**User Interface and USB microcontroller system for high-speed control of the Interestim-2B neurological microstimulator array.**

**Phase I**

# Final Report

*Revised to include new UI features by FBM – 8/30/05*

**July 10, 2005**

*Students:*
Frankie Myers, fbmyers@ncsu.edu
Jim Simpson, jasimps3@ncsu.edu
Chris Thomas, cmthomas@ncsu.edu

*Faculty Advisor:*
Dr. Maysam Ghovanloo
NCSU Bionics Group
445 Engineering Graduate Research Centier
2410 Campus Shore Drive
Raleigh, NC 27695-7914
mghovan@ncsu.edu

This work was sponsored by the NCSU 2005 Spring/Summer Undergraduate Research Award Program.

## Abstract

This project focused on the development and verification of a USB microcontroller system and Windows user interface that allows high speed serial data to be sent to the Interestim-2B's wireless interface. The microcontroller we chose, the Cypress CY7C68013 EZ-USB FX2, is capable of High Speed USB data transfer rates of 480Mbit/s. It provides a unique feature called the general-purpose interface (GPIF) which is a micro-coded state machine that runs at 48MHz and automatically places received data from a given endpoint FIFO queue on an output bus and allows us to insert wait states, effectively downclocking the output rate. Our UI reads in a text script file that describes the commands to be sent to the device and represents this file in memory as a linked-list data structure. Data is buffered into an output queue and sent using asynchronous bulk mode USB transfers via the CyAPI library provided by Cypress. Our preliminary testing showed that it is possible to achieve 6MBit/s prolonged continuous data transfers, interleaving data transmission and rebuffering, and 24MBit/s continuous burst transmissions for pre-buffered data.

## Table of Contents

Abstract ........ ii
Table of Contents ........ iii
Introduction ........ 1
System Hardware ........ 1
Microcontroller Selection ........ 1
PCB Design ........ 2
User Interface ........ 2
Windows GUI ........ 3
CInterestimInterface ........ 4
CInterestimUI ........ 4
CInterestimApp ........ 5
Script File Format ........ 5
Nomenclature ........ 5
Header Directives (@ commands) ........ 5
*@triggerwidth n* ........ 6
*@beginscript* ........ 6
Transmission Script Commands ........ 6
Sample .INT Script: ........ 7
Representing Neurological Signals with the Scripting Language ........ 7
File Parsing ........ 8
File Representation Data Structure ........ 8
CLoopList ........ 8
CLoop ........ 8
CLoopPulseNode ........ 8
CPulse ........ 8
CPulseList ........ 8
CPulseListNode ........ 9
Asynchronous Queueing and Transmission ........ 9
Previous Revision ........ 10
Device Driver: CyUSB.sys ........ 10
Signal Pinouts ........ 11
USB Microcontroller Firmware ........ 12
Overview ........ 12
Endpoint Configuration ........ 13
Responding to host requests ........ 13
Configuring, Controlling and Monitoring GPIF ........ 14
FIFO Write Without Clock Division (48MHz) ........ 14
FIFO Write With Clock Division (<48MHz) ........ 15
Keil uVision2 IDE ........ 16
Performance Analysis ........ 17
Throughput versus GPIF Clock Rate and Packet Size ........ 17
Burst transfers ........ 19
Continuous transfers ........ 19
Delays ........ 20

CPU Interruptions ........ 20
Development Goals for Fall 2005/Spring 2006 ........ 21
Ideas For Further Development ........ 21
Known Bugs ........ 22
Conclusions ........ 22
Appendix A: Hardware Keep-alive solution ........ 23

## Introduction

Previous experiments with the Interestim-2B has required the use of a LabView applet and National Instruments DAQ module. Not only is the cost of LabView and the DAQ hardware prohibitively expensive for widespread experimentation among physiologists, but it is difficult to use. In order to promote greater acceptance of the platform within the biomedical community, the need to develop a more streamlined and cost effective PC interface was realized. USB 2.0 was chosen due to its unparalleled speed, familiarity, and ease of use. In keeping with the design motivation of the Interestim-2B itself, the USB interface aims to provide the necessary data throughput for a visual prosthesis application. Given the data format of the Interestim-2B, this was determined to be in the 1-20Mbit/s range. This project successfully demonstrates continuous data transmission at 6Mbits, with data burst rates of any finite duration (provided that the PC has enough memory to buffer the burst) of 24MBits. Given that the length of an Interestim-2B command frame is 19 bits, FSK modulation is performed in software (dividing the throughput by 4), and each biphasic pulse takes around 4 command frames to generate, this translates into approximately 79,000 continuous pulses per second.

## System Hardware

### *Microcontroller Selection*

The Cypress CY7C68013 EZ-USB FX2 microcontroller was chosen because of it's low cost, extensive documentation and sample applications, reputation, C++ APIs, and general-purpose interface (GPIF). In addition, Cypress provides a hardware development kit for this microcontroller, which we used while developing our firmware.

*Fig 1: Cypress development kit used during prototyping*

### *PCB Design*

We designed a 3x3 inch PCB which contains the microcontroller, USB port connection, EEPROM which stores the firmware, DC level shifters to convert the 3.3V logic signals from the microcontroller to 5V logic to control the Interestim-2B (when it is not connected wirelessly), power regulator, and clock oscillator. At the time of this writing, the PCB had been sent off to be fabricated. All of the CPU's data ports are brought out to headers in this PCB, and the PCB could be made smaller if only the essential 5 pins were brought out.

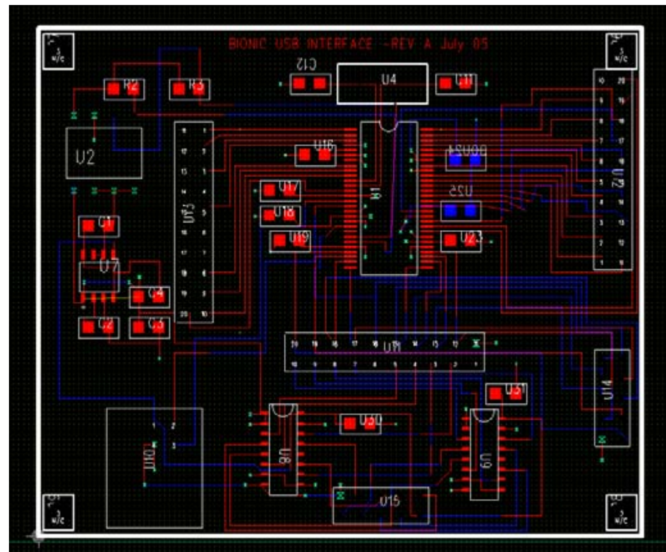



*Fig. 2: PCB Layout*

## User Interface

The UI is a Windows application written in Visual C++ 7.0 (Visual Studio .NET). It uses the CyAPI.lib/.h library provided by Cypress to perform asynchronous bulk mode transfers to the microcontroller. Figure 3 shows a complete UML diagram.

