SOFTWARE ENGINE DEVELOPMENT FOR SDR

Magdalena Sánchez Mora (CTVR, IMWS, NUI Maynooth, Kildare, Ireland, msanchez@eeng.nuim.ie); Ignacio Ruiz (CTVR, IMWS, NUI Maynooth, Co. Kildare, Ireland, iruiz@eeng.nuim.ie); Gerard Baldwin (CTVR, IMWS, NUI Maynooth, Co. Kildare, Ireland, gbaldwin@eeng.nuim.ie); Ronan Farrell (CTVR, IMWS, NUI Maynooth, Co. Kildare, Ireland, rfarrell@eeng.nuim.ie)

1. ABSTRACT

This paper focuses on the development of the software engine for an SDR hardware platform [1][2]. This SDR hardware system operates across the frequency band from 1.6GHz to 2.5GHz with the capability to support the GSM1800, PCS 1900, UMTS-FDD, UMTS-TDD and 802.11b standards. It consists of TX/RX RF front-ends, data converters and the USB 2.0 PHY interface.

2. INTRODUCTION

Software Defined Radio gains more interest with the increasing number of communication standards and the associated requirements for individual base stations with specific needs. In traditional wireless devices, most radio functionality is implemented in hardware. The SDR solution has the advantage of reconfigurability; the base station can implement many different standards by software reconfiguration.

Our software engine for SDR hardware has a maximum speed of 384Mbps (24 MHz). This performance is much better than the one obtained by the Universal Software Radio Peripheral (USRP) created by the GNU Radio project. The USRP is capable of processing signal up to 16 MHz wide [3].

The software engine can be divided into three main parts each of them is executed in different software spaces:

1) User space: an application-programming interface (API) provides the configuration and data transport interfaces to the SDR hardware platform.
2) Kernel space: the USB driver allows communication between PC and SDR hardware via the USB 2.0 interface.
3) Embedded space: the firmware in the Cypress CY7C68013A chip implements the communication between the USB device and the external logic.

The three modules have been written in C language due to its low-level capabilities, speed and portability between a wide variety of PCs and operating systems.

The API integrates all digital signal processing and recovery functions, such as modulation/demodulation and carrier/timing recovery, hardware control functions and communication functions. The SDR hardware is reconfigured through the hardware control functions. These control functions communicate with the embedded software on the hardware platform to implement changes to the ADC and DAC sampling rates, and to the local oscillator frequencies and the gain in both the transmitter and receiver. In this way the performance of the hardware is under software control from the PC. The communication functions enable the connection between user-space applications and the SDR hardware platform. Adding new signal processing functions or new hardware reconfigurable functions can easily extend this API.

The second part of the software engine is the USB driver, which follows the USB 2.0 specifications and runs as a kernel-module of the Linux operating system. It implements the bulk transfer type in high-speed mode, which ideally has a maximum speed of 480Mbps. Kernel buffering mechanisms are needed in order to achieve the highest USB speed. The buffering mechanisms are also useful for caching data in kernel space, which avoids lost packets due to USB device 'not ready for next packet' errors. The kernel buffer implemented on our USB driver follows a FIFO processing scheme, the packet that comes first is sent first to the USB host controller. In summary, the information from the user-space applications is temporarily stored on the FIFO kernel buffer and sent to the USB host controller by the USB driver.

The third part is the firmware which provides a high speed communication path between USB driver and the rest of the SDR hardware components, such as data converters and local oscillator.

The result is a software engine that hides hardware specific details and provides a consistent platform to develop user-space applications for controlling SDR hardware platforms.

	User Space	*Kernel Space*	*Embedded Space*
Data Path	Data transport functions Digital signal processing & recovery signal	Read Operations Write Operations	GPIF (data path)
Control Path	Hardware control	Control Operations	I^2C (control path)

Figure 1. Software engine modules

3. USER SPACE: API

The three parts of the API can be seen on Figure 2. Digital signal processing/recovery functions and hardware control functions make use of the data transport functions in order to communicate with the USB driver at kernel space.

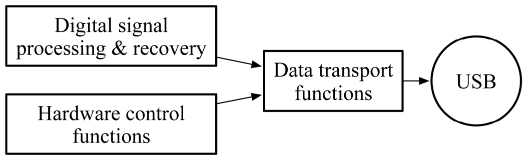
Digital signal processing & recovery
Hardware control functions
Data transport functions
USB

Figure 2. User space block diagram

3.1. Data transport functions

Three basic functions have been implemented to enable communication with the USB driver from user-space applications (digital signal processing and hardware control functions). These are *write_sock()*, *read_sock()* and *ioctl_sock()* functions. In the Linux operating system the USB device is treated as a socket, which greatly simplifies the communication process. Basic socket operations such as open, write, read and close are enough to interface with the USB driver.

