

AN57322

Author: Bhuvaneshwara Rao Nalla, Praveen Kumar C P
Associated Project: Yes
Associated Part Family: CY7C68013A
Software Version: None
Associated Application Notes: None

Application Note Abstract

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.

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.

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 Chapter10 (GPIF) of the [EZ-USB FX2LP Technical Reference Manual](#) is beneficial in designing the waveforms.

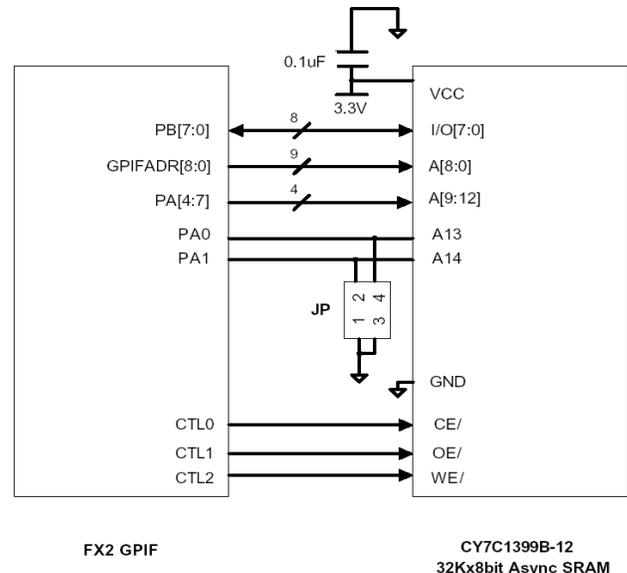
Hardware Connections

This section discusses the required hardware interconnect 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] is wired to A[12:9] and PA[1:0] is wired to A[14:13].

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. PA0 and PA1 are used to access four such 8K byte banks, providing access to the entire 32K of space.

Note Only A[8:0] is shown in this document because the GPIF only has nine address lines.

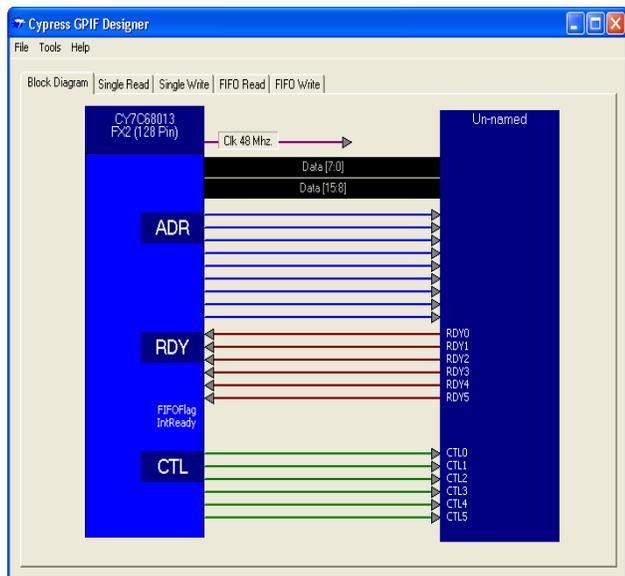
Figure 1. Hardware Connection Diagram



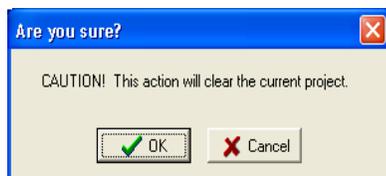
Designing GPIF Interconnect

The GPIF Designer utility is used to create the waveform descriptors to read and write from the SRAM. The following steps demonstrate how to define the interface and create the waveforms.

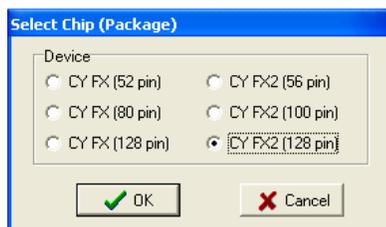
1. Start the GPIF designer tool.



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

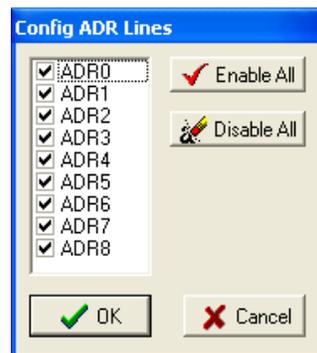


3. In the window that pops up, select the appropriate part and click **OK**.



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 8 bits of the data bus are used.

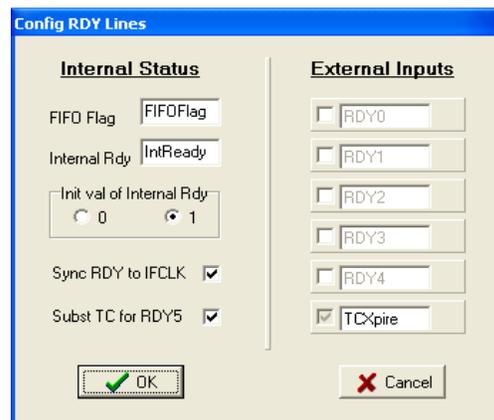
6. Right-click on the ADR trace. The Config ADR Lines dialog box appears. All nine address lines of the GPIF are used and must remain selected. Click **OK**.



