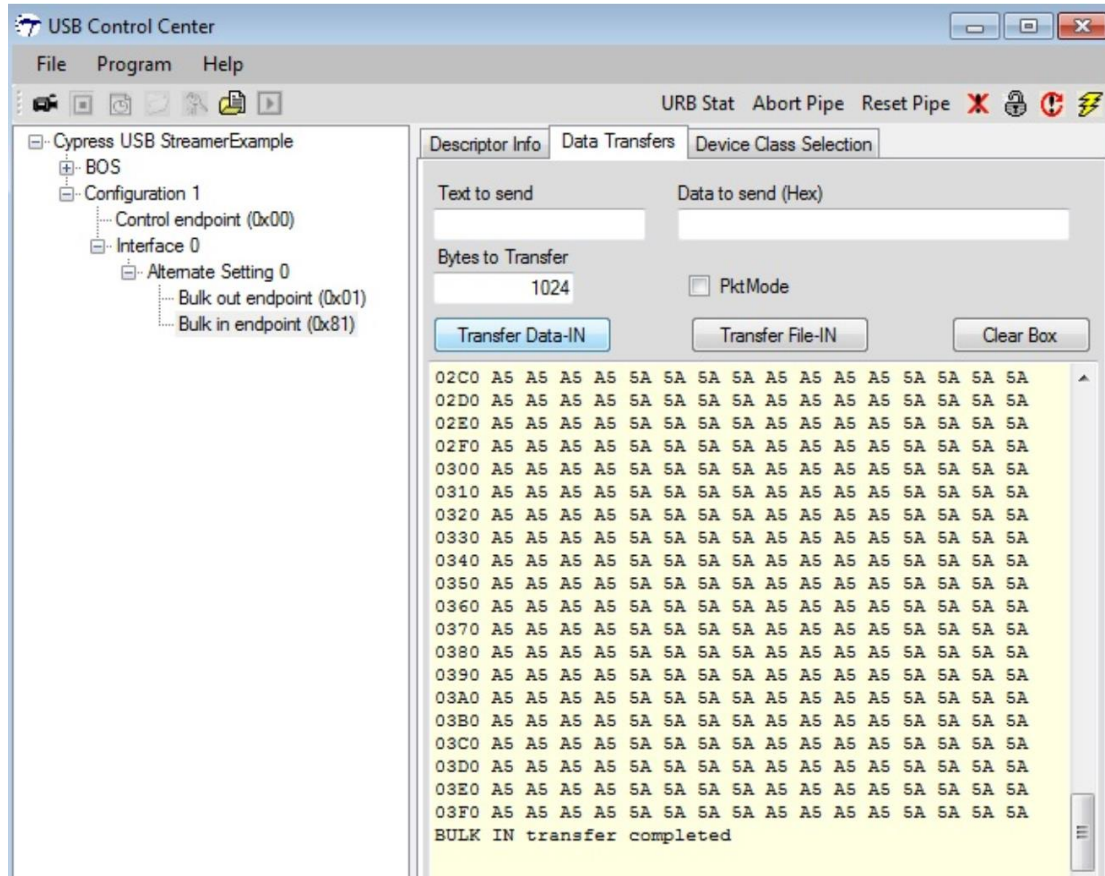
这样，您可浏览并选择包含需要传输的数据的文件。在本应用笔记的附件中，您可在回送文件夹中找到 `TEST.txt` 文件。该文件包含一个以交替方式发送“0xA5A5A5A5 0x5A5A5A5A”的数据内容，文件总大小为 1024 字节。双击选择该文件，然后发送数据。

采用 Transfer File-OUT 选项时通过选择 `TEST.txt` 文件所实现的数据图案传输



6. FPGA 正在等待 FLAGA 变为 1。只要 PIB_SOCKET_0 的缓冲区中的数据可用，FPGA 会立即读取该数据。然后，FPGA 会回送同样的数据，并将其写入到 FX3 的 PIB_SOCKET_3。
7. 然后我们可从 USB 主机发送一个 BULK IN 传输。在 Control Center 中选择 BULK IN endpoint 项，并点击 **Transfer Data-IN** 按钮。这样，先前写入的数据将被读取并显示在信息框中。

通过使用 Transfer Data-IN 按钮启动 BULK IN 传输，以完成回送测试



串流传输的测试步骤

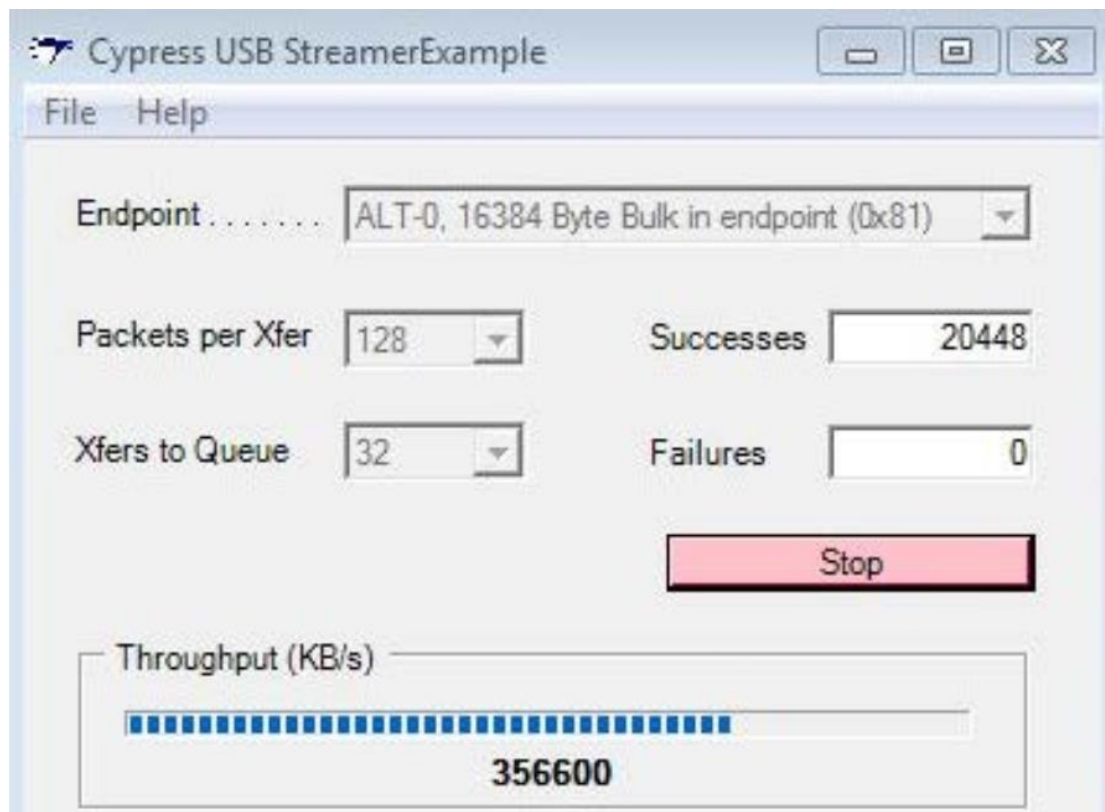
注意： 始终使用下面路径中 FX3 SDK 所提供的 C++ Streamer 工具：

