Cypress FX3 Firmware for INI Sensors

Luca Longinotti
on behalf of the
Institute of Neuroinformatics, UZH/ETHZ

August 29, 2013

Contents

1 Introduction **3**
- 1.1 Project Goals . . . 3
- 1.2 Hardware Situation . . . 3

2 USB 3.0 **4**
- 2.1 Comparison with USB 2.0 . . . 6

3 Cypress FX3 **7**
- 3.1 Comparison with Cypress FX2LP . . . 7
- 3.2 Development Kit . . . 8
- 3.3 Development Tools . . . 9

4 FX3 Firmware **11**
- 4.1 Comparison with FX2 Firmware . . . 11
- 4.2 Architecture . . . 11
- 4.3 File Structure . . . 12
- 4.4 Debugging and Status End-Point . . . 15
- 4.5 Slave FIFO Interface and Data End-Point . . . 15
- 4.6 Custom DMA Processing . . . 16
- 4.7 Common Vendor Requests . . . 17
- 4.8 Custom Vendor Requests . . . 19
- 4.9 GPIO . . . 19
- 4.10 Heartbeat . . . 21
- 4.11 I2C Bus . . . 21
- 4.12 SPI Bus . . . 23
- 4.13 Custom Initialization . . . 25
- 4.14 Implementing a Device . . . 25

5 jAER Integration **27**
- 5.1 USBIO Library . . . 27

6 Summary **28**
- 6.1 Lessons Learned . . . 28

A Pin Assignment **29**

B VR Assignment **31**

C SPI Memory Map **33**

D Clock Frequencies **35**

Chapter 1

Introduction

The growing resolution of the sensors developed at the Insitute of Neuroinformatics (INI) requires ever increasing bandwidth to be able to transfer data from the sensor to the host. The current generation of sensors supports USB 2.0 for data transfer using the Cypress FX2LP micro-controller. An update of the infrastructure to support USB 3.0 (see chapter 2) using the new Cypress FX3 micro-controller (see chapter 3), to leverage the features and possibilities offered by both the new standard and the new micro-controller, was essential to the continued growth of the hardware developed by INI for its research.

1.1 Project Goals

The three main project goals, listed below, were met successfully:

1. The firmware for the Cypress FX3 chip was implemented (available at `https://sourceforge.net/p/jaer/code/HEAD/tree/trunk/deviceFirmwarePCBLayout/CypressFX3/firmware/`), see chapter 4 for more details.

2. The general documentation is provided both as this report, and as comments within the code. Further, all used data-sheets, as well as spread-sheets with additional details, as well as this report, are available for download at `https://sourceforge.net/p/jaer/code/HEAD/tree/trunk/deviceFirmwarePCBLayout/CypressFX3/docs/`.

3. The firmware and its test-board were experimentally integrated with the jAER framework using the Thesycon USBIO library, see chapter 5 for more details.

1.2 Hardware Situation

All the development work for the firmware was executed on the FX3 development kit (see section 3.2), which provided for fairly minimal testing opportunities (firmware loading, single IO pins). The real board that will make use of the developed firmware has just completed its final design stages and is not projected to be produced, assembled and tested before mid-September 2013.

All work done here is based on data-sheets for the hardware that is expected to be part of the final product.

Chapter 2

USB 3.0

The USB 3.0 standard was finalized in November 2008, but started seeing a wider adoption only during the years 2011-2012, with the wider distribution and integration of host-controller chips for USB 3.0, notably by Intel and NEC.

The two most visible new features are certainly the dramatically increased bandwidth of 5 Gbit/s (compared to the earlier 480 Mbit/s of USB 2.0), as well as the better power management available to devices, with now four power states available to devices (U0 to U3). Several other new features were also introduced with the new protocol under the hood, notably asynchronous notifications (through the NRDY/ERDY packets), which enable the host-controller to be notified of traffic, instead of having to continuously poll the device; also the traffic now only goes to the targeted device, and is not broadcast anymore to all devices connected to the USB hub. Further new interesting features are support for Bursts and Streams when transferring data, allowing respectively for increased data rates and logical data transfer streams.

The USB 3.0 architecture is fully backwards compatible with USB 2.0 and prior standards. It achieves this by also integrating all the necessary hardware for USB 2.0 and by also having the signals required for USB 2.0 inside the 3.0 cable. This so called dual-bus approach, as can be seen in Figure 2.1, specifies compatibility with USB 2.0 by including and integrating it almost 1:1 alongside the new USB 3.0 bus. Figure 2.2 shows clearly how the signals for USB 2.0 (UTP Signal Pair) are also present inside the USB 3.0 composite cable, alongside the signals needed for USB 3.0 (SDP Signal Pair 1/2). The eXtensible Host Controller Interface (xHCI) describes how to interface with the new USB 3.0 host controllers, and is capable of interfacing with USB 3.0, 2.0 and 1.X, supporting all speeds in a single driver stack. Drivers are available from host controller vendors directly for Windows XP, Vista and 7, while Windows 8 offers an integrated USB 3.0 driver out-of-the-box. Linux supports it since kernel 2.6.31 and Mac OS X at least since release 10.8 Mountain Lion.

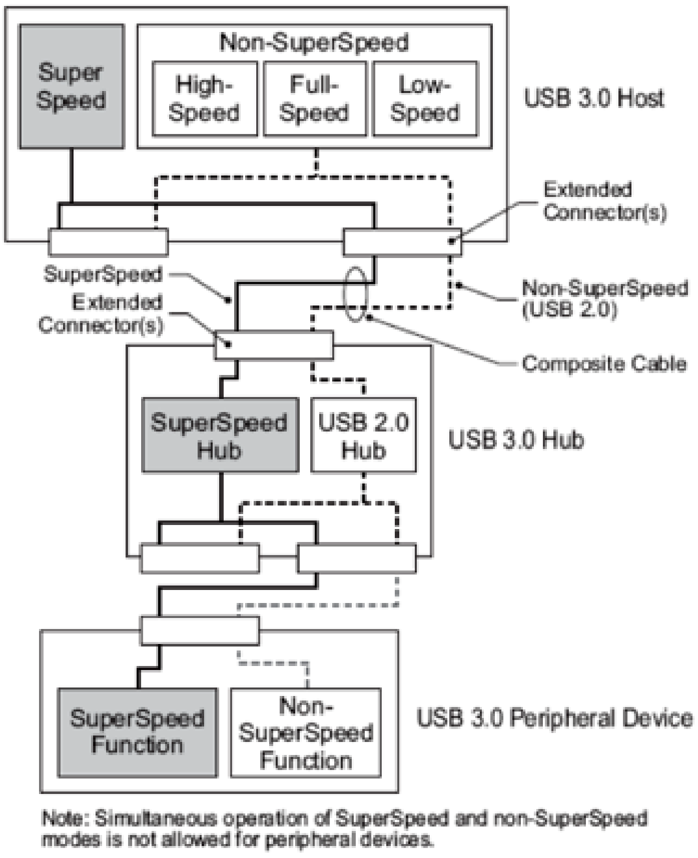


Figure 2.1: USB 3.0 dual-bus architecture (http://www.usb.org/)

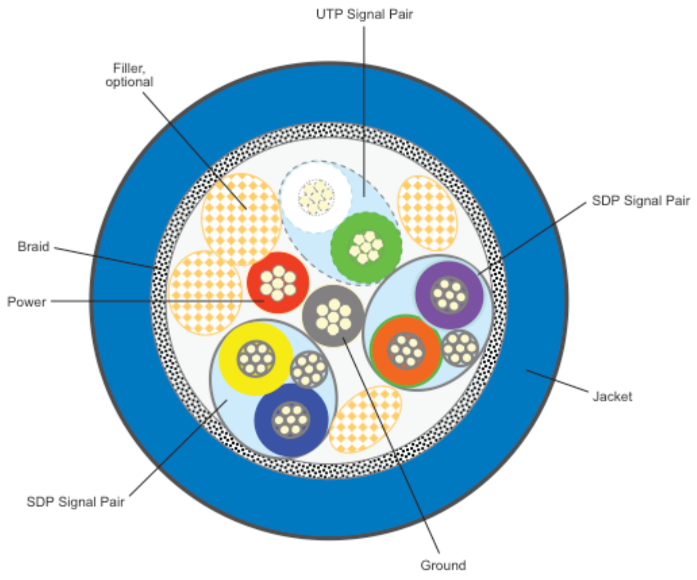


Figure 2.2: USB 3.0 cable (http://www.usb3expresscard.com/)

2.1 Comparison with USB 2.0

Table 2.1 compares the most important characteristics of USB 3.0 and 2.0, showing the significant differences between the two protocols [3]:

Feature	USB 2.0	USB 3.0
Bandwidth	480 Mbit/s, 12 Mbit/s, 1.5 Mbit/s	5 Gbit/s, 480 Mbit/s, 12 Mbit/s, 1.5 Mbit/s
Data interface	Half-duplex over two-wire differential signaling	Dual-simplex over four-wire (twice two-wire) differential signaling; plus USB 2.0 interface for backwards compatibility
Signals (cable)	Four: two for USB 2.0 data, two for VBUS and GND	Ten: four for USB 3.0 data, two for USB 3.0 GND, two for USB 2.0 data, two for VBUS and GND
Bus protocol	Host-directed, polled traffic, packets broadcast to all downstream devices, no multiplexing of data-streams	Host-directed, asynchronous notifications, packets routed to target device only, multiple data-streams for bulk mode possible
Power management	Two modes: Active, Suspend	Four modes: Active (U0), Idle Fast (U1), Idle Slow (U2), Suspend Slow (U3)
Bus power	Low-power device: 100 mA, High-power device: 500 mA	Low-power device: 150 mA, High-power device: 900 mA
Cable length	max. 5 meters	max. 3 meters (depends on electrical specification)
Data transfer types	Four: Control, Bulk, Interrupt, Isochronous	Four: Control, Bulk, Interrupt, Isochronous

Table 2.1: USB 2.0 and 3.0 comparison

Chapter 3

Cypress FX3

The FX3 [7] is a USB 3.0 micro-controller developed and marketed by Cypress Semiconductor Corporation. It enables peripherals to be accessed and controlled over USB 3.0, providing the means to easily build USB-enabled devices. It supports connecting to a wide variety of devices thanks to the several serial buses it implements:

- I2C master at 100 KHz/400 KHz/1 MHz
- UART full-duplex
- SPI master up to 33 MHz
- I2S master at 32/44.1/48 KHz (mostly audio applications)
- GPIO general purpose IO pins (partially shared with GPIF2)

as well as the flexible, programmable parallel interface GPIF2, which enables data exchange up to 32bit at 100 MHz (400 MByte/s) and allows the implementation of master/slave interfaces common in the industry. Appendix D encloses a list of all clock frequencies and their default values. A new Direct Memory Access (DMA) engine controls the data transfer between the peripherals, freeing up the processor after initialization for other tasks. A real-time operating system with support for multiple threads, ThreadX, significantly enhances and simplifies initialization and working with devices, as well as providing a much better and cleaner API to program with.

3.1 Comparison with Cypress FX2LP

Table 3.1 compares the most important characteristics of the Cypress FX3 and its predecessor, the Cypress FX2LP, showcasing the relevant differences between the two micro-controllers [3]:

Feature	FX2LP	FX3
USB support	USB 2.0, up to 6 physical endpoints	USB 3.0 and 2.0, USB 2.0 On-The-Go (OTG), battery charging supported, up to 32 physical endpoints
CPU	Enhanced Intel 8051 at 48 MHz	32-bit ARM926EJ at 200 MHz
SRAM	16 KB	512 KB
Boot from	USB, I2C	USB, I2C, SPI, GPIF2 [5]
GPIF	8/16-bit data bus at 48 MHz	8/16/32-bit data bus at 100MHz
Peripherals	I2C (up to 400 KHz), UART	I2C (up to 1 MHz), UART, SPI (up to 33 MHz), I2S
Operating System	-	ThreadX RTOS
Compiler / IDE	Keil C for 8051	Eclipse with GNU ARM toolchain (GCC)

Table 3.1: Cypress FX2LP and FX3 comparison

3.2 Development Kit

The FX3 development kit [6] consists of the FX3 development board, shown in Figure 3.1, the corresponding USB cable and power adapter, as well as the FX3 Software Development Kit (SDK), detailed in section 3.3. It provides connectivity to all the interfaces that the FX3 micro-controller supports, as well as debugging facilities, and incorporates a 4 Mbit SPI Flash memory. It can load its firmware easily from either USB or the SPI Flash mentioned above, providing for easy testing of new firmware.

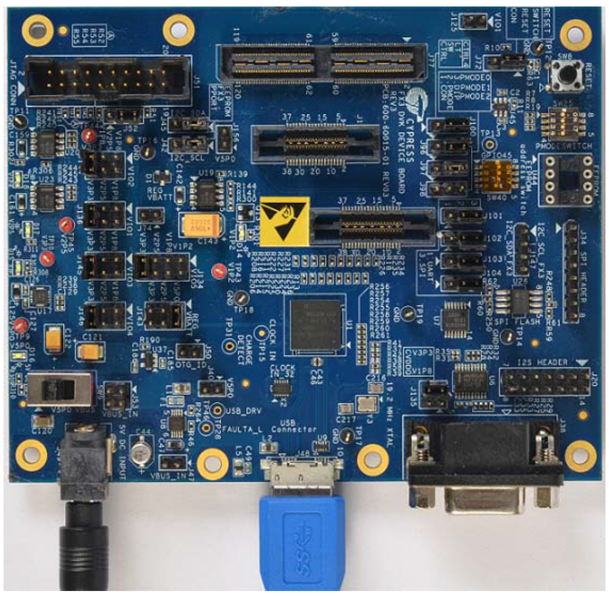


Figure 3.1: FX3 development board (http://www.cypress.com/)

3.3 Development Tools

The FX3 SDK was completely revamped and features multiple improvements over the SDK for its predecessor, which was based on the Keil C compiler for 8051.

The central development application is now based on the Eclipse IDE, as demonstrated in Figure 3.3, and uses the ARM port of the GNU GCC compiler to generate the target binaries. This gives the developer an up-to-date environment in which to work, easily supporting modern features such as auto-completion, multiple projects, integrated API documentation and so on, using a well-known and established interface.

Thanks to a layered, modular approach, developing firmware for the FX3 has become easier and less error-prone. As seen in Figure 3.2, the user writes his own firmware by leveraging the FX3 firmware framework and its well-documented APIs, which in turn run on the ThreadX real-time operating system. Lots of examples provide a starting point and a kind of documentation on how certain types of problems are to be solved using the FX3 framework. This enables the programmer to focus more on logic and features, than on which bit set to what value to provide some effect, as it was back in the register-driven era of the FX2LP firmware. High-level APIs are provided to program and setup the USB, GPIF2 and serial (I2C, UART, SPI, I2S, GPIO) interfaces, as well as controlling the DMA engine, how data is exchanged and how to handle USB requests. APIs to access debug facilities as well as the underlying ThreadX RTOS are also provided, as well as advanced APIs to access and control USB and DMA in a much more fine-grained way. A quick example to illustrate the much cleaner approach possible thanks to the well defined and documented function-based API is shown in Table 3.2.

FX2LP
EP0BUF[0] = SETUPDAT[1];
EP0BUF[1]= TIMESTAMP_MASTER; // PC1 (Port C Pin 1)
EP0BCH = 0;
EP0BCL = 2;
EP0CS |= bmHSNAK;
FX3
uint8_t Buffer[2];
Buffer[0] = bRequest;
CyU3PGpioGetValue(GPIO_ID, &Buffer[1]); // GPIO Pin
CyU3PUsbSendEP0Data(2, &Buffer);

Table 3.2: Cypress FX3 function-based API versus FX2LP register-driven API

Other tools are also included, such as the GPIF2 Designer, which provides an easy way to specify the various characteristics of the GPIF2 interface and its programmable states, using a simple graphical user interface; or the ControlCenter application, which enables easy uploading and flashing of new firmware to the FX3 for testing purposes.

The FX3 firmware is now generated in only one format directly by the tool-chain, and this format can then be uploaded directly to the micro-controller or to a memory device from which it can later read it and boot from. No multiple formats or conversions between them (BIX, HEX, IIC formats) are needed anymore. Several examples, guides, various documentation, as well as an auto-update feature for the SDK complete the offering.

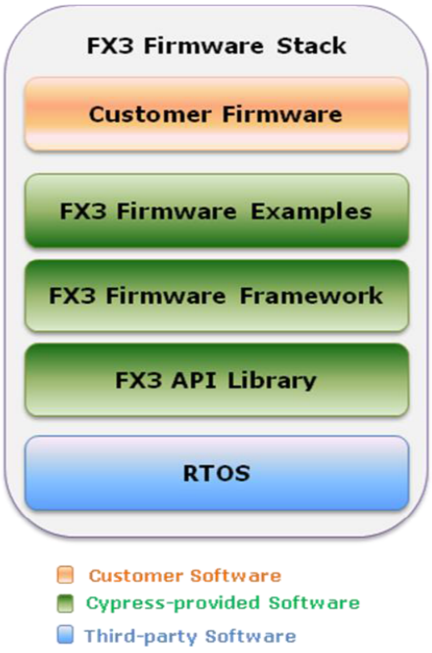
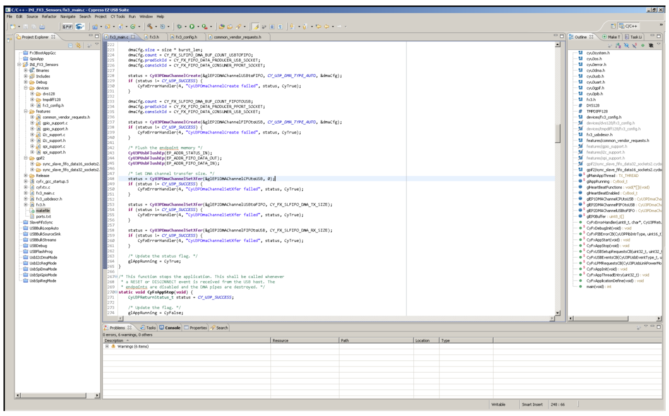