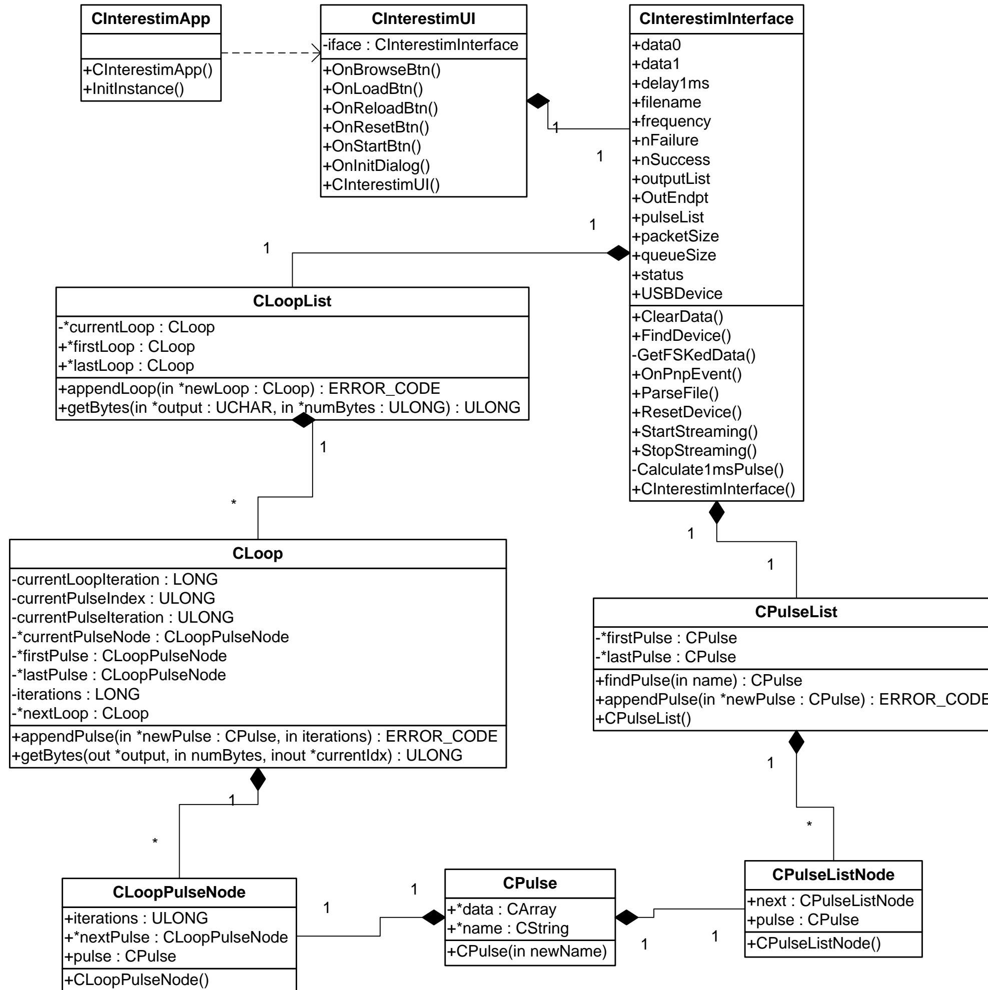



*Fig. 3: UML diagram of UI software*

## ***Windows GUI***

The Windows graphical user interface (GUI) provides a simple, intuitive interface for users to load script files and send data to the probe (Figure 4). The program receives plug-and-play events (PnP) from Windows and when the Interestim-2B is connected it automatically detects it and indicates that it is connected. The user may then load a script file (.int extention) and edit the parameters of the file in the textboxes to the right. Clicking the "start" button initiates data transfers, and the number of successful transfers updates in real time as data is sent out.

Several aspects of this GUI are not yet implemented such as the "browse" button which will display a file browsing dialog window, the message feedback which will show detailed messages about the status of the file parsing and data transmission, and sufficient

error handling. The messaging window will display each loop and the number of bytes contained in that loop. This will aid the user in sending burst transfers where each buffer must completely contain a burst of data.

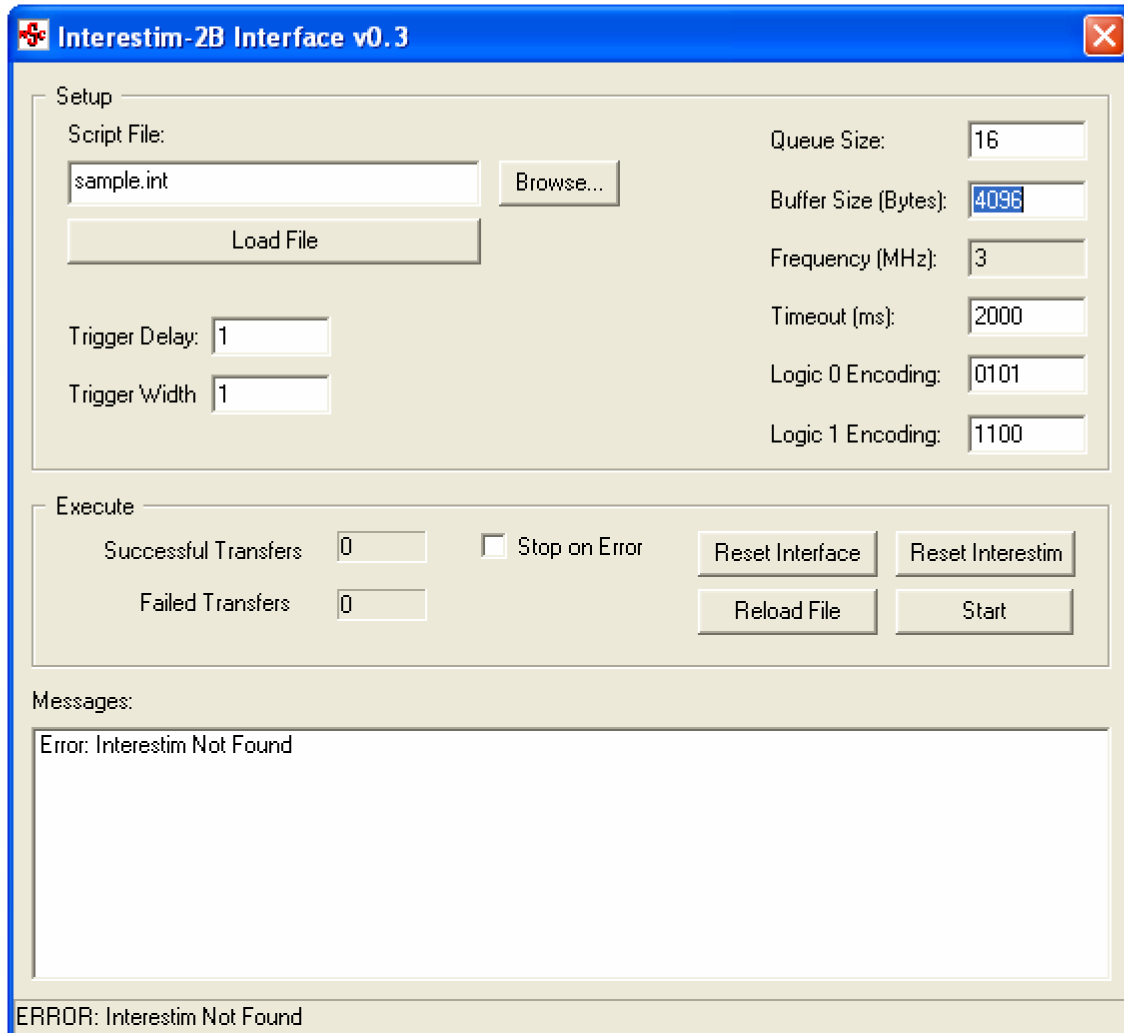



*Fig. 4: Windows GUI*

The UI is composed of three classes:

**CInterestimInterface**

This is an API that encapsulates file parsing and threaded data transmission. It automatically calculates the appropriate byte values to populate the data structure (see below) with based on the data1/data0 directives, and causes the UI to update based on new events via a function pointer to an update method in the UI.

**CInterestimUI**

This contains implementation code for the UI's buttons and input fields. The UI's graphical layout is contained in interestim.rc. This class instantiates an CInterestimInterface, where all functionality is encapsulated.

#### CInterestimApp

This is the top level class that instantiates the UI. It is the main point of entry and it instantiates a CInterestimUI and makes the necessary Windows API calls to create the program's message loop.

## *Script File Format*

The .INT script file format was created to allow very basic test scenarios to be represented. It is not intented, in its current form, for use by physiologists as it requires that commands be entered directly in the format of the Interestim-2B's binary command frame. Additional keywords may be added to allow physiologists to describe pulses and bursts more naturally.

Commands must appear on separate lines, but otherwise spaces, tabs, and newline characters are ignored. Comments are denoted by the hash symbol (#). Lines cannot be longer than 100 characters.