<FX3_SDK_installation_path>\EZ-USB FX3 SDK\1.3\application\cpp\streamer\x86\

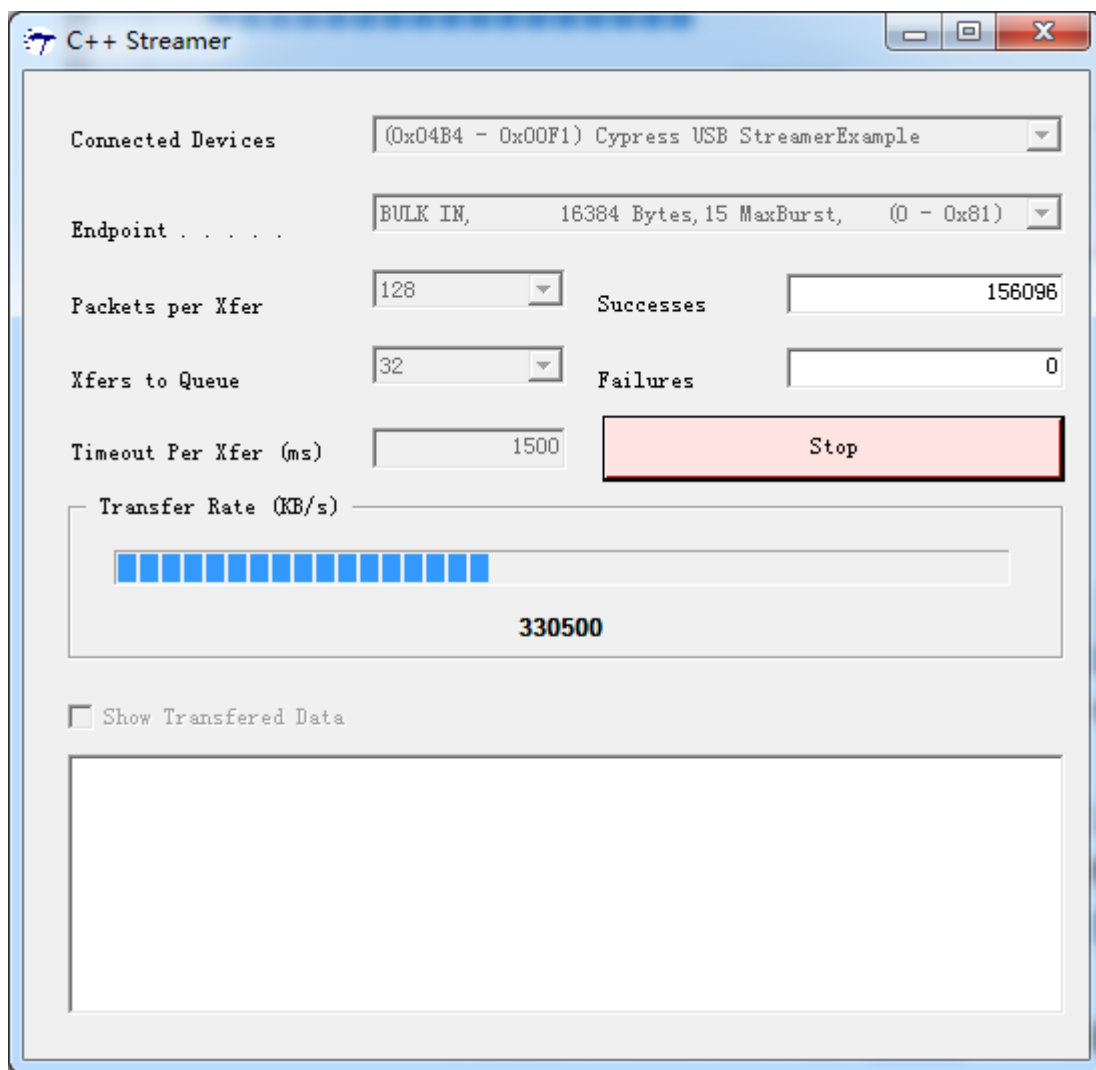
该路径中的 1.3 是 FX3 SDK 版本编号。FX3 SDK 将来版本的编号可高于本版本的。

- 1、 使用 Control Center 工具通过 USB 主机给 FX3 下载 *SF_streamIN.img* 固件文件。下载固件后，FX3 器件会作为一个 SuperSpeed 器件进行枚举(前提是设备连接到 USB 3.0 端口)。
- 2、 对于串流 IN 传输和串流 OUT 传输，分别使用 *usb30_sfifo_streamIN.sof* 文件和 *usb30_sfifo_streamOUT.sof* 文件编程 AC6102 上的 FPGA 芯片。
- 3、 在串流 IN 情况下，FPGA 正在等待 FLAGA 变为 1。当缓冲区可用时，FPGA 立即将数据连续写入到 FX3 的 PIB_SOCKET_0。您可从 USB 主机发送连续的 BULK IN 传输。在赛普拉斯 Streamer 工具中选择 BULK IN 端点，然后点击 **Start**。这时，将显示性能数值。

赛普拉斯 Streamer 工具中显示的串流 IN 性能



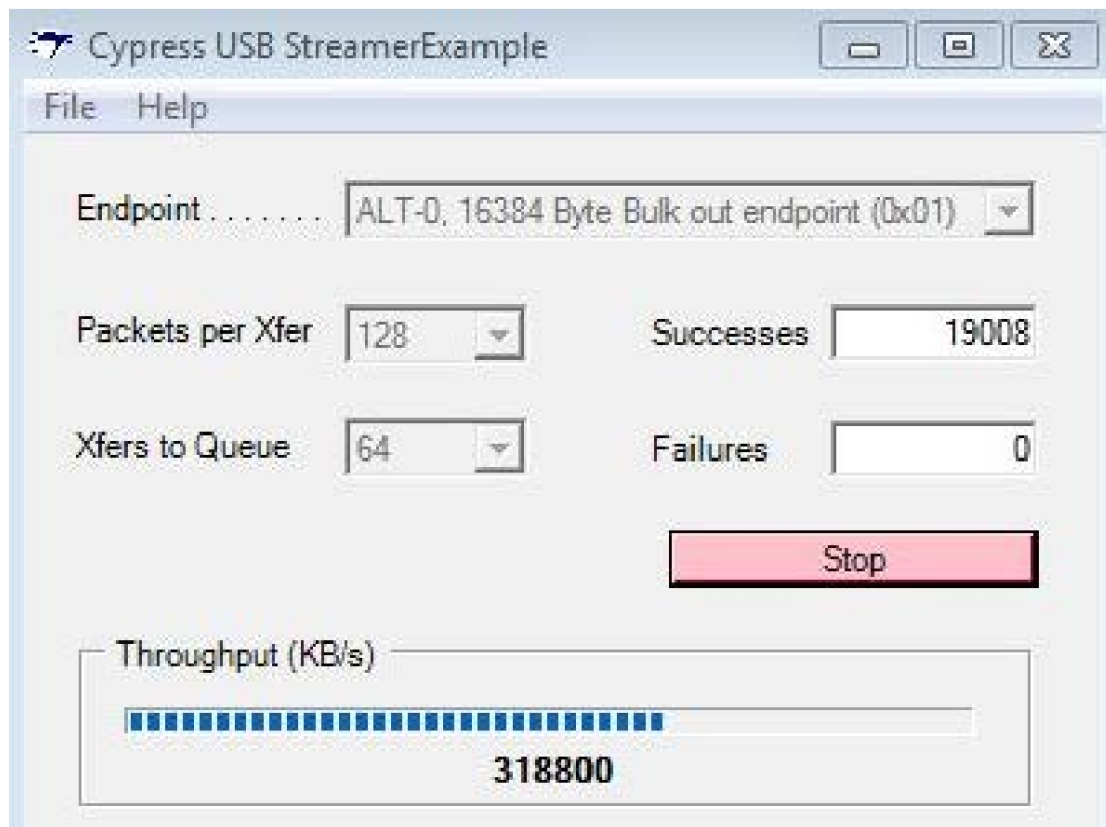
Cypress 官方测试的性能，Z77 Express Chipset 的 Win7 64 位电脑



