

Interfacing SRAM with FX2LP over GPIF

Associated Project: Yes

Associated Part Family: CY7C68013A

Related Application Notes: [AN65209](#), [AN66806](#), [AN15456](#)

To get the latest version of this application note, or the associated project file, please visit <http://www.cypress.com/go/AN57322>.

More code examples? We heard you.

To access a variety of FX2LP code examples, please visit our [USB High-Speed Code Examples webpage](#).

Are you looking for USB 3.0 peripheral controllers?

To access USB 3.0 product family, please visit our [USB 3.0 Product Family webpage](#).

This application note discusses how to connect Cypress SRAM CY7C1399B to FX2LP over the General Programmable Interface (GPIF). It describes how to create read and write waveforms using the GPIF Designer. This application note is also useful as a reference to connect FX2LP to other SRAMs.

Contents

1	Introduction.....	1	4.4	uVision2 Project and Firmware	12
2	Hardware Connections	2	5	Testing with Control Center	17
3	Designing GPIF Interconnect.....	2	6	Summary	21
4	GPIF Waveforms	5	7	References	21
4.1	Write Waveform	5		Appendix A.....	22
4.2	Read Waveform	9		Document History.....	30
4.3	Exporting GPIF Waveforms	12		Worldwide Sales and Design Support.....	31

1 Introduction

The GPIF is an 8-bit or 16-bit programmable parallel interface that helps to reduce system costs by providing a glueless interface between the EZ-USB FX2LP™ and an external peripheral. It is a highly configurable and flexible piece of hardware that allows you to get the most out of your USB 2.0 design. GPIF fits into applications that need an external mastering device to exchange information. The GPIF allows the EZ-USB FX2LP to perform local bus mastering to external peripherals implementing a wide variety of protocols. For example, EIDE/ATAPI, printer parallel port (IEEE P1284), Utopia, and other interfaces are supported using the GPIF block of the EZ-USB FX2LP. Refer to [AN66806](#) for more insight on the FX2LP GPIF topic.

GPIF Designer is a utility that Cypress provides to create GPIF waveform descriptors. This is done according to the read and write cycle timing of the peripherals, to connect them with FX2LP. When created, these waveforms can be exported to a C file, which is included into the project workspace. This document explains the process of defining the interface, creating waveforms, exporting them, and including them in the project framework. Familiarity with the examples and documentation on the EZ-USB FX2LP development kit and Chapter 10 (GPIF) of the [EZ-USB FX2LP Technical Reference Manual](#) is useful in designing the waveforms. For complete list of application notes, click [here](#).

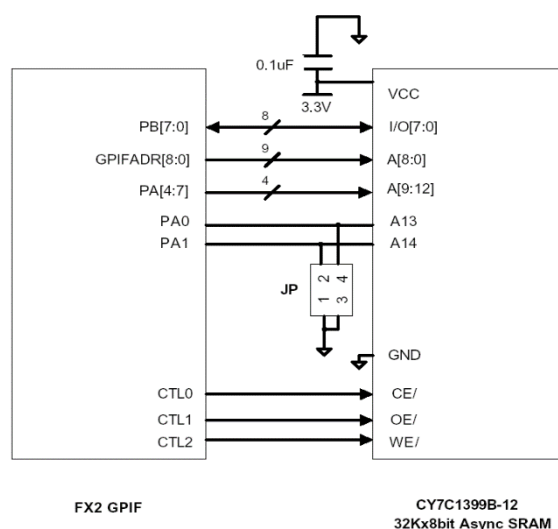
Note: The example explained in the next section can be reproduced only on 100-pin or 128-pin devices of FX2LP. This is because the GPIFADR[0:8] pins are available in 100-pin and 128-pin packages only.

2 Hardware Connections

This section discusses the required hardware interconnection between FX2LP and the SRAM. According to the SRAM data sheet, communicating with this device requires three control signals, an address, and a data bus. The SRAM's three control signals are a chip enable CE/, an output enable OE/, and a write enable WE/. The address and data buses are fifteen and eight bits wide, respectively. To address memory locations greater than 512 (only 9-bit address bus is provided by GPIF Designer), additional port I/O pins are required. Therefore, PA[7:4] and PA[1:0] of FX2LP are wired to A[12:9] and A[14:13] of SRAM respectively.

PA[7:4] is used to control A[12:9]. This gives the firmware access to 16, 512 byte banks for a total contiguous space of 8K. PA[0] and PA[1] are used to access four such 8K byte banks, providing access to the entire 32K of space.

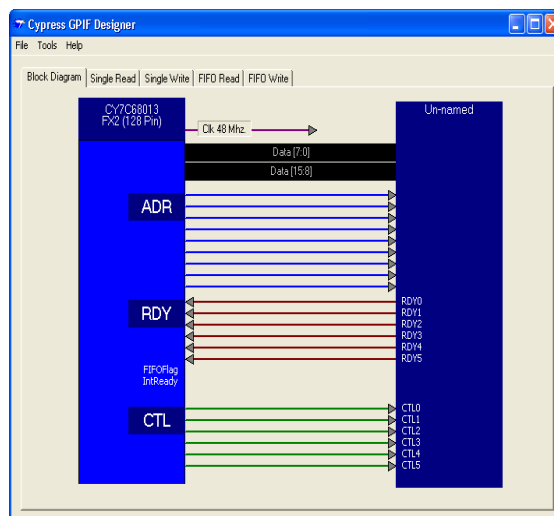
Figure 1. Hardware Connection Diagram



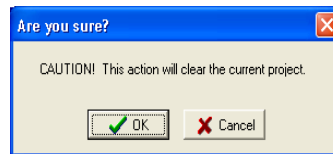
3 Designing GPIF Interconnect

The GPIF Designer utility is used to create the waveform descriptors to read and write from the SRAM. It is available along with the [CY3684 EZ-USB FX2LP DVK](#) setup. The following steps demonstrate how to define the interface and create the waveforms.

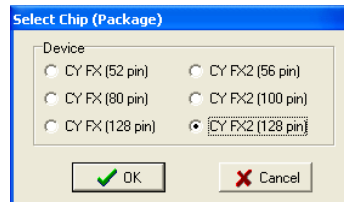
1. Start the Cypress GPIF Designer tool.



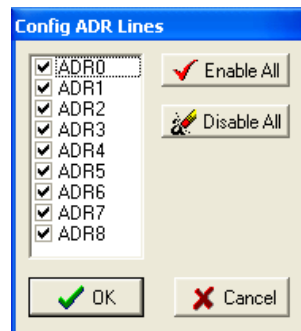
2. Go to **File > New**. The following window appears; Click **OK**.



3. In the window that pops up, select the appropriate part and click **OK**.



4. Right-click the **Un-named** label and rename it as **SRAM**.
5. Right-click the **Data [15:8]** band and clear **Use this bus**. Only the lower eight bits of the data bus are used.
6. Right-click on the **ADR** trace. The Config ADR Lines dialog box appears. All nine address lines of the GPIF are used and must remain selected. Click **OK**.

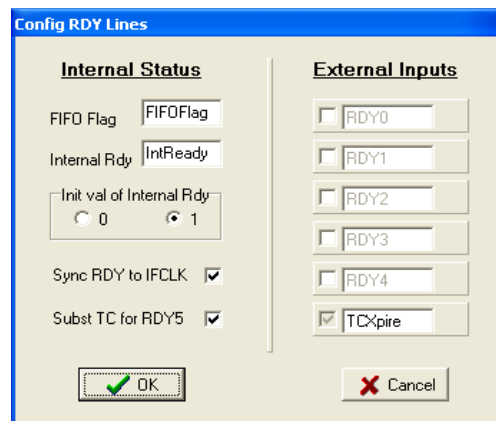


7. Right-click on the **RDY** trace. The Config RDY Lines dialog box appears. SRAM does not have any status indicators. Therefore, there is no need to define any RDY lines. Clear the checkboxes under "External Inputs". Select the following checkboxes:

- Sync RDY to IFCLK
- Subst TC for RDY5

where IFCLK stands for Interface Clock and TC stands for Transaction Count.

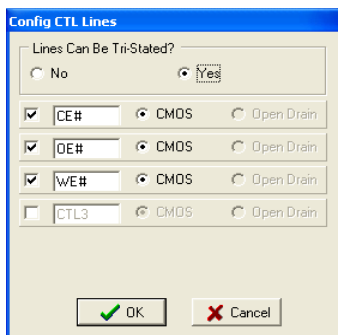
The window appears as follows. Click **OK**.



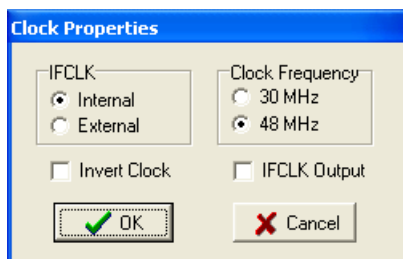
8. Right-click on **CTL** trace. The Config CTL Lines window is displayed. In this window,

- Select **Yes** for 'Lines can be tristated?'. When CTL lines are configured to be tristate, CTL[5:4] are not available.
- Rename CTL [0-2] lines to **CE#**, **OE#**, and **WE#** respectively.
- Uncheck the "unused" label (CTL3).

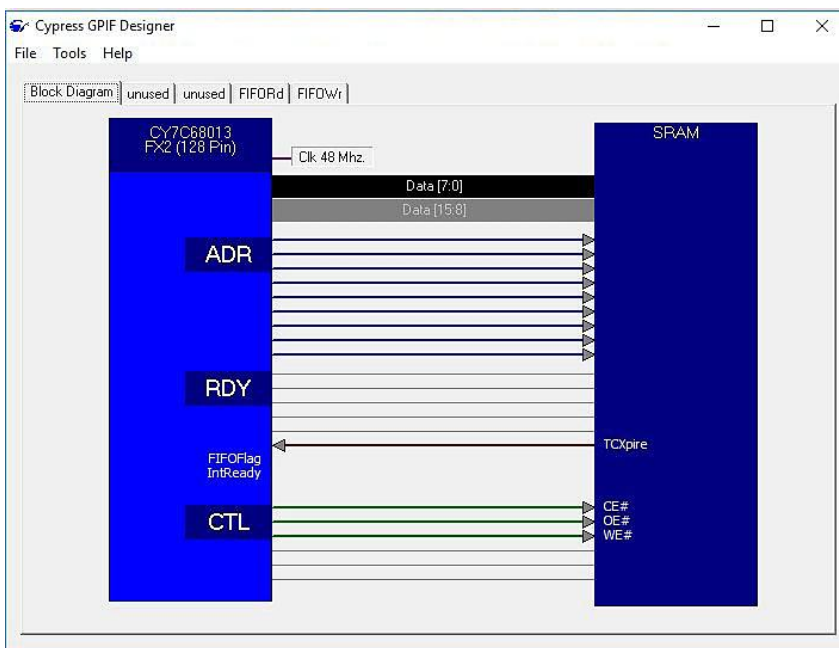
The window appears as follows. Click **OK**.