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 for any RDY lines to be defined. Clear the checkboxes under "External Inputs". Select the following checkboxes:
 - Sync RDY to IFCLK
 - Subst TC for RDY5

- Sync RDY to IFCLK
- Subst TC for RDY5

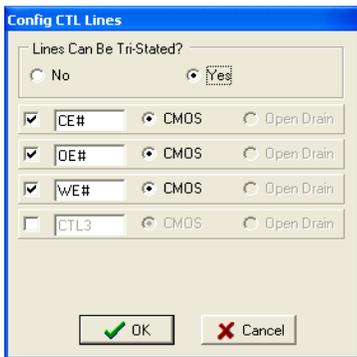
The window appears as follows. Click **OK**



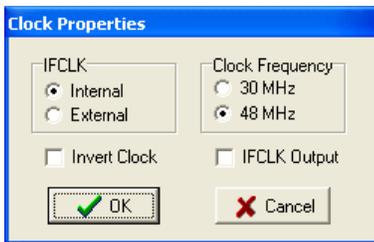
8. Right-click on CTL trace. The Config CTL Lines window is displayed. In this window,

- Select **Yes** for 'Lines can be tristated'.
- Rename CTL [0-2] lines to **CE#**, **OE#**, and **WE#** respectively.
- Uncheck the "unused" label (CTL3).

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



- Right-click on the '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 next step is to design the read and write waveforms.

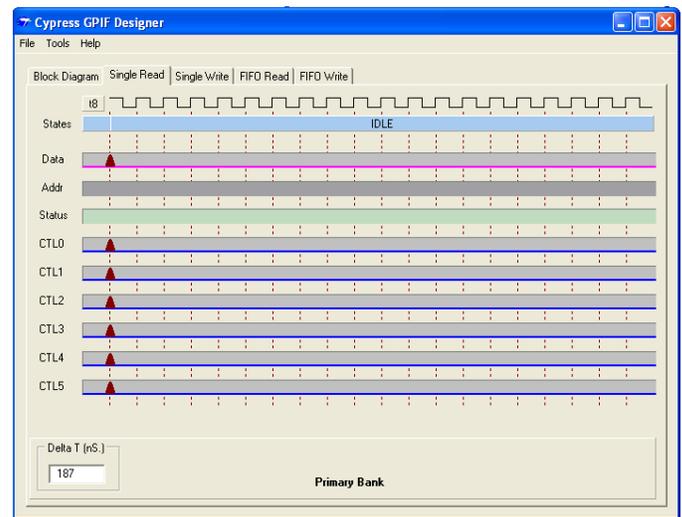
GPIF Waveforms

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

Write Waveform

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

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".



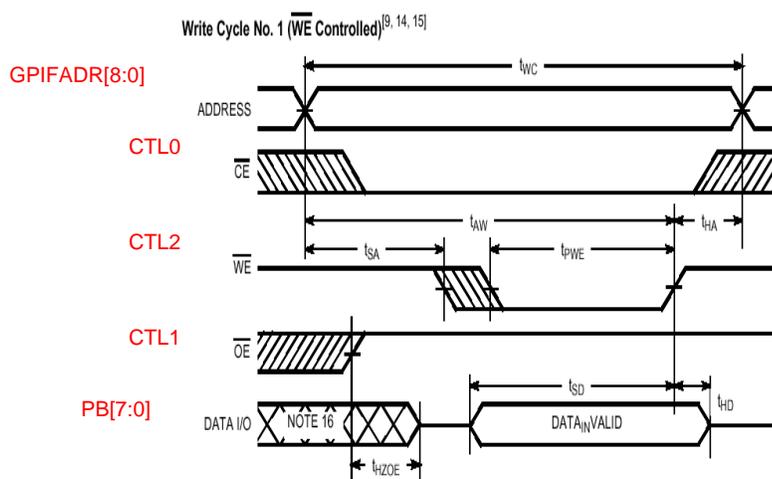
Select **Tools > Map Waveforms to WFSELECT**.



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 GPIFWSELECT register. The waveforms are already mapped appropriately; click **OK**.

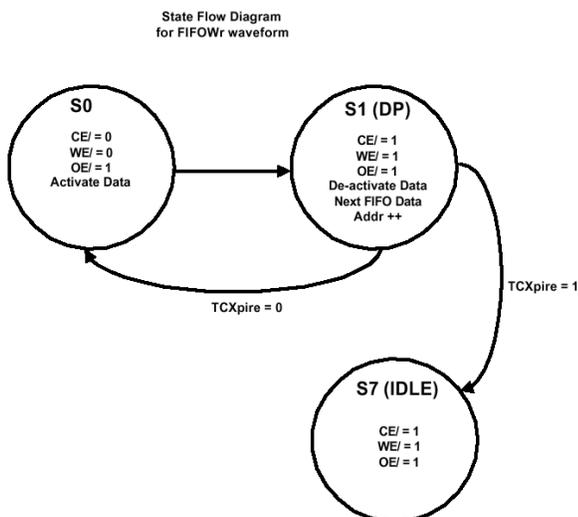
To construct the FIFO Write waveform, first review the write cycle timing for the SRAM and its timing parameters.