By using *ioctl_sock()*, the user selects the source or destination endpoint. After that, *read_sock()* and *write_sock()* calls can be made indicating the buffer and block size of the information to be sent or received.

3.2. Digital signal processing and recovery functions

The FIR (finite impulse response) digital filter along with the 16QAM modulation scheme, m-power non data aided carrier recovery functions and early-late gate timing recovery functions are implemented in C language as part of a library of digital signal processing functions. The use of Matlab is quite limited by its low performance at real time execution speed. Consequently the C programming language is used instead; hence the bandwidth will not be limited by the user space applications.

3.3. Hardware control functions

The operating rates of the ADC and DAC, the gain of the transmitter and the automatic gain control at the receiver are some of the configurable variables at the SDR hardware from user space. As we will explain later in section 5, these configurations are performed using the I^2C bus controller.

4. KERNEL SPACE: USB DRIVER

The developed USB driver follows the USB 2.0 specification [4] and is configured for high-speed mode, maximum speed of 480Mbps. The USB driver is a loadable kernel module (LKM) which extends the running kernel capabilities of the Linux operating system to enable communication with the SDR hardware via the USB device. It is basically an object file which has been written in the C language.

It has the following characteristics:

- Bulk transfer type: this kind of USB transfer guarantees delivery of data but does not guarantee bandwidth or latency. The access to the USB will be on a bandwidth-available basis. It is mostly suitable for transferring large amounts of data at highly variable times and bandwidth.

- The maximum bandwidth for bulk transfer at high-speed mode is 53MBps (425Mbps) according to the USB 2.0 specifications.

- Alternate Settings 0 to 3: these settings define the endpoint (ep) characteristics. Our USB driver utilizes alternate setting 1 at high-speed mode, which has ep2 and ep4 as double bulk-out buffers (512bytesx2), ep6 and ep8 as double bulk-in buffers (512bytesx2), and ep1out/ep1in as 64 bytes bulk buffers (see Table 1). Notice that *in* means source and *out* destination, so the USB driver writes to ep2-ep4 and reads from ep6-ep8.

The functions implemented in this USB driver can be divided into two groups: control functions and transfer functions. The control functions allow endpoint selection from the user application. Therefore, it is possible to select the source/destination endpoint before doing any reading/writing operation from the user space.

Table 1. High-speed alternate setting 1

Alternate Setting	**1**
ep0	64
ep1out	64
ep1in	64
ep2	512 bulk out (2x)
ep4	512 bulk out (2x)
ep6	512 bulk in (2x)
ep8	512 bulk in (2x)

The transfer functions perform read/write operations over the endpoints (bulk-in and bulk-out buffers). In order to communicate between our USB driver and the USB device, we use USB request blocks (URBs)[5][6]. These are extremely useful data structures to send/receive data to/from a USB endpoint. The main advantage of using URBs is that many of them can be sent to a particular endpoint creating a queue of URBs, which is continuously processed by the USB controller improving the achieved data rate. As opposed to other procedures such as bulk messages, URBs allow our driver to achieve the highest possible data transfer speeds.

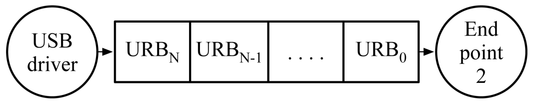

Figure 3. Queue of URBs on out-ep2

The size of each URB is limited, therefore the USB driver needs to split the user message into smaller pieces and creates a FIFO buffer in kernel space. This feature avoids packets being missed because they are too big to be sent on a single URB to the USB controller.

5. EMBEDDED SPACE: FIRMWARE

As mentioned previously, the SDR hardware consists of TX/RX RF front-ends, data converters and the USB 2.0 PHY. The Cypress EZ-USB FX2LP (CY7C68013A) chip has been selected to interface between SDR hardware and USB driver [7]. The main objectives are to provide a high speed communication path between the USB driver and the rest of the SDR hardware components, such as data converters and the local oscillator. In order to implement this behavior, the firmware software has to be designed and written for the FX2LP device. The next subsection outlines its main characteristics.

5.1. USB FX2LP Characteristics

The following figure shows the simplified architecture of the FX2LP device.

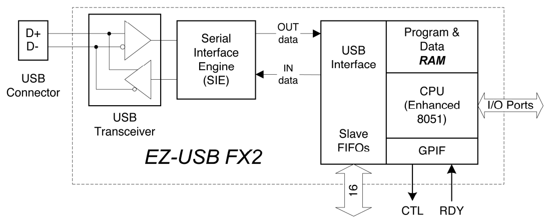

Figure 4. EZ-USB FX2LP diagram (source [7])

A brief explanation of each main part is included below:

1) USB transceiver and Serial Interface Engine (SIE).