9. Right-click on **48 MHz CLK**. Uncheck **IFCLK Output**. The Clock Properties window is displayed. The interface is asynchronous and GPIF uses the internal 48 MHz clock.



These steps define and configure the GPIF interface for the SRAM. The following figure shows how the interface looks after the configuration.



The next step is to design the read and write waveforms.

4 GPIF Waveforms

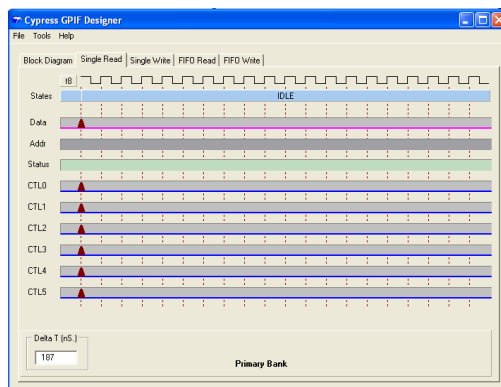
After the interface is configured, create the read and write waveforms using which communication takes place over the interface.

4.1 Write Waveform

Write waveforms are designed to write data from the Endpoint FIFO into the SRAM. It must satisfy the timing requirements of the various signals involved in the write cycle of the SRAM.

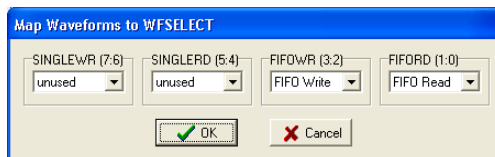
1. In the GPIF Designer window, click the **Single Read** tab to select it. Right-click and select **Set Tab Label**; rename as “Unused”. Repeat with “Single Write” tab, also renaming it as “Unused”.

Figure 2. Single Read Tab



2. Select **Tools > Map Waveforms to WFSELECT**.

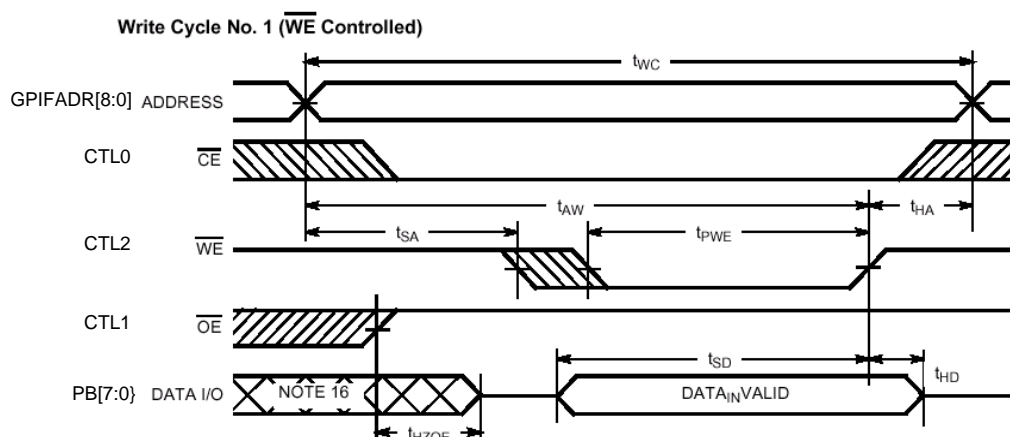
Figure 3. Map Waveforms



Make sure that the FIFO Write waveform is mapped to FIFOWR and the FIFO Read waveform is mapped to FIFORD. This ensures that when GPIF FIFO Write operation is launched, the FIFO Write waveform is executed and when a GPIF FIFO Read operation is launched, the FIFO Read waveform is executed. The mapping of bit fields is identical to the bit fields in the GPIFWFSELECT register. The waveforms are already mapped appropriately; click **OK**.

To construct the FIFO Write waveform, first review the write cycle timing for the SRAM (Figure 4) and its timing parameters (Table 1).

Figure 4. Write Cycle Timing for SRAM



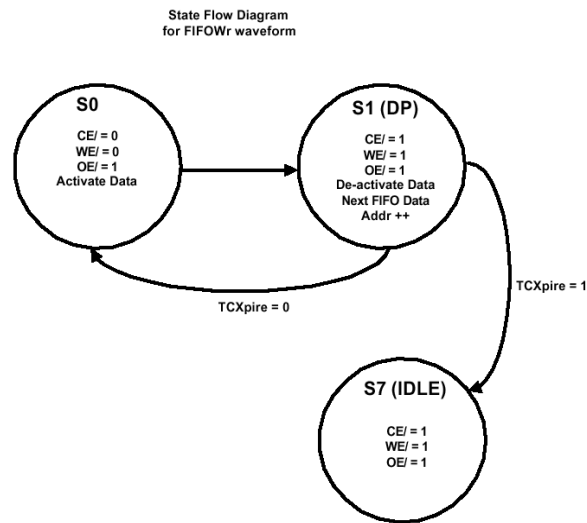
Note: PB[7:0] pins of FX2LP GPIF are interfaced to the Data I/O[7:0] pins of SRAM. This is shown in [Figure 1](#).

Table 1. Write Cycle Timing Parameters

Parameter (Minimum value)	Time (ns)	Notes
t_{WC} (Write Cycle Time)	12	When IFCLK=48 MHz, each GPIF cycle is 20.83 ns. Therefore, it only takes one cycle to write a byte.
t_{PWE} (WE/ Pulse Width)	8	When IFCLK=48 MHz, each GPIF cycle is 20.83 ns. Therefore, WE/ only needs to be driven low for one cycle.
t_{SD} (Data Setup to Write End)	7	Driving data together with WE/ LOW meets the setup time easily.
t_{HA} (Address Hold from Write End)	0	It is not required to keep the address asserted after WE/ goes HIGH.
t_{SA} (Address Set-Up to Write Start)	0	No setup time required for address with respect to WE/ going LOW. This means that Address and WE/ can be asserted at the same time.
t_{AW} (Address Set-Up to Write End)	8	Because address is asserted for one GPIF cycle and WE/ is de-asserted in the next cycle, this setup time is easily met.
t_{HD} (Data Hold from Write End)	0	It is not required to keep driving data after WE/ is de-asserted.

Now that the timing parameters involved are defined, the write waveform can be designed in GPIF designer. The following state flow diagram must be accomplished:

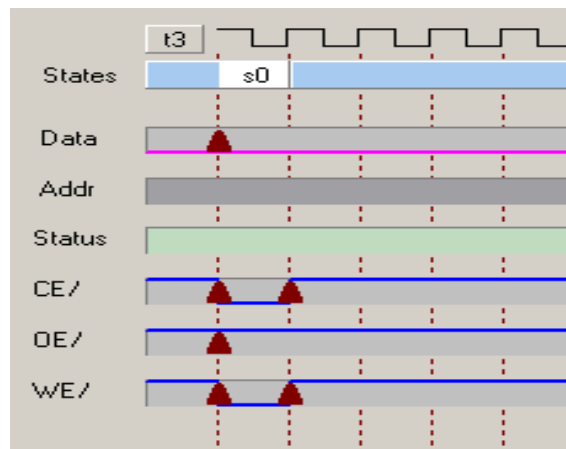
Figure 5. State Diagram



Note: In Figure 5, DP stands for Decision Point.

Follow these steps to complete the FIFO write waveform.

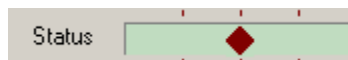
3. Click the **FIFO Write** waveform tab.
4. Click on the **WE/** trace one clock cycle from the left boundary. This places an action point and creates the WE/ waveform. State 0 (s0) is generated automatically and lasts for 1 IFCLK cycle (20.83 ns). WE/ is asserted for 20.83 ns. This easily satisfies the tPWE requirement.
5. Assert and de-assert CE/ along with WE/. To do this, click on the **CE/** trace one clock cycle from the left boundary.
6. OE/ must be HIGH throughout the waveform. To ensure this, right-click on the action point on the **OE/** trace and select **High (1)**. This considers the CTL line activity and the waveform appears as shown in the following diagram.



7. The data bus is also driven in s0. To do this, right-click on the data action point, and select **Activate Data**.
8. The data bus should only be driven for one clock cycle. To stop driving the data after one clock cycle, place another action point on the data trace after one clock cycle. Notice that the data trace is high for just the duration of s0 now. The waveform should appear as follows.



9. The next step is to add a decision point (DP) state to loop through this waveform until the GPIF transaction count (GPIFTC) expires. To do this, test the internal TCXpire flag in a DP state and only branch to the IDLE state when the transaction count expires. In the DP state, the GPIFADR lines are also incremented.
10. A DP must be implemented after s0. To do this, set an action point on the Status Trace by clicking at the right boundary of s0.