#### Nomenclature

Neurological pulses occur with periods on the order of milliseconds, within bursts with periods on the order of seconds. These pulses generally fit the biphasic pulse model, and they can be generated by the Interestim-2B using a sequence of 19-bit command frames. In our scripting language, a *pulse* is simply a collection of bits, defined with an *@define* directive, to send to the probe as one unit. It may or may not represent a biphasic pulse. It could, for instance, be a resynchronization packet. Pulses may be sent in rapid succession using the *send* command, one after the other, with no added delay, for an integer number of times. A *delay* is a period of time (on the order of milliseconds) where no data is sent, but the FSK clock lines are still toggled. Internally, this is simply represented as a pulse itself. A *loop* is a collection of pulses and/or delays that is sent repeatedly, back to back, either for an integer number or infinitely many iterations.

When we refer to the actual data that is sent, we describe it in terms of *packets*. Each packet is a *buffer* populated with a certain number of bytes (specified in the UI) and inserted into a *queue* for transmission. Placing data in a queue makes the software more resilient to fluctuations in CPU availability because as long as the queue does not completely empty, there remains a constant stream of data ready to be sent without any processing required.

#### Header Directives (@ commands)

The header section of the file contains data that controls the behavior of the software and the definition of data values.

***@data1/0***

These directives set the bitstream representation of a logic 1 or 0. FSK modulation can be setting 1/0 to be represented as a series of pulses—in the case below 1 would be represented as a 24MHz pulse and 0 would be represented by a 12MHz pulse. These definitions must be the same length.

***@frequency***
Sets the frequency of the GPIF output clock (not implemented).

***@timeout***
Sets the timeout value for an asynchronous packet transfer.

***@queuesize***
Sets the number of asynchronous buffers to hold in queue.

***@packetsize***
The size of a transfer buffer.

***@define pulseName***
defines a series of commands that can be sent in batch during a loop process

***@end***
marks the end of a pulse definition

***@triggerdelay n***
+/- integer value indicating the number of milliseconds before the first data is sent that a trigger pulse should occur on the TRIG pin. If this value is positive, n milliseconds of delay are inserted before the first transmission loop to provide the delay between the trigger and the data. If this value is negative, it occurs n milliseconds after the first pulse is started.

***@triggerwidth n***
Positive integer indicating the width of the trigger strobe in milliseconds. The trigger strobe is active high.

***@beginscript***
Marks the end of the header (pulse definitions and all other @ commands) and the beginning of the transmission script.

#### Transmission Script Commands

***loop n***
loops the block n times, if n is 'infinite', the loop goes on forever

***end***
marks the end of a loop block

***send pulseName n***
sends the specified pulse (must be @define'd) n times

***delay n***
sends NO DATA (not even resync packets) for n milliseconds

**Sample .INT Script:**

```
# ------ Header Directives ------

@data1 1010
@data0 1100
@frequency 48
@timeout 100
@queuesize 100
@packetsize 8192
@triggerdelay 10
@triggerwidth 1

@define pulse1
        1111111111111111111
        0000000000000000000
        1010101010101010101
        0101010101010101010
@end

@define pulse2
        1100101011
@end

@define pulse3
        1010
@end


# ------ Transmission script begins here ------
@beginscript

loop 20
        send pulse1 5
        send pulse2 12
        delay 3
end

delay 10

loop infinite
        send pulse1 10
end

# ------ End of Script ------
```

#### Representing Neurological Signals with the Scripting Language

Consider the case where a pulse which is defined with the @define directive contains the commands necessary to generate a biphasic pulse between two stimulating sites on the Interestim-2B. A loop containing this pulse and some delay might represent a standard neurological burst. Several loops could be added to the file to produce the desired number of bursts. This scheme, however, would not allow for a burst to loop indefinitely. Alternatively, a pulse may contain within its definition the appropriate amount of delay before the next pulse, allowing a single *send* command to send a series of pulses at the desired frequency (a burst), and a single loop to send repeated bursts. Since this scripting language was developed for testing the system, this two-level looping scheme is sufficient. However, we plan on adding additional commands that more closely resemble the physiological signals and their associated parameters. An @*pulse* command, for instance, might be provided to "design" a biphasic pulse, and an @*burst* command might describe a collection of these pulses and their pulse frequency. The *send*

command might provide an additional parameter that selects the stimulating sites to transmit the pulse across.

### *File Parsing*

The UI reads in .INT files and parses them into a simple linked-list based data structure. This data structure allows for 2-level looping and delays. Lines are read in one at a time and tokenized. The first token of a line is assumed to be a command unless the IN_DEFINE flag has been set, in which case it may either be an @end or a string of 1's and 0's. Error conditions are returned if an unexpected token is encountered. The IN_LOOP flag is set when a loop has been opened, and all send/delay commands for that loop will be place inside it until an 'end' command is encountered.

### *File Representation Data Structure*

The following is a brief description of each class in the file representation data structure (also, observe code comments for more detailed info). See figure 3 for a UML relationship diagram of these classes.

#### CLoopList

This is the top level class. It represents the entire file. It is a linked list of CLoops. It provides a method, getBytes(), which returns a specified number of bytes from the data structure. It handles iterating through each loop and pulse for the required number of times, and remembers where it last left off from the last transfer. The loop list also remembers the start and stop locations that the trigger should be raised and lowered. It passes these values down to the current loop and maintains a static byte count variable that increments during transmission. The current loop takes the start, stop, and byte count values and determines, on the fly, whether the trigger line should be high or low.

#### CLoop

This represents a single loop in the input file. It maintains a linked-list of LoopPulseNode's. It also contains a reference to the next loop in the file (or NULL if this is the last loop). The loop class also handles dynamic triggering in its getBytes function.

#### CLoopPulseNode

This is a simple linked-list node class that contains a reference to a pulse and a number of iterations, as well as a reference to the next node in the list (NULL if this is the last node in the list)

#### CPulse

This contains the name and the data for a pulse that has been @define'd in the input file.

#### CPulseList

When the file is read in, each new pulse that is @define'd is instantated and placed in a CPulseList. When the script portion of the file is read in, the pulses are looked up using their names with this list (it is similar to a Dictionary in this regard).

### CPulseListNode

This is a single linked-list node in the CPulseList.

## *Asynchronous Queueing and Transmission*

According to Cypress and our USB handbook, asynchronous bulk-mode transfers on an otherwise idle High Speed (USB 2.0) bus should result in 98% bus utilization, or approximately 54Mbytes/sec. Since this is greater than 48 (our output rate), we should see no gaps at all, and the timeout value on the asynchronous transfers should cause the UI to wait on the device to finish (as it is, the device is finishing transfers faster than the UI is supplying them). However, we were unable to achieve a 48Mbit continuous data stream with this software (see figure 5), but we were able to achieve a 24Mbit continuous data stream, provided the data we're sending is repetitive (i.e. it can be placed in a single buffer and then resent continuously without modifying that buffer). When the data is not repetitive, the software overhead of fetching each new buffer causes a delay in the output stream, and our maximum continuous data throughput drops to 6Mbit. The UI and script have two parameters which control asynchronous queuing and transmission, Queue Size and Buffer Size. Queue Size specifies the number of asynchronous buffers to queue up simultaneously, and Buffer Size is the size in bytes of each data packet. The Begin/Wait/End Xfer methods of the CyAPI library were used to queue up data transmissions.