Figure 2. Write Cycle Timing for SRAM



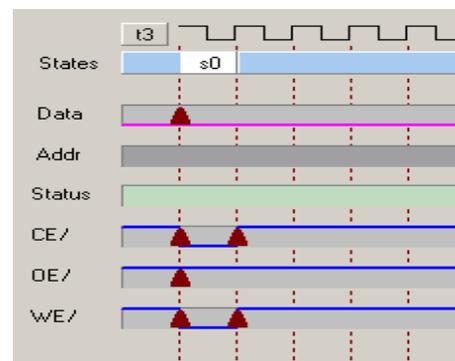
Parameter	Time (ns)	Notes
tWC (Write Cycle Time) (min)	12	When IFCLK=48 MHz, each GPIF cycle is 20.83 ns. Therefore, it only takes one cycle to write a byte.
tPWE (WE/ Pulse Width) (min)	8	When IFCLK=48 MHz, each GPIF cycle is 20.83 ns. Therefore, WE/ only needs to be driven low for one cycle.
tSD (Data Setup to Write End) (min)	7	Driving data together with WE/ LOW meets the setup time easily.
tHA (Address Hold from Write End) (min)	0	It is not required to keep the address asserted after WE/ goes HIGH.
tSA (Address Set-Up to Write Start) (min)	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.
tAW (Address Set-Up to Write End) (min)	8	Because address is asserted for one GPIF cycle and WE/ is de-asserted in the next cycle, this setup time is easily met.
tHD (Data Hold from Write End) (min)	0	It is not required to keep driving data after WE/ is de-asserted.

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

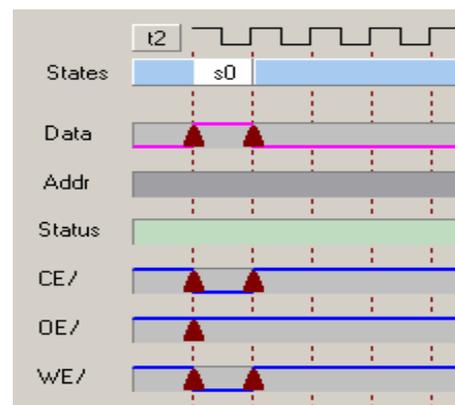


Follow these steps to complete the FIFO Write waveform

1. Click the FIFO Write waveform tab.
2. 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.
3. Assert and de-assert CE/ along with WE/. To do this, click on the CE/ trace one clock cycle from the left boundary.
4. 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.

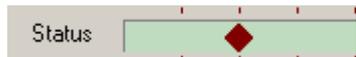


5. The data bus is also driven in s0. To do this, right-click on the data action point, and select **Activate Data**.
6. The data bus should only be driven for one clock cycle. To stop driving the data after one clock cycle, place another action point on the data trace after one clock cycle. Notice that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



7. 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.

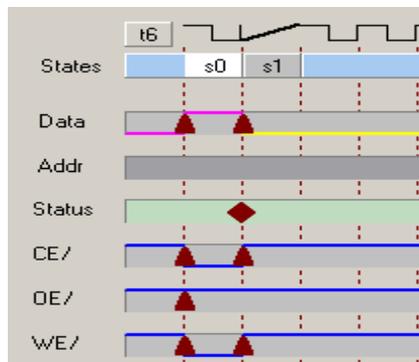
- A DP must be implemented after s0. To do this, set an action point on the Status Trace by clicking at the right boundary of s0.



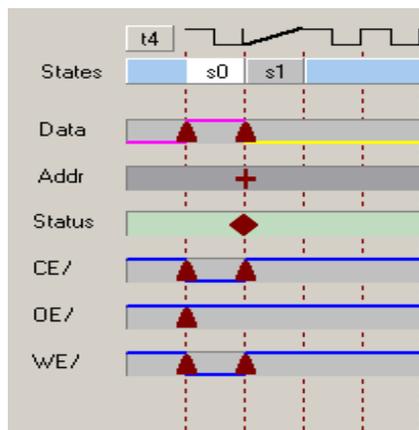
- A dialog box appears prompting you for DP branch conditions. Select the condition as follows.



- 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.
- 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.