11. A dialog box appears prompting you for DP branch conditions. Select the condition as follows.

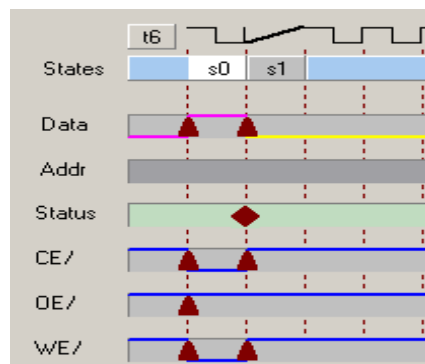
Specify Decision Point

IF (TCXpire = 1 AND TCXpire = 1)

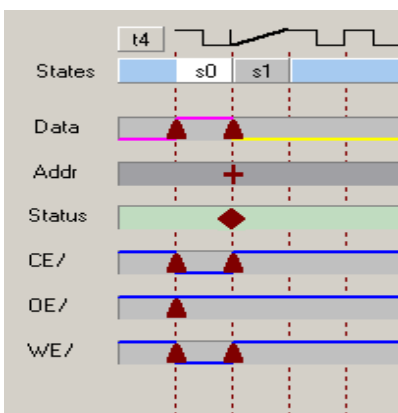
THEN GOTO IDLE ELSE GOTO S0

☐ LOOP (Re-Execute)

12. This sets the GPIF to look at only one signal, the TCXpire flag. When the transaction count expires, the GPIF sets the TCXpire flag to 1. To set the branch condition, branch to the IDLE state and terminate the waveform. Otherwise, loop back to S0 and continue with the waveform. The transaction count decrements with every "Next FIFO Data" operation.
13. The internal FIFO pointer must be incremented in the waveform. To do this, right-click on the action point at the end of s0 on the data trace and select **Next FIFO Data**. This is highlighted by the yellow trace. The waveform should appear as follows.



14. Next, increment the GPIFADR lines. To do this, click on the **Addr** trace at the left boundary of s1. The final waveform appears as follows:



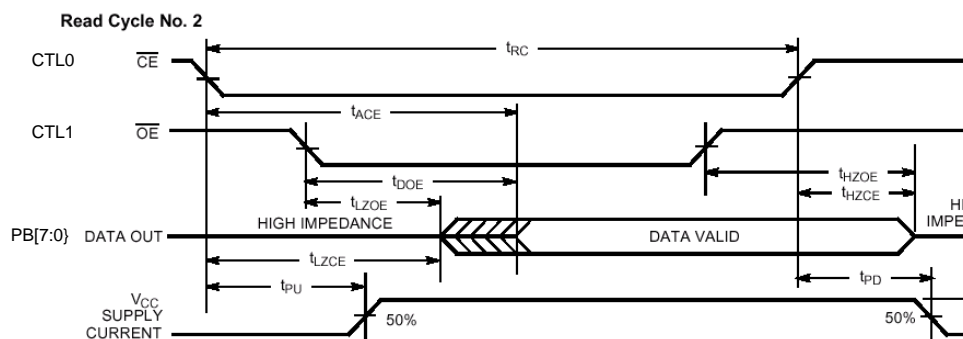
Save your waveform by selecting **File > Save**.

4.2 Read Waveform

Read waveforms are designed to read data from the SRAM into the Endpoint FIFO. It must satisfy the timing requirements of the various signals involved in the read cycle of the SRAM.

The process is similar to designing the write waveform. First, review the read cycle timing for the SRAM (Figure 6) and its timing parameters (Table 2).

Figure 6. Read Cycle Timing for SRAM



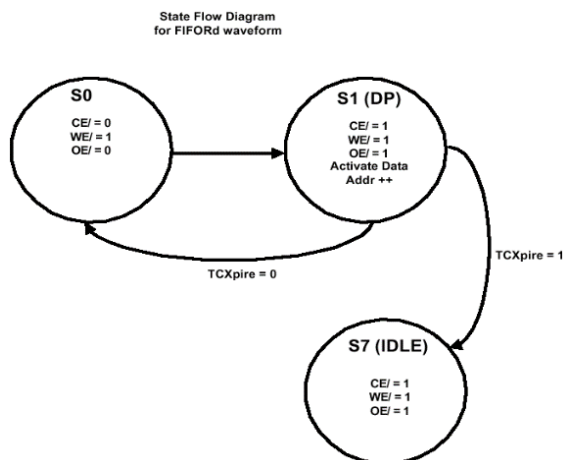
Note: The WE/ signal must be kept HIGH throughout the read cycle.

Table 2. Read Cycle Timing Parameters

Parameter	Time (ns)	Notes
tRC - Read Cycle Time (min)	12	When IFCLK = 48 MHz, each GPIF cycle is 20.83 ns.
tACE - CE/ Low to Data Valid (max)	12	When IFCLK = 48 MHz, each GPIF cycle is 20.83 ns. Therefore, CE/ must be driven low for one cycle in s0; sample data on the next cycle. Data is sampled on rising edge of IFCLK entering the state.
tDOE - OE/ Low to Data Valid (max)	5	Data is valid worst case 5 ns after OE/ is asserted. Data is valid in the next GPIF cycle; sample data in s1.
tHZOE - OE/ High to High-Z (max)	5	It is also possible to de-assert OE/ in s1 because the data is already sampled. Data hold time is not an issue.

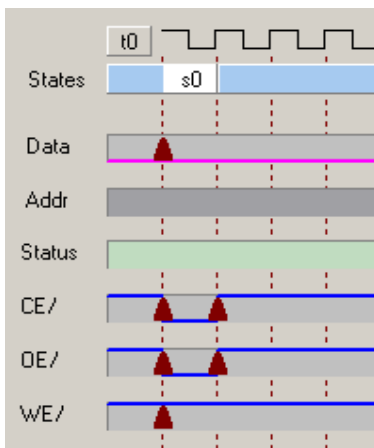
Now that the timing parameters involved are defined, the read waveform can be designed in GPIF designer. The following state flow diagram must be accomplished:

Figure 7. State Diagram

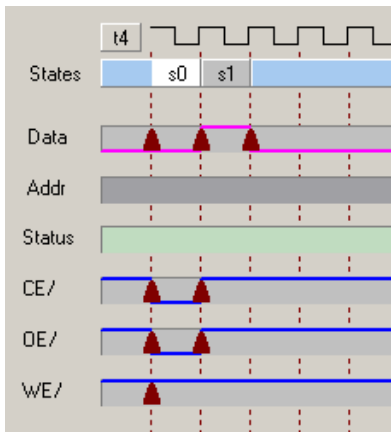


Follow these steps to complete the FIFO Read waveform.

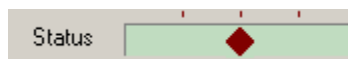
10. Click the **FIFO Read** tab.
11. Right-click the left boundary of the **OE/** trace and select Low (0). Click on the **OE/** trace one clock cycle from the left boundary. This places an action point and creates the OE/ waveform. State 0 (s0) is generated automatically and lasts for 1 IFCLK cycle (20.83 ns). Thus, OE/ is asserted for 20.83 ns.
12. Assert and de-assert CE/ along with OE/. To do this, right-click on **CE/** trace and select Low (0). Now, click on the **CE/** trace one clock cycle from the left boundary.
13. WE/ must be kept HIGH throughout the waveform. From the waveform observe that WE/ is high by default.
14. In the data trace, observe a yellow line on the data trace. This is because GPIF Designer forces all the four waveforms to be in the same IDLE state. Right-click the action point on the data trace and click **Same Data**. Observe that the yellow line has disappeared even in the write waveform. Change this after completing the read waveform.
15. The waveform should appear as follows:



16. The data bus must be sampled one clock cycle after asserting the CE/ to ensure that data is valid before sampling (tACE). To do this, click the **Data** trace on the right boundary of s0. This causes the data trace to toggle HIGH (placing an "Activate Data" event).
17. The data bus should only be sampled for one clock cycle. To stop sampling after one clock cycle, place another action point on the data trace after another clock cycle. Notice that the data trace is high for just the duration of s1 now. The waveform should appear as follows:



18. Next, add a decision point (DP) state to loop through this waveform until the GPIF transaction count (GPIFTC) expires. To do this, test the internal TCXpire flag in a DP state and only branch to the IDLE state when the decision count expires. In the DP state, increment the GPIFADR lines.
19. A DP must be implemented at the beginning of s1. To do this, set an action point on the Status Trace by clicking on the left boundary of s1.



20. A dialog box appears prompting you for DP branch conditions. Select the condition as follows:

Specify Decision Point

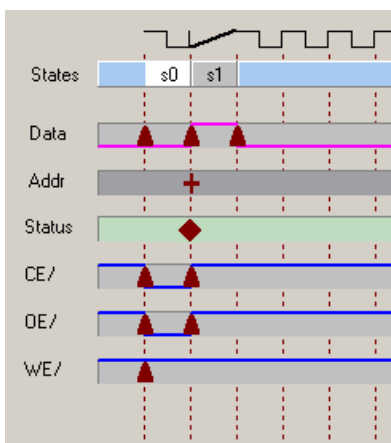
IF (TCXpire = 1 AND TCXpire = 1)

THEN GOTO IDLE ELSE GOTO S0

☐ LOOP (Re-Execute)

OK Cancel

21. This sets the GPIF to look at only one signal, the TCXpire flag. When the transaction count expires, the GPIF sets the TCXpire flag to 1. To set the branch condition, branch to the IDLE state and terminate the waveform. Otherwise, loop back to S0 and continue with the waveform. The transaction count decrements with every "Activate Data" operation for FIFO Reads. Note that this is different for FIFO Writes, which specifically requires a "Next Data" event to decrement the transaction count.
22. Next increment the GPIFADR lines. To do this, click on the **Addr** trace at the left boundary of s1. The final waveform should appear as follows:



23. Save your waveform by selecting **File > Save**.

This completes the read waveform. The write waveform is modified while creating the read waveform. To change this back, click the **FIFO Write** tab and put an action point on data trace at the end state s1. This causes the data trace to toggle HIGH. Right-click on the action point at the end of s1 on the data trace and select **De-activate Data**. Right-click the action point at the beginning of s1 and select the **Next FIFO** data. This is done because data action points located on the left edge of the IDLE state cannot present any form of "Next Data".

This completes the designing of GPIF waveforms. This waveform can now be exported to a *gpif.c* file and included in a project.