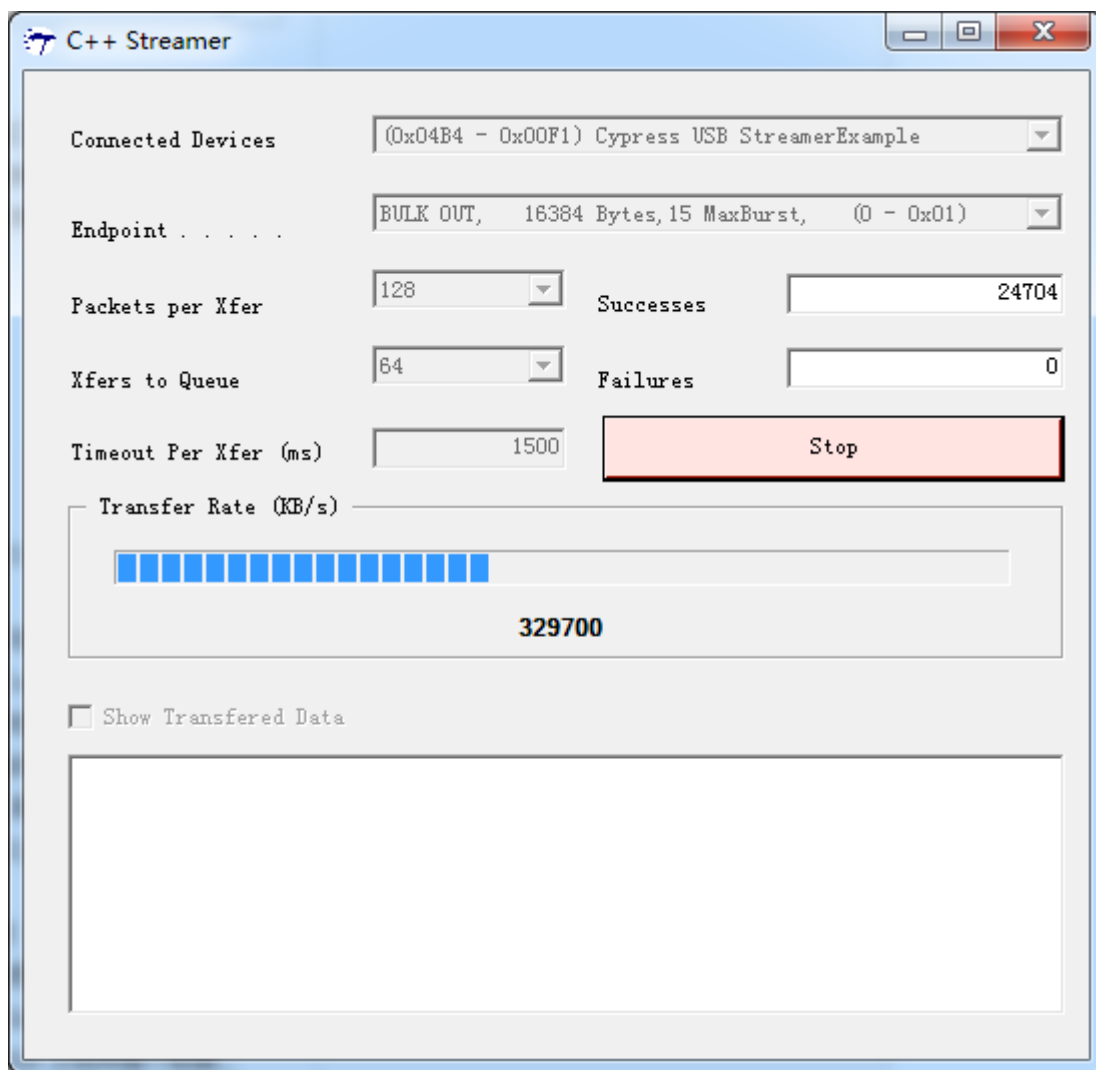
芯航线 AC6102 开发板的测试性能，测试主机为戴尔 Precision 3510 移动工作站，win7 64 位

5. 在串流 OUT 情况下，FPGA 正在等待 FLAGC 变为 1。当数据可用时，FPGA 立即连续读取 FX3 的 PIB_SOCKET_3 中的数据。您可从 USB 主机发送连续的 BULK OUT 传输。在赛普拉斯 Streamer 工具中选择 BULK OUT 端点，然后点击 **Start**。这时，将显示性能数值。图 56 中显示了在配有 Intel Z77 Express Chipset 的 Win7 64 位电脑上观察到的性能。

赛普拉斯 Streamer 工具中显示的串流 OUT 性能



Cypress 官方测试的性能，Z77 Express Chipset 的 Win7 64 位电脑



芯航线 AC6102 开发板的测试性能，测试主机为戴尔 Precision 3510 移动工作站，win7 64 位

SF_streamIN.img 文件可使用于串流 IN 和串流 OUT。在 SF_streamIN.img 中，将 8 个缓冲区和 4 个缓冲区分别分配给 P2U DMA 通道和 U2P 通道。在 SF_streamOUT.img 中，则将 8 个缓冲区和 4 个缓冲区分别分配给 U2P DMA 通道和 P2U 通道。因此，SF_streamIN.img 固件文件将显示更高的 P2U 性能； SF_streamOUT.img 固件文件将显示更高的 U2P 性能。

FX3 应用程序（固件）架构

FX3 所有的固件程序都包含两部分（初始化代码和应用代码）

初始化代码：所有应用的初始化代码基本都是相同的

应用代码：每个应用的专用代码

下面以同步 fifo（SlaveFifoSync）的应用为例，讲解其代码架构。



应用固件结构

Slave Fifo 例程包含以下文件：

- cyfxgpif_synscf.h: 本文件是 GPIF II 工作在 16 位或 32 位 Slave Fifo 模式下的描述文件，GPIF II 的各种配置由该文件定义。一般通过 GPIF II Designer 软件生成。
- cyfxslfifousbdscr.c: 本文件包含 USB 的枚举描述
- cyfxslfifosync.h: 本文件主要是 cyfxslfifosync.c 文件中用到的一些参数的宏定义。例如，CY_FX_SLFIFO_GPIF_16_32BIT_CONF_SELECT 定义了 Slave Fifo 是使用 16 位模式还是 32 位模式，当该定义为 0 时，选择使用 16 位模式，为 1 则使用 32 位模式。当然，该常量也被用来定义 16 位和 32 位模式时 IO 的功能。

