Reconfigurable Wireless Handset Realization Based on a Universal API

BABAK D. BEHESHTI
School of Engineering and Technology
New York Institute of Technology
Old Westbury, NY11568
U.S.A.

Abstract: - Reconfigurable radio systems are radios that can change to different communication protocols as they move between different radio environments. An example would be moving from a wireless LAN 802.11b to 802.11a and then to EV-DO (Evolution Data – Optimized). Researching the development of a Reconfigurable Radio Architecture that will concurrently support multiple radio protocols over multiple frequency bands across multiple wireless networking environments is an active area of R&D in the industry. The Reconfigurable radio realizes the convergence of computing and communications by allowing a flexible communications for any handheld computing device. As more digital processing is applied to the radio system, the promise of a software based digital baseband processor controlling a reconfigurable RF front end approaches reality. The focus of this paper is the development and testing of a generic API (Application Program Interface) on a software based baseband processor to access and control any arbitrary RF front end.

Key-Words: - **Wireless, Physical layer, Baseband processing, Software Defined Radio.**

1 Introduction

Software Defined Radios (SDR) have the potential of changing the fundamental usage model of wireless communications devices. These transceivers are often conceptually divided into two major sections: the Baseband Processing Section and the RF Front End. This division is simply a matter of convenience as the technological state of the two sections are at different stages. The baseband section which is responsible for all symbol level and bit level computations is typically implemented as reconfigurable hardware architecture or a digital signal processor (DSP). The RF front end on the other hand, requires wide bandwidth filters with high selectivity throughout the band as well as highly accurate synthesizers and ADC/DACs over a large range of frequencies. While much work has been done to enable most air interfaces to be digitally implemented on signal processors, SDR's still use conventional RF designs for their front end, and thus can only operate over limited frequency ranges. The ideal SDR RF unit would be frequency-agile, tunable and could be incorporated directly into a radio to replace multiple banks of microwave sources, thus reducing the size, weight and (ideally) cost of the unit. [5]

In addition to the capability of the RF front end to configure to various air-interface standards, the RF front end is also required to quickly perform transition from one standard to another.

There are a number of commercially available configurable RF front ends available. However, currently there is no standard that has emerged for the interface between the baseband processor and the RF front end. As a result for every possible RF front end in a SDR implementation, there is the requirement for developing a custom device driver and interface software to configure and communicate with the RF front end.

In this paper we discuss the development of a generic API that can be used to unify the RF device driver interface of any application running on the baseband processing DSP platform.

2 API Architecture

The API consists of two separate sub-layers, The Application Sub-Layer that remains mostly uniform for any wireless standard (with parametric variations to accommodate different modulations and frequency ranges), and the Hardware Specific Sub-Layer that is specific to a particular hardware realization of the RF front end. The Hardware Specific Sub-Layer needs to be rewritten and ported to any new RF platform; however, the Application Sub-Layer remains the same for all applications requiring services from the RF.

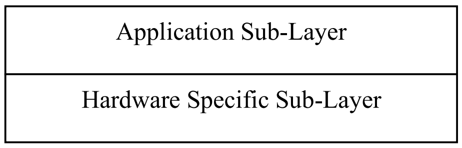


Figure 1 - Generic RF API Breakdown

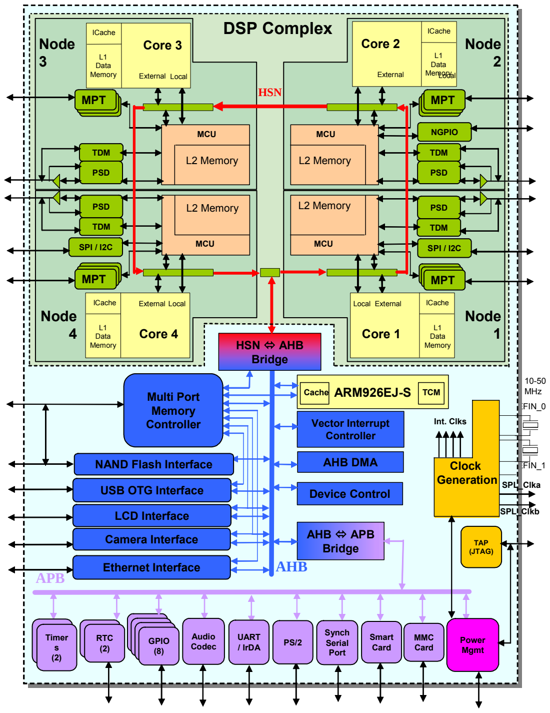
The goal of this project was to test the developed API on a specific hardware platform (currently focusing on an EV-DO RF card), as connected to a baseband processing card, and measure and analyze the transmitter and receiver characteristics via the use of this API. Once this API was tested, the application sublayer would then be reused for interfacing this baseman processor to any other RF front end.