4.3 Exporting GPIF Waveforms

To export the waveforms to a C file and include it in the firmware project, follow these steps:

1. Select Tools > Export to gpif.c File.
2. Save file as *gpif.c* in a temporary location.

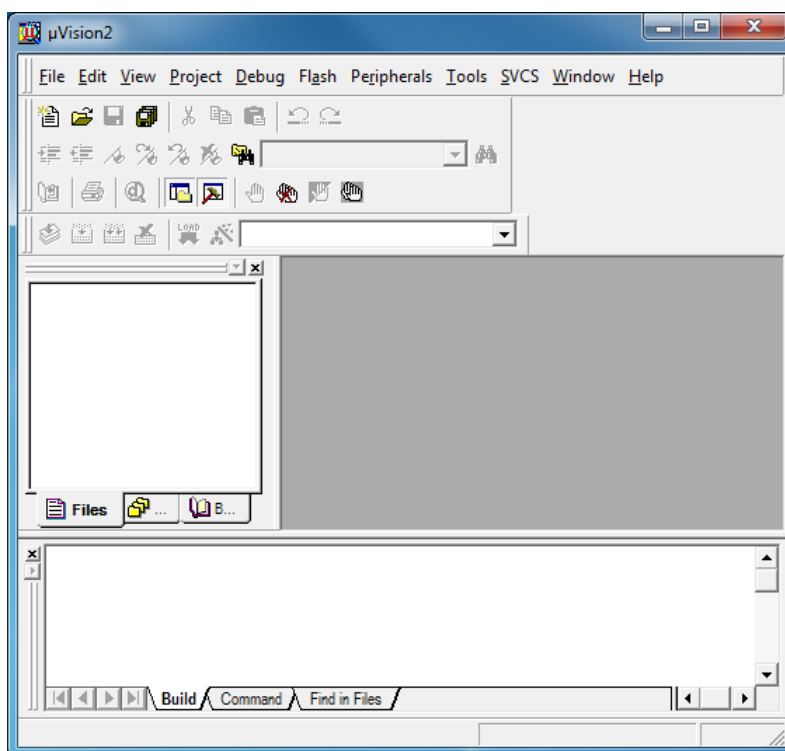
4.4 uVision2 Project and Firmware

The next step is to create a project that will include the GPIF waveform file (*gpif.c*) and write the firmware to interface SRAM to the FX2LP device.

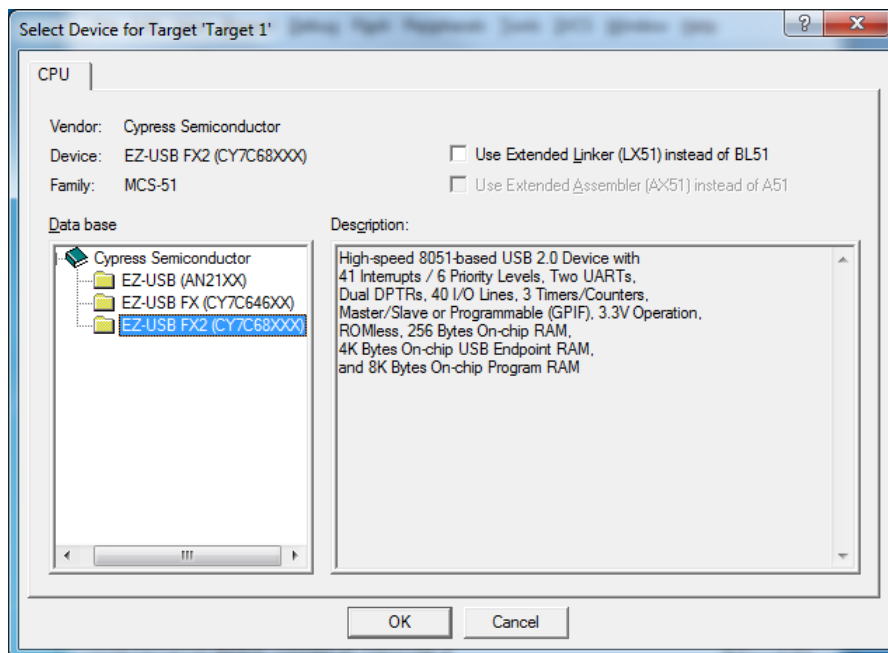
Note: You can either directly use the uVision Project 'FX2_SRAM_GPIF' available along with this application note in Cypress webpage or create a new project as shown below.

The following steps explain how you can create a new uVision project using a sample project (Bulkloop example project is used as reference here) available with the [FX2LP DVK](#).

1. Create a new folder called SRAM_GPIF.
If you have already installed the EZ-USB FX2LP development tools, create a new folder in the path C:\Cypress\USB\CY3684_EZ-USB_FX2LP_DVK\1.1\Firmware and name it as "SRAM_GPIF".
2. Copy the following project files from C:\Cypress\USB\CY3684_EZ-USB_FX2LP_DVK\1.1\Firmware\Bulkloop to the newly created folder.
 - dscr.a51
 - fw.c
 - bulkloop.c
3. Rename bulkloop.c as FX2_SRAM_GPIF.c.
4. Move the gpif.c file saved when exporting the GPIF waveforms section to this directory.
5. Start Microvision (uv2). The project window appears as follows.



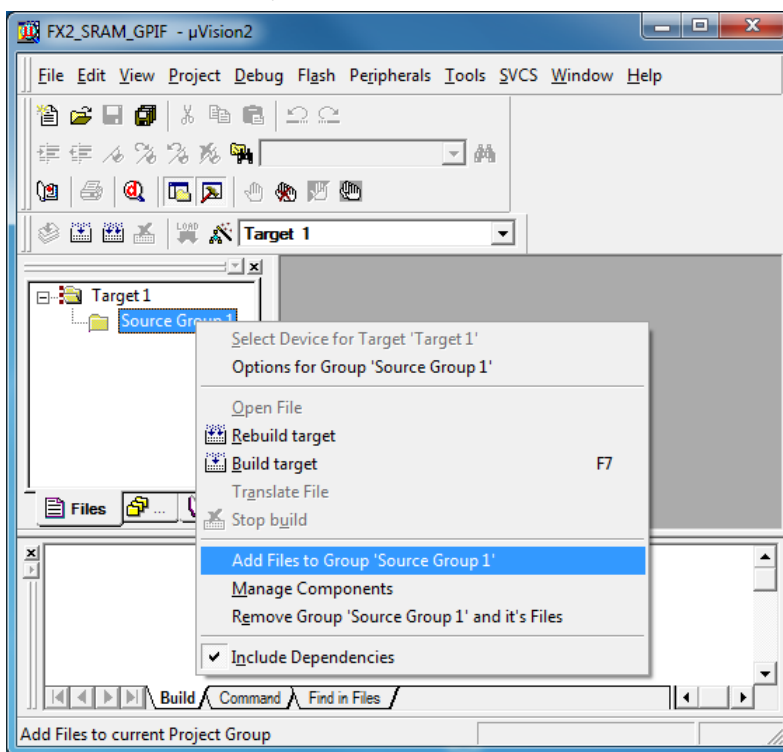
6. Go to **Project > New Project**. The tool prompts you to name the project and save it. Browse to the newly created SRAM_GPIF directory and save the project as FX2_SRAM_GPIF.
7. The following window is displayed prompting to select the type of device. Select **EZ-USB FX2 (CY7C68013)** from the list under "Cypress Semiconductor" and click **OK**.



8. You are prompted with the following dialog box. Click **No**.



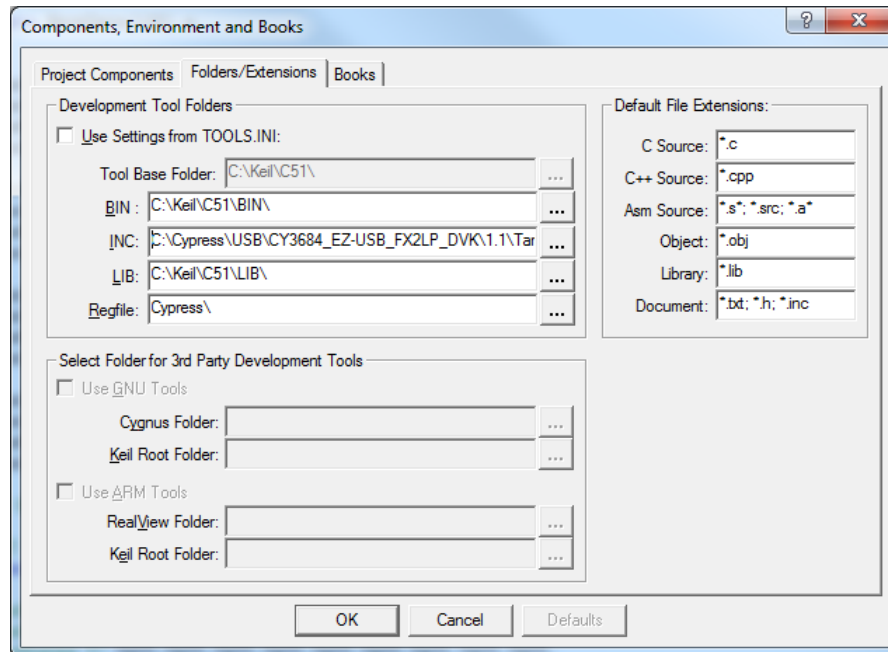
9. The following window is displayed. Add all the relevant source files to the FX2_SRAM_GPIF project. To add files, right click on **Source Group1** directory and select **Add Files** to “Source Group1”.