2) Enhanced 8051 CPU running at up to 48 MHz.

3) The endpoint and interface FIFOs: double-, triple- and quad-buffered endpoint FIFOs to achieve the 480 Mbps USB data rate.

4) General Programmable Interface (GPIF): a programmable state machine which provides the highest possible bandwidth achievable over the physical layer.

Due to the fact that the standard 8051 CPU runs at 48 MHz and uses four clocks per instruction cycle, it does not participate in our high speed data path. The GPIF is used instead in order to achieve the best performance between the internal FIFOs and the external logic.

There are no speed requirements for the control communication path so the programmable I²C bus controller is utilized. In this case, the 8051 CPU participates in the communication.

5.2. GPIF: data path

The GPIF [7] is a programmable state machine which generates up to six control and nine address outputs, and accepts six external and two internal ready inputs. It is powerful enough to implement such interface such as the one between USB and an IDE hard drive. However, its flexibility comes with added complexity thus a very good understanding of the GPIF architecture and implementation is required by the developer. Figure 5 shows the input/output signals between GPIF and the external logic.

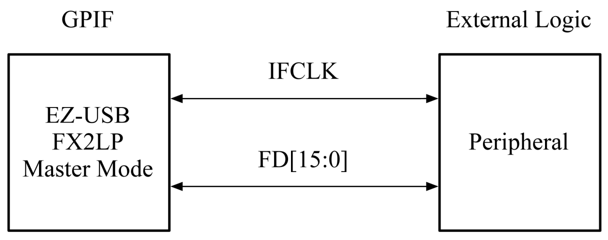

Figure 5. GPIF interfacing with the peripheral

The GPIF allows the definition of up to four user-defined waveform descriptors which control the state machine. In general, one is written for FIFO reads, one for FIFO writes, one for single reads and one for single writes. In our case, we have created only two different waveforms: one for FIFO reads and one for FIFO write. This is mainly because the low performance obtained by using single read/write operations.

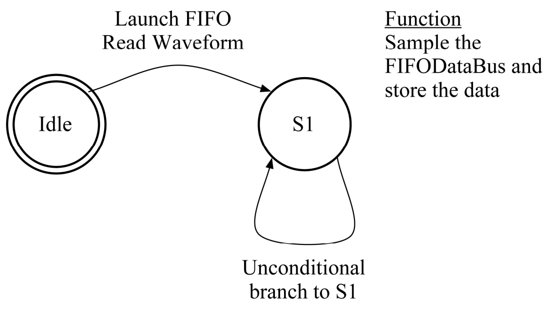

Figure 6. State diagram for FIFO read operations

The maximum performance was achieved by different embedded code versions; one for FIFO reads and one for FIFO writes. Figure 6 and Figure 7 show the state diagram for both FIFO operations. Notice that the waveforms never go to idle state, so the 8051 CPU is never involved in the communication path.

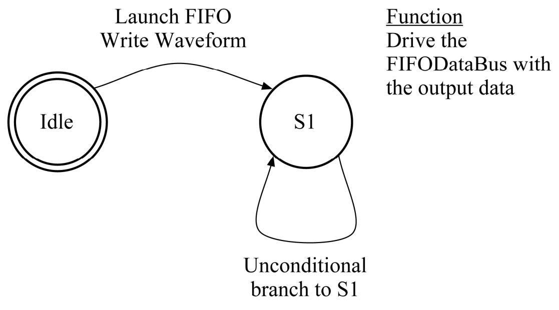

Figure 7. State diagram for FIFO write operations

In terms of the GPIF and endpoint configuration is very important to mention that the GPIF is externally clock and the endpoints are respectively in AUTOIN and AUTOOUT mode.

5.3. I²C bus: control path

I²C (Inter-Integrated Circuit) is a serial bus controller invented by Philips and is one of the communication systems provided by the EZ-USB FX2LP device. I²C uses two bidirectional lines: Serial Data (SDA) and Serial Clock (SCL) [7]. We have selected the I²C standard mode which has a speed of 100 kbps. The master is configured in *master transmit mode* and the slave in *slave receive mode*. Thus, the master is in control of the clock and is sending data to a slave in order to perform the configurations.