The hardware platform to test and verify the API was the Sandbridge Technologies SB3011 configurable baseband processor. Sandbridge's flagship product is the SB3011 Flexible Baseband Processor. The SB3011 uniquely provides the capability to operate on any network and any communication protocol.
Featuring the integration of FOUR Sandblaster™ cores into a single SoC, the innovative Sandblaster™ architecture enables the SB3011 to implement the latest 3G protocols including W-CDMA, CDMA2000, and TD-SCDMA. Additionally, since the physical layers of these protocols are implemented in software, creating 'derivative' device designs is a relatively inexpensive software task, rather than a costly hardware integration effort. [2]

The SB3011 features the Sandblaster™ DSP for execution of baseband in software - including physical layer. It has a programmable RF interface, with the capability to capture raw data at 100 million samples/sec. It includes interfaces to LCD, keypad, USIM, SmartCard, Audio codec, IrDA, plus emerging 'critical' features such as add-on memory cards, camera interface, and USB. [3]

Figure 2 shows the SB3011 chip which has four SandBlaster™ DSP cores. The SB3011 has the following features:

- Low-power-consumption design
- Four SandBlaster™ DSP cores connected by a high speed bus in a ring topology
- Eight time-multiplexed hardware thread execution units for each DSP core
- SIMD/Vector operation unit



- 600MHz (1.67ns instruction cycle)

Figure 2 - SB3011 Internal Block Diagram

- 90nm CMOS process
- 32Kbytes instruction cache per core
- 64Kbytes L1 data memory per core
- 256Kbytes L2 data memory per core
- Integrated ARM processor

SB3011 DSP core has a SIMD vector operation unit which provides a parallel-execution capability of four 16-bit multiply-accumulate (MAC) operations per single instruction cycle. [7]
The peripheral devices which require high-speed access are connected with Advanced High-Performance Bus (AHB) and most peripherals are connected via Advanced Peripheral Bus (APB). Both AHB and APB are compliant with Advanced Microcontroller Bus Architecture (AMBA).

The API developed and the associated data obtained from verification of the RF front end is discussed in the following.

2 The Driver API Development

A device driver is software that abstracts the functionality of a physical or virtual device. A device driver manages the operation of these devices. Examples of physical devices are network adapters, timers, and universal asynchronous receiver-transmitters (UARTs). Implementing a device driver allows the functionality of your device to be exposed to applications and other parts of the operating system (OS). While developing a device driver, take advantage of the services provided by the OS. One of the most basic rules to remember when writing portable code is to be aware of how big one needs to make the variables. Different processors define different variable sizes for int and long data types. They also differ in specifying whether a variable size is signed or unsigned. Because of this, if one knows the variable size has to be a specific number of bits, and it has to be signed or unsigned, then one needs to use the built-in data types. The following typedefs can be used anywhere in kernel code and are defined in the linux/types.h header file:

u8	unsigned byte (8 bits)
u16	unsigned word (16 bits)
u32	unsigned 32-bit value
u64	unsigned 64-bit value

s8	signed byte (8 bits)
s16	signed word (16 bits)
s32	signed 32-bit value
s64	signed 64-bit value

All of these functions return a signed 32-bit value and take a number of values for either a data or command parameters. Because these data types are used, this code is portable to any processor type.

If the variables are going to be used in any code that can be seen by user-space programs, then one needs to use the following exportable data types. Examples of this are data structures that get passed through ioctl() calls. Once again they are defined in the linux/types.h header file:

__u8	unsigned byte (8 bits)
__u16	unsigned word (16 bits)
__u32	unsigned 32-bit value
__u64	unsigned 64-bit value

__s8	signed byte (8 bits)
__s16	signed word (16 bits)
__s32	signed 32-bit value
__s64	signed 64-bit value

3 RF API Description

The general form of the API is a series of function calls, where each function call returns an error code and receives one or more parameters. For example, the following API is designed to set the automatic frequency control setting of the RF front end. It receives a 16-bit parameter that corresponds to a number between -5000 to plus 5000 Hz. If the API is successful it will return a value of zero. Any nonzero return value corresponds to a particular error that has been identified in a header file. [6]

```
RF_Drv_Rf_Error_t Drv_Rf_Set_Afc (int afc_val)
Sets the value of AFC.
Parameters:
afc_val
Units:  Hz
Range:  -5000 to +5000 Hz (approx.)
Mapping:  <absolute gain> + 100
0 = -5000 Hz
256 = +5000 Hz

Return Value:
•        RF_DRV_RF_SUCCESS
•        Or Error code
```