- 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:



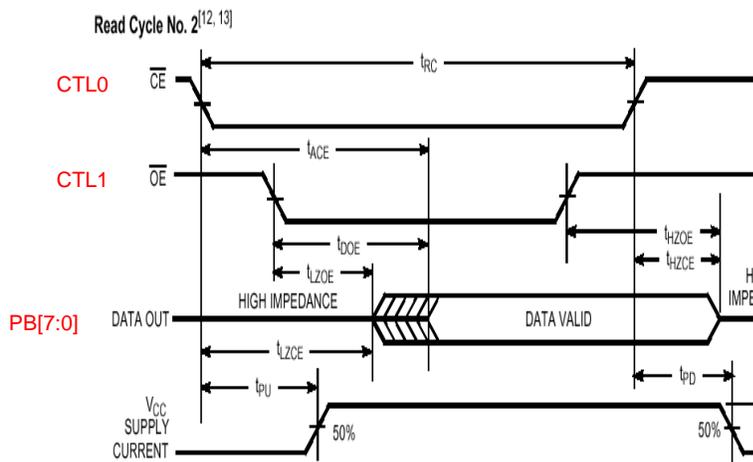
- Save your waveform by selecting **File > Save**.

Read Waveform

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

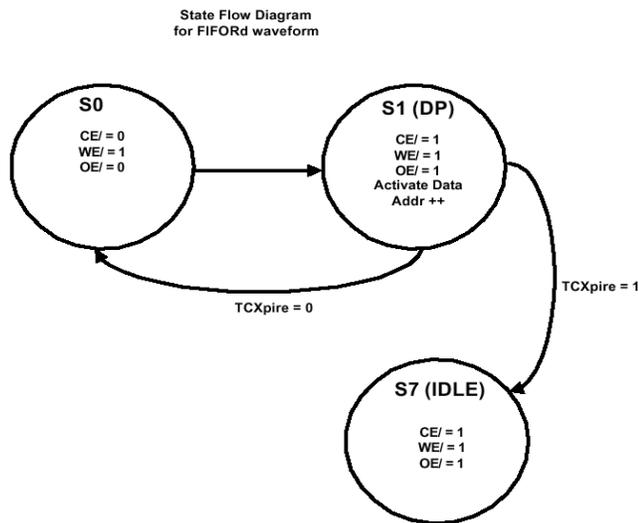
The process is similar to designing the write waveform. First, review the read cycle timing for the SRAM and its timing parameters.

Figure 3. Read Cycle Timing for SRAM



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.

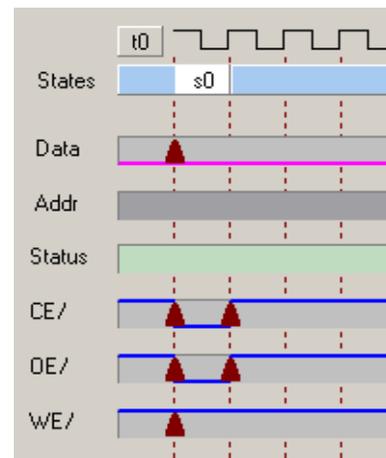
Now that the timing parameters involved are understood, the read waveform can be designed in GPIF designer. The following state flow diagram must be accomplished:



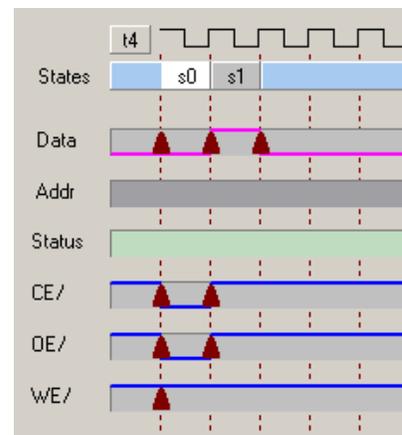
Follow these steps to complete the FIFO Read waveform.

1. Click the FIFO Read tab
2. 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.
3. 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
4. WE/ must be kept HIGH throughout the waveform. From the waveform observe that WE/ is high by default.
5. 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.

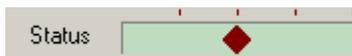
6. The waveform should appear as follows:



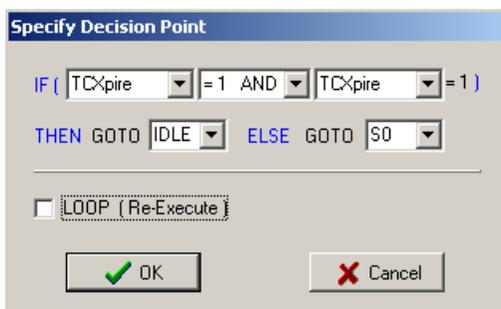
7. The data bus must be sampled one clock cycle after asserting the CE/ to ensure that data is valid before sampling (tACE). To do this, click the data trace on the right boundary of s0. This causes the data trace to toggle HIGH (placing an "Activate Data" event).
8. The data bus should only be sampled 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:



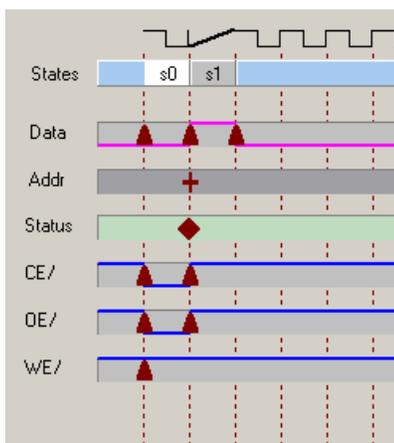
9. Next, add a decision point (DP) state to loop through this waveform until the GPIF transaction count (GPIFTC) expires. To do this, test the internal TCXpire flag in a DP state and only branch to the IDLE state when the decision count expires. In the DP state, increment the GPIFADR lines.
10. A DP 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.