Figure 3.2: FX3 firmware stack (http://www.cypress.com/)



```
    dmaCfg.size = size * burst_len;
    dmaCfg.count = CY_FX_SLFIFO_DMA_BUF_COUNT_USBTOFIFO;
    dmaCfg.prodSckId = CY_FX_FIFO_DATA_PRODUCER_USB_SOCKET;
    dmaCfg.consSckId = CY_FX_FIFO_DATA_CONSUMER_PPORT_SOCKET;

    status = CyU3PDmaChannelCreate(&glEP2DMAChannelUSBtoFIFO, CY_U3P_DMA_TYPE_AUTO, &dmaCfg);
    if (status != CY_U3P_SUCCESS) {
        CyFxErrorHandler(4, "CyU3PDmaChannelCreate failed", status, CyTrue);
    }

    dmaCfg.count = CY_FX_SLFIFO_DMA_BUF_COUNT_FIFOTOUSB;
    dmaCfg.prodSckId = CY_FX_FIFO_DATA_PRODUCER_PPORT_SOCKET;
    dmaCfg.consSckId = CY_FX_FIFO_DATA_CONSUMER_USB_SOCKET;

    status = CyU3PDmaChannelCreate(&glEP2DMAChannelFIFOtoUSB, CY_U3P_DMA_TYPE_AUTO, &dmaCfg);
    if (status != CY_U3P_SUCCESS) {
        CyFxErrorHandler(4, "CyU3PDmaChannelCreate failed", status, CyTrue);
    }

    /* Flush the endpoint memory */
    CyU3PUsbFlushEp(EP_ADDR_STATUS_IN);
    CyU3PUsbFlushEp(EP_ADDR_FIFO_DATA_OUT);
    CyU3PUsbFlushEp(EP_ADDR_FIFO_DATA_IN);

    /* Set DMA channel transfer size. */
    status = CyU3PDmaChannelSetXfer(&glEP1DMAChannelCPUtoUSB, 0);
    if (status != CY_U3P_SUCCESS) {
        CyFxErrorHandler(4, "CyU3PDmaChannelSetXfer failed", status, CyTrue);
    }

    status = CyU3PDmaChannelSetXfer(&glEP2DMAChannelUSBtoFIFO, CY_FX_SLFIFO_DMA_TX_SIZE);
    if (status != CY_U3P_SUCCESS) {
        CyFxErrorHandler(4, "CyU3PDmaChannelSetXfer failed", status, CyTrue);
    }

    status = CyU3PDmaChannelSetXfer(&glEP2DMAChannelFIFOtoUSB, CY_FX_SLFIFO_DMA_RX_SIZE);
    if (status != CY_U3P_SUCCESS) {
        CyFxErrorHandler(4, "CyU3PDmaChannelSetXfer failed", status, CyTrue);
    }

    /* Update the status flag. */
    glAppRunning = CyTrue;
}

/* This function stops the application. This shall be called whenever
 * a RESET or DISCONNECT event is received from the USB host. The
 * endpoints are disabled and the DMA pipes are destroyed. */
static void CyFxAppStop(void) {
    CyU3PReturnStatus_t status = CY_U3P_SUCCESS;

    /* Update the flag. */
    glAppRunning = CyFalse;
```

0 errors, 6 warnings, 0 others

Description	Resource	Path	Location	Type
Warnings (6 items)				

Figure 3.3: FX3 Eclipse-based SDK

Chapter 4

FX3 Firmware

The firmware for the FX3 is a complete re-development. It would have been difficult [4] to port the old FX2 firmware in a straight-forward manner, since everything changed so radically, from the possibilities that the hardware offers, to the way it's programmed and the tools used. Further, as explained below, a re-engineering effort was desirable. As such, the old firmware was just used as a place to gather the requirements that a new, general-purpose firmware would need to satisfy at the very least, and the new firmware is based upon a completely new architecture and new code.

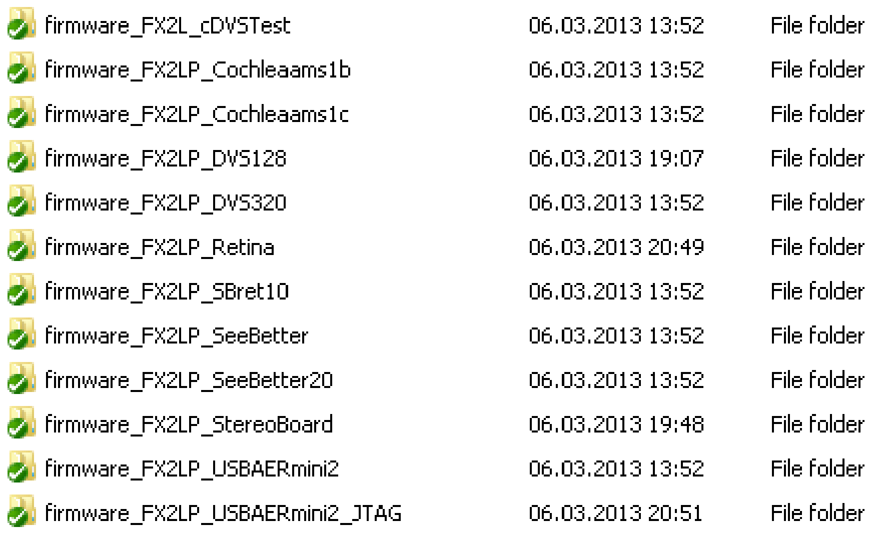
4.1 Comparison with FX2 Firmware

When looking at the FX2 firmware, it became quickly evident, as shown in Figure 4.1, that it had suffered major fragmentation during its life-cycle; indeed about 14 different firmware projects exist currently for almost the same number of sensor devices and variants thereof. Most of those consist of an older version of the working firmware being copied and some code here and there being adjusted to the new requirements of the device currently being worked on. This approach to software development is sub-optimal, as it heavily splits the effort and doesn't provide for a common way to maintain the older code bases, due to heavy code duplication, making it even more difficult to understand and compare the functionality of the various FX2 firmware implementations.

To remedy the above situation, it was decided to write not one firmware for each device in the case of the FX3, but instead a firmware framework, offering common, as well as optional, functionality in a well-defined way, and then enable each new device to simply be a new set of configuration files to control the framework's behavior, enable functionality as needed and, depending on the application, hook into certain parts of the framework and supply device-specific code-paths for certain tasks.

4.2 Architecture

The core concepts of the new architecture are: small common core; additional functionality as optional, separate modules; and per-device configuration and customization. The minimal core is composed of the device initialization and configuration code, as well as the configuration for the GPIF2 interface to exchange data, either 16 or 32 bits at a time,



firmware_FX2L_cDVSTest	06.03.2013 13:52	File folder
firmware_FX2LP_Cochleaams1b	06.03.2013 13:52	File folder
firmware_FX2LP_Cochleaams1c	06.03.2013 13:52	File folder
firmware_FX2LP_DVS128	06.03.2013 19:07	File folder
firmware_FX2LP_DVS320	06.03.2013 13:52	File folder
firmware_FX2LP_Retina	06.03.2013 20:49	File folder
firmware_FX2LP_SBret10	06.03.2013 13:52	File folder
firmware_FX2LP_SeeBetter	06.03.2013 13:52	File folder
firmware_FX2LP_SeeBetter20	06.03.2013 13:52	File folder
firmware_FX2LP_StereoBoard	06.03.2013 19:48	File folder
firmware_FX2LP_USBAERmini2	06.03.2013 13:52	File folder
firmware_FX2LP_USBAERmini2_JTAG	06.03.2013 20:51	File folder

Figure 4.1: Multitude of FX2 firmware projects

on an USB end-point (4.5). An additional USB end-point provides status data exchange (4.4). Optionally, one can enable communication over various serial protocols, such as I2C (4.11), SPI (4.12) and GPIO (4.9). I2S and UART were not taken into consideration, as no present or future sensor presented any use case for them. All of this is configured per-device, using a mixture of #defines, functions and arrays, enabling high flexibility in the deployment of customized firmware. The following sections will explore each feature in higher detail. Three USB end-points are provided: EP 0, the control end-point, is responsible for managing vendor requests, which can be seen as the commands a USB device can react and respond to. This end-point is always present, and actually mandated by the protocol standard. EP1, the status end-point, and EP2, the data end-point, will be explained in detail in sections 4.4 and 4.5.

4.3 File Structure

Let's go over each file and directory in the new structure, as seen in Figure 4.2 in more detail:

- cyfx_gcc_startup.S is an assembly file provided by Cypress, which has to be part of any firmware project and should never be touched by the user. It initializes the device and jumps to the user-supplied initialization code.
- cyfxtx.c is another file provided by Cypress, containing parts of the API the user is going to have access to later, mostly concerning memory management. It can be edited if particular memory management requirements have to be met.
- fx3.h is the main configuration file for global settings. Several framework-wide settings can be changed here in the form of #defines, such as the default log level, the default thread stack-sizes and priorities, as well as all USB end-point and DMA

settings. Several helper macros for general usage, as well as several global variables, are also declared here as to be accessible everywhere; like the global error handler or the KILO/MEGA macros for improved readability. All of these settings can also be tweaked on a per-device basis.