Another example of an API call is two sets be out of my gain control of the transmitter. As can be seen in the example the API has a uniform and consistent format.

```
RF_Drv_Rf_Error_t       Drv_Rf_Set_Tx_Agc    (int
tx_agc_val)
Sets the value of transmit AGC.
Parameters:
tx_agc_val
Units:  dB
Range:  -60 dBm to +30 dBm (approx.)
Mapping:

Return Value:
•        RF_DRV_RF_SUCCESS
•        Or Error code
```

Contents of rf_api h are any data definitions, datatypes and function prototypes, as well as an enumeration of all return error codes. A sample of the return error codes is shown below

```
enum
{
RF_DRV_RF_SUCCESS = 0,
MAX_RF_DRV_RF_ERRORS
} RF_Drv_Rf_Error_t;

enum
```

```
{
RF_OFF_STATE,
RF_ON_STATE,
RF_MAX_STATE
} RF_On_State_t;
```

4 API Testing

The API test suites consisted of a battery of tests to test the transmitter, and another battery of tests to test the receiver.

Figure 3 shows a test configuration for the transmitter. As can be seen, a PC connected to the baseband processor board is used to send commands to exercise various transmitter APIs. The baseband processor then exercises the appropriate functionality in the RF transmitter, and a test equipment (Agilent PSA) is used to view various characteristics of the transmitted signal. As can be seen in **Figure 4** the transmitter was configured to transmit a QPSK signal at 21dBm power level, and the I/Q contents of the signal where measured on the Agilent equipment.

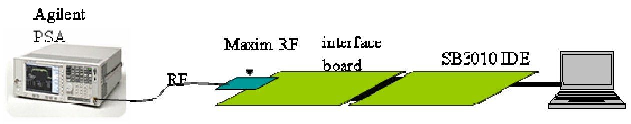
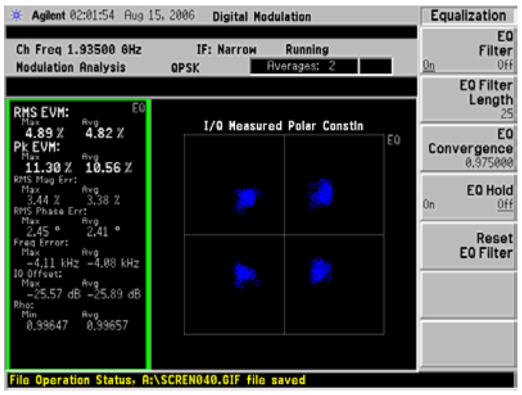


Figure 3 – Transmitter Test Suite Configuration



```
Agilent 02:01:54  Aug 15, 2006   Digital Modulation
Ch Freq 1.93500 GHz        IF: Narrow     Running
Modulation Analysis        QPSK           Averages: 2

RMS EVM:                         EQ
  Max        Avg
  4.89 %     4.82 %
Pk EVM:
  Max        Avg
  11.30 %    10.56 %
RMS Mag Err:
  Max        Avg
  3.44 %     3.38 %
RMS Phase Err:
  Max        Avg
  2.45 °     2.41 °
Freq Error:
  Max        Avg
  -4.11 kHz  -4.08 kHz
IQ Offset:
  Max        Avg
  -25.57 dB  -25.89 dB
Rho:
  Min        Avg
  0.99647    0.99657

File Operation Status, A:\SCREN040.GIF file saved
```

Figure 4 – EVM for QPSK 21 dBm

A sample test Scenario is listed below:

1. Set transmitter frequency to 1935[MHz].
2. Output the EV-DO signal from transmitter using the EV-DO test program
3. Set the transmitting power to 0[dBm], 21[dBm] and 24[dBm] one by one.
4. Set PSA.
   Amplitude key -> Attenuation -> Auto
   MODE key -> 1xEV-DO
   Mode Setup key -> Radio -> Device -> MS(Rev) Frequency key -> Center Freq -> 1.935GHz
   Input/Output key -> Max Total Pwr -> [Corresponding to transmitting power]

The receiver test suite configuration is shown in **Figure 5**. As can be seen, a vector signal generator is configured to generate various EV-DO signals. The signals or received by the RF front end and the receive RF API is used to configure the receiver as well as capture the received data. The characteristics of the captured data are used in the tests battery.

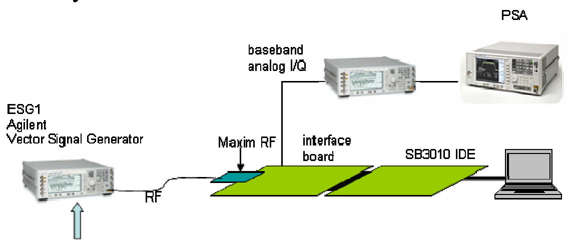


Figure 5 – Receiver Test Suite Configuration

As can be seen in **Figure 6**, the power spectrum of the CW signal is received and analyzed.