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



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 "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.
13. 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:



14. 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.

Exporting GPIF Waveforms

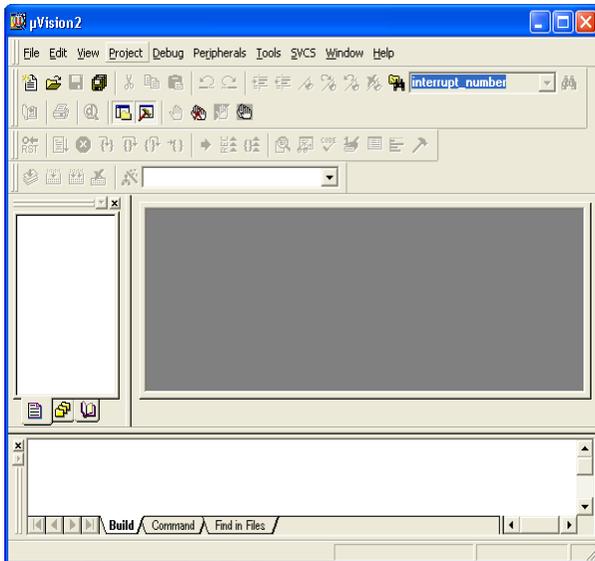
To export the waveforms to a C file and include it in the firmware project, follow these steps:

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

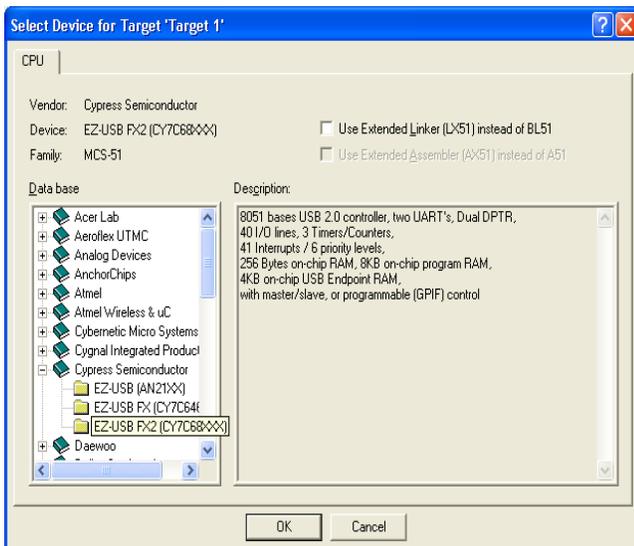
uVision2 Project and Firmware

The next step is to create a project and write firmware to interface the SRAM to the FX2LP device.

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\Examples\FX2LP and name it "SRAM_GPIF"
2. Copy the following project files from C:\Cypress\USB\Examples\FX2LP\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.



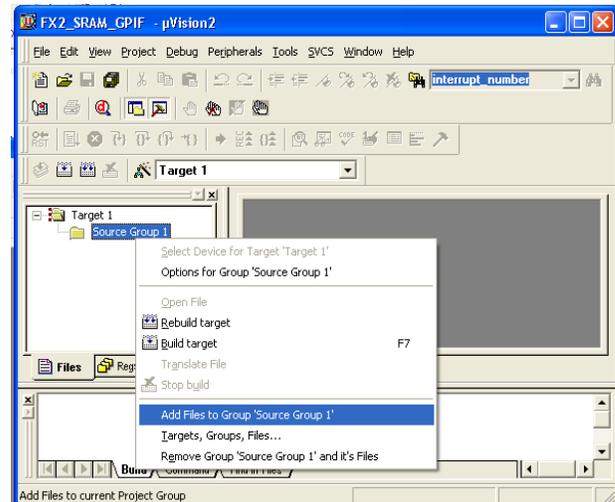
6. Go to **Project > New Project**. The tool prompts you to name the project and save it. Browse to the newly created SRAM_GPIF directory and save the project as FX2_SRAM_GPIF.
7. The following window is displayed prompting to select the type of device. Select **EZ-USB FX2 (CY7C68013)** from the list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



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



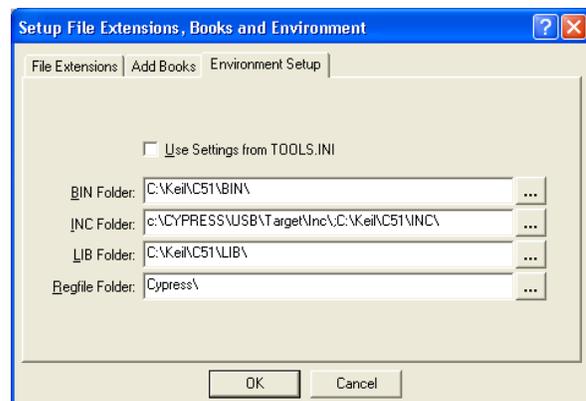
9. The following window is displayed. Add all the relevant source files to the FX2_SRAM_GPIF project. To add files, right click on Source Group1 directory and select **Add Files** to “Source Group1”.