- fx3_main.c includes the main firmware functionality. It is responsible for initializing the device according to the configuration, for creating the various threads that will handle requests, and initializing the various configured serial and USB sub-systems. It further provides the global error handler, as well as brings up the USB endpoints for data exchange. Most of the above is usually done by calling out to the appropriate functions implemented within the feature or device-specific modules.

- fx3_usbdescr.h contains a series of C-structs, which represent the various USB descriptors of the device. USB descriptors are pieces of information that the host system uses to discover what abilities, services and features the connected device has to offer, such as the maximum USB version supported, or how much electrical current is needed, or who the manufacturer of the device is. The most important settings, such as the Vendor ID or Product ID, can be overridden per-device.

- gpif2 directory contains the configuration files for the GPIF2 interface, once for data exchange of 16 bits per clock cycle (the default setting), and once for 32 bits per cycle. Both configurations were generated using the GPIF2 Designer tool from Cypress.

- features directory contains the implementation of the various additional features, so as to keep them well separated from the core. The header files present usually define which vendor requests are supported by the module in question and which functions are exported and visible to other modules.

- features/common_vendor_requests.h,c handles several general-purpose vendor requests, ranging from status information to debugging helpers. See section 4.7 for a more detailed explanation.

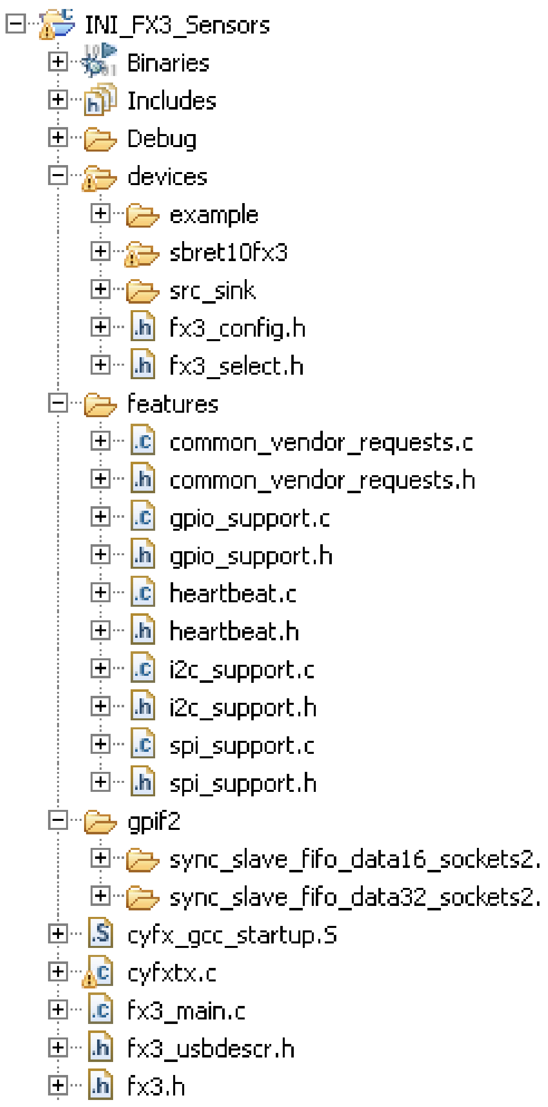
- features/gpio_support.h,c implements the handling of General-Purpose I/O (GPIO) pins, as well as their initial configuration. Section 4.9 contains more information on all supported modes.

- features/heartbeat.h,c contains the heartbeat system, which enables the execution of defined functions at timed intervals to accomplish some task, such as flashing a LED. Section 4.10 enters into more detail.

- features/i2c_support.h,c enables the firmware to support the I2C serial bus to exchange data with connected peripheral devices. Section 4.11 has a detailed explanation of the protocol and the involved code.

- features/spi_support.h,c handles support for the SPI bus, another serial data bus with slightly different characteristics than I2C and higher speeds. See section 4.12 for details.

- devices directory contains the device-specific configuration, which makes this firmware so flexible and powerful.
- devices/fx3_select.h lists all supported devices, and selects the one for which the firmware should be compiled at this moment in time.
- devices/fx3_config.h holds all the #defines and function declarations for the various optional features and configuration settings, that have to be set and implemented by a device to enable certain functionality.
- devices/$device/$device_config.h contains the per-device #defines.
- devices/$device/$device_config.c, on the other hand, contains the per-device function implementations and array definitions, as required by the enabled features.



```
INI_FX3_Sensors
  Binaries
  Includes
  Debug
  devices
    example
    sbret10fx3
    src_sink
    fx3_config.h
    fx3_select.h
  features
    common_vendor_requests.c
    common_vendor_requests.h
    gpio_support.c
    gpio_support.h
    heartbeat.c
    heartbeat.h
    i2c_support.c
    i2c_support.h
    spi_support.c
    spi_support.h
  gpif2
    sync_slave_fifo_data16_sockets2.
    sync_slave_fifo_data32_sockets2.
  cyfx_gcc_startup.S
  cyfxtx.c
  fx3_main.c
  fx3_usbdescr.h
  fx3.h
```

Figure 4.2: FX3 firmware structure

4.4 Debugging and Status End-Point

The status end-point, EP1, is a special end-point meant for quick notifications to be sent out from the device to the host. As such, it only operates in that one direction (from FX3 to USB), and can only send data packets of up to 64 bytes. These limitations are imposed by the fact that this is an INTERRUPT type end-point, which, on the other hand, offers guaranteed latency, which is of critical importance for being able to send status messages in a timely manner.

On top of the status-endpoint, the standard debug facility is implemented, enabling easy status and error messages to be sent in a common format. The following function is responsible for sending error messages; it expects a valid log level (see section 4.7, VR_LOG_LEVEL), as well as a message explaining the problem and an appropriate error code.

```
void CyFxErrorHandler(uint8_t log_level, const char *
    debug_message, CyU3PReturnStatus_t error_code);
```

The format of the sent message is as follows:

Byte	Content
0	0x00, which signifies a standard error message
1	the error code
2-5	the current time-stamp
6-63	the error message, up to 58 characters (64 maximum minus 6)

Table 4.1: Error message format

It can happen that a message is not delivered to the host, because, to avoid blocking the handler threads, the logging function returns right away. If, when it is called, no buffers are free to receive the new log message, because their number is limited and they could have already been consumed by other messages that haven't yet been delivered to the host either, the message will be dropped. The occurrence of such drops is counted and can be queried with the VR_STATUS vendor request. In practice this rarely happens, unless a veritable storm of errors is produced, while the device is running and communicating; but is more easily possible if the device is running, but not connected to USB. In such a case, only the first few messages are preserved for later sending over USB when the device is again connected, any successive ones are lost.

4.5 Slave FIFO Interface and Data End-Point

For data transfers, the data end-point is provided. EP2 is a BULK type end-point, which is perfect for streaming large amounts of data. It is directly linked to the GPIF2 interface, and is used for high-speed data exchange between the external device connected to the GPIF2 interface and USB. The GPIF2 interface's default configuration is that of a 16 bit bi-directional Slave FIFO, meaning it can transfer data to and from the external device, in FIFO order, and it's the device that initiates read or write cycles, not the FX3 micro-controller. The FX3 provides a series of buffers onto which data is written to or read from, and several status flags so that the external device knows when it can access the FX3 for those operations and when not. Those four flags signal if the currently selected buffer is either FULL, ALMOST_FULL, ALMOST_EMPTY or EMPTY. Please refer to [1] for more detailed information, especially on how external devices access the buffer and the required timings thereof. Through configuration (#define GPIF_32BIT_SUPPORT_ENABLED), it is possible to switch to a 32 bit wide data exchange, but sacrificing at the same time sixteen of the limited GPIO pins to provide for the needed additional data lines.

4.6 Custom DMA Processing

Normally, data packets are forwarded quickly and directly from the external interfaces to USB and vice versa, thanks to the DMA engine. It is though possible to bypass this mechanism to gain greater control over the data packets in transit, and being then able to examine them, modify them or discard them directly on the FX3's ARM CPU, by deviating the data flow in question through it. Support for this functionality is provided by the following #defines and functions:

```
#define DMA_USBTOFX3_CALLBACK (0)

void CyFxDmaUSBtoFX3Callback(CyU3PDmaChannel *chHandle,
    CyU3PDmaCbType_t type, CyU3PDmaCBInput_t *input);

void CyFxDmaUSBtoFX3CallbackInit(CyU3PDmaChannel *chHandle);
void CyFxDmaUSBtoFX3CallbackDestroy(CyU3PDmaChannel *chHandle);

#define DMA_FX3TOUSB_CALLBACK (0)

void CyFxDmaFX3toUSBCallback(CyU3PDmaChannel *chHandle,
    CyU3PDmaCbType_t type, CyU3PDmaCBInput_t *input);

void CyFxDmaFX3toUSBCallbackInit(CyU3PDmaChannel *chHandle);
void CyFxDmaFX3toUSBCallbackDestroy(CyU3PDmaChannel *chHandle);
```

As you can see, it is possible to enable the functionality for the IN and OUT channels separately. The user is expected to implement a main function to manipulate the data packets moving over the specified DMA channel, as well as two helper functions that are called when the DMA channel is either initialized or destroyed. Please use this feature sparingly, as having to move the high-speed data streams from EP2 through the CPU adds a heavy load to it and slows down the data transfer. The more complex the inspection and algorithms being applied on the data, the higher the slow-down will be; keep in mind, this is only a 200 MHz RISC CPU.