10. The files from the following path are added.

- C:\Cypress\USB\CY3684_EZ-USB_FX2LP_DVK\1.1\Firmware\SRAM_GPIF
 - gpif.c
 - fw.c
 - FX2_SRAM_GPIF.c
 - dscr.a51
- C:\Cypress\USB\CY3684_EZ-USB_FX2LP_DVK\1.1\Target\Lib\LP
 - USBImpTb.OBJ
 - Ezusb.lib

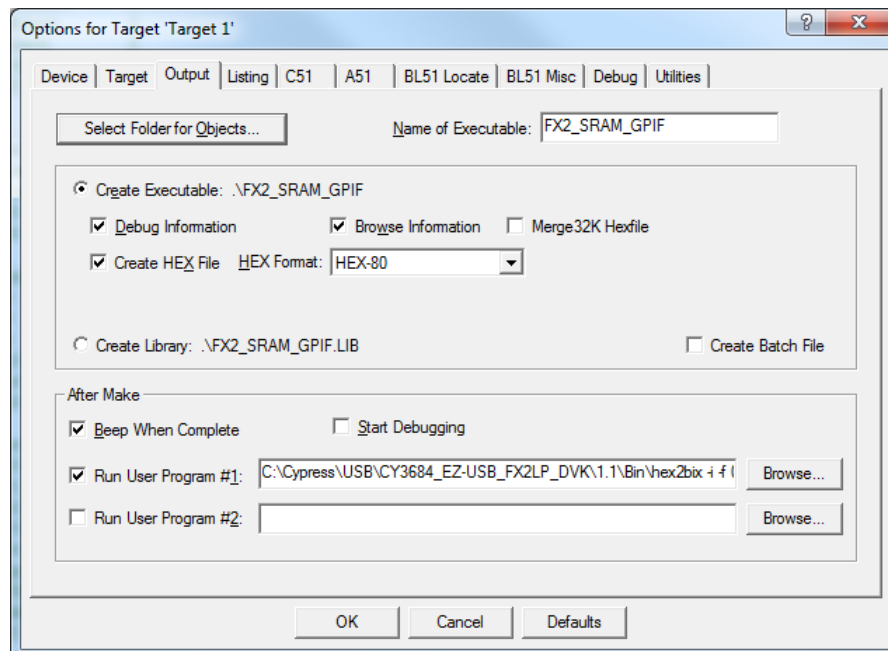
12. To set the Microvision setup environment, select **Project > Components, Environment and Books**.



13. In the 'Folders/Extensions' tab, edit the fields as below and then click **OK**.

- BIN Folder: C:\Keil\C51\BIN\
- INC Folder: C:\Cypress\USB\CY3684_EZ-USB_FX2LP_DVK\1.1\Target\Inc\;C:\Keil\C51\INC\
- LIB Folder: C:\Keil\C51\LIB\

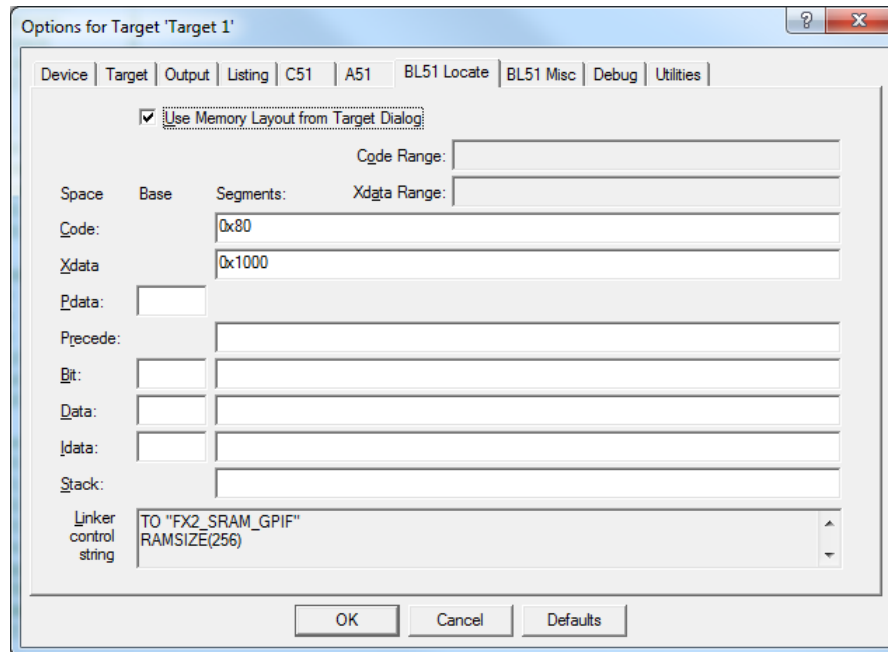
14. Go to **Project > Options for Target 'Target1'**. Select the Output tab and check the **Create Hex File** box.




'Run User Program #1' field should be filled with the line below:

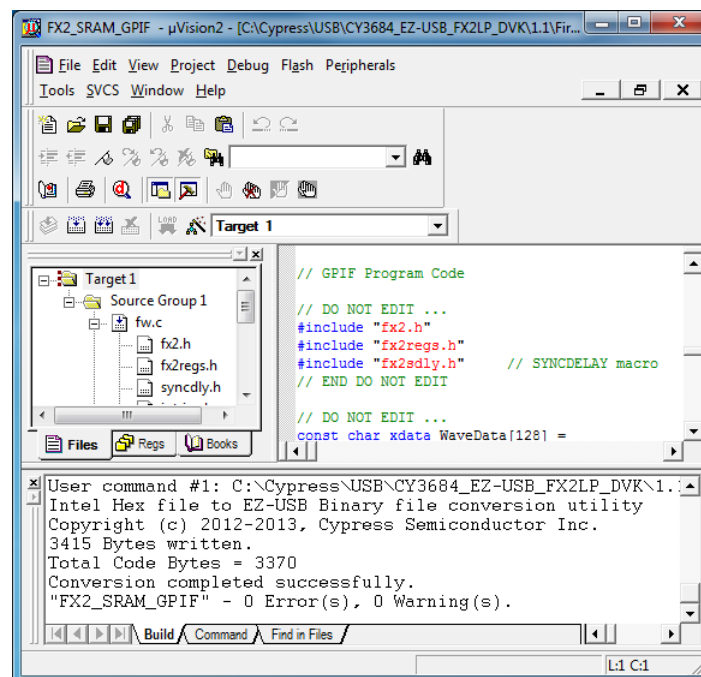
```
C:\Cypress\USB\CY3684_EZ-USB_FX2LP_DVK\1.1\Bin\hex2bix -i -f 0xC2 -o FX2_SRAM_GPIF.iic
FX2_SRAM_GPIF.hex
```

15. Click the **BL51 Locate** tab and locate the code and xdata as shown. Then click **OK**.



FX2LP has an interrupt vector located from 0x00 to 0x7C. So, it is recommended to set the Code location from 0x80 and the Xdata location from 0x1000.

16. Build the project by clicking the Rebuild all target files button . Make sure the code build is error-free. Click OK for warnings.
17. If the build is successful, the following screen appears in the output window. The firmware build is ready and you have to add firmware for the GPIF transfers.



18. The following sections of firmware are now added to the *FX2_SRAM_GPIF.c* file to create the final firmware for interfacing SRAM to the FX2LP device:

- Initialization code in the TD_Init() routine.
- Routines to turn LEDs on and off in TD_Poll() to indicate that the firmware is running.
- Vendor specific commands to do the following:
 - Trigger a GPIF FIFO Write transfer to the SRAM (handles both single and block writes)
 - Trigger a GPIF FIFO Read transfer to read from the SRAM (handles both single and block reads)

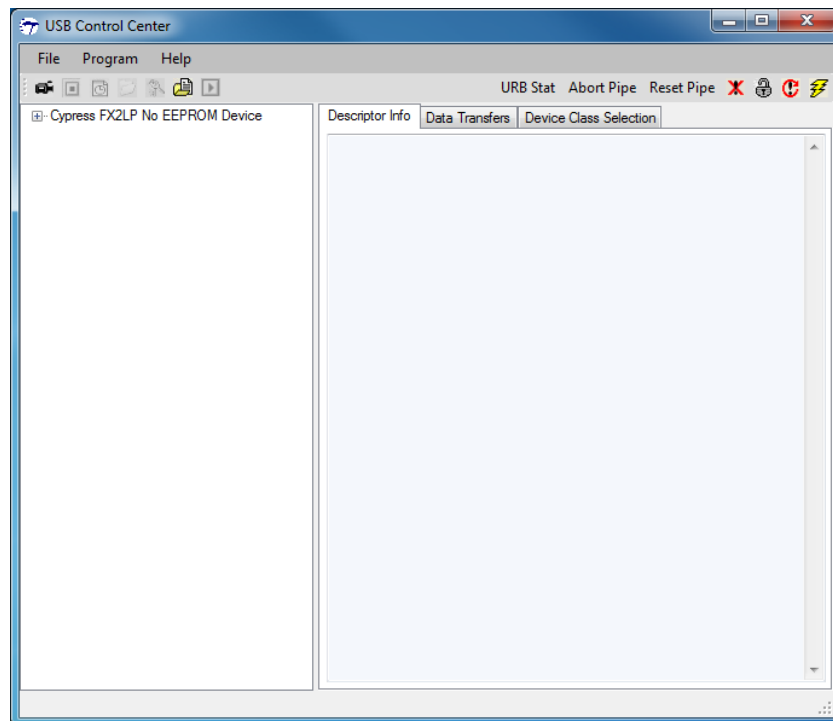
The code for this is provided in [Appendix A](#). Copy the code to the *FX2_SRAM_GPIF.c* file in the appropriate positions. For example, replace the existing TD_poll() with the one in this document. Also copy all the #defines and functions related to LEDs.

19. Rebuild the project. If you are using the 4K evaluation Keil tools, it might complain about the 4K code size limit. In this case, comment out the other vendor commands that are remnants from the bulkloop example and then rebuild the project.

5 Testing with Control Center

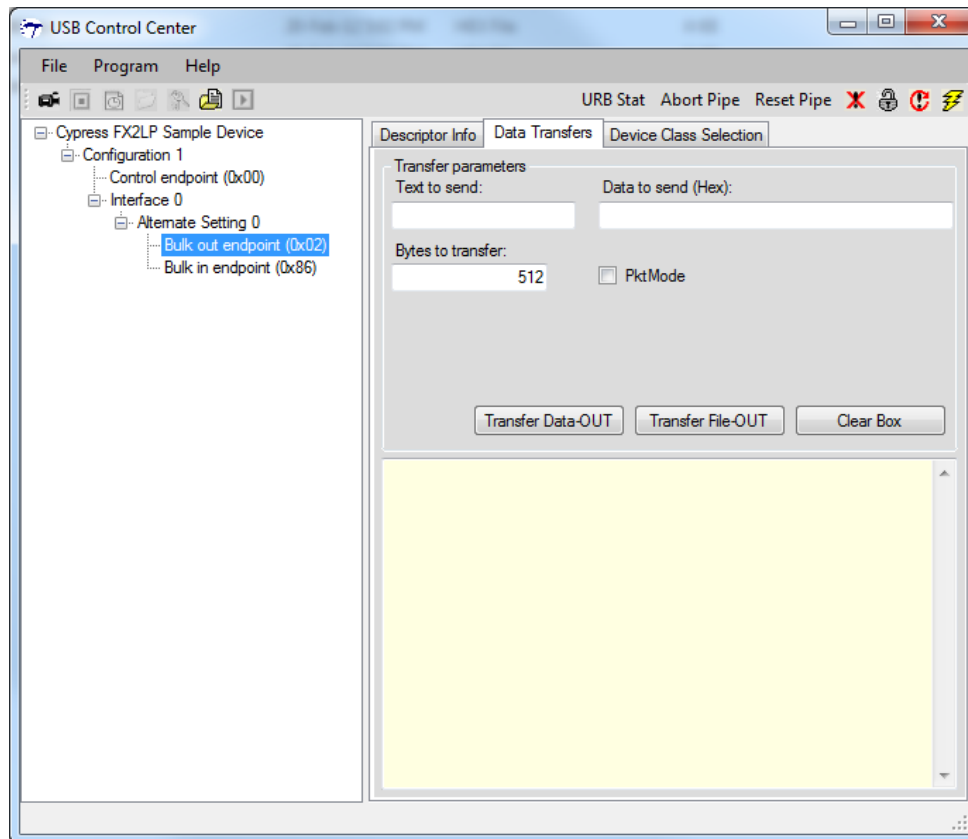
The functionality of the project created is verified using the CY3684 Development board and Control Center host application.