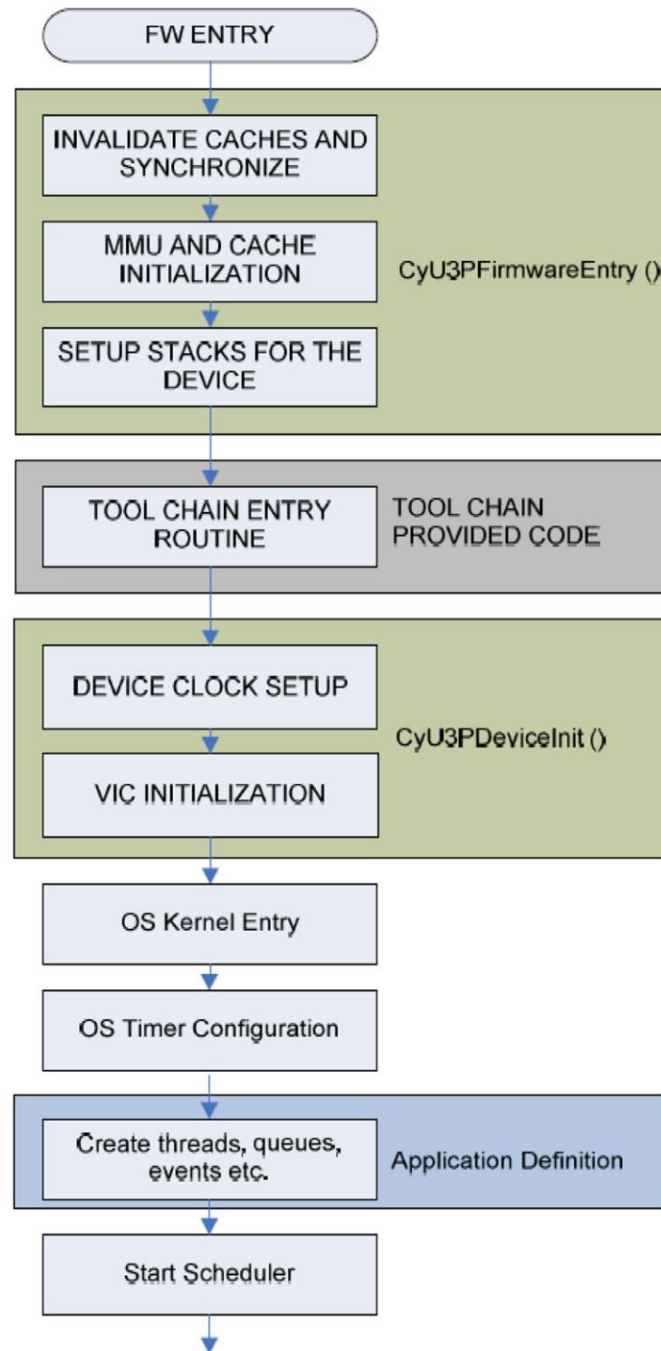
```
the FX3 system clock has to be set to a frequency greater
#if (CY_FX_SLFIFO_GPIF_16_32BIT_CONF_SELECT == 0)
    clockConfig.setSysClk400 = CyFalse;
#else
    clockConfig.setSysClk400 = CyTrue;
#endif

io_cfg.usespi = CyTrue;
#if (CY_FX_SLFIFO_GPIF_16_32BIT_CONF_SELECT == 0)
    io_cfg.isDQ32Bit = CyFalse;
    io_cfg.lppMode = CY_U3P_IO_MATRIX_LPP_UART_ONLY;
#else
    io_cfg.isDQ32Bit = CyTrue;
    io_cfg.lppMode = CY_U3P_IO_MATRIX_LPP_DEFAULT;
#endif
```

- cyfxslfifosync.c: 本文件包含 Slave Fifo 应用实例的主程序逻辑,关于该程序的解释，将在后续章节进行讲解。

初始化代码

下图展示了 FX3 应用程序的初始化流程，本节将对流程中每个步骤的功能进行详细介绍。



固件入口（Firmware Entry）

FX3 的固件入口是 `CyU3PFirmwareEntry()`，该函数定义在了 *FX3* API 库中，对用户不可见。该函数实现了以下功能。

- 1、初始化缓存（cache）（这个 cache 将在启动的时候被 bootloader 使用）
- 2、初始化内存管理单元（MMU）和对应的缓存
- 3、初始化 SYS、FIQ、IRQ 和 SVC 的堆栈
- 4、然后，执行接口将会被转交给工具链初始化函数 `CyU3PToolChainInit()`。

工具链初始化 (Tool Chain Initialization)

当 `CyU3PFirmwareEntry()` 执行完成以后，初始化流程中的下一个步骤就是初始化工具链。由于所有需要的栈初始化是由固件入口函数定义的，因此工具链初始化不再需要执行这些过程，`CyU3PToolChainInit` 所执行的唯一操作就是将 BSS 数据段的值全部清零。BSS 数据段保存了所有应用固件的未初始化数据。针对 ARM 处理器的 GUN GCC 编译器的工具链初始化函数示例如下（本代码可在 `cyfx_gcc_startup.S` 文件中找到）：

```
.global CyU3PToolChainInit
CyU3PToolChainInit:

# clear the BSS area
__main:
    mov R0, #0
    ldr R1, =_bss_start
    ldr R2, =_bss_end
1: cmp R1, R2
    strlo R0, [R1], #4
    blo 1b

    b main
```

在该函数中，执行了以下两个过程

- 1、清零 BSS 数据段
- 2、将程序控制权转交给 `main()` 函数

main()函数

`main()` 函数是应用程序的 C 程序入口，该函数在 `cyfxslfifosync.c` 文件中定义。该函数主要实现了四个操作

- 1、器件初始化：这是固件的第一个步骤，该部分功能代码如下所示：

```
status = CyU3PDeviceInit (&clockConfig);
if (status != CY_U3P_SUCCESS)
{
    goto handle_fatal_error;
}
```

设备初始化主要完成以下工作

- a) 设置 CPU 时钟，`clockConfig` 是与时钟参数有关的结构图，当对该结构体中与时钟相关的参数（如 PLL 的分频和倍频系数）进行设置后，将该结构图传给 `CyU3PDeviceInit` 函数，就能完成时钟的设置。如果传递一个 NULL 给该函数，则将按照默认时钟设置进行设置。当给 FX3 芯片外接一个 19.2M 的时钟，并选择芯片的时钟源为该时钟信号时，则按照时钟默认设置，芯片即可正常工作。