10. The files from the following path are added.

- C:\Cypress\USB\Examples\FX2LP\SRAM_GPIF
 - *gpif.c*
 - *fw.c*
 - *FX2_SRAM_GPIF.c*
 - *dscr.a51*
- C:\Cypress\USB\Target\Lib\LP
 - *USBImpTb.OBJ*
 - *Ezusb.lib*

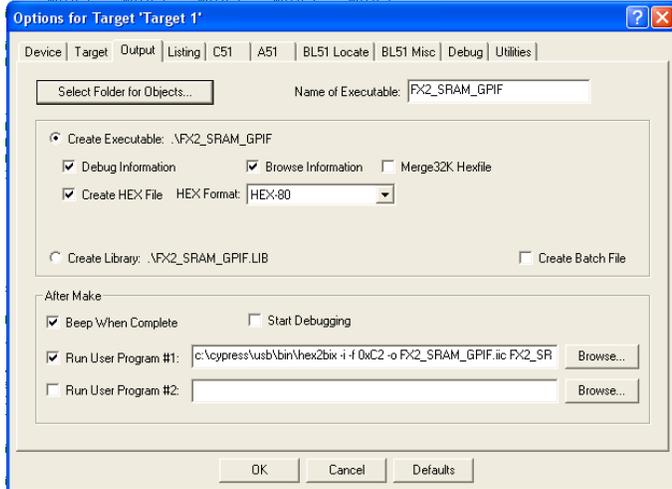
11. To set the Microvision setup environment, select **Project > File Extensions, Books and Environment**.



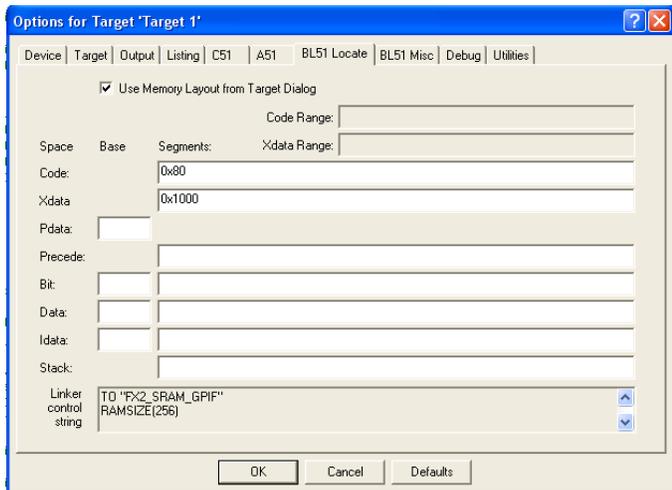
In the Environment Setup tab, edit the fields as follows:

- BIN Folder: C:\Keil\C51\BIN\
- INC Folder: C:\CYPRESS\USB\Target\Inc\C:\Keil\C51\INC\
- LIB Folder: C:\Keil\C51\LIB\

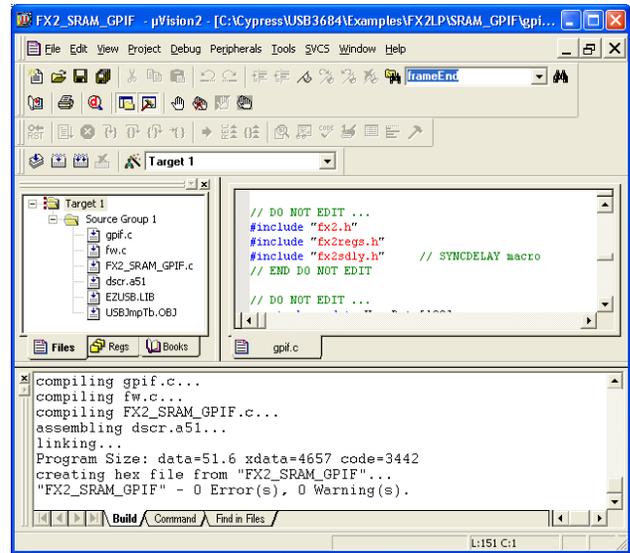
12. Go to **Project > Options for Target 'Target1'**. Select the **Output** tab and check the **Create Hex File** box.



13. Click the **BL51 Locate** tab and locate the code and xdata as shown.



14. Build the project by clicking the **Rebuild all target files** button . Make sure the code build is error-free. Click **OK** for warnings.
15. 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.