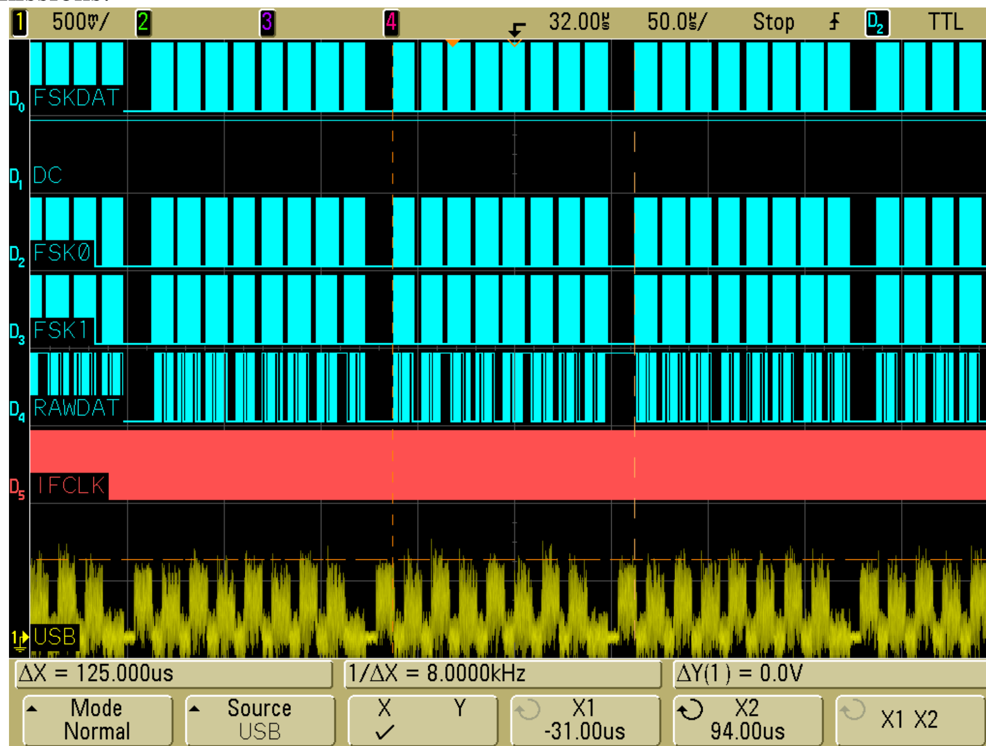



*Fig 5: Even with a repeating buffer, where no overhead is required in fetching new data, and the output buffer is completely loaded from the data structure, using 48Mbit/s GPIF output does not yield a constant data stream*

### *Previous Revision*

The software described above is the second incarnation of the UI. The first was a modified version of the CyAPI BulkLoop sample application (see figure 6).

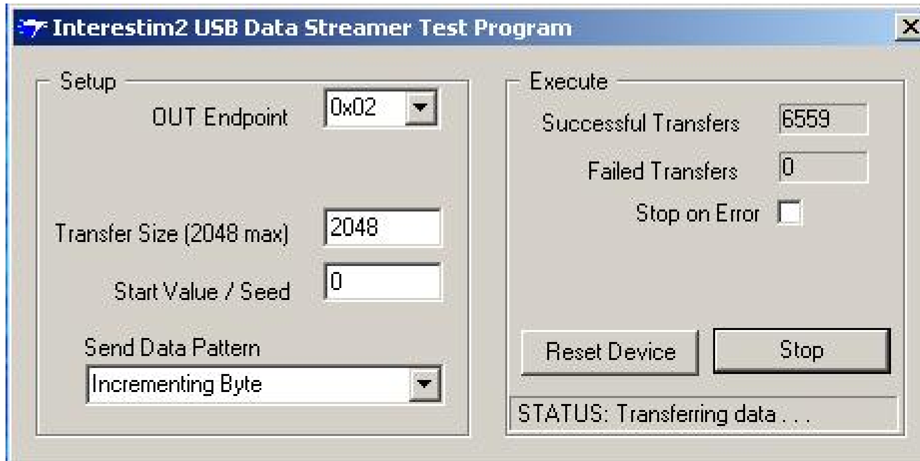



*Fig. 6: First software revision, based on BulkLoop sample application*

This software read in a script similar to the one used by our second revision, but instead of populating an in-memory data structure and sending data asynchronously from a queue, it read the data line by line and sent it as it was read. It populated a single buffer, and when that buffer became full (or until the @flush command was encountered), it would send the contents of the buffer using the blocking XferData function of CyAPI. The drawback of this approach is that it is not resilient to delays caused by other software executing on the system, and requires a great deal of processor overhead to send data as the output data bytes are calculated on-the-fly. A copy of this software is included. A sample .int file for this revision follows:

## Device Driver: CyUSB.sys

We used the CyUSB.sys generic Windows device driver that Cypress provides, along with the C++ CyAPI.lib/.h to interface with it from C++. By configuring the vendor/product ID in the descriptor tables in the firmware to be the same as those in the .INF file of the driver, we can ensure that the correct driver is loaded when the device is plugged in (enumeration).

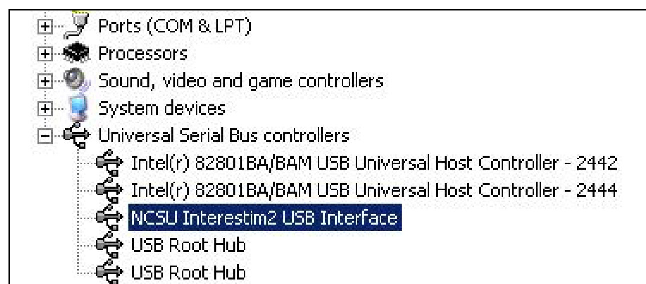



*Fig. 7: Shows enumeration in Windows device manager*

## Signal Pinouts

The following shows the order of the data pinouts. These are brought to the lower 8-bits of the GPIF output bus.

MSB LSB

| 0 | RESET | TRIG | RAWDAT | FSK0 | FSK1 | DC | FSKDAT |
|---|---|---|---|---|---|---|---|

***RESET***
Active-low signal to reset the Interestim chip. When the user presses the "Reset Interestim" button on the GUI, this line is temporarily lowered.

***TRIG***
General-purpose, active-high pulse for synchronizing to other devices (such as a camera). Software handles delay and pulse width parameters.

***RAWDAT***
Data that has not been FSKed, but extends the full duration of the FSK pulse

***FSK0***
Continuous transmission of the FSK0 parameter (can be used for clocking)

***FSK1***
Continuous transmission of the FSK1 parameter (can be used for clocking)

***DC***
Goes high when data is being transmitted, low during delays

***FSKDAT***
Data that has been FSKed

## USB Microcontroller Firmware

### *Overview*

The two main functions of code on the FX2 USB microcontroller are:

1) To identify USB peripheral (itself) to host computer upon enumeration (connection). This is referred to as Descriptor tables and contains information like Vendor ID (VID), Product ID (PID), etc.

2) To run code to transfer data coming into the USB peripheral to devices interfaced with it. This is referred to as firmware.

The FX2 is different from most other USB controllers in the sense that its configuration is "soft". Firmware is stored in its internal RAM, which can be loaded from an external EEPROM or from the host itself. If an EEPROM is present, the FX2 automatically loads the firmware and descriptor tables from the EEPROM onto the FX's on-chip RAM. In the absence of an external EEPROM, the FX2 is capable of enumerating as a Cypress USB device without firmware, and download firmware and descriptor tables from the host into the FX2's on-chip RAM. It then electrically simulates a physical disconnect/re-connect and causes the FX2 to renumerate as the second device. Once the enumeration process is completed as specified in the descriptor tables, the firmware starts running.

USB data enters and exits the FX2 through endpoint buffers. Incoming data can be external logic (device) two different ways:

1) The firmware can access the endpoint buffers either as blocks of RAM or as a "FIFO" using special auto-incrementing pointers. However, by using the CPU to process data, the rate of transfer is limited by the CPU clock which can run at 12, 24 or 48 MHz to process each individual instruction.

2) External logic can read incoming data by direct connection to the endpoint FIFO's without any participation by the FX2's CPU (AUTOOUT mode). There are two different ways the FX2 can be setup to handle such transactions:

    a. The FX2's General Programmable Interface (GPIF) can act as an internal master providing data as well as control signals to external logic.
    b. The FX2 can be in "slave FIFO" mode enabling it to be controlled by an external master which contains a standard FIFO interface.