1. Set the EEPROM Enable switch in the development board to **OFF** and plug it into the host.
2. The development board enumerates “Cypress FX2LP No EEPROM Device”, which is seen in the device manager.
3. Download and install [Cypress Suite USB 3.4](#). This installs the Control Center utility, shown below.

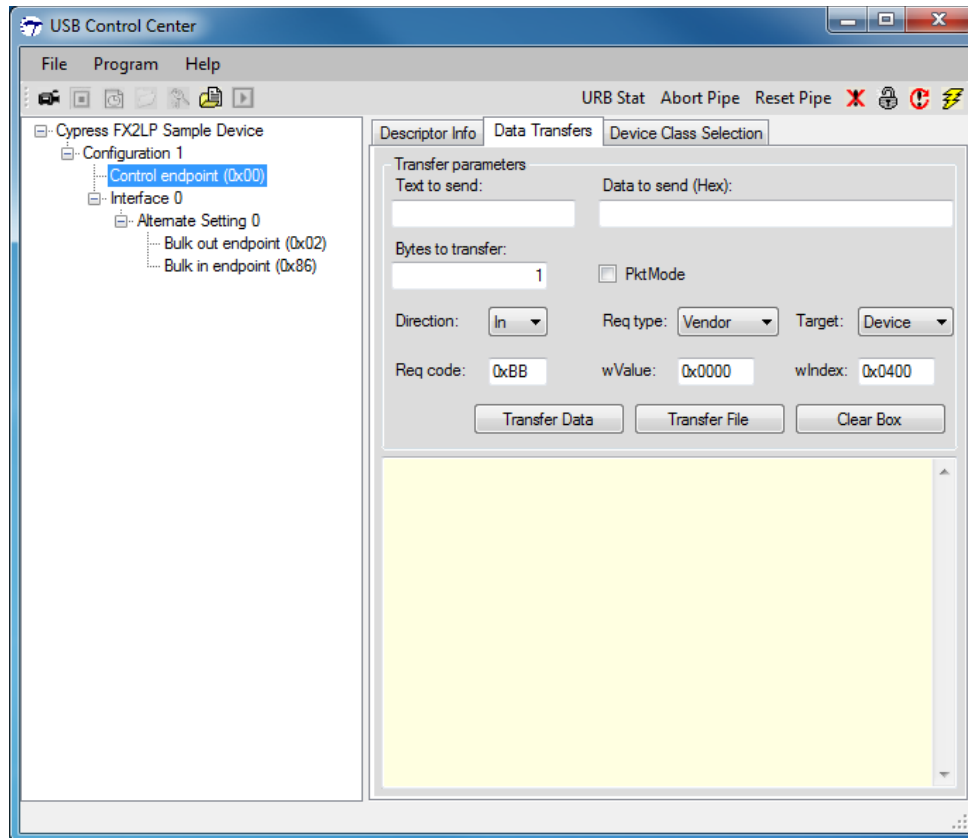


4. Select the device in the left pane and Click **Program > FX2 > RAM** and navigate to the newly created *FX2_to_SRAM.hex* file. Wait for the firmware to re-enumerate and reconnect as a Cypress FX2LP sample device.
5. When the device has re-enumerated successfully, it is possible to connect to the SRAM via the vendor IN commands 0xBB and 0xBC.

Transfer data to EP2OUT endpoint. This is done in two ways: use the bulk transfer bar to specify the endpoint to use data value and request length or use the **Transfer File-OUT** button to transfer a file of known data pattern. Make sure you select **Bulk out endpoint (0x02)** in the drop down in the left pane. Click on **Data Transfers** tab in the right pane and click on **Transfer File-OUT** button and select **1024_count.hex** located in the path C:\Cypress\USB\CY3684_EZ-USB_FX2LP_DVK\1.1\Target\File_Transfer.

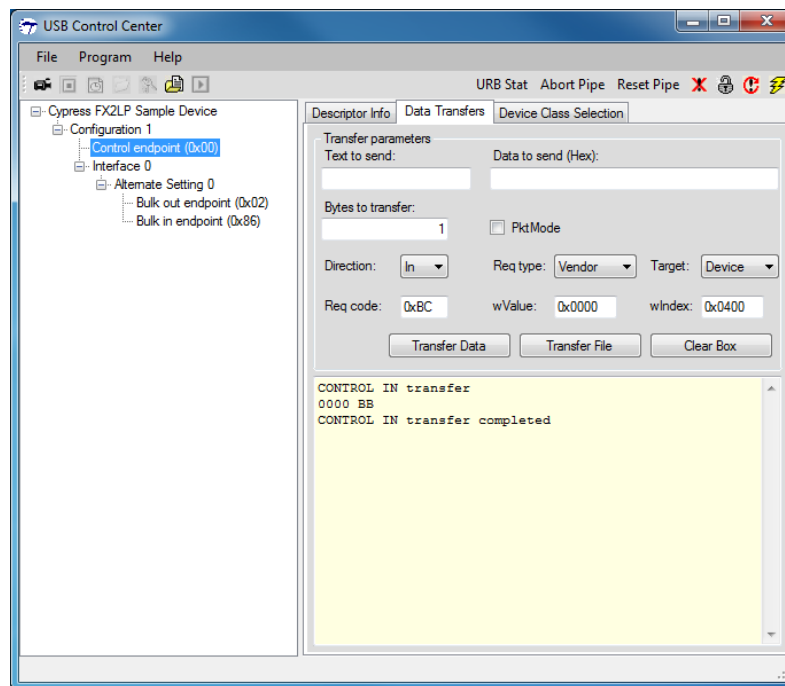


6. Select **Control endpoint (0x00)** in the left pane and click on **Data Transfers** tab in the right pane. Choose the **Req type** as Vendor. The **Req code** field represents the type of request. Enter either **0xBB** or **0xBC** in this field, depending on whether you want to actuate a write or read respectively. The **wValue** field specifies the SRAM address. The **wIndex** field specifies the transfer length in HEX. The length field must be the equivalent HEX value of the transfer length that is specified in step 7. For example, the figure below shows how to write 1K (0x0400) bytes starting from address 0x0000 in the SRAM in the Vend Req transfer bar.



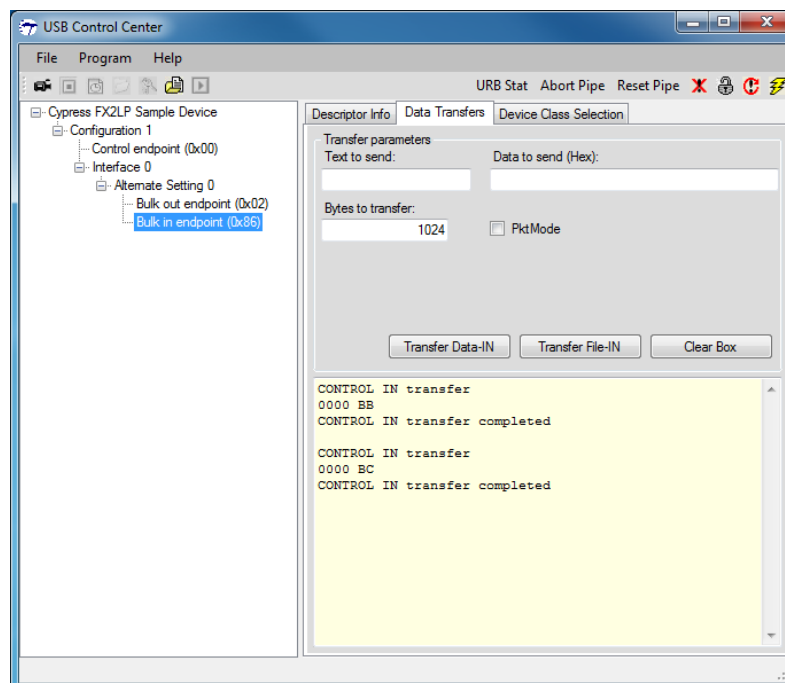
Click the **Transfer Data** button to trigger the write to the SRAM. When the request is processed, the output window displays the vendor request value of 0xBB.

7. To read 1K (0x0400) bytes starting from address 0x0000 in the SRAM, specify the following in the Vend Req transfer bar.