4.7 Common Vendor Requests

USB vendor requests are used to send commands to USB devices, always over end-point zero, and have a well defined format [14]; the setup packet consisting of eight bytes, with the following meaning: byte 0 encodes the direction of the data transfer, as well as its type and the intended recipient (usually the device); byte 1 contains the code of the USB request (bRequest field), bytes 2-3 represent the wValue field, containing any 16 bit value; bytes 4-5 contain the wIndex field, again with 16 bits of user-defined information; and lastly, bytes 6-7 contain the wLength field, which specifies the length of the data portion to come afterwards. In other words, one can have a few hundred commands per device (the 0x00 to 0x3F codes are reserved), each of which can be supplied up to 32 bits of data to configure their behavior (wValue plus wIndex fields), to then transfer up to 65 KB of data to the device or from it, in a single request. All vendor requests supported by this firmware are listed in Appendix B.

The common vendor requests module supports general purpose functionality, concerning itself with device information and debugging. All vendor requests that can't be handled by a module, because they are not recognized, are forwarded to the next, until either a module is found which is able to handle the specified request, or an error is sent back to the user.

The VR_MS_FEATURE_DSCR vendor request provides an implementation for the WCID feature offered by the Microsoft Windows WinUSB driver, which allows a device to specify that it is compatible with and wants the WinUSB driver to be responsible on the host-side for communication, freeing the consumer from having to worry about driver installation and offering a well tested and solid way to exchange data. This is done by having the device supply a particular USB descriptor, which in turn specifies a particular vendor request that the Windows USB sub-system will then execute to query the needed information from the device. The format of the response is defined by Microsoft documentation, and consists of 40 bytes containing various pieces of data about the device and how it can interact with the Windows USB sub-system. This request is never initiated by users of the device themselves, and support for it has to be explicitly enabled in the device configuration with the MS_FEATURE_DESCRIP TOR_ENABLED #define.

VR_TEST implements a quick and simple way to test the device: any message sent along with it, is then sent back over the status end-point, giving an easy way to test communication in both directions over different end-points. Only the first 58 bytes are sent back, as that's the limit of external data the status end-point can send in one go (see section 4.4 for a more detailed explanation on why this is so).

The VR_LOG_LEVEL request sets the default log level, delimiting which log messages generated by the firmware are sent out over the status end-point to the host. The wValue field encodes one of the following supported values:

Log Level	Value
LOG_EMERGENCY	0
LOG_ALERT	1
LOG_CRITICAL	2
LOG_ERROR	3
LOG_WARNING	4
LOG_NOTICE	5
LOG_INFO	**6**
LOG_DEBUG	7

Table 4.2: Supported log levels

The default value is set in fx3.h to LOG_INFO. Further, setting the value to LOG_DEBUG, will also enable a heart-beat message to be sent over the status end-point every five seconds approximately.

The VR_FX3_RESET command simply hard-resets the device.

VR_STATUS sends back some information on the run-time firmware status. Its returned message currently consists of eight bytes, divided as follows:

Byte	Content
0-3	the current time-stamp
4	whether the application is running (end-points successfully enabled or not)
5	the currently set log level
6	the number of log messages that could not be delivered
7	the current USB connection speed

Table 4.3: VR_STATUS response format

VR_SUPPORTED instead contains information about some of the compile-time flags that influence which optional features are available. It currently consists of the following seven bytes:

Byte	Content
0	32 bit GPIF2 interface enabled instead of 16 bit
1	I2C support enabled
2	SPI support enabled
3	GPIO support enabled
4	Custom Vendor Requests defined
5	Custom DMA processing defined on the IN path (from USB to FX3)
6	Custom DMA processing defined on the OUT path (from FX3 to USB)

Table 4.4: VR_SUPPORTED response format

4.8 Custom Vendor Requests

It is possible to add new, custom vendor requests for each different device, so as to provide support for specific features or react differently to certain requests. This can be achieved by specifying a particular handler function, whose signature is shown below, that then receives all the relevant information about the request being handled, as well as enabling the appropriate #define. It is the first vendor request handler being called, and can as such also override the common vendor requests from section 4.7 if so desired, though it is highly discouraged. All useful vendor request information is passed along to the function, already split up, and the glEP0Buffer global buffer can be used to read or write data with the appropriate functions.

```
#define DEVICE_SPECIFIC_VENDOR_REQUESTS (0)

CyBool_t CyFxHandleCustomVR_DeviceSpecific(uint8_t bDirection,
    uint8_t bRequest, uint16_t wValue, uint16_t wIndex, uint16_t
    wLength);
```

4.9 GPIO

Support for General-Purpose Input/Output (GPIO) control represents the backbone of controlling external hardware, by giving fine-grained control over several pins and wires: the current value can either be sensed (input), or set to on/off (output), for specified periods of time. This is used to send or receive single bits of information to connected peripherals, such as controlling LEDs, or resetting an external sensor, or reacting to some interrupt from outside. The VR_GPIO_GET and VR_GPIO_SET vendor requests are responsible for, respectively, reading the current value of a GPIO pin and setting the value to something supplied by the user. The GET request simply expects the wValue field to be set to the ID of the GPIO one wants to query, and sends back one byte containing the current status. The SET request also expects the wValue field to contain the GPIO ID, but further accepts various modes to be passed through the wIndex field:

Value	Mode	Description
0	OFF	turns the specified GPIO off
1	ON	turns the specified GPIO on
2	TOGGLE	turns the specified GPIO on and off again
3	TIMED	turns the specified GPIO on, waits for the amount of time (in milliseconds) specified in the first two bytes of the data portion (in network order), then turns it off again
4	RECURRING	sets up a timed, recurring GPIO switch; the first two bytes of the data portion (in network order) specify the time in milliseconds between on/off cycles. Useful for flashing LEDs or resetting timers. See the heartbeat documentation in section 4.10 to understand the limitations of the timer used for this.

Table 4.5: VR_GPIO_SET modes

A list of which GPIO IDs [7] are available depending on the configuration can be found in Appendix A. All requests are checked internally against this list, so it's impossible to set invalid GPIO pins via USB vendor requests. GPIO support needs to be enabled in the device configuration, and an initial configuration has to be supplied, so that the firmware can know which GPIO pins are used in which mode, and then initialize them appropriately, and later check that accesses to them are valid. As shown with the following code, GPIO configuration consists of an array of small structs, each containing a valid GPIO ID and the corresponding type setting (refer to table 4.6 for a list of all possible type settings). Further, the length of the array has to be supplied in the appropriately named variable. The configuration array is meticulously checked for invalid values at start-up and the firmware refuses to boot if any misconfiguration is detected. The values must be those indicated in the valid GPIO list, there can be no duplicates, and the types must be correct.

```
#define GPIO_SUPPORT_ENABLED (0)

typedef struct {
    const uint8_t gpioId;
    const uint8_t gpioType;
} gpioConfig_DeviceSpecific_Type;

extern gpioConfig_DeviceSpecific_Type gpioConfig_DeviceSpecific
    [];
extern const uint8_t gpioConfig_DeviceSpecific_Length;

void CyFxHandleCustomGPIO_DeviceSpecific(uint8_t gpioId);

#define GPIO_DEBUG_LED_ENABLED (0)
#define GPIO_DEBUG_LED_NUMBER (0xFF)
```

For interrupt type GPIOs (up to 24 of them can be configured as such), the CyFx-HandleCustomGPIO_DeviceSpecific() function has to be defined: each time an interrupt is detected on an appropriately configured input GPIO, this function gets called and the ID of the GPIO that generated the event is supplied. The user can then react to it inside that function in any way he sees fit, since, through a mechanism called event-flags, the function is called from the main firmware thread, and not from the internal interrupt handler, which would not have provided any useful access to send data over USB etc. from inside the restricted interrupt handler context.

It is further possible to enable and specify a GPIO ID as a debug LED, it will then be toggled every time an error message is sent over the status end-point. This GPIO has to be configured as an output before-hand.