In the absence of an external device capable of controlling the FX2's endpoint FIFO's, the GPIF AUTOOUT mode is the most logical choice for high speed transfer of data using the FX2 and it is the one we chose. A high level view of such a transfer can be illustrated as:

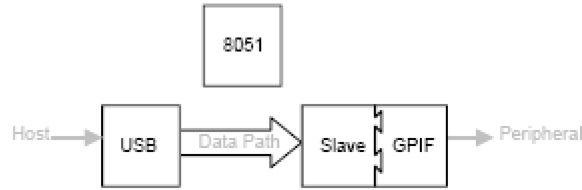



*Figure 8: GPIF acts as internal master keeping CPU out of the data path.*

Although absent from the data path, the firmware still needs to:

- Configure the endpoints.
- Respond to host requests on endpoint 0.
- Configure, control and monitor GPIF.

### *Endpoint Configuration*

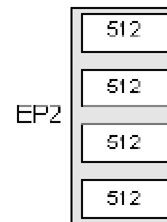
The FX2 offers two small endpoints, EP0 and EP1 for control packets and four large endpoints, EP2, EP4, EP6 and EP8 to be used for bulk, interrupt or isochronous transfers. The maximum packet sizes available for each of these endpoints are:

| Transfer Type | Max Packet Size | |
|---|---|---|
| | USB 1.1 | USB 2.0 |
| CONTROL (EP0 only) | 8,16,32,64 | 64 |
| BULK | 8,16,32,64 | 512 |
| INTERRUPT | 1-64 | 1-1024 |
| ISOCHRONOUS | 1-1023 | 1-1024 |

*Table 1: Max. Sizes for EP buffers*

However, they can be configured various ways to suit requirements. Since data is only sent out to the USB peripheral, EP2 was configured as an out endpoint with the following parameters:

- Receives BULK transfer from the host
- Accommodates a maximum of 512 bytes (max available).
- Allows quad-buffering.
  Double buffering means that one packet of data can be filling or emptying with USB data while another packet (from the same endpoint) is being serviced by external interface logic. Quad buffering adds an extra buffer to either side.




### *Responding to host requests*

Control transfers sent to the FX2 arrive in CONTROL packets to EP0. The arrival of such a packet triggers the interrupts SUTOK and SUDAV. When such an interrupt is asserted, setup data is available in the SETUPDAT register which needs to be immediately serviced.

During enumeration, a query from the host for descriptor table information is received as a request through this endpoint. The three types of descriptor information that needs to be provided are device, configuration and string data.

In the future, a "Set Configuration" request could be sent from the host to change parameters on the FX2 like GPIF speed, etc.

### *Configuring, Controlling and Monitoring GPIF*

The GPIF is a programmable state machine that transfers data from the endpoint buffers to external logic/device through its FD port. Figure 9 shows a block diagram of the FX2 highlighting the path taken by data from the USB transceivers to the FD pins on the chip.

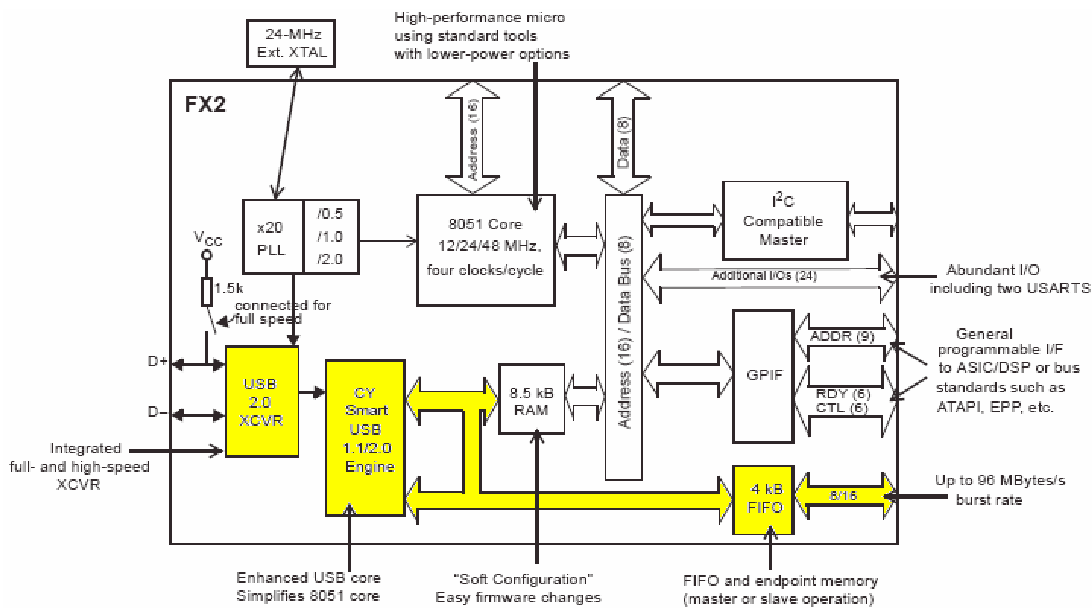



*Figure 9: Data path using GPIF internal master mode*

The GPIF state machine waveforms generate the control signals and specify how data moves from the FIFO to the pins. The waveforms can be easily generated using the GPIF Tool utility.
Since only a single type of transaction was required, only one waveform was defined for our application. However, to enable the GPIF to operate at different clock speeds, a series of GPIF waveform descriptor files were defined to achieve this. The state machines for these waveforms are:

#### FIFO Write Without Clock Division (48MHz)

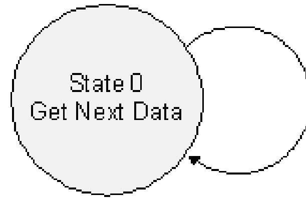



*Fig 10: GPIF State Transition Diagram @ 48MHz*

As shown in the state machine above, each clock cycle sends a new data (byte) onto the output pins (FD [0:8]). This corresponds to a bandwidth of 48Mhz. The GPIF Tool waveform definition for the above state machine corresponds to:

```
// GPIF Waveform 3: FIFO Wri
//
// Interval     0          1          2          3          4          5          6       Idle (7)
//          _________  _________  _________  _________  _________  _________  _________  _________
//
// AddrMode Same Val   Same Val   Same Val   Same Val   Same Val   Same Val   Same Val
// DataMode Activate   Activate   Activate   Activate   Activate   Activate   Activate
// NextData NextData   NextData   NextData   NextData   NextData   SameData   SameData
// Int Trig No Int     No Int     No Int     No Int     No Int     No Int     No Int
// IF/Wait  IF         Wait 1     Wait 1     Wait 1     Wait 1     Wait 1     Wait 1
//   Term A IntReady
//   LFunc  AND
//   Term B IntReady
// Branch1  Then 0
// Branch0  Else 0
// Re-Exec  Yes
// Sngl/CRC Default    Default    Default    Default    Default    Default    Default
// Clock        0          0          1          0          1          1          1          1
// CTL1         0          0          0          0          0          0          0          0
// CTL2         0          0          0          0          0          0          0          1
// CTL3         0          0          0          0          0          0          0          0
// CTL4         0          0          0          0          0          0          0          0
// CTL5         0          0          0          0          0          0          0          0
//
```

*Fig 11: GPIF State Transition Descriptor Table @ 48MHz*

**FIFO Write With Clock Division (<48MHz)**

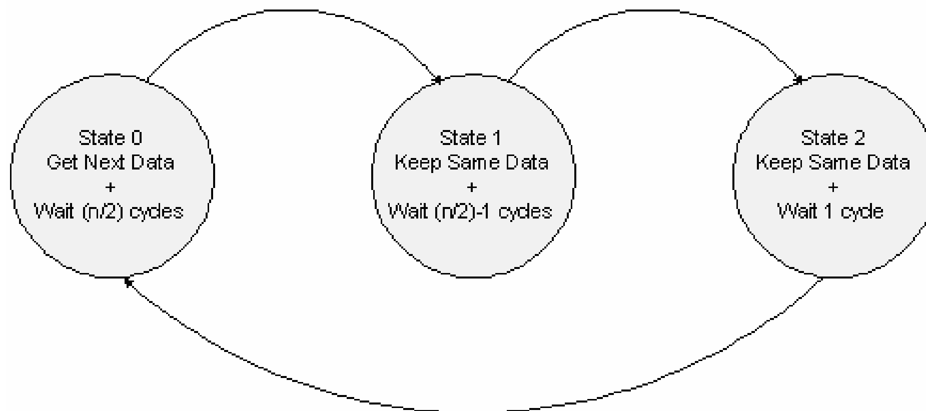



*Fig 12: GPIF State Transition Diagram @ < 48MHz w/ Wait States*

To achieve an output data rate below 48 MHz, the GPIF can be slowed down by following the division method as shown in the state machine above. To divide the clock by n, State 0 outputs new data for half the delayed time and States 1 and 2 outputs the previous data for half the delayed time.