- b) 初始化向量中断控制器
- c) 配置 GCTL 和 PLL

CyU3PDeviceInit()是 FX3 库的一部分，因此我们也无法查看到其中的内容。

2、器件缓存（cache）配置。main()函数的第一个步骤是配置器件的缓存。器件包含 8K 字节的数据缓存区和 8K 字节的指令缓存区。在本例中，仅使能了指令缓存区。因为数据缓存区仅在需要 CPU 进行大量的数据存取工作的时候才能体现出效果，而本例是基于 DMA 的数据传输，当完成初始化后，CPU 基本处于旁路状态，无需进行大量的数据存取操作，因此不需要使能数据缓存。以下为初始化缓存的代码：

```
/* Initialize the caches. Enable both Instruction and Data Caches.
 */
status = CyU3PDeviceCacheControl (CyTrue, CyFalse, CyFalse);
if (status != CY_U3P_SUCCESS)
{
    goto handle_fatal_error;
}
```

CyU3PDeviceCacheControl (CyTrue, CyTrue, CyTrue)函数的三个参数分别为指令缓存、数据缓存、DMA 是否关联数据缓存的开关，使用宏定义的 CyTrue 或 CyFalse 来设置开关状态。

注意，在 Cypress 官方提供的固件源码中，此部分实际是将数据缓存打开了的，与官方文档讲解不一致哈，例程中的实际代码如下所示。

```
/* Initialize the caches. Enable both Instruction and Data Caches.
 */
status = CyU3PDeviceCacheControl (CyTrue, CyTrue, CyTrue);
if (status != CY_U3P_SUCCESS)
{
    goto handle_fatal_error;
}
```

3、IO 矩阵配置：第三步是配置 IO 矩阵，所谓 IO 矩阵配置就是配置 GPIF 或串行外设接口的对应 IO 功能 IO。被配置的对象包括 GPIF 和串行接口（SPI, I2C, I2S, GPIO, 和 UART）。该部分代码如下所示：

```
//存储接口 0 不使用，以使这部分 IO 用于其他接口功能
io_cfg.s0Mode = CY_U3P_SPORT_INACTIVE;
//存储接口 1 不使用，以使这部分 IO 用于其他接口功能
io_cfg.s1Mode = CY_U3P_SPORT_INACTIVE;
io_cfg.useUart = CyTrue; //使能串口
io_cfg.useI2C = CyFalse; //不使能 I2C 接口
io_cfg.useI2S = CyFalse; //不使能 I2S 接口
io_cfg.useSpi = CyFalse; //不使能 SPI 接口
#if (CY_FX_SLFIFO_GPIF_16_32BIT_CONF_SELECT == 0) //16bit 模式
io_cfg.isDQ32Bit = CyFalse; //关闭 32 位数据总线模式
io_cfg.lppMode = CY_U3P_IO_MATRIX_LPP_UART_ONLY;
```

```

#else    //32bit 模式
    io_cfg.isDQ32Bit = CyTrue; //使能 32 位数据总线模式
    io_cfg.lppMode    = CY_U3P_IO_MATRIX_LPP_DEFAULT;
#endif

/* No GPIOs are enabled. */
io_cfg.gpioSimpleEn[0] = 0;
io_cfg.gpioSimpleEn[1] = 0;
io_cfg.gpioComplexEn[0] = 0;
io_cfg.gpioComplexEn[1] = 0;
status = CyU3PDeviceConfigureIOMatrix (&io_cfg);
if (status != CY_U3P_SUCCESS)
{
    goto handle_fatal_error;
}

```

在本例中，

- a) 通过 CY_FX_SLFIFO_GPIF_16_32BIT_CONF_SELECT 定义来选择并配置 16 位或 32 位模式；
- b) GPIO、I2C、I2S 和 SPI 都没有使用；
- c) 使用了 UART；
- d) 两个存储端口因为本模式下无法支持，所以也没有使能。

通过本步骤，IO 配置结构体被初始化了，而且 CyU3PDeviceConfigureIOMatrix（）函数也被调用并执行。另外注意，CyU3PDeviceConfigureIOMatrix（）也是 FX3 库的一部分，我们没法看到源码。

4、main() 函数的最后一步通过调用 CyU3PKernelEntry()函数来启动操作系统。该函数在库中被定义，是一个没有返回值的函数。该函数实际上是 Threadx 操作系统入口函数的一个封装。其主要实现以下功能：

- a) 初始化操作系统；
- b) 设置操作系统定时器；
- c) 启动并将控制权交给操作系统调度器。

应用定义（ApplicationDefine）

当操作系统被调用以后，FX3 的库会调用 CyFxApplicationDefine()函数，在这个函数中，将创建具体的应用线程。

在 Slave Fifo 实例中，在应用定义函数中仅创建了一个线程，该部分代码如下所示：

```

/* Application define function which creates the threads. */
void CyFxApplicationDefine(void) {
    void *ptr = NULL;
    uint32_t retThrdCreate = CY_U3P_SUCCESS;

    /* 为线程申请一块内存用作堆栈 */

```

```

ptr = CyU3PMemAlloc(CY_FX_SLFIFO_THREAD_STACK);

/* 为应用创建线程 */
retThrdCreate = CyU3PThreadCreate(
/* Slave Fifo 应用线程的结构体 */
&slFifoAppThread,
/* 线程 ID 和线程名称 */
"21:Slave_FIFO_sync",
/*Slave Fifo 应用线程的入口函数*/
SlFifoAppThread_Entry,
/*没有需要传给线程的参数*/
0,
/*为线程申请的栈存储器地址*/
ptr,
/*应用线程的栈大小*/
CY_FX_SLFIFO_THREAD_STACK,
/*应用线程的优先级*/
CY_FX_SLFIFO_THREAD_PRIORITY,
/*应用线程的抢占优先级*/
CY_FX_SLFIFO_THREAD_PRIORITY,
/* 应用无时间片 */
CYU3P_NO_TIME_SLICE,
/*启动应用线程 */
CYU3P_AUTO_START);

/* 检查线程创建返回值*/
if (retThrdCreate != 0) {
    /* 线程创建失败，并返回了一个错误码*/

    /* Add custom recovery or debug actions here */

    /* Application cannot continue */
    /* Loop indefinitely */
    while (1)
        ;
}
}