Another useful feature that was implemented is automatic transformation of active high and low signals. Electrical signals represent bits by their voltage: a low voltage represents 0 and a high voltage represents 1. This is called an active-high signal: the 1 is the high voltage. Many devices on the other hand work with or expect also active-low signals: the meaning is inverted, and the logical 1 is represented by the low voltage. To free the user from having to manually figure out what each signal needs to be set to in order for it to be active, the GPIO configuration abstracts the concept of voltage away and the user simply has to know that ON means logical 1 and OFF means logical 0. The firmware converts the values automatically internally, thanks to the supplied initial GPIO configuration's type setting: if the type is written in upper-case, it means the GPIO in question shall be active-high, if the type is written in lower-case, it shall be active-low. Once this initial configur ation is done, the user doesn't have to care about voltage meanings anymore, but can simply work with the logical meaning of the signals (ON/OFF), trusting that the firmware will do the right conversions.

Value	Description
'O', 'o'	simple output (has to be set manually)
'I', 'i'	simple input (has be queried manually)
'P', 'p'	pos-edge interrupt, triggers every time the GPIO transitions from OFF to ON
'N', 'n'	neg-edge interrupt, triggers every time the GPIO transitions from ON to OFF
'B', 'b'	both-edge interrupt, triggers on both pos-edge and neg-edge transitions
'L', 'l'	low interrupt, triggers when the GPIO is in OFF state
'H', 'h'	high interrupt, triggers when the GPIO is in ON state

Table 4.6: GPIO configuration: types

4.10 Heartbeat

The heartbeat system provides a general framework for executing functions at defined time intervals. This was developed as a general solution to implement flashing LEDs or keep-alive messages, for which it is currently used in the firmware. Two functions enable the programmer to add or remove functions, with a 32 bit input argument, from the heartbeat system. When adding a function, a 16 bit value specifies the time interval in milliseconds between calls to the function. For performance and resource consumption reasons, the interval will be converted to the nearest multiple of 512 internally, so about 0.5 seconds of precision are given, called timer ticks (one timer tick = 512 milliseconds). This functionality is intended for long running, repeating tasks, and not for hard-real-time tasks or tasks which require a high degree of precision in the timer.

```
CyU3PReturnStatus_t CyFxHeartbeatFunctionAdd(void (*func)(
    uint32_t input), uint32_t input, uint16_t timer);
void CyFxHeartbeatFunctionRemove(void (*func)(uint32_t input),
    uint32_t input);
```

4.11 I2C Bus

Inter-Integrated Circuit (I2C) is a serial data bus invented by Philips in the eighties and standardized in the nineties. It provides support for multiple master devices addressing several slave devices over a two wire bus. The two wires are Serial Data (SDA) and Serial Clock (SCL), the clock can run at 100 KHz, 400 KHz or 1 MHz. In the case of the FX3

I2C support, only one master can be present: the FX3 itself. Slave devices are usually selected with a 7 bit address, meaning that up to 128 different devices could be connected to the bus theoretically. The master initiates communication by sending a START bit, followed by the 7 bit address, followed by 1 bit indicating whether it wants to read (1) or write (0). If the indicated slave exists, it will respond with an ACK bit, and the two will then exchange data as indicated.

To easily support the configuration of multiple devices, the firmware requires an array of structs to be populated with the relevant information on the connected slave devices. As most I2C devices are either EEPROMs or provide an EEPROM-like interface for sending commands and reading/writing data, special support was included to simplify working with them, by being able to specify the length of the memory address that's sent to the device, the page size for optimal performance, and the total memory size of the device. In detail, the required information consists of the slave's address (7 lower bits, MSB has to be zero), the length in bytes of the memory address, the page size (which has to be a multiple of two) and the total size of the device, again both in bytes, as well as the maximum frequency in Hertz the device can operate on. This last piece of data is used to compute the highest common frequency at which all devices connected to the bus can operate, which is then used as default D.

```c
#define I2C_SUPPORT_ENABLED (0)

typedef struct {
    const uint8_t deviceAddress;
    uint8_t addressLength;
    uint16_t pageSize;
    const uint32_t deviceSize;
    const uint32_t maxFrequency;
} i2cConfig_DeviceSpecific_Type;

extern i2cConfig_DeviceSpecific_Type i2cConfig_DeviceSpecific[];
extern const uint8_t i2cConfig_DeviceSpecific_Length;
```

On the USB side, two vendor requests are provided to interact with I2C devices from the host: VR_I2C_CONFIG and VR_I2C_TRANSFER. The configuration request sets up all necessary parameters for the subsequent transfer requests, enabling several checks to be done on them to guarantee correctness and performance. The lower byte of wValue contains the address of the device to talk to, the upper byte specifies the length in bytes of the memory address that can be supplied later on in transfer calls (maximum length is four bytes, zero means to not change the default value coming from the configuration array above). The wIndex field contains the 16 bit page-size value, which has to either be a power of two (up to the maximum transfer size, usually 4096 bytes), or zero for no change. Usually, if the configuration array above was filled correctly, the latter two values should always be zero.

The VR_I2C_TRANSFER request performs either a read or a write from the configured I2C device [11], depending on its direction. The wValue and wIndex fields are concatenated to form a memory address of up to 32 bits (upper 16 bits wValue, lower 16 bits wIndex), from which up to wLength bytes of data are going to be either read or written. As already mentioned, particularities of EEPROM memory such as how to split a big request into appropriate pieces to take advantage of the memory's page-size alignment, or the length or format of the I2C commands to request a read or write at a particular address, are automatically taken care of by the firmware.

4.12 SPI Bus

The Serial Peripheral Interface (SPI) bus is a synchronous serial bus that is able to operate in full-duplex mode. It follows the master/slave model, where the master initiates the data exchange with the slave. Multiple slaves are supported by providing multiple slave-select lines, that can be toggled to enable a particular slave to receive commands. The standard implementation consists of four wires: Serial Clock (SCLK, from master), Master Output/Slave Input (MOSI, from master), Master Input/Slave Output (MISO, from slave) and Slave Select (SS, from master). No explicit address is used as it was with I2C to select a slave, but instead a slave device is selected by toggling its SS line. The higher frequency of up to 100 MHz enables much faster data exchange than is possible with I2C, as well as full-duplex exchange, albeit by using more wires (four, plus one more for each additional slave in the simplest configuration). The FX3 supports being a SPI master at frequencies up to 33 MHz and provides out-of-the-box support for one SS line and thus one connected slave device, using the provided SPI access API.

Given the above advantages of the SPI bus, it is often used in the industry, especially for accessing Flash memory, or for sending configuration to FPGAs, both of which are concrete use cases for the firmware presented here. Like I2C, SPI is an optional feature that has to be enabled and configured, again by supplying an array of structs that contain the needed information about connected SPI devices. The device address field here contains either a zero to indicate to use the default SS line, or a valid GPIO ID that can be used for additional SS lines if multiple devices are to be supported. The default SS line has to always be used (else we'd be wasting a pin!), and the additional GPIO IDs can't be used or configured in the GPIO sub-system, to avoid interference between the two. Of course, no duplicates are allowed either. The address length, page-size and device size fields have the same meaning and format as the I2C ones (see section 4.11), as they are used for the sa me special, enhanced handling of read/write requests to memory-like devices. The maxFrequency field specifies the maximum frequency in Hertz at which the device in question can operate, so that the bus can be set to the highest common frequency supported by all devices D. The SSPolarity field indicates the polarity of the SS line: TRUE means active-high, FALSE active-low.

```
#define SPI_SUPPORT_ENABLED (0)

typedef struct {
    const uint8_t deviceAddress;
    uint8_t addressLength;
    uint16_t pageSize;
    const uint32_t deviceSize;
    const uint32_t maxFrequency;
    const CyBool_t SSPolarity;
} spiConfig_DeviceSpecific_Type;

extern spiConfig_DeviceSpecific_Type spiConfig_DeviceSpecific[];
extern const uint8_t spiConfig_DeviceSpecific_Length;
```

Four USB vendor requests are provided to interact with SPI devices. VR_SPI_CONFIG is similar to the I2C configuration command, it sets up all necessary parameters for the subsequent transfer requests, enabling several checks to be done on them to guarantee performance and correctness. The lower byte of wValue contains the address of the device to talk to (its SS line), the upper byte specifies the length in bytes of the memory address that can be supplied later on in transfer calls (maximum length is four bytes, zero means to not change the default value coming from the configuration array above). The wIndex field contains the 16 bit page-size value, which has to either be a power of two (up to the maximum transfer size, usually 4096 bytes), or zero for no change. Further, up to wLength provided bytes of data (up to a maximum of 256) are stored so as to support arbitrary length SPI command sequences to be sent, for devices which do not follow memory-device like conventions.

The VR_SPI_CMD request enables arbitrary commands to be sent to an SPI device [9]. If any command was previously set with the VR_SPI_CONFIG request, it will be sent to the device, accompanied by either writing the data sent with this request (wLength bytes), or reading up to wLength bytes back from the device. It is both possible to inhibit the sending of the command by setting the wIndex field to one, or to inhibit the sending of any additional data by setting wLength to zero. This enables sending commands without any subsequent data, or one command, followed by multiple data requests, for example in case the size of the data to be transmitted exceeds 65 KB. The type of the request, read or write, is determined by its direction.