Firmware to Interface with SRAM

The following sections of firmware are added to the *FX2_SRAM_GPIF.c* file:

- 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 write 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)

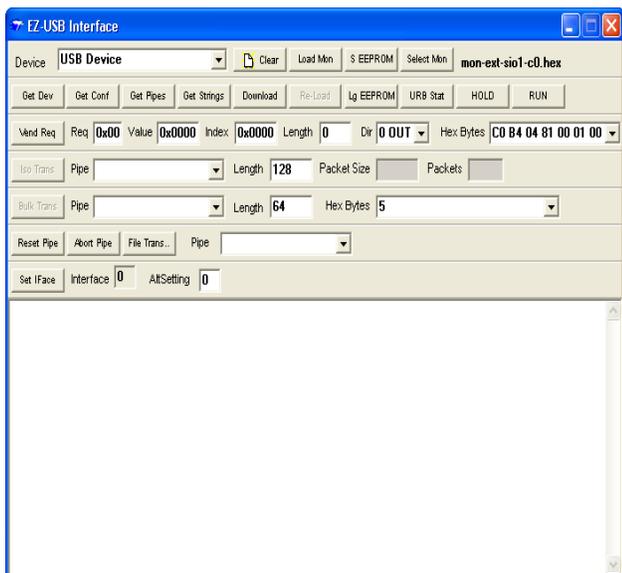
The code for this is provided in [Appendix A](#) of this document. Copy the code to the *FX2_SRAM_GPIF.c* file in the appropriate positions. For example, replace the existing `TD_poll()` with the one in this document. Also copy all the `#defines` and functions related to LEDs.

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.

Testing with CyConsole

The functionality of the project created is verified using the CY3686 Development board and CyConsole 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 EZ-USB FX2LP – EEPROM Missing”, which is seen in the device manager.
3. Start the CyConsole host application. In the Options menu, select **EZ-USB Interface**. The following window is displayed



4. Click the **Download** button and navigate to the newly created *FX2_to_SRAM.hex* file. Wait for the firmware to re-enumerate and reconnect as a Cypress EZ-USB 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 **File Trans** button to transfer a file of known data pattern. Make sure you select **Endpoint 2 OUT** in the drop down box. Click the **File Trans** button and select *1024_count.hex* located in the path C:\Cypress\USB\Util\Test.



6. On the Vend Req transfer bar, the “Req” 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. The “Value” field specifies the SRAM address. The “Index” 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 **Vend Req** 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.

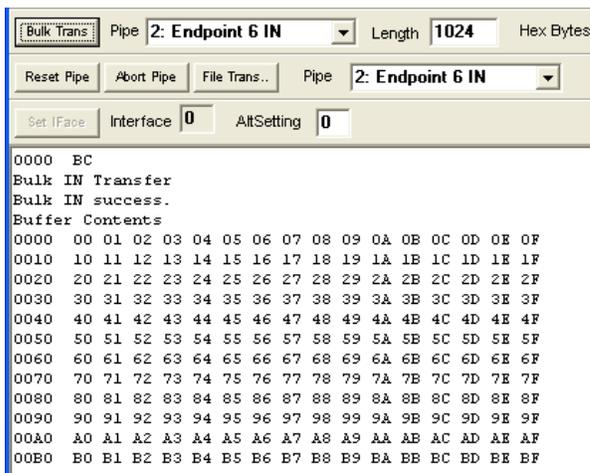


After the request is processed, the output window displays the vendor request value of 0xBC.

To transfer data to the host via EP6IN (the data read from the SRAM) endpoint, use the bulk transfer bar to specify the endpoint to use, data value and request length.



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).

Appendix A

```

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

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

BOOL enum_high_speed = FALSE; // flag
to let firmware know FX2 enumerated at high
speed
static WORD xFIFOBC_IN = 0x0000; //
variable that contains EP6FIFOBCH/L value
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;
    }
}

void TD_Init(void) // Called
once at startup
{
    // 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

    FIFORESET = 0x80; // set NAKALL bit to
NAK all transfers from host
    SYNCDELAY;
    FIFORESET = 0x02; // reset EP2 FIFO
    SYNCDELAY;
    FIFORESET = 0x06; // reset EP6 FIFO
    SYNCDELAY;
}

```

```

FIFORESET = 0x00; // clear NAKALL bit to
resume normal operation
SYNCDELAY;

EP2FIFOCFG = 0x00; // allow core to see
zero to one transition of auto out bit
SYNCDELAY;
EP2FIFOCFG = 0x10; // auto out mode,
disable PKTEND zero length send, byte ops
SYNCDELAY;
EP6FIFOCFG = 0x08; // auto in mode,
disable PKTEND zero length send, byte ops
SYNCDELAY;
// enable dual autopointer feature
AUTOPTRSETUP |= 0x01;

GpifInit (); // initialize GPIF
registers
PORTACFG = 0x00;
OEA |= 0xF3;
IOA &= 0xFC;
Rwuen = TRUE; // Enable
remote-wakeup
}