For example, to achieve an output rate of 6 MHz, n = 48/6 = 8. Hence State 0 should last 4 GPIF clock cycles, State 1 should last 3 clock cycles and State 2 should last 1 clock cycle. The GPIF Tool inputs corresponding to these values is:

```
// GPIF Waveform 3: FIFO Wri
//
// Interval     0         1         2         3         4         5         6     Idle (7)
//          _________ _________ _________ _________ _________ _________ _________ _________
//
// AddrMode Same Val  Same Val  Same Val  Same Val  Same Val  Same Val  Same Val
// DataMode Activate  Activate  Activate  Activate  Activate  Activate  Activate
// NextData NextData  SameData  SameData  NextData  NextData  SameData  SameData
// Int Trig No Int    No Int    No Int    No Int    No Int    No Int    No Int
// IF/Wait  Wait 4  IFWait 3  WaIF      WaWait 1  WaWait 1  WaWait 1  WaWait 1
//   Term A                     FIFOFlag
//   LFunc                      AND
//   Term B                     FIFOFlag
// Branch1                      Then 2
// Branch0                      Else 0
// Re-Exec                      Yes
// Sngl/CRC Default   Default   Default   Default   Default   Default   Default
// Clock        1         0         0         0         0         0         0         0
// CTL1         0         0         0         0         0         0         0         0
// CTL2         0         0         0         0         0         0         0         0
// CTL3         0         0         0         0         0         0         0         0
// CTL4         0         0         0         0         0         0         0         0
// CTL5         0         0         0         0         0         0         0         0
//
```

*Fig 13: GPIF State Transition Descriptor Table @ <48MHz*

### *Keil uVision2 IDE*

The Keil uVision2 IDE is an excellent environment to compile, build and debug a project containing these files. It can connect through the serial port of the FX2 development board to step through and debug the code. In order to properly compile, the project requires the files as listed in the figure below as well as the include libraries fx2.h, fx2regs.h, and fx2sdly.h from the Cypress include folder.

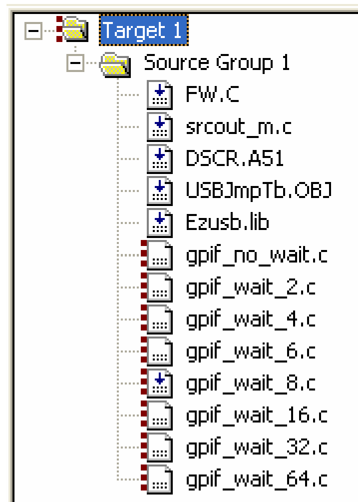



*Figure 14: Keil project window*

## Performance Analysis

### *Throughput versus GPIF Clock Rate and Packet Size*

Even without CPU intervention, the output of the GPIF is bursty at 48 MHz. This is why we have multiple GPIF waveforms for different clock rates. An analysis of the different divided clock rates on a Pentium 4 2.8GHz PC with 512MB ram and USB 2.0 is given below. All transfers were made with the same infinite large looping data set sent using asynchronous buffers.

| Filename | GPIF Clock | Buffer Size | Period | Re-buffering time | Live time |
|---|---|---|---|---|---|
| .c | (in MHz) | (in bytes) | (in s) | (in s) | (in s) |
| | 48 | 8192 | 3.68E-03 | 3.42E-03 | 2.60E-04 |
| | | 16384 | 5.30E-03 | 4.80E-03 | 5.00E-04 |
| | | 32768 | 8.64E-03 | 7.68E-03 | 9.60E-04 |
| | | 65536 | 1.26E-02 | 1.07E-02 | 1.90E-03 |
| | | 131072 | 2.20E-02 | 1.84E-02 | 3.60E-03 |
| | | 262144 | 3.64E-02 | 2.84E-02 | 8.00E-03 |
| | | 524288 | 7.16E-02 | 5.56E-02 | 1.60E-02 |
| gpif_wait_2 | 24 | 8192 | 2.64E-03 | 2.30E-03 | 3.40E-04 |
| | | 16384 | 5.32E-03 | 4.64E-03 | 6.80E-04 |
| | | 32768 | 8.68E-03 | 7.32E-03 | 1.36E-03 |
| | | 65536 | 1.05E-02 | 7.78E-03 | 2.72E-03 |
| | | 131072 | 1.90E-02 | 1.36E-02 | 5.40E-03 |
| | | 262144 | 3.68E-02 | 2.58E-02 | 1.10E-02 |
| | | 524288 | 7.16E-02 | 4.96E-02 | 2.20E-02 |
| gpif_wait_4 | 12 | 8192 | 3.78E-03 | 3.08E-03 | 7.00E-04 |
| | | 16384 | 5.72E-03 | 4.32E-03 | 1.40E-03 |
| | | 32768 | 8.84E-03 | 6.08E-03 | 2.76E-03 |
| | | 65536 | 1.37E-02 | 8.20E-03 | 5.50E-03 |
| | | 131072 | 1.93E-02 | 8.40E-03 | 1.09E-02 |
| | | 262144 | 3.68E-02 | 1.48E-02 | 2.20E-02 |
| | | 524288 | 7.47E-02 | 3.12E-02 | 4.35E-02 |
| gpif_wait_8 | 6 | 8192 | 3.72E-03 | 2.34E-03 | 1.38E-03 |
| | | 16384 | 5.48E-03 | 2.76E-03 | 2.72E-03 |
| | | 32768 | 8.64E-03 | 3.20E-03 | 5.44E-03 |
| | | 65536 | Beyond here continuous but OS interruptable | | |
| | | 131072 | | | |
| | | 262144 | | | |
| | | 524288 | | | |
| gpif_wait_16 | 3 | 8192 | | | |
| | | 16384 | | | |
| | | 32768 | | | |
| | | 65536 | | | |
| | | 131072 | Beyond here continuous and not easily OS interruptable | | |

| | | | | | |
|---|---|---|---|---|---|
| | | 262144 | | | |
| | | 524288 | | | |
| gpif_wait_32 | 1.5 | 8192 | | | |
| | | 16384 | | | |
| | | 32768 | Beyond here continuous and not easily OS interruptable | | |
| | | 65536 | | | |
| | | 131072 | | | |
| | | 262144 | | | |
| | | 524288 | | | |
| gpif_wait_64 | 0.75 | 8192 | Beyond here continuous and not easily OS interruptable | | |
| | | 16384 | | | |
| | | 32768 | | | |
| | | 65536 | | | |
| | | 131072 | | | |
| | | 262144 | | | |
| | | 524288 | | | |

*Table 2: Throughput Experiment Results w/ Various Packet Sizes, GPIF Rates*

Values for each of these parameters at a buffer size of a 32768 bytes was plotted to obtain the following chart (figure 15). Continuous data is obtained where live time meets period. It can be observed that, for a buffer size of 32 768 bytes, continuous data for an infinitely large stream can be obtained when GPIF output is at a little less than 6 MHz.