VR_SPI_TRANSFER was specifically created to ease handling of SPI Flash memory devices [12], which offer higher storage capabilities than I2C EEPROMs at higher transfer rates, and are usually preferred for storing data, such as the firmware itself, to enable the device to boot on its own. It performs either a read or a write from the SPI device, depending on its direction. The wValue and wIndex fields are concatenated to form a memory address of up to 32 bits (upper 16 bits wValue, lower 16 bits wIndex), from which up to wLength bytes of data is going to be either read or written as directed. Particularities of Flash memory such as how to split a big request into appropriate pieces to take advantage of the memory's page-size alignment, or the length or format of the common SPI commands to request a read or write at a particular address from Flash memory, are automatically taken care of by the firmware. Also protocol details such as sending write-enable commands before any write and selecting/deselecting the SS line are taken care of internally.

VR_SPI_ERASE is another, complementary request to VR_SPI_TRANSFER, specifically for Flash memory devices. A Flash memory device needs to be erased before it can be written to, and this erasing is always done in big chunks (blocks) of memory, for performance reasons. The wValue and wIndex fields are concatenated to form a memory address of up to 32 bits (upper 16 bits wValue, lower 16 bits wIndex), at which a 64 KB block of memory will then be erased. Please note that an aligned 64 KB block of memory will be erased, meaning that if the address is not 64 KB aligned, it will erase memory both before and after the supplied address, as Flash memory chips then usually align the address to the lower 64 KB block internally.

4.13 Custom Initialization

When, at boot-up [5], the firmware is loaded and initialized, support is provided for additional, device-specific initialization steps to be performed through the following function. The function is called after all serial interfaces have been brought up, and can thus be used for example to load data from Flash memory or EEPROMs. The USB support on the other hand is not yet initialized, so there can be no data exchange with the host performed at this stage. If enabled, CY_U3P_SUCCESS must be returned on successful execution, or the firmware will not continue initialization.

```
#define DEVICE_SPECIFIC_INITIALIZATION (0)

CyU3PReturnStatus_t CyFxHandleCustomINIT_DeviceSpecific(void);
```

4.14 Implementing a Device

Now that all features of the framework have been explained, it's time to see how a specific device can be implemented within it. It's a very simple process; at its most basic, it consists of the following steps:

1. Navigate to the devices/ directory.
2. Create a new directory named like the device you want to implement (convention: lower-case!).
3. Inside it, create two files, named after the device, with suffixes _config.h and _config.c (again, convention: lower-case!). The header file will contain the needed #defines, and the C file will contain the source code defining the required functions as explained in the sections above. Make sure to include the proper #define guards, as shown in the provided examples. #defines are by convention, always upper-case!
4. Add your device to the devices list, in devices/fx3_select.h, by adding the correct defines and including the right header file.

Three working devices are already provided and well commented, and should definitely be looked through to understand how to configure the firmware and all the offered functionality.

The example/ directory contains a minimal example, suitable as a starting point for new devices, by just copying and renaming the appropriate parts, and then changing and expanding as desired. It is based on the FX3 development kit board (3.2), and provides support for accessing the 4 Mbit Flash memory present on the kit, as well as toggling some GPIOs. It also shows how to set the Serial Number to some static value.

The Src_Sink project is again based on the development kit board and provides access to the integrated Flash memory, as well as defining the MS Feature Descriptor flag to support automatic driver installation. It further implements the custom DMA overrides mentioned in section 4.6, to provide a platform for USB transfer performance testing: all traffic sent to the device is dropped right away (the sink), and the device permanently sends as much traffic as possible to the host (the source).

SBRet10FX3 is the current implementation to support the latest INI sensors, it is based on the actual knowledge of the board design and its data-sheets. It provides access to an 8 Mbit SPI Flash memory [12], on which to store the firmware, as well as the configuration bit-stream for the FPGA and other data, such as the Serial Number. See Appendix C for a detailed map of how the available memory is split up to support the various requirements. Support for a Lattice ECP3-17 FPGA [8] is also present, so that it can be configured over SPI [9] by being an additional slave device. Its configuration can be either supplied via USB, thanks to custom vendor requests, or it can be loaded at initialization time from the Flash memory. Further, an Inertial Measurement Unit (IMU) [10] is connected to the I2C bus, to provide acceleration and gyro data. This data can either be directly accessed or sent out automatically when it changes, thanks to GPIO interrupt handling. Multiple GPIO lines are used for various purposes, from controlling the sensor's behavior to flashing LEDs. Last but not least, the MS Feature Descriptor feature is also enabled to support automatic driver installation on the host.

Chapter 5

jAER Integration

jAER is an open-source real time sensory-motor processing framework for event-based sensors and systems written in Java, available at `http://sourceforge.net/projects/jaer/`. It receives data from various sensors and processes it in a multitude of ways, such as showing the results to the users graphically, or by controlling servo motors.

An experimental, proof-of-concept only implementation of the new hardware interface, based on the support already present for FX2 devices, was developed (cypressfx3 and retina3 packages, available at `https://dev.longi.li/UZH/browser/UZH_SoPro_Cypress_FX3/jaer/`), using the Thesycon USBIO 5.1 library, to test basic functionality such as flashing new firmware, as well as cleaning up and re-factoring the implementation a little. Due to the unavailability of hardware, deeper testing was not possible. Further, the USBIO library is being faded out in favor of the open-source libusbx alternative, and the jAER code-base is being rewritten to support new requirements and features. Both of these facts invalidate any further development of the experimental integration, which will have to be re-done once the new technologies and APIs stabilize.

5.1 USBIO Library

The Thesycon USBIO library [13] is a commercial Windows-only driver and library combination that provides access to USB devices via a C interface, with support also for C# and Java. Windows 8 is supported in the newer versions, and USB 3.0 works since the abstraction provided over USB is at a high enough level (the drivers themselves regulate the connection speed). Advanced USB 3.0 features are not yet supported (streams, getSpeed(), BOS descriptors, ...). Open-source alternatives exist in the excellent libusbx (`http://libusbx.sourceforge.net/`, [2]) C library, which provides a unified API on Windows, Linux and Mac OS X, and fully supports all USB 3.0 features starting with release 1.0.16. Further, the usb4java (`https://github.com/kayahr/usb4java`) project provides a Java-JNI wrapper around the libusbx library for direct access to USB devices from the Java language.

Chapter 6

Summary

Even with the original deadlines sliding far into the future, all the initial goals of the project were met successfully, and the new software provides a clear framework that should last long into the future, enabling easy and dynamic expansion to new sensor hardware as needed. This was one of the most interesting software projects I've ever worked on, mostly due to the breadth and variety of involved tasks and technology: from looking over the shoulders of printed circuit-boards (PCBs) and chip designers, to discover how the hardware is connected, and how certain tasks are implemented on FPGAs using Hardware-Description Languages (HDLs, such as VHDL or Verilog), so that the firmware, written in C, could connect to the devices, and send the collected data on to the host computer, running the jAER Java framework to process it. It really gave a comprehensive view of all stages of device development, from silicon to seeing the data in running software, and everything in-between.

6.1 Lessons Learned

During the course of the project, much was learned about the technologies involved, relevant to the embedded computing world. At the same time, some important lessons from an organizational point of view were understood, which I'd like to summarize here:

- Never plan projects during time slots when other, intensive engagements are already present (exams, military service).
- Be more generous with self-determined dead-lines. More time can only help, within reason.
- Write the report already during the coding phase as an integral part of it, not afterwards as a consequence.
- Communicate more with the various stakeholders (being physically closer to them helps; workplace should preferably not be at home, but in an office).

Appendix A

Pin Assignment

Table A.1: FX3 firmware expected pin assignment

FX3 IO	FX3 DVK Pin	16-bit SlvFIFO	32-bit SlvFIFO	Direction
GPIO [0]	68	DATA 0	DATA 0	IN / OUT
GPIO [1]	74	DATA 1	DATA 1	IN / OUT
GPIO [2]	76	DATA 2	DATA 2	IN / OUT
GPIO [3]	70	DATA 3	DATA 3	IN / OUT
GPIO [4]	50	DATA 4	DATA 4	IN / OUT
GPIO [5]	80	DATA 5	DATA 5	IN / OUT
GPIO [6]	32	DATA 6	DATA 6	IN / OUT
GPIO [7]	42	DATA 7	DATA 7	IN / OUT
GPIO [8]	66	DATA 8	DATA 8	IN / OUT
GPIO [9]	89	DATA 9	DATA 9	IN / OUT
GPIO [10]	61	DATA 10	DATA 10	IN / OUT
GPIO [11]	79	DATA 11	DATA 11	IN / OUT
GPIO [12]	71	DATA 12	DATA 12	IN / OUT
GPIO [13]	81	DATA 13	DATA 13	IN / OUT
GPIO [14]	65	DATA 14	DATA 14	IN / OUT
GPIO [15]	83	DATA 15	DATA 15	IN / OUT
GPIO [16]	44	PCLK	PCLK	IN / OUT
GPIO [17]	33	SLCS	SLCS	IN
GPIO [18]	51	SLWR	SLWR	IN
GPIO [19]	55	SLOE	SLOE	IN
GPIO [20]	31	SLRD	SLRD	IN
GPIO [21]	60	SOCK 0 FULL	SOCK 0 FULL	OUT
GPIO [22]	39	SOCK 0 PART FULL	SOCK 0 PART FULL	OUT
GPIO [23]	45	SOCK 1 EMPTY	SOCK 1 EMPTY	OUT
GPIO [24]	37	PKTEND	PKTEND	IN
GPIO [25]	49	SOCK 1 PART EMPTY	SOCK 1 PART EMPTY	OUT
GPIO [26]	41	GPIO	GPIO	IN / OUT
GPIO [27]	43	GPIO	GPIO	IN / OUT
GPIO [28]	54	A 1	A 1	IN