```

```

BOOL DR_SetConfiguration(void) // Called
when a Set Configuration command is
received

```

```

{
    if( EZUSB_HIGHSPEED( ) )
    {
        // FX2 enumerated at high speed
        SYNCDELAY;
        EP6AUTOINLENH = 0x02; // set
        AUTOIN commit length to 512 bytes
        SYNCDELAY;
        EP6AUTOINLENL = 0x00;
        SYNCDELAY;
        enum_high_speed = TRUE;
    }
    else
    { // FX2 enumerated at full speed
        SYNCDELAY;
        EP6AUTOINLENH = 0x00; // set AUTOIN
        commit length to 64 bytes
        SYNCDELAY;
        EP6AUTOINLENL = 0x40;
        SYNCDELAY;
        enum_high_speed = FALSE;
    }

    Configuration = SETUPDAT[2];
    return(TRUE); // Handled by
user code
}

```

```

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;
    }
}

```

```

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;
        case VX_BB: // actuate write to
SRAM
            {
                EPOBUF[0] = VX_BB;

                IOA = (IOA & 0x0F) + (SETUPDAT[3] <<
3); // select bank of 16x512 (bit shift
MSB of wValue by 3

                // and OR it
                with PA[7:4]
                GPIFADRH = SETUPDAT[3];
                // set GPIFADR[8:0] to address passed down
                in wValue
                GPIFADRL = SETUPDAT[2];

                len = ( (SETUPDAT[5] << 8) +
SETUPDAT[4] ); // get transfer length from
wIndex field

                while (len)
                // while the transfer length is non-zero,
                {

```



```

    GPIFADRH = SETUPDAT[3];
// set GPIFADR[8:0] to address passed down
in wValue
    GPIFADRL = SETUPDAT[2];

    len = ( (SETUPDAT[5] << 8) +
SETUPDAT[4] ); // get transfer length from
wIndex field

    while (len)
// while the transfer length is non-zero,
    {
        if( GPIFTRIG & 0x80 )
// if GPIF interface IDLE
        {
            if( !( EP68FIFOFLGS &
0x01 ) ) // if EP6 FIFO is not
full
            {
                if(enum_high_speed)
// if the FX2 enumerated at high-speed
                {
                    if ( len >
0x0200 ) // if the transfer length
is greater than 512 bytes,
                    {
                        GPIFTCB1 = 0x02; // set GPIF transaction
count to 512, since
//
GPIFADR can only access 512
SYNCDELAY; // locations at a time

= 0x00; GPIFTCB0

SYNCDELAY; Tcount =
0x0200;
                    }
                    else
                    {
                        GPIFTCB1 = MSB(len);

SYNCDELAY; GPIFTCB0
= LSB(len);

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

SYNCDELAY; GPIFTCB0
= LSB(len);

SYNCDELAY; Tcount =
len;
        }
    }
    GPIFTRIG = GPIFTRIGRD |
GPIF_EP6; // launch GPIF FIFO READ
Transaction to EP6IN
    SYNCDELAY;

    while( !( GPIFTRIG &
0x80 ) ) // poll GPIFTRIG.7 GPIF Done bit
    {
        ;
    }
    SYNCDELAY;

    xFIFOBC_IN = ( ( EP6FIFOBCH
<< 8 ) + EP6FIFOBCL ); // get EP6FIFOBCH/L
value

    if( ( xFIFOBC_IN > 0 ) && (
xFIFOBC_IN < 0x0200 ) ) // if pkt is short,
    {
        INPKTEND = 0x06;
// force a commit to the host
    }

    len = len - Tcount;
// decrement transfer length by Tcount

    if(!(len % 0x0200))
// if the transfer length is not a modulus
of 512, no need to
    {
        // reset GPIFADR[8:0] to access next bank
of 512 bytes,
        GPIFADRH = 0x00;
// handles full-speed case and high-speed
case
        GPIFADRL = 0x00;
// reset GPIFADR[8:0] to access the next
bank at offset 0
    }
}

```

```

        IOA = ( ( IOA >> 4 ) + 1
) << 4 ); // increment the bank
address by 1 to access

// next bank of 512
    }

    }

    }

    EPOBCH = 0;
    EPOBCL = 1;
    EPOCS |= bmHSNAK;
    break;
}
default:
return(TRUE);
}
return(FALSE);
}

```

Summary

This application note describes an easy to follow procedure to connect and create waveforms using the GPIF Designer. It demonstrates this procedure using an implementation to connect an SRAM to the FX2LP Development Kit board.

About the Author

Name: Praveen Kumar C P
Title: Applications Engineer
Contact: cppk@Cypress.com

Document History

Document Title: 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.

EZ-USB FX2LP is a trademark of Cypress Semiconductor Corp. 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
 Phone: 408-943-2600
 Fax: 408-943-4730
<http://www.cypress.com/>

© Cypress Semiconductor Corporation, 2009. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.