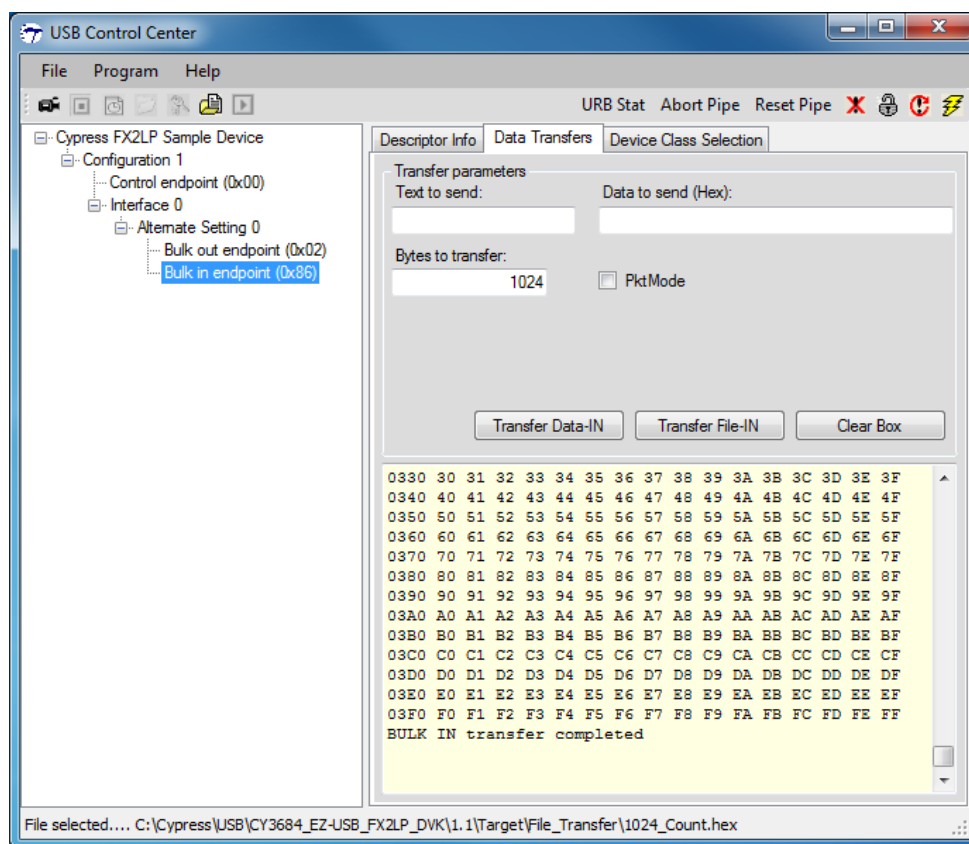


After the request is processed, the output window displays the vendor request value of 0xBC.

8. To transfer data to the host via **Bulk in endpoint (0x86)** (the data read from the SRAM), select the endpoint in the left pane and use the **Data Transfers** tab to specify the request length and click on **Transfer Data-IN**.



9. After the bulk transfer is complete, the data received on the host end is displayed in the control panel as shown.



10. Now, you can repeat the exercise for any SRAM address from 0x0000 to 0x1FFF (8K range), and transfer length from 0x0001 to 0x2000 (1 to 8192 bytes).

6 Summary

This application note describes an easy-to-follow procedure on how to create the read and write waveforms using GPIF Designer. It then explains how to create a firmware project with these waveform files to interface an SRAM to the FX2LP Development Kit board.

7 References

For more information on GPIF, refer to the following documents available on our website:

- [FX2LP GPIF Master-Slave FIFO Back-to-Back Setup](#)
- [Manual Mode Operation of EZ-USB FX2LP™ in GPIF and Slave FIFO Configuration](#)
- [EZ-USB FX2 GPIF Primer](#)
- [AN63787 - EZ-USB® FX2LP™ GPIF and Slave FIFO Configuration Examples Using an 8-Bit Asynchronous Interface](#)

For information on GPIF II, refer to the following application notes:

- [AN87216 - Designing a GPIF II Master Interface](#)
- [AN65974 - Designing with the EZ-USB® FX3™ Slave FIFO Interface](#)
- [AN68829 - Slave FIFO Interface for EZ-USB® FX3™: 5-Bit Address Mode](#)
- [AN75779 - How to Implement an Image Sensor Interface Using EZ-USB® FX3™ in a USB Video Class \(UVC\) Framework](#)

Appendix A.

```

/* GPIF write */
#define VX_BB 0xBB
/* GPIF read */
#define VX_BC 0xBC

#define GPIFTRIGRD 4
#define GPIF_EP2 0
#define GPIF_EP4 1
#define GPIF_EP6 2
#define GPIF_EP8 3

/* flag to let firmware know FX2 enumerated at high speed */
BOOL enum_high_speed = FALSE;
/* variable that contains EP6FIFOBC/L value */
static WORD xFIFOBC_IN = 0x0000;
static WORD xdata LED_Count = 0;
static BYTE xdata LED_Status = 0;

WORD addr, len, Tcount;
/* ...debug LEDs: accessed via movx reads only ( through CPLD ) */
/* it may be worth noting here that the default monitor loads at 0xC000 */
xdata volatile const BYTE LED0_ON_at_ 0x8000;
xdata volatile const BYTE LED0_OFF_at_ 0x8100;
xdata volatile const BYTE LED1_ON_at_ 0x9000;
xdata volatile const BYTE LED1_OFF_at_ 0x9100;
xdata volatile const BYTE LED2_ON_at_ 0xA000;
xdata volatile const BYTE LED2_OFF_at_ 0xA100;
xdata volatile const BYTE LED3_ON_at_ 0xB000;
xdata volatile const BYTE LED3_OFF_at_ 0xB100;
/* use this global variable when (de)asserting debug LEDs...*/

BYTE xdata ledX_rdvar = 0x00;
BYTE xdata LED_State = 0;

void LED_Off (BYTE LED_Mask);
void LED_On (BYTE LED_Mask);
void GpifInit ();

void LED_Off (BYTE LED_Mask)
{
    if (LED_Mask & bmBIT0)
    {
        ledX_rdvar = LED0_OFF;
        LED_State &= ~bmBIT0;
    }
    if (LED_Mask & bmBIT1)
    {
        ledX_rdvar = LED1_OFF;
        LED_State &= ~bmBIT1;
    }
    if (LED_Mask & bmBIT2)
    {
        ledX_rdvar = LED2_OFF;
        LED_State &= ~bmBIT2;
    }
    if (LED_Mask & bmBIT3)
    {
        ledX_rdvar = LED3_OFF;
        LED_State &= ~bmBIT3;
    }
}

void LED_On (BYTE LED_Mask)
{
    if (LED_Mask & bmBIT0)
    {

```

```

    ledX_rdvar = LED0_ON;
    LED_State |= bmBIT0;
}
if (LED_Mask & bmBIT1)
{
    ledX_rdvar = LED1_ON;
    LED_State |= bmBIT1;
}
if (LED_Mask & bmBIT2)
{
    ledX_rdvar = LED2_ON;
    LED_State |= bmBIT2;
}
if (LED_Mask & bmBIT3)
{
    ledX_rdvar = LED3_ON;
    LED_State |= bmBIT3;
}
}

//-----
// Task Dispatcher hooks
// The following hooks are called by the task dispatcher.
//-----

/* Called once at startup */
void TD_Init(void)
{
    /* set the CPU clock to 48MHz*/
    CPUCS = ((CPUCS & ~bmCLKSPD) | bmCLKSPD1) ;
    SYNCDELAY;
    /* set the slave FIFO interface to 48MHz*/
    IFCONFIG |= 0x40;

    /*change EP configuration */
    EP2CFG = 0xA0;
    SYNCDELAY;
    EP4CFG = 0x00;
    SYNCDELAY;
    EP6CFG = 0xE0;
    SYNCDELAY;
    EP8CFG = 0x00;

    /* out endpoints do not come up armed */
    /* set NAKALL bit to NAK all transfers from host */
    FIFORESET = 0x80;
    SYNCDELAY;
    /* reset EP2 FIFO */
    FIFORESET = 0x02;
    SYNCDELAY;
    /* reset EP6 FIFO */
    FIFORESET = 0x06;
    SYNCDELAY;
    /* clear NAKALL bit to resume normal operation */
    FIFORESET = 0x00;
    SYNCDELAY;
    /* allow core to see zero to one transition of auto out bit*/
    EP2FIFOCFG = 0x00;
    SYNCDELAY;
    /* auto out mode, disable PKTEND zero length send, byte ops*/
    EP2FIFOCFG = 0x10;
    SYNCDELAY;
    /* auto in mode, disable PKTEND zero length send, byte ops */
    EP6FIFOCFG = 0x08;
    SYNCDELAY;

    /* nable dual autopointer feature */
    AUTOPTRSETUP |= 0x01;

```

```

/* initialize GPIF registers */
GpifInit ();
PORTACFG = 0x00;
OEA |= 0xF3;
IOA &= 0xFC;

/* Enable remote-wakeup */
Rwuen = TRUE;
}

/* Called repeatedly while the device is idle */
void TD_Poll(void)
{
  /* blink LED0 to indicate firmware is running */
  if (++LED_Count == 10000)
  {
    if (LED_Status)
    {
      LED_Off (bmBIT0);
      LED_Status = 0;
    }
    else
    {
      LED_On (bmBIT0);
      LED_Status = 1;
    }
    LED_Count = 0;
  }
}

/* Called before the device goes into suspend mode */
BOOL TD_Suspend(void)
{
  return(TRUE);
}

/* Called after the device resumes */
BOOL TD_Resume(void)
{
  return(TRUE);
}

//-----
// Device Request hooks
// The following hooks are called by the end point 0
//device request parser.
//-----

BOOL DR_GetDescriptor(void)
{
  return(TRUE);
}

/* Called when a Set Configuration command is received */
BOOL DR_SetConfiguration(void)
{
  if( EZUSB_HIGHSPEED( ) )
  {
    /* FX2 enumerated at high speed */
    SYNCDELAY;
    /* set AUTOIN commit length to 512 bytes */
    EP6AUTOINLENH = 0x02;
    SYNCDELAY;
    EP6AUTOINLENL = 0x00;
    SYNCDELAY;
    enum_high_speed = TRUE;
  }
}