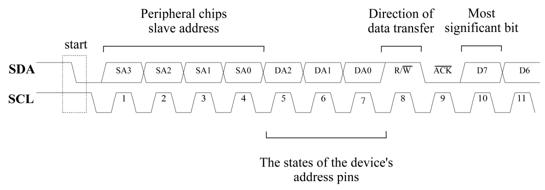

Figure 8. Master transmit mode (source [7])

The transmission starts by sending a start bit followed by the 7-bit address of the slave device, and a single bit representing write to or read from the slave device. Each peripheral device on the I²C has a unique address. After that, the slave will respond with an acknowledgment bit (ACK). Consecutive master write operations are followed by the slave ACK bit.

The configuration data to be sent across the I²C bus is obtained from the specification data sheets of each slave device (slave's address together with configuration information). This information will be sent through the hardware control functions from the user application (see section 3).

From the point of view of the embedded code, the address line, data line and bus configuration are addressed by writing on the I2CTL, I2CS and I2DAT registers. The 64-byte ep1 is the communication channel between USB driver and I²C bus. Thus, once EP1 is full, the embedded code will take (see Figure 9):

- ep1[0] as the slave's device address
- ep1[1] as the communication mode (*S* for sending individual bytes and *M* for sending blocks of data)
- ep1[2] as the number of bytes to be transmitted
- ep1[3 to 63] as the configuration data itself

After that, the serial communication starts through the I²C bus.

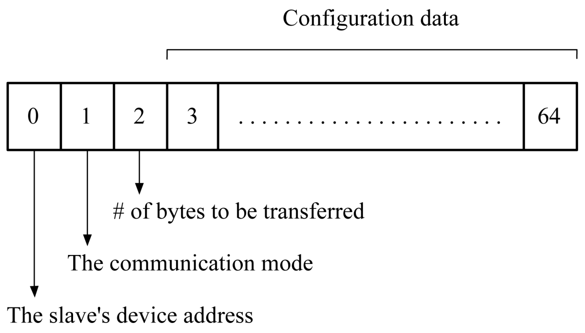

Figure 9. Enpoint 1 structure

Due to the I²C bus will be used only for short periods of time and for too few data, it will not be any speed variation through the GPIF channel.

6. CONCLUSIONS

This paper shows the description of a software engine for SDR hardware which achieves a speed of 384Mbps. This data rate corresponds which the 90% of the highest data rate that could be obtained using USB bulk transfer type. This high performance is achieved by a combination of different techniques at all levels of the software spaces: user, kernel and firmware. Even higher performance than the current USRP from GNU Radio is achieved.

The hardware performance is totally under software control from the PC side. Therefore, a software engine has been built that hides hardware specific details and provides a high speed platform for controlling SDR hardware platforms.

7. FUTURE WORK

The future work can be divided into the three main software spaces:

- The user-space applications, the library of signal processing and recovery functions will be continuously extended with more components such as different modulation schemes.

- The kernel space: a more flexible and versatile USB driver version will be developed, which will include functions such as selection of alternate settings or transfer types from user space. It is planned to develop a driver for the Windows operating system as well as an open source code driver for GNU radio software [3].This will make the great potential of our USB driver available for the wide community of GNU radio users.

- The embedded space: the next generation of the SDR hardware [8], which includes a FPGA, will be finished in the near future. Therefore, a more sophisticated state machine is needed to include the control and ready signals that interface with the FPGA. In relation to the current SDR hardware version, an improvement is to combine read and write operations in the same embedded code maintaining at least the current speed rate.

8. REFERENCES

[1] G. Baldwin, L. Ruíz, R. Farrell, "Low-Cost Experimental Software Defined Radio System", *SDR Technical Forum 2007*, 5-9 November 2007, Denver, Colorado.

[2] L. Ruíz, G. Baldwin, R. Farrell, "*Reconfigurable Radio Testbed*", Irish Signal and System Conference (ISSC'06), pp. 237-240, June 28-30, 2006.

[3] www.gnu.org/software/gnuradio/

[4] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC and Philips, *Universal Serial Bus Specification*, Revision 2.0, April 27, 2000.

[5] J. Corbet, A. Rubini, G. Kroah-Hartman, *Linux Device Drivers*, Third Edition, O'Reilly, 2005.

[6] G. Kroah-Hartman, *Linux kernel in a nutshell*, O'Reilly, 2006.

[7] Cypress Semiconductor Corporation, *EZ-USB Technical Reference Manual*, version 1.4, 2000-2006. Cypress Semiconductor Corporation, *EZ-USB® FX2™ GPIF Primer*, 2003.

[8] L. Barrandon, G. Corley, G. Baldwin, R. Farrell, "*Hardware implementation of a versatile low-cost mixed-signal platform for SDR experimentation*", *SDR Technical Forum 2007*, 5-9 November 2007, Denver, Colorado.