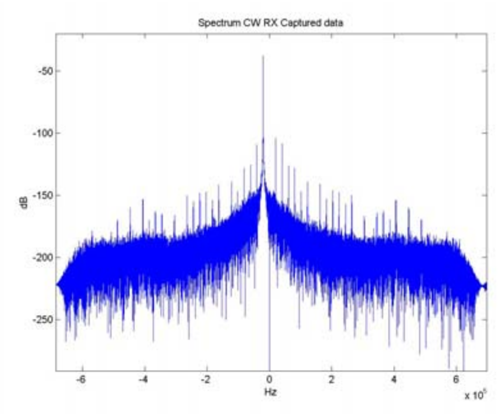


Figure 6 - Power spectrum of data captured on SB DSP (CW Rx test)

5 Physical Layer Receiver Chain Description

The baseband transmit and receive processing chains used to exercise the EV-DO RF front end are shown in **Figure 7** below.

6 Conclusion

As demonstrated in this paper, a generic RF API can be written for a software defined radio platform to unify and standardize the control of the RF front end. This API can be extended to reconfigurable RF front end easily by simply modifying the underlying Hardware Specific Sublayer. Results of this API development indicated speed of development, and efficient integration of the RF API into the application code.

7 References

[1] B. Beheshti, T. Raja, "Software Defined Radio Implementation Considerations and Principles Using the Sandblaster™ SDR Baseband Processor", Proceedings of Software Defined Radio Technical Forum, 16-18 November, 2005, Anaheim, California.

[2] B. Beheshti, J. Glossner, D. Routenberg, L. Zannella, and P. Steensma, "Evaluation of Military Waveform Processing on a COTS Reconfigurable SDR Processing Platform", Proceedings of Software Defined Radio Technical Forum, Volume A, pp. 147-151, 16-18 November, 2004, Scottsdale, Arizona.

[3] B. Beheshti, S. Jinturkar, "Simultaneous Baseband Processing Considerations in a Multi-Mode Handset Using the Sandblaster™ Baseband Processor", Proceedings of Software Defined Radio Technical Forum, 16-18 November, 2006, Orlando, Florida.

[4] D. Iancu, J. Glossner, V. Kotlyar, H. Ye, M. Moudgill, and E. Hokenek, "Software Defined Global Positioning Satellite Receiver", Proceedings of the 2003 Software Defined Radio Technical Conference (SDR'03), HW-2-001, 6 pages, Orlando, Florida, 2003.

[5] J. Glossner, D. lancu, J. Lu, E. Hokenek, and M. Moudgill, "A Software Defined Communications Baseband Design", IEEE Communications Magazine, Vol. 41, No.1, pp. 120-128, Jan., 2003.

[6] J. Glossner, S. Dorward, S. Jinturkar, M. Moudgill, E. Hokenek, M. Schulte, and S. Vassiliadis, "Sandbridge Software Tools", in Proceedings of the 3rd International Worksop on Systems, Architectures, Modeling, and Simulation (SAMOS.p3), July 21-23,2003, pp. 142-147, Samos, Greece.

[7] J. Glossner, T. Raja, E. Hokenek, and M. Moudgill, "A Multithreaded Processor Architecture for SDR", The Proceedings of the Korean Institute of Communication Sciences, Vol. 19, No. 11, pp. 70-84, November, 2002.

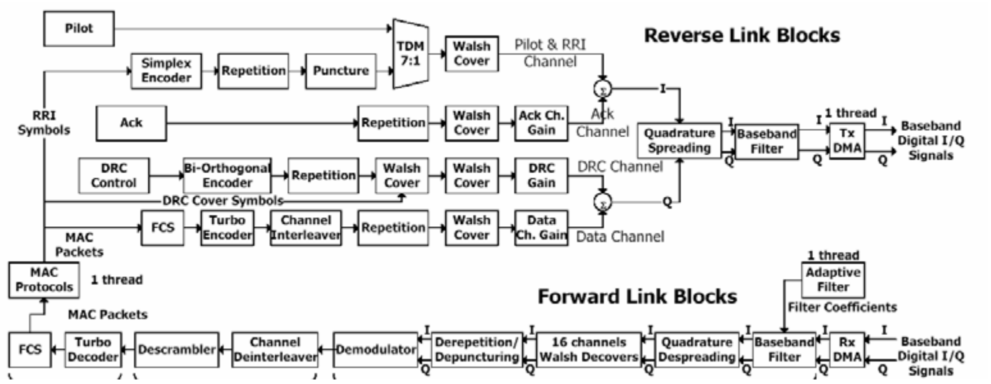


Figure 7 – EV-DO Forward and Reverse Link Processing Chains