```



```
else
{
    /* FX2 enumerated at full speed */
    SYNCDELAY;
    /* set AUTOIN commit length to 64 bytes */
    EP6AUTOINLENH = 0x00;
    SYNCDELAY;
    EP6AUTOINLENL = 0x40;
    SYNCDELAY;
    enum_high_speed = FALSE;
}

Configuration = SETUPDAT[2];
return(TRUE);
}

/* Called when a Get Configuration command is received */
BOOL DR_GetConfiguration(void)
{
    EP0BUF[0] = Configuration;
    EP0BCH = 0;
    EP0BCL = 1;
    return(TRUE);
}

/* Called when a Set Interface command is received */
BOOL DR_SetInterface(void)
{
    AlternateSetting = SETUPDAT[2];
    return(TRUE);
}

/* Called when a Set Interface command is received */
BOOL DR_GetInterface(void)
{
    EP0BUF[0] = AlternateSetting;
    EP0BCH = 0;
    EP0BCL = 1;
    return(TRUE);
}

BOOL DR_GetStatus(void)
{
    return(TRUE);
}

BOOL DR_ClearFeature(void)
{
    return(TRUE);
}

BOOL DR_SetFeature(void)
{
    return(TRUE);
}

BOOL DR_VendorCmnd(void)
{
    BYTE tmp;

    switch (SETUPDAT[1])
    {
        case VR_NAKALL_ON:
            tmp = FIFORESET;
            tmp |= bmNAKALL;
            SYNCDELAY;
            FIFORESET = tmp;
            break;
        case VR_NAKALL_OFF:
```

```

    tmp = FIFORESET;
    tmp &= ~bmNAKALL;
    SYNCDELAY;
    FIFORESET = tmp;
    break;
/* actuate write to SRAM */
case VX_BB:
{
    EPOBUF[0] = VX_BB;
    /* select bank of 16x512 (bit shift MSB of wValue by 3*/
    /* and OR it with PA[7:4] */
    IOA = (IOA & 0x0F) + (SETUPDAT[3] << 3);
    /* set GPIFADR[8:0] to address passed down in wValue */
    GPIFADRH = SETUPDAT[3];
    GPIFADRL = SETUPDAT[2];

    /* get transfer length from wIndex field */
    len = ( (SETUPDAT[5] << 8) + SETUPDAT[4] );
    /* while the transfer length is non-zero */
    while (len)
    {
        /* if GPIF interface IDLE */
        if( GPIFTRIG & 0x80 )
        {
            /* if there's a packet in the peripheral domain for EP2 */
            if ( ! ( EP24FIFOFLGS & 0x02 ) )
            {
                /* if the FX2 enumerated at high-speed */
                if(enum_high_speed)
                {
                    /* if the transfer length is greater than 512 bytes */
                    if ( len > 0x0200 )
                    {
                        /* set GPIF transaction count to 512, since*/
                        /* GPIFADR can only access 512 locations at a time */
                        GPIFTCB1 = 0x02;
                        SYNCDELAY;
                        GPIFTCB0 = 0x00;
                        SYNCDELAY;
                        Tcount = 0x0200;
                    }
                    else
                    {
                        /* else set GPIF transaction count to EP2 */
                        GPIFTCB1 = EP2FIFOBCH;
                        SYNCDELAY;
                        GPIFTCB0 = EP2FIFOBCL;
                        SYNCDELAY;
                        Tcount = len;
                    }
                }
                /* if the FX2 enumerated at full-speed */
                else
                {
                    /* if the transfer length is greater than 64 bytes, */
                    if ( len > 0x040 )
                    {
                        /* set GPIF transaction count to 64 */
                        GPIFTCB1 = 0x00;
                        SYNCDELAY;
                        GPIFTCB0 = 0x40;
                        SYNCDELAY;
                        Tcount = 0x0040;
                    }
                    else
                    {
                        GPIFTCB1 = EP2FIFOBCH;
                        SYNCDELAY;
                        GPIFTCB0 = EP2FIFOBCL;
                        SYNCDELAY;
                    }
                }
            }
        }
    }
}

```

```

        Tcount = len;
    }
}
/* launch GPIF FIFO WRITE Transaction from EP2 FIFO*/
GPIFTRIG = GPIF_EP2;
SYNCDelay;
/* poll GPIFTRIG.7 GPIF Done bit */
while( !( GPIFTRIG & 0x80 ) )
{
    ;
}
SYNCDelay;
/* decrement transfer length by Tcount*/
len = len - Tcount;
/* if the transfer length is not a modulus of 512, no need to */

/* reset GPIFADR[8:0] to access next bank of 512 bytes,*/
/* handles full-speed case and high-speed case */
/* reset GPIFADR[8:0] to access the next bank at offset 0*/
if (!(len % 0x0200))
{
    GPIFADRH = 0x00;
    GPIFADRL = 0x00;
    /* increment the bank address by 1 to access next bank of 512*/
    IOA = ( ( IOA >> 4 ) + 1 ) << 4 );
}
}
}

EPOBCH = 0;
EPOBCL = 1;
EPOCS |= bmHSNAK;
break;
}
/* actuate read from SRAM */
case VX_BC:
{
    EPOBUF[0] = VX_BC;

    /* select bank of 16x512 (bit shift MSB of wValue by 3
    and OR it with PA[7:4]*/
    IOA = (IOA & 0x0F) + (SETUPDAT[3] << 3);
    /* set GPIFADR[8:0] to address passed down in wValue */
    GPIFADRH = SETUPDAT[3];
    GPIFADRL = SETUPDAT[2];
    /* get transfer length from wIndex field */
    len = ( (SETUPDAT[5] << 8) + SETUPDAT[4] );
    /* while the transfer length is non-zero */
    while (len)
    {
        /* if GPIF interface IDLE */
        if( GPIFTRIG & 0x80 )
        {
            /* if EP6 FIFO is not full */
            if( !( EP6FIFOFLGS & 0x01 ) )
            {
                /* if the FX2 enumerated at high-speed */
                if(enum_high_speed)
                {
                    /* if the transfer length is greater than 512 bytes, */
                    if ( len > 0x0200 )
                    {
                        /*set GPIF transaction count to 512, since
                        GPIFADR can only access 512*/
                        GPIFTCB1 = 0x02;
                        SYNCDelay;
                        GPIFTCB0 = 0x00;
                        SYNCDelay;
                        Tcount = 0x0200;
                    }
                }
            }
        }
    }
}

```

```

    }
    else
    {
        GPIFTCB1 = MSB(len);
        SYNCDELAY;
        GPIFTCB0 = LSB(len);
        SYNCDELAY;
        Tcount = len;
    }
}
/* if the FX2 enumerated at full-speed */
else
{
    /* if the transfer length is greater than 64 bytes*/
    if ( len > 0x0040 )
    {
        GPIFTCB1 = 0x00;
        SYNCDELAY;
        GPIFTCB0 = 0x40;
        SYNCDELAY;
        Tcount = 0x0040;
    }
    else
    {
        GPIFTCB1 = MSB(len);
        SYNCDELAY;
        GPIFTCB0 = LSB(len);
        SYNCDELAY;
        Tcount = len;
    }
}

/* launch GPIF FIFO READ Transaction to EP6IN */
GPIFTRIG = GPIFTRIGRD | GPIF_EP6;
SYNCDELAY;
/* poll GPIFTRIG.7 GPIF Done bit */
while( !( GPIFTRIG & 0x80 ) )
{
    ;
}
SYNCDELAY;
/* get EP6FIFOBCH/L value */
xFIFOBC_IN = ( ( EP6FIFOBCH << 8 ) + EP6FIFOBCL );
/* if pkt is short*/
if( ( xFIFOBC_IN > 0 ) && ( xFIFOBC_IN < 0x0200 ) )
{
    INPKTEND = 0x06;
}
/* decrement transfer length by Tcount */
len = len - Tcount;
/* if the transfer length is not a modulus of 512, no need to
   reset GPIFADR[8:0] to access next bank of 512 bytes, */
if(!(len % 0x0200))
{
    /* handles full-speed case and high-speed case */

    /* reset GPIFADR[8:0] to access the next bank at offset 0 */
    GPIFADRH = 0x00;
    GPIFADRL = 0x00;
    /* increment the bank address by 1 to access */
    /* next bank of 512*/
    IOA = ( ( ( IOA >> 4 ) + 1 ) << 4 );
}
}
}

EPOBCH = 0;

```

```
        EPOBCL = 1;
        EPOCS |= bmHSNAK;
        break;
    }
    default:
        return(TRUE);
}
return(FALSE);
}
```

Document History

Document Title: AN57322 - Interfacing SRAM with FX2LP over GPIF

Document Number: 001-57322

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2798921	CPPK	11/04/09	New application note.
*A	3086428	CPPK	11/15/10	Updated Introduction (Added a note in the end). Updated GPIF Waveforms (Edited Write Cycle and Read Cycle timing diagrams). Added References.
*B	3817378	CPPK	11/20/2012	Updated to new template. Completing sunset review.
*C	4428591	NIKL	09/08/2014	Updating Document Title to "AN57322 - Interfacing SRAM with FX2LP over GPIF".
*D	5035776	NIKL	01/08/2016	Added a reference to related Application Notes "AN65209, AN15456" in page 1. Added a note on the link to additional ANs on FX2LP in Introduction. Updated all snapshots with better resolution pictures. Updated the section "Testing with CyConsole" (To use "Control Center"). Updated References: Replaced AN6077 with AN63787. Updated to new template. Updated attached Associated Project (To use correct DVK installation paths in the project properties). Completing Sunset Review.
*E	5701534	AESATP12	04/26/2017	Updated logo and copyright.
*F	6060411	HENA	02/07/2018	Added reference to AN66806 in the related application notes. Added the links to 'USB High Speed Code examples' and 'USB 3.0 Product Family' webpages. Corrected typos and added notes for better clarity. Added the links to download the GPIF Designer Utility and Control Center. Added a figure in the 'Designing GPIF Interconnect' section. Corrected the file paths in the 'uVision Project and Firmware' section. Updated the associated project (Added all source files correctly) Updated References

Worldwide Sales and Design Support

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

Products

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

[PSoC 1](#) | [PSoC 3](#) | [PSoC 4](#) | [PSoC 5LP](#) | [PSoC 6 MCU](#)

Cypress Developer Community

[Community](#) | [Projects](#) | [Videos](#) | [Blogs](#) | [Training](#) | [Components](#)

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.



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

© Cypress Semiconductor Corporation, 2009-2018. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. No computing device can be absolutely secure. Therefore, despite security measures implemented in Cypress hardware or software products, Cypress does not assume any liability arising out of any security breach, such as unauthorized access to or use of a Cypress product. In addition, the products described in these materials may contain design defects or errors known as errata which may cause the product to deviate from published specifications. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.