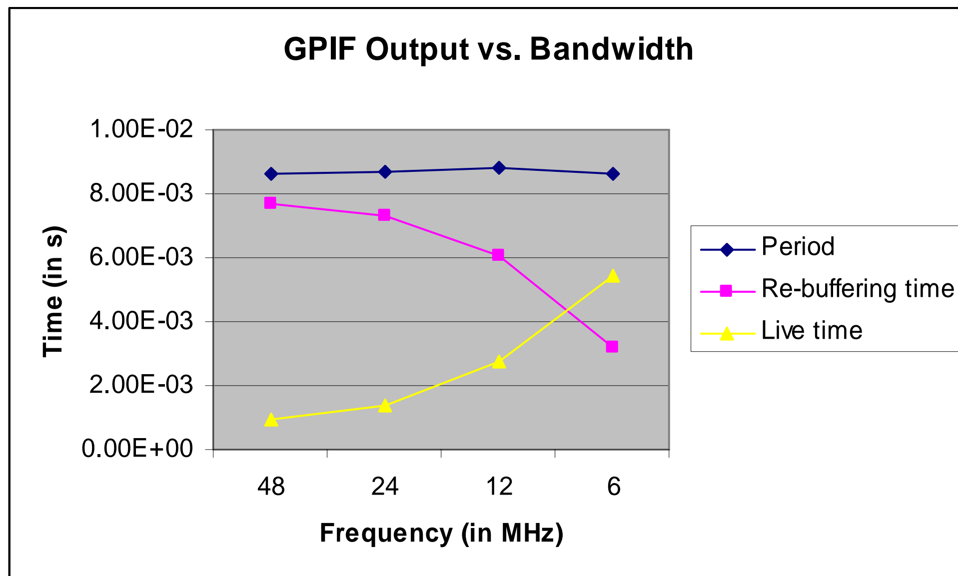



*Figure 15: Infinitely large data stream at buffer size = 32768 bytes*

In addition to generating the waveforms, the GPIF Tool can also be used to configure all the registers associated with using GPIF. The default settings for these registers are sufficient for a standard FIFO write as long as the GPIFTRIG register is set soon after enumeration. The output file generated from GPIF Tool is a standard C file which can be included with the rest of the firmware C code.

### *Burst transfers*

The software can send a burst of data at any data rate up to 48MHz provided that the computer has enough memory to buffer the entire burst and has enough setup time between bursts to populate this buffer. For example, to send continuous data for 1 second at 48Mhz, the computer would require 48Mbytes of memory. The user would specify a 48*2^10 or 50331648 byte packet size (queue size would be 1). When the software started, it would read the input .INT file and cache 48Mbytes of data into memory before sending. The recycle time, or the time it takes to build a new buffer, increases linearly with the number of bytes per packet (O(n)). While it may be possible to decrease this overhead to nearly O(1) with a static file by analyzing the file more thoroughly and automatically synchronizing the output buffers with the loops, this would only work for certain scenarios where the total loop size does not exceed the memory capacity of the PC. In addition, this kind of performance would never be possible with a real-time system such as one that read data from a camera, since in this case, data is not static.

Since real-time operation at 48MHz is the ultimate goal of this project, attention needs to be paid to how this can be achieved. Real-time operation at clock speeds up to 48MHz can be achieved with this system provided that maximum data burst size is bounded, and some amount of recycle time is acceptable between bursts. Recycle time could be spent with the output lines completely dead or, with some additional hardware, could be spent sending a keep-alive command to the probe. In the first case, each burst would begin with a resynchronization packet after a certain amount of delay has elapsed. In the second case, such resynchronization would be unnecessary because the probe would have never lost synchronization, and the new packet could just begin transmitting. In either case, the recycle time must not require CPU overhead, as it does currently, so that the CPU can focus on preparing the next data packet. The data burst size could be a fixed amount (i.e. the size of an image frame in a visual prosthesis application), and the period of these bursts could be fixed as well (i.e. the frame rate of the video stream).

### *Continuous transfers*

Continuous transfers—ones that end with an infinite loop or contain too much data to be wholly contained in a single buffer—are possible with this software and have been demonstrated at 6MHz (using a clock divide ratio of 8, see figure 16) on a Pentium 4, 2.8GHz with 512MB ram using USB 2.0. In other words, this is the minimum speed at which we've been able to successfully interleave transmission and rebuffering code. While the USB hardware and firmware is capable of continuously streaming at 48MHz, there is a bottleneck in the UI software at speeds greater than 6MHz because of the overhead in reading packets out of the data structure. This bottleneck could easily be eliminated with a few minor software changes provided that the file's data could fit within the computer's memory (including delays) and the data was repetitive. Of course, in any real-time scenario, this would not be possible because the data would be aperiodic.

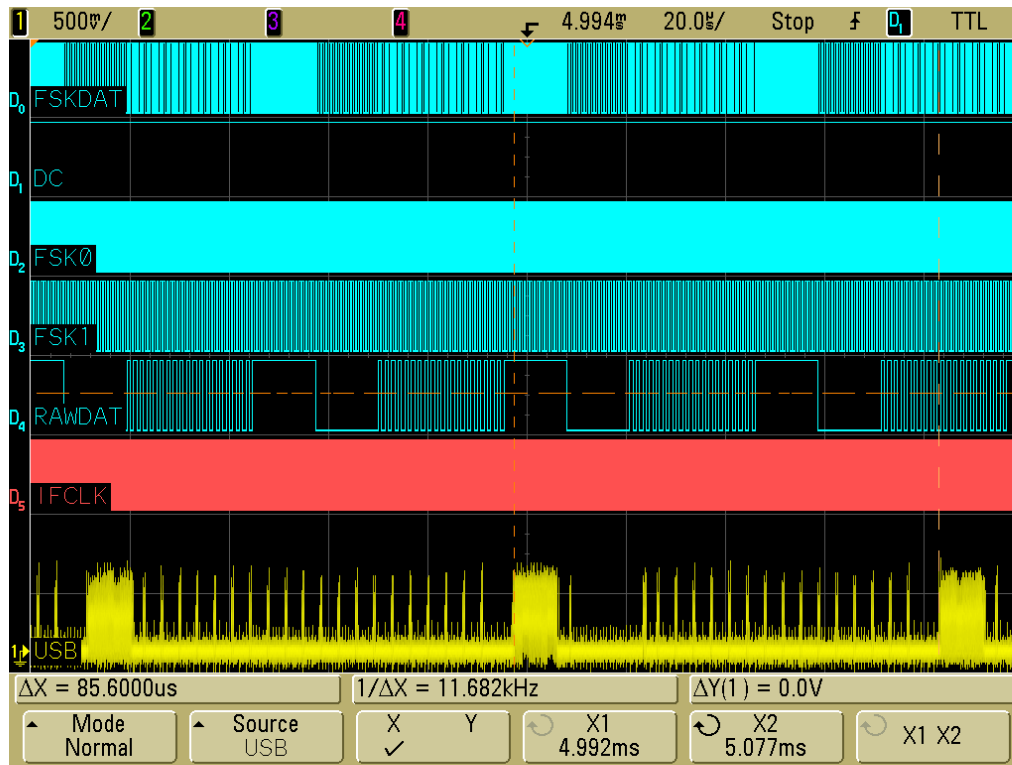



*Fig 16: Continuous data transmission w/ GPIF clock @ 6MHz (clock divider = 8)*

### Delays

Delays provide space between pulse commands and bursts. Currently, delays are simply data packets themselves that put 0's on the data line and loop for however long it takes to produce a given delay (specified in milliseconds). But, this is not a workable option when very large amounts of data are being sent (and hence lots of memory would be required to buffer the pulses and delays) or when the software needs to do other work during these periods of time (i.e. read a frame from a camera). For these cases, a new delay scheme needs to be developed that allows the CPU to do other work while reliably returning to transmission when the desired delay has been reached. This would mean resynchronization commands would have to be inserted at the start of a new data frame, because the clock would stop, and this too could be done automatically with the software.