```

用户可以在应用程序定义函数中根据自己的需求创建更多的线程。所有其他的具体程序必须在其自己线程中完成。

应用代码

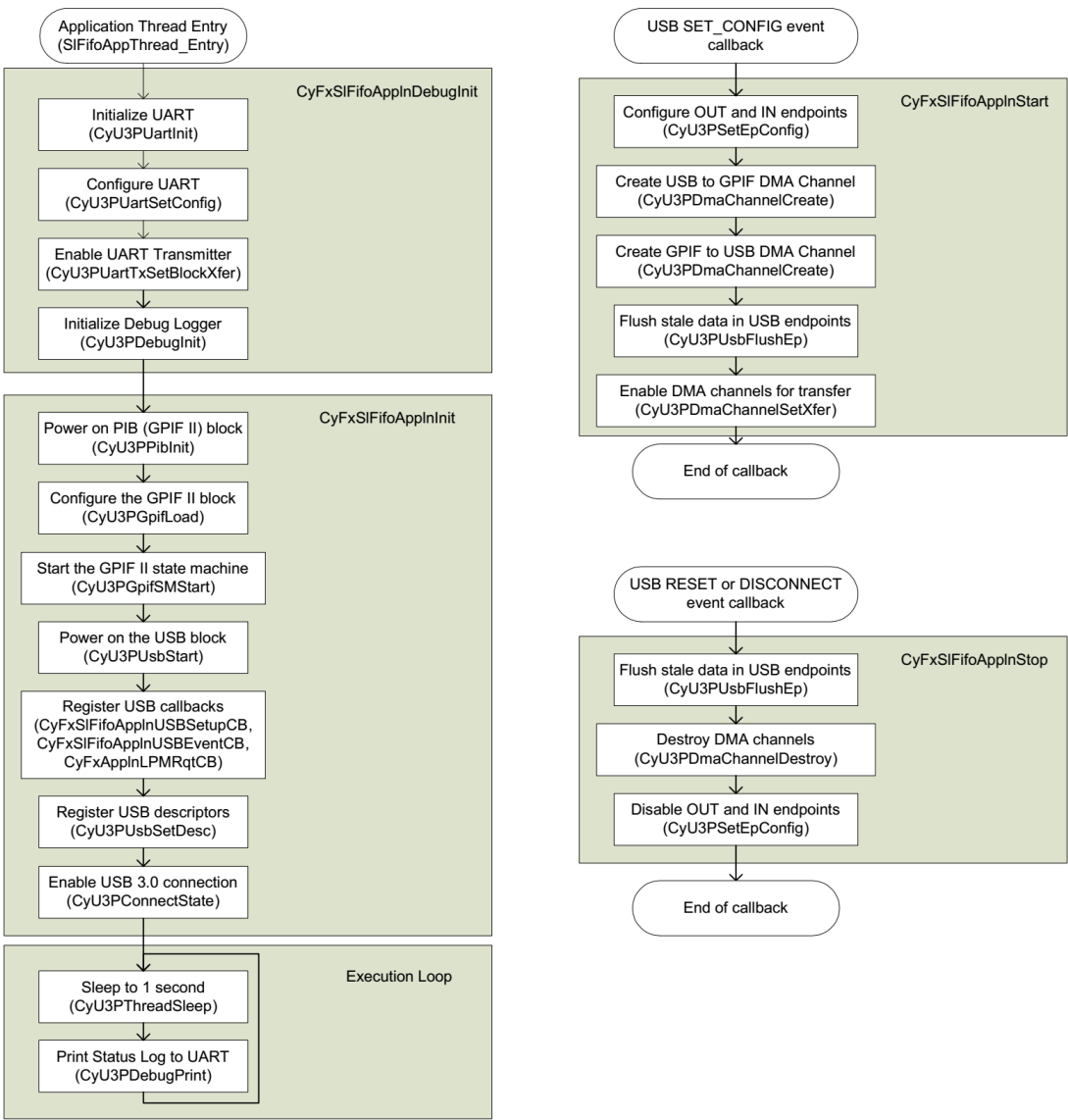
Slave Fifo 应用代码由 3 部分组成：

应用进程实现具体应用相关的初始化操作，例如 GPIF II 和 USB 模块的配置。

在收到 USB SET_CONFIGURATION 请求后，CyFxFifoApplnStart 函数被调用，设置 USB 和 GPIF 互通所需的端点和 DMA 通道。

当 USB 复位或者连接断开被检测到后，CyFxFifoApplnStop 函数被调用，关闭 USB 端点、释放 DMA 通道。

Slave fifo 应用代码结构如下所示：



应用线程（Application Thread）

针对 Slave fifo 实例，应用部分的程序入口是 SlFifoAppThread_Entry ()函数，该函数代

码如下所示：

```
/* Entry function for the slFifoAppThread. */
void
SlFifoAppThread_Entry (
    uint32_t input)
{
    /* Initialize the debug module */
    CyFxsSlFifoApplnDebugInit();

    /* Initialize the slave FIFO application */
    CyFxsSlFifoApplnInit();

    for (;;)
    {
        CyU3PThreadSleep (1000);
        if (glIsApplnActive)
        {
            /* Print the number of buffers received so far from the
            USB host. */
            CyU3PDebugPrint (6, "Data tracker: buffers received: %d,
            buffers sent: %d.\n",
                            glDMARxCount, glDMATxCount);
        }
    }
}
```

该线程内主要完成以下操作：

1. 初始化调试机制（debug mechanism）
2. 初始化 Slave Fifo 应用的主体

以下将对这两个步骤分别进行讲解

初始化（Debug）

调试模块使用串口输出调试信息。串口必须在调试机制初始化之前被配置好，实现该功能是通过调用 CyU3PUartInit（）函数来实现的。

```
CyU3PUartConfig_t uartConfig;
CyU3PReturnStatus_t apiRetStatus = CY_U3P_SUCCESS;

/* Initialize the UART for printing debug messages */
apiRetStatus = CyU3PUartInit();
if (apiRetStatus != CY_U3P_SUCCESS)
```

```

{
    /* Error handling */
    CyFxAppErrorHandler(apiRetStatus);
}

```

下一步就是配置 UART 串口，UART 的数据结构体首先被填充，然后传递给 CyU3PUartSetConfig () 函数来完成配置。

```

/* Set UART configuration */
CyU3PMemSet ((uint8_t *)&uartConfig, 0, sizeof (uartConfig));
uartConfig.baudRate = CY_U3P_UART_BAUDRATE_115200; //波特率 115200
uartConfig.stopBit = CY_U3P_UART_ONE_STOP_BIT; //1 位停止位
uartConfig.parity = CY_U3P_UART_NO_PARITY; //无校验位
uartConfig.txEnable = CyTrue; //使能串口发送功能
uartConfig.rxEnable = CyFalse; //关闭串口接收功能
uartConfig.flowCtrl = CyFalse; //关闭串口流控功能
uartConfig.isDma = CyTrue; //使能 DMA 传输

apiRetStatus = CyU3PUartSetConfig (&uartConfig, NULL);
if (apiRetStatus != CY_U3P_SUCCESS)
{
    CyFxAppErrorHandler(apiRetStatus);
}

```

下一步，设置 UART 传输的大小，这里将大小设置为无限大，这样调试的打印信息就不会受到尺寸的限制。

```

/* Set the UART transfer to a really large value. */
apiRetStatus = CyU3PUartTxSetBlockXfer (0xFFFFFFFF);
if (apiRetStatus != CY_U3P_SUCCESS)
{
    CyFxAppErrorHandler(apiRetStatus);
}

```

最后，调试模块被初始化，与之相关的两个重要的参数是：

Debug 打印的描述符，该描述符指向 UART Socket。

Debug 打印等级，该值被设置为 8，因此所有低于该等级的调试信息将会被打印出来。

```

/* Initialize the debug module. */
apiRetStatus = CyU3PDebugInit (CY_U3P_LPP_SOCKET_UART_CONS, 8);
if (apiRetStatus != CY_U3P_SUCCESS)
{
    CyFxAppErrorHandler(apiRetStatus);
}

```

应用初始化

应用的初始化包含以下两个步骤：

- GPIF-II 初始化
- USB 初始化

GPIF-II 初始化

GPIF-II 模块首先被初始化该部分代码如下所示：

```
CyU3PPibClock_t pibClock;
CyU3PReturnStatus_t apiRetStatus = CY_U3P_SUCCESS;

/* Initialize the p-port block. */
pibClock.clkDiv = 2;
pibClock.clkSrc = CY_U3P_SYS_CLK;
pibClock.isHalfDiv = CyFalse;
/* Disable DLL for sync GPIF */
pibClock.isDllEnable = CyFalse;
apiRetStatus = CyU3PPibInit(CyTrue, &pibClock);
if (apiRetStatus != CY_U3P_SUCCESS)
{
    CyU3PDebugPrint (4, "P-port Initialization failed, Error Code
= %d\n", apiRetStatus);
    CyFxAppErrorHandler(apiRetStatus);
}
```