GPIO [29]	57	A 0	A 0	IN
GPIO [30]	36	PMODE 0	PMODE 0	IN
GPIO [31]	46	PMODE 1	PMODE 1	IN
GPIO [32]	30	PMODE 2	PMODE 2	IN
GPIO [33]	56	GPIO	DATA 16	IN / OUT
GPIO [34]	85	GPIO	DATA 17	IN / OUT
GPIO [35]	73	GPIO	DATA 18	IN / OUT
GPIO [36]	87	GPIO	DATA 19	IN / OUT
GPIO [37]	77	GPIO	DATA 20	IN / OUT
GPIO [38]	75	GPIO	DATA 21	IN / OUT
GPIO [39]	69	GPIO	DATA 22	IN / OUT
GPIO [40]	67	GPIO	DATA 23	IN / OUT
GPIO [41]	78	GPIO	DATA 24	IN / OUT
GPIO [42]	72	GPIO	DATA 25	IN / OUT
GPIO [43]	63	GPIO	DATA 26	IN / OUT
GPIO [44]	58	GPIO	DATA 27	IN / OUT
GPIO [45]	TP 13	GPIO	GPIO	IN / OUT
GPIO [46]	38	GPIO	DATA 28	IN / OUT
GPIO [47]	62	GPIO	DATA 29	IN / OUT
GPIO [48]	40	GPIO	DATA 30	IN / OUT
GPIO [49]	64	GPIO	DATA 31	IN / OUT
GPIO [50]	95	GPIO	GPIO	IN / OUT
GPIO [51]	93	GPIO	GPIO	IN / OUT
GPIO [52]	91	GPIO	GPIO	IN / OUT
GPIO [53]	COM port	SPI_CLK	GPIO	OUT
GPIO [54]	COM port	SPI_SSN	GPIO	OUT
GPIO [55]	COM port	SPI_MISO	GPIO	IN
GPIO [56]	COM port	SPI_MOSI	GPIO	OUT
GPIO [57]	90	GPIO	GPIO	IN / OUT
GPIO [58]	88	I2C-SCL	I2C-SCL	OUT
GPIO [59]	86	I2C-SDA	I2C-SDA	IN / OUT
int#	52	INT	INT	IN / OUT
reset#	59	RESET	RESET	IN

Appendix B

VR Assignment

Table B.1: Vendor Request assignment map

VR Byte	Name	Description
A0	VR_FIRMWARE	Read/Write firmware to Cypress FX3 RAM and execute it. ONLY ON BLANK DEVICES!
A1	FX3 BOOT RESERVED	
A2	FX3 BOOT RESERVED	
A3	FX3 BOOT RESERVED	
A4	FX3 BOOT RESERVED	
A5	FX3 BOOT RESERVED	
A6	FX3 BOOT RESERVED	
A7	FX3 BOOT RESERVED	
A8	FX3 BOOT RESERVED	
A9	FX3 BOOT RESERVED	
AA	FX3 BOOT RESERVED	
AB	FX3 BOOT RESERVED	
AC	FX3 BOOT RESERVED	
AD	FX3 BOOT RESERVED	
AE	FX3 BOOT RESERVED	
AF	VR_MS_FEATURE_DSCR	Microsoft OS Feature Descriptor request for advanced functionality (WCID, ...).
B0	VR_TEST	Read test byte sequence from Control pipe (EP0) and write it back to the Status pipe (EP1).
B1	VR_LOG_LEVEL	Set log level to wValue.
B2	VR_FX3_RESET	Perform a cold-reset of the Cypress FX3 device.
B3	VR_STATUS	Return various status information (like USB speed, log_level, ...).
B4	VR_SUPPORTED	Return information on supported firmware features.
B5	VR_GPIO_GET	Read GPIO wValue and get back the state.

B6	VR_GPIO_SET	Execute operation wIndex on GPIO wValue (supported are OFF, ON, TOGGLE, TIMED, RECURRING).
B7	VR_I2C_CONFIG	Configure I2C device.
B8	VR_I2C_TRANSFER	Read/Write from I2C device.
B9	VR_SPI_CONFIG	Configure SPI device, set command to execute with VR_SPI_CMD.
BA	VR_SPI_CMD	Send any command to SPI device and read/write result (more general than VR_SPI_TRANSFER).
BB	VR_SPI_TRANSFER	Read/Write from SPI memory-like device.
BC	VR_SPI_ERASE	Erase a 64KB block on SPI memory-like device.
BD	VR_BIAS	Write Bias configuration to chip.
BE	VR_FPGA_CONFIG	Configure FPGA via USB, by loading a new bitstream into it.

Appendix C

SPI Memory Map

Table C.1: SPI Flash memory map and notes

32 KByte Blocks	Address (bytes)	Usage	Notes
1	00000	FX3 Firmware Image.	Directly written here, no
2	08000	(192 KB)	preamble, to boot from.
3	10000		
4	18000		
5	20000		
6	28000		
7	30000	FPGA Bitstream.	8 byte preamble:
8	38000	(576 KB)	4 byte string "FPGA" +
9	40000		4 byte length (LE)
10	48000		then bitstream as
11	50000		exported by tools.
12	58000		
13	60000		
14	68000		
15	70000		
16	78000		
17	80000		
18	88000		
19	90000		
20	98000		
21	A0000		
22	A8000		
23	B0000		
24	B8000		
25	C0000	Data.	Each piece of data has
26	C8000	(256 KB)	an 8 byte preamble:
27	D0000		4 byte string identifier +
28	D8000		4 byte length (LE)
29	E0000		then the data as specified.

30	E8000		Memory regions have to be
31	F0000		reserved by marking them
32	F8000		in the SPI Memory Map doc.

Appendix D

Clock Frequencies

Clock	Frequency	Notes
SYS_CLK	400 MHz	
CPU	200 MHz	divider on SYS_CLK, 2 to 16, default 2
DMA	100 MHz	divider on CPU, 2 to 16, default 2
MMIO	100 MHz	divider on CPU, 2 to 16, default 2
PIB (GPIF2)	100 MHz	divider on SYS_CLK, 2 to 1024, default 4
FAST_CLK (GPIO)	100 MHz	for Complex GPIO, divider on SYS_CLK, 2 to 16, default 4
SLOW_CLK (GPIO)	-	for Complex GPIO, divider on FAST_CLK, 2 to 64, 0 to disable, default 0
SIMPLE_CLK (GPIO)	50 MHz	for Simple GPIO, divider on FAST_CLK, 2/4/16/64, default 2
I2C	1 MHz	freely configurable 100 KHz/400 KHz/1 MHz, default 1 MHz or highest device
SPI	33 MHz	freely configurable 10 KHz-33 MHz, default 33 MHz or highest device

Table D.1: Clock Frequencies

Bibliography

[1] Sonia Gandhi, *AN65974 - Designing with the EZ-USB FX3 Slave FIFO Interface*, Rev. G

[2] Dhanraj Rajput, *AN73609 - EZ-USB FX2LP/FX3 Developing Bulk-Loop Example on Linux*, Rev. A

[3] Sonia Gandhi, *AN75705 - Getting Started with EZ-USB FX3*, Rev. A

[4] Rama Sai Krishna V, *AN76348 - Migrating from EZ-USB FX2LP Based Design to EZ-USB FX3 Based Design*, Rev. -

[5] Sonia Gandhi, *AN76405 - EZ-USB FX3 Boot Options*, Rev. B

[6] Cypress Semiconductor, *CYUSB3KIT-001 - EZ-USB FX3 Development Kit Guide*, Rev. B

[7] Cypress Semiconductor, *CYUSB3014 - EZ-USB FX3 SuperSpeed USB Controller*, Rev. L

[8] Lattice Semiconductor, *DS1021 - LatticeECP3 Family Data Sheet*, Version 02.2EA, April 2012

[9] Lattice Semiconductor, *TN1222 - LatticeECP3 Slave SPI Port User's Guide*, May 2013

[10] InvenSense Inc., *MPU-6000 and MPU-6050 Product Specification*, Rev. 3.3

[11] InvenSense Inc., *MPU-6000 and MPU-6050 Register Map and Descriptions*, Rev. 4.0

[12] Winbond, *W25Q80BW - SPIFlash 1.8V 8M-Bit Serial Flash Memory with Dual and Quad SPI*, Rev. 1

[13] Thesycon GmbH, *USBIO - USB Software Development Kit for Windows - Reference Manual*, Version 2.71

[14] Craig Peacock, *http://www.beyondlogic.org/usbnutshell/usb6.shtml#SetupPacket*, 24.06.2011