“Lessons learnt in developing a SDR Platform with USB interface”

Magdalena Sánchez Mora, Gerry Corley, Ronan Farrell
Centre for Telecommunications Value-Chain Research (CTVR),
National University of Ireland,
Maynooth, Ireland

{msanchez, gcorley, rfarrell@eeng.nuim.ie}

1. Introduction

Building a new Software Defined Radio (SDR) system requires multidisciplinary research covering the engineering disciplines of communication systems, radio frequency, digital and analog hardware, software and digital signal processing. This paper focuses on the efforts at the low-level software development, such as device drivers, embedded source code at firmware-space and Application-Programming Interfaces (APIs) at user-space.

In the early stages of constructing a SDR platform, design decisions are made regarding the interface between the SDR hardware and the PC. These decisions are of great importance and will determine the complexity of the low-level software development, its interoperability with third-party tools for waveform development and its efficiency in terms of bandwidth and configurability.

This position paper reviews the experiences in using a USB interface between the PC and the SDR platform and the corresponding impact in the software development stage.

2. SDR platform to PC interface

The SDR platform [1] consists of four hardware elements: a radio transmitter, a radio receiver, a baseband interface and a PC to perform signal processing and configuration. Data and control communication is performed via a USB 2.0 interface between the transceiver and a laptop PC. The USB 2.0 communication is carried out by the CY7C68013A USB microcontroller [2].

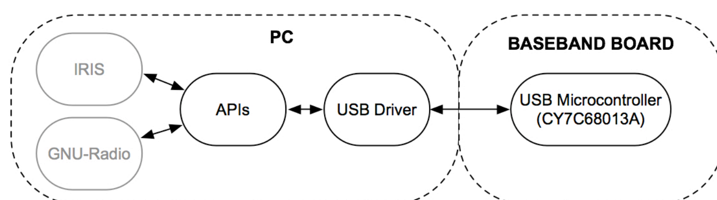

Figure 1. Software development overview

The main needs in terms of software are an optimized USB driver and embedded code running on the USB microcontroller [3] (see Figure 1). The following is a list of lessons learnt when developing these software modules:

1. Steep learning curve required to understand the Cypress USB microcontroller.

   In order to achieve the highest possible bandwidth over the physical layer, the General Purpose Interface (GPIF) of the USB microcontroller needs to be programmed. While the GPIF is very flexible it is also complex in terms of architecture and implementation, which makes its full understanding harder for the software developer.

2. Theoretical USB 2.0 throughput is affected by factors out of our control.

   Assuming a data rate of 480 Mbps is a common mistake. Operating system (OS) modules between our USB driver and the firmware in the physical device also affect the whole system performance. Such as the USB Host Controller Driver (HCD) driver and the USB core module (see Figure 2).

3. An open source USB driver template is available for the Linux OS.

   A basic USB driver can be implemented in a short period of time. Further modifications are needed to improve memory management, the multiprocess mechanism and data rate. The use of a queue of URBs (USB Request Blocks) and bulk transfer type, and the implementation of semaphores allow us to achieve the highest data rate.

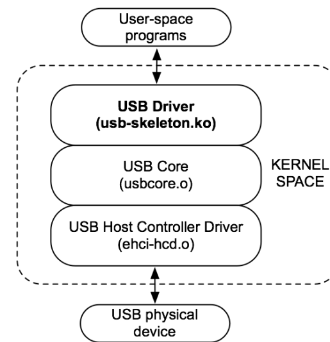

Figure 2. USB overview

4. Linux device drivers: Simplicity versus Lack of Portability.

   Linux is free and open source software, has good performance, high stability and a much simpler driver model than in the Windows OS. This is partly due to the fact that device drivers are treated as files by user-space applications in Linux, therefore to access the USB device is as easy as performing I/O file operations (open, read, write, close and ioctl). On the other hand, Linux drivers are harder to maintain because they need to be updated in accordance with the kernel changes.

5. Open source software weak in terms of reusability.

   The lack of documentation and support are some of the fundamental problems of open source software. Basing the firmware development on studying existing open source code lead to a waste of time and resources.

6. Data transfer bottlenecks in the system are difficult to detect and to anticipate.

   Since SDR needs software development at all the levels: firmware, device drivers and applications at user-space. Extensive tests regarding system integration were carried out to meet the data rate requirements.

7. Adopting software standards for our SDR platform helps us to integrate with third-party waveform development tools, such as GNU-Radio or IRIS (Implementing Radio in Software) [5].

   Software standards regarding device interface, documentation, folder hierarchy, Makefiles, APIs, general purpose, flexibility, etc.. All this will result in a more efficient, maintainable and reusable software solution.

3. Conclusions

The choice of the USB interface in our SDR platform created a number of challenges, especially when it came to achieving the highest data rate with the minimum firmware and device driver development. On the other hand, the USB interface made the integration with existing waveform development tools easier. Finally, regarding the Linux device driver development, a simple, optimum and open source driver solution was carried out for the SDR platform.

References

[1] G. Corley, M. Sánchez Mora, R. Farrell, *"Software Defined Radio Transceiver Implementation"*, RIA Colloquium, Dublin, Ireland, 2008.

[2] Cypress Semiconductor Corporation, *"EZ-USB Technical Reference Manual v 1.4"*, www.cypress.com, 2006.

[3] M. Sánchez Mora, G. Baldwin, R. Farrell, *"Software Engine Development for SDR",* SDR Technical Forum 2007, 5-9 November 2007, Denver, Colorado.

[4] S. Kolokowsky, T. Davis,*"Common USB Development Mistakes",* Cypress Semiconductor Copyright, www.cypress.com, 2006.

[5] P. MacKenzie, *"Software and reconfigurability for software radio systems"*, Ph.D. dissertation, Trinity College Dublin, Ireland, 2004.