Slave Fifo 描述符会被载入 GPIF-II 的寄存器，然后 GPIF-II 的寄存器被启动。

```
/* Load the GPIF configuration for Slave FIFO sync mode. */
apiRetStatus = CyU3PGpifLoad
(&Sync_Slave_Fifo_2Bit_CyFxGpifConfig);
if (apiRetStatus != CY_U3P_SUCCESS)
{
    CyU3PDebugPrint (4, "CyU3PGpifLoad failed, Error Code
= %d\n", apiRetStatus);
    CyFxAppErrorHandler(apiRetStatus);
}

/* Start the state machine. */
apiRetStatus = CyU3PGpifSMStart (SYNC_SLAVE_FIFO_2BIT_RESET,
SYNC_SLAVE_FIFO_2BIT_ALPHA_RESET);
if (apiRetStatus != CY_U3P_SUCCESS)
{
    CyU3PDebugPrint (4, "CyU3PGpifSMStart failed, Error Code
= %d\n", apiRetStatus);
}
```

```
CyFxAppErrorHandler(apiRetStatus);  
}
```

USB 初始化

USB 栈在 FX3 的库中首先被初始化，该功能通过调用 USB 启动函数 CyU3PUsbStart 来实现。

```
/* Start the USB functionality. */  
apiRetStatus = CyU3PUsbStart();  
if (apiRetStatus != CY_U3P_SUCCESS)  
{  
    CyU3PDebugPrint (4, "CyU3PUsbStart failed to Start, Error  
code = %d\n", apiRetStatus);  
    CyFxAppErrorHandler(apiRetStatus);  
}
```

下一步就是注册回调函数，在本例中，回调函数为 USB 请求和 USB 时间注册。

```
/* The fast enumeration is the easiest way to setup a USB  
connection,  
 * where all enumeration phase is handled by the library. Only the  
 * class / vendor requests need to be handled by the application.  
 */  
CyU3PUsbRegisterSetupCallback(CyFxSlFifoApplnUSBSetupCB, CyTrue);  
  
/* Setup the callback to handle the USB events. */  
CyU3PUsbRegisterEventCallback(CyFxSlFifoApplnUSBEventCB);
```

回调函数和回调处理部分将在后续部分讲解。

设置 USB 描述符，这是通过调用 USB 设置描述符来完成的。

```
/* Set the USB Enumeration descriptors */  
  
/* Super speed device descriptor. */  
apiRetStatus = CyU3PUsbSetDesc(CY_U3P_USB_SET_SS_DEVICE_DESCR,  
0, (uint8_t *)CyFxUSB30DeviceDscr);  
if (apiRetStatus != CY_U3P_SUCCESS)  
{  
    CyU3PDebugPrint (4, "USB set device descriptor failed, Error  
code = %d\n", apiRetStatus);  
    CyFxAppErrorHandler(apiRetStatus);  
}  
.....
```

上述代码是设置器件描述符，示例代码中还设置了其他描述符包括 Device Qualifier, Other Speed, Configuration, BOS (for Super Speed)和 String Descriptors。

USB 引脚连接后，FX3 USB 设备只有在此过程之后才能被主机（PC）识别到。

```
/* Connect the USB Pins with super speed operation enabled. */
apiRetStatus = CyU3PConnectState(CyTrue, CyTrue);
if (apiRetStatus != CY_U3P_SUCCESS)
{
    CyU3PDebugPrint (4, "USB Connect failed, Error code = %d\n",
apiRetStatus);
    CyFxAppErrorHandler(apiRetStatus);
}
```

端点设置

端点在接收到一个 SET_CONFIGURATION 请求的时候被配置，一个 IN 和一个 OUT 两个端点被配置为 bulk 端点。端点的最大包长度基于 USB 的连接速度来设置。该部分代码如下：

```
void
CyFxs1FifoApplnStart (void)
{
    uint16_t size = 0;
    CyU3PEpConfig_t epCfg;
    CyU3PDmaChannelConfig_t dmaCfg;
    CyU3PReturnStatus_t apiRetStatus = CY_U3P_SUCCESS;
    CyU3PUSBSpeed_t usbSpeed = CyU3PUsbGetSpeed();

    /* First identify the usb speed. Once that is identified,
    * create a DMA channel and start the transfer on this. */

    /* Based on the Bus Speed configure the endpoint packet size */
    switch (usbSpeed)
    {
        case CY_U3P_FULL_SPEED:
            size = 64;
            break;

        case CY_U3P_HIGH_SPEED:
            size = 512;
            break;

        case CY_U3P_SUPER_SPEED:
            size = 1024;
            break;
    }
```

```

        default:
            CyU3PDebugPrint (4, "Error! Invalid USB speed.\n");
            CyFxAppErrorHandler (CY_U3P_ERROR_FAILURE);
            break;
    }

    CyU3PMemSet ((uint8_t *)&epCfg, 0, sizeof (epCfg));
    epCfg.enable = CyTrue;
    epCfg.epType = CY_U3P_USB_EP_BULK;
    epCfg.burstLen = 1;
    epCfg.streams = 0;
    epCfg.pcktSize = size;

    /* Producer endpoint configuration */
    apiRetStatus = CyU3PSetEpConfig(CY_FX_EP_PRODUCER, &epCfg);
    if (apiRetStatus != CY_U3P_SUCCESS)
    {
        CyU3PDebugPrint (4, "CyU3PSetEpConfig failed, Error code
= %d\n", apiRetStatus);
        CyFxAppErrorHandler (apiRetStatus);
    }

    /* Consumer endpoint configuration */
    apiRetStatus = CyU3PSetEpConfig(CY_FX_EP_CONSUMER, &epCfg);
    if (apiRetStatus != CY_U3P_SUCCESS)
    {
        CyU3PDebugPrint (4, "CyU3PSetEpConfig failed, Error code
= %d\n", apiRetStatus);
        CyFxAppErrorHandler (apiRetStatus);
    }
}

```

USB 设置的回调函数

每当 FX3 固件框架需要应用程序来响应一个 USB 控制请求 (SETUP command) 时该回调函数将会被调用。由于使用了快速遍历模式，只有被指定到该接口或端点的类和厂家特定的请求以及标准的请求会被应用程序处理，其他标准的请求将会被固件库处理，本例没有实现类和厂家请求。该部分代码如下所示：

```

/* Callback to handle the USB setup requests. */
CyBool_t
CyFxSlFifoApplnUSBSetupCB (
    uint32_t setupdat0,
    uint32_t setupdat1

```

```

    )
{
    /* Fast enumeration is used. Only requests addressed to the
    interface, class,
        * vendor and unknown control requests are received by this
    function.
        * This application does not support any class or vendor
    requests. */
    CyBool_t isHandled = CyFalse;

    return isHandled;
}