### CPU Interruptions

Interruptions from OS functions or from other software running on the system can occur between packets, but never in the middle of a packet (in all the testing we have done, we haven't seen it happen). This indicates that the CyAPI transmissions are not interruptable, but the asynchronous queueing seems to be. So, by setting the packet size large, one can guarantee that that amount of data will be sent in one continous stream. But setting the queue size high seems to have no effect on the reliability of the stream to remain constant. Setting the Windows thread priority to its highest level ("realtime") did seem to improve stream constancy, but there were still breaks when the CPU was under heavy loads.

## Development Goals for Fall 2005/Spring 2006

1. Optimize UI code to improve throughput when reading from the data structure (getBytes() method).
2. Add functionality to "browse" button and message list, including printing the number of pulses defined in the file and the number of bytes for each loop.
3. For static file reading, include buffer synchronization technique that would size the output data buffer according to the length of the loop and resend it as many times as the loop dictates (potentially infinite). This way, there is no overhead in retrieving the next byte, provided that the entire loop can fit into memory.
4. Provide delay routines which do not require CPU overhead and leave the CPU free to do other work (like process the next frame of data and populate the next output buffer)
5. Include firmware messages (Vendor Commands) to select divisor ratio (an arbitrary integer from 1-256). Also, a command to select the external clock source (CLKIN) would be nice. The UI needs to be modified to send these commands as well.
6. Improve UI's error handling capability. Add a "File Open" dialog box, and send feedback messages to the textbox in the UI. Implement "Reload" and "Reset Device" buttons
7. Populate PCB, download code to EEPROM, and test.
8. Add biphasic pulse generating code in the script file syntax (@pulse), with all of the relevant parameters to describe pulses and bursts.
9. Build LED matrix demonstration board.
10. Build and test filter/amplifier board.
11. Write code to convert images from the Video4Windows camera API to a series of command frames that produce an image on the LED matrix.
12. Port the UI to a Pocket PC platform.

## Ideas For Further Development

1. Employ a PLL clock multiplier and custom hardware to shift out bits from the 16-bit GPIF parallel bus, thereby fully utilizing it and dramatically increasing throughput. Also, the PLL could be used to generate the FSK modulation automatically, increasing throughput by at least a factor of 4
2. Use a microcontroller with more memory (several megabytes) to store pulse definitions on-chip, cutting down on USB traffic. This would also allow for a larger output buffer.
3. Add custom output logic to automatically loop through a keep-alive command when the bus is idle, ensuring that the probe never loses power and guaranteeing that communications is never lost during long data transfers, no matter how busy Windows gets (see Appendix A).

## Known Bugs

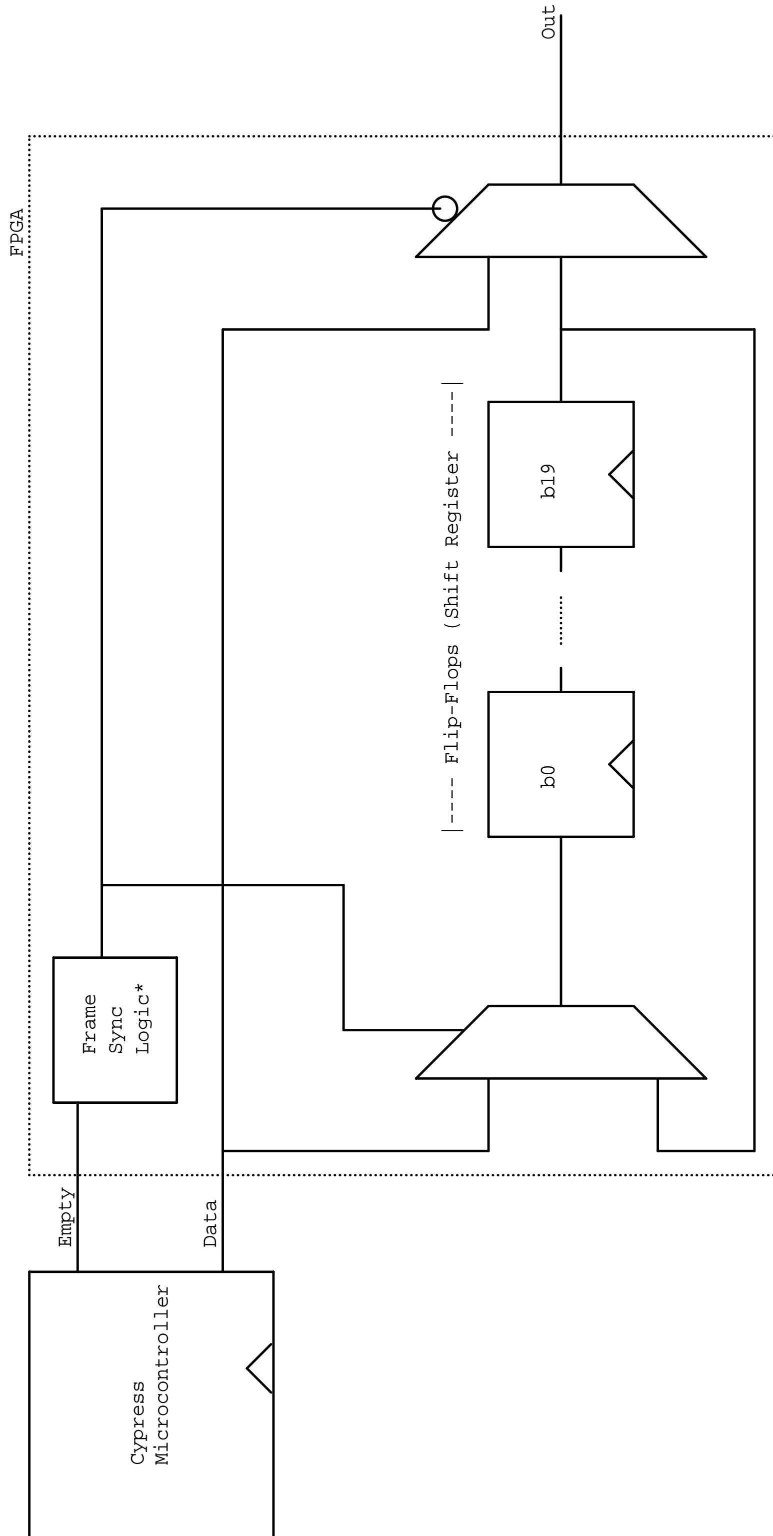
1. Files must end in an infinite loop. Finite data sets cause the program to crash. This is most likely due to a problem with how the software detects and marks the end of the buffer within the xfer thread.

## Conclusions

Phase I of this project showed that high-speed data transfers in the range required for a visual prosthesis application are indeed possible with USB 2.0. The most important thing to keep in mind is that since the microcontroller itself is essentially a pass-through component in this case, it is up to the Windows UI to supply data fast enough and with enough precision to generate the desired stimulation pulse rate. More work should be done to improve the UI and make it more intuitive and useful for physiologists. If speed becomes a problem—either for continuous streaming or for burst throughput, there are several options for dramatically improving performance. Bearing in mind that data is being sent at 480Mbits/s, we have a lot of headroom for throughput, particularly if we employ additional hardware.

Phase II will focus on demonstrating the interface with the complete system and adding functionality to the UI so it can be employed in a physiology lab setting.

# Appendix A: Hardware Keep-alive solution




* Frame sync logic maintains a counter that tracks the current position in the frame of data being transmitted.  If the empty flag is transitioning from 0 to 1, it immediatly triggers the muxes to switch the output to the shift registers and cycle the bits around in the shift register.  If the empty flag transitions from 1 to 0, it waits until the end of the current frame before switching back over to the uC's data output.