```

USB 事件的回调函数

USB 关注的事件包括：设置配置，复位和断开连接。Slave Fifo 环路在接收到一个 SETCONF 时间后被启动，在 USB 被复位或断开的时候停止。该部分代码如下所示

```

/* This is the callback function to handle the USB events. */
void
CyFxsIFifoApplnUSBEventCB (
    CyU3PUsbEventType_t evtype,
    uint16_t          evdata
)
{
    switch (evtype)
    {
        case CY_U3P_USB_EVENT_SETCONF:
            /* Stop the application before re-starting. */
            if (glIsApplnActive)
            {
                CyFxsIFifoApplnStop ();
            }
            /* Start the loop back function. */
            CyFxsIFifoApplnStart ();
            break;

        case CY_U3P_USB_EVENT_RESET:
        case CY_U3P_USB_EVENT_DISCONNECT:
            /* Stop the loop back function. */
            if (glIsApplnActive)
            {
                CyFxsIFifoApplnStop ();
            }
    }
}

```

```

        }
        break;

default:
    break;
}
}

```

DMA 设置

Slave Fifo 应用使用 2 个 DMA 用户通道。这些通道在收到 USB 主机的设置配置后被设置。DMA 缓冲区大小由 USB 连接速度决定。

```

/* Create a DMA MANUAL channel for U2P transfer.
 * DMA size is set based on the USB speed. */
dmaCfg.size = size;
dmaCfg.count = CY_FX_SLFIFO_DMA_BUF_COUNT;
dmaCfg.prodSckId = CY_FX_PRODUCER_USB_SOCKET;
dmaCfg.consSckId = CY_FX_CONSUMER_PPORT_SOCKET;
dmaCfg.dmaMode = CY_U3P_DMA_MODE_BYTE;
/* Enabling the callback for produce event. */
dmaCfg.notification = CY_U3P_DMA_CB_PROD_EVENT;
dmaCfg.cb = CyFxFifoUtoPDmaCallback;
dmaCfg.prodHeader = 0;
dmaCfg.prodFooter = 0;
dmaCfg.consHeader = 0;
dmaCfg.prodAvailCount = 0;

apiRetStatus = CyU3PDmaChannelCreate (&glChHandleSlFifoUtoP,
                                       CY_U3P_DMA_TYPE_MANUAL, &dmaCfg);
if (apiRetStatus != CY_U3P_SUCCESS)
{
    CyU3PDebugPrint (4, "CyU3PDmaChannelCreate failed, Error
code = %d\n", apiRetStatus);
    CyFxAppErrorHandler(apiRetStatus);
}

/* Create a DMA MANUAL channel for P2U transfer. */
dmaCfg.prodSckId = CY_FX_PRODUCER_PPORT_SOCKET;
dmaCfg.consSckId = CY_FX_CONSUMER_USB_SOCKET;
dmaCfg.cb = CyFxFifoPtoUDmaCallback;
apiRetStatus = CyU3PDmaChannelCreate (&glChHandleSlFifoPtoU,
                                       CY_U3P_DMA_TYPE_MANUAL, &dmaCfg);

```

```

    if (apiRetStatus != CY_U3P_SUCCESS)
    {
        CyU3PDebugPrint (4, "CyU3PDmaChannelCreate failed, Error
code = %d\n", apiRetStatus);
        CyFxAppErrorHandler(apiRetStatus);
    }

```

使能 DMA 通道

```

/* Set DMA channel transfer size. */
apiRetStatus = CyU3PDmaChannelSetXfer (&glChHandleSlFifoUtoP,
CY_FX_SLFIFO_DMA_TX_SIZE);
if (apiRetStatus != CY_U3P_SUCCESS)
{
    CyU3PDebugPrint (4, "CyU3PDmaChannelSetXfer Failed, Error
code = %d\n", apiRetStatus);
    CyFxAppErrorHandler(apiRetStatus);
}
apiRetStatus = CyU3PDmaChannelSetXfer (&glChHandleSlFifoPtoU,
CY_FX_SLFIFO_DMA_RX_SIZE);
if (apiRetStatus != CY_U3P_SUCCESS)
{
    CyU3PDebugPrint (4, "CyU3PDmaChannelSetXfer Failed, Error
code = %d\n", apiRetStatus);
    CyFxAppErrorHandler(apiRetStatus);
}

```

DMA 回调函数

在本例中，有两个数据通道分别为 U to P 和 P to U。每个数据通路上的 DMA 事件都会为之注册一个 DMA 回调函数，当一个 DMA 的缓存中有数据时，该事件发生。该缓存中的数据可以使来自 USB 输出 或者 GPIF II 套接字。在 DMA 回调函数中，该缓存中的数据被传递给 GPIF-II 套接字或者 USB 输入端点。

这两个回调函数代码如下所示：

```

/* DMA callback function to handle the produce events for U to P
transfers. */
void
CyFxSlFifoUtoPDmaCallback (
    CyU3PDmaChannel *chHandle,
    CyU3PDmaCbType_t type,
    CyU3PDmaCBInput_t *input
)
{

```

```

    CyU3PReturnStatus_t status = CY_U3P_SUCCESS;

    if (type == CY_U3P_DMA_CB_PROD_EVENT)
    {
        /* This is a produce event notification to the CPU. This
notification is
        * received upon reception of every buffer. The buffer will
not be sent
        * out unless it is explicitly committed. The call shall
fail if there
        * is a bus reset / usb disconnect or if there is any
application error. */
        status = CyU3PDmaChannelCommitBuffer (chHandle,
input->buffer_p.count, 0);
        if (status != CY_U3P_SUCCESS)
        {
            CyU3PDebugPrint (4, "CyU3PDmaChannelCommitBuffer failed,
Error code = %d\n", status);
        }

        /* Increment the counter. */
        glDMARxCount++;
    }
}

/* DMA callback function to handle the produce events for P to U
transfers. */
void
CyFxs1FifoPtoUDmaCallback (
    CyU3PDmaChannel *chHandle,
    CyU3PDmaCbType_t type,
    CyU3PDmaCBInput_t *input
)
{
    CyU3PReturnStatus_t status = CY_U3P_SUCCESS;

    if (type == CY_U3P_DMA_CB_PROD_EVENT)
    {
        /* This is a produce event notification to the CPU. This
notification is
        * received upon reception of every buffer. The buffer will
not be sent
        * out unless it is explicitly committed. The call shall
fail if there

```

```
        * is a bus reset / usb disconnect or if there is any
application error. */
        status = CyU3PDmaChannelCommitBuffer (chHandle,
input->buffer_p.count, 0);
        if (status != CY_U3P_SUCCESS)
        {
            CyU3PDebugPrint (4, "CyU3PDmaChannelCommitBuffer failed,
Error code = %d\n", status);
        }

        /* Increment the counter. */
        glDMATxCount++;
    }
}
```

通过上述内容，我们已经分析了 FX3 的 Slave Fifo 的应用示例固件代码的框架和细节，理解该框架和处理流程，有助于我们根据实际应用，灵活的更改应用内容。

小梅哥

2016 年 12 月 17 日于成都市电子科技大学

联系我们：

芯航线 FPGA 技术支持群：472607506

邮箱：xiaomeige_fpga@foxmail.com

网址：<https://xiaomeige.taobao.com>