ADOPTING WINNF TRANSCEIVER FACILITY FOR SPECTRUM SENSORS

Tomaž Šolc ("Jožef Stefan" Institute, Ljubljana, Slovenia; tomaz.solc@ijs.si)

ABSTRACT

An implementation of the Wireless Innovation Forum Transceiver Facility was developed for VESNA SNE-ESHTER, a specialized spectrum sensor deployed in wireless testbeds within the FP7 CREW project. The goal was to simplify experimentation and portability of experiments. A C++ library was developed that runs on the host PC and implements an event-based scheduler and an asynchronous interface conforming to the Transceiver Facility specification. The library communicates with the sensor over a serial line and does not require modification of the spectrum sensor's firmware. We present the challenges encountered and show results of some latency benchmarks of this Transceiver Facility implementation. Finally, we provide some suggestions on how the Transceiver Facility could be improved in future versions to better support such hardware.

1. INTRODUCTION

1.1. Transceiver Facility

In a software defined radio architecture, all waveform processing tasks are implemented in software. To make software portable between different transceiver hardware, a standardized programming interface is desired. The Wireless Innovation Forum Transceiver Facility [1] is an effort to develop such a standardized programming interface. The version 1.0 of the specification is available on-line.

Transceiver Facility describes in detail a modular interface that allows the software to control the radio hardware, the hardware to describe itself to the software and a streaming interface for passing digital baseband data between hardware and software. Hardware control functions of the Transceiver Facility include radio-frequency (RF) front-end control (for example, central frequency for up- and down- conversion, gain, channel filter, etc.), analog-to-digital and digital-to-analog conversion details (for example, sampling rate, etc.). An event-based mechanism is provided for accurate time synchronization between hardware and software. While the Transceiver Facility specification is language-agnostic, it includes reference examples of the interface in C++ and VHDL.

1.2. FP7 CREW project

The FP7 CREW project [2] developed a federation of wireless testbeds. CREW testbeds allow experimentation in diverse environments, technologies and frequency bands. A testbed consists of a number of remotely controlled computing nodes with attached radio hardware. Environments range from RF shielded rooms to out-door deployments. For example, nodes in the LOG-a-TEC testbed [3] are mounted on street lights in several urban areas. An experimenter develops their application, uploads it to one or more nodes in a testbed and performs measurements using testbed instrumentation.

Radio hardware in CREW testbeds can be roughly grouped into: a) SDR front-ends like the Ettus Research USRP in combination with high-performance general-purpose computers, b) narrow-band radios mounted on low-power wireless sensor nodes and c) specialized spectrum sensing devices like the VESNA SNE-ESHTER [4] and Imec Sensing Engine [5].

Each of these devices typically provides its own programming interface. One of the goals of the FP7 CREW project was to develop a common programming interface across the federation. A common interface to testbed hardware simplifies application development for experimenters and enables easy portability of experiments from one testbed to another. Early in the project, it has been decided to adopt the Transceiver Facility as the common interface to SDR nodes in testbeds [6].

To develop further the concept of a unified interface, we have decided to also adopt the Transceiver Facility for other categories of radio hardware in our testbeds. Adopting the Transceiver Facility for sensor node radios seemed impractical. On the other hand, adopting the Transceiver Facility for our specialized spectrum sensing hardware seemed feasible. Still, this posed several challenges. Spectrum sensors are specialized devices that differ from general-purpose SDR frontends in some significant ways. For example, they are receive-only devices with on-board signal processing. They are optimized for continuous scanning of a radio-frequency band and typically report only statistical data to the host PC. They are typically incapable of providing an uninterrupted stream of unprocessed baseband samples due to bitrate restrictions in various parts of the system.

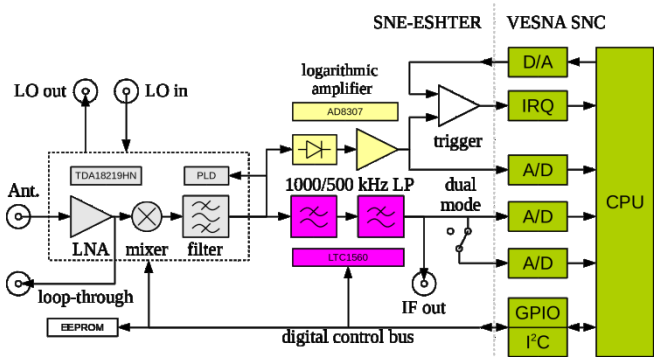

Figure 1: Block diagram of the SNE-ESHTER receiver, showing analog front-end and the interface to the VESNA sensor node core.

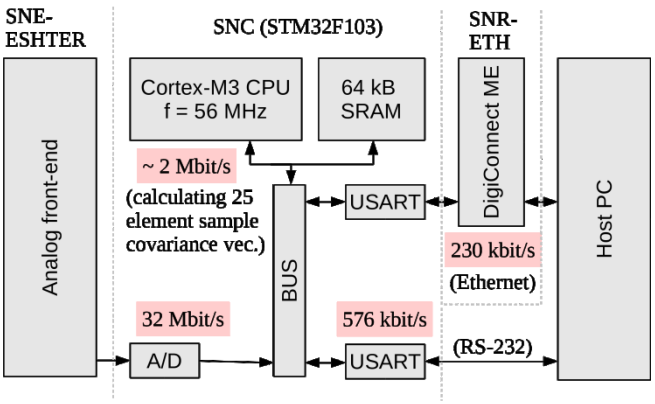

Figure 2: Block diagram showing different bottlenecks in passing the data from the front-end to the host PC

The rest of this paper is structured as follows: In section 2 we introduce VESNA SNE-ESHTER spectrum sensor we targeted in this extension of the Transceiver Facility support in the FP7 CREW project. In section 3 we describe the software architecture behind the current implementation of the Transceiver Facility. In section 4 we show some latency benchmarks of the implementation. In section 5 we comment on the possible future improvements. Finally, we conclude the paper in section 6.

2. VESNA SNE-ESHTER

VESNA SNE-ESHTER is a low-cost, compact spectrum sensor for the VHF and UHF frequency range that was developed at the Jožef Stefan Institute. It is based on VESNA [7], a low-power sensor node core. These devices are deployed in the LOG-a-TEC testbed. A VESNA SNE-ESHTER setup typically consists of three parts: the SNE-ESHTER analog front-end, the VESNA sensor node core (SNC) and a host PC running a GNU/Linux operating system.

2.1. Analog Front-End

The analog front-end contains the radio frequency analog electronic circuit that performs the frequency down-conversion and signal conditioning before analog-to-digital conversion. A simplified block diagram is presented in Figure 1. The front-end is a custom designed single-conversion, low-IF receiver based on the NXP TDA18219HN integrated circuit. The receiver has a specified input frequency range between 42 MHz and 870 MHz. The local oscillator (LO) signal is generated by a FRAC-N phase-locked loop (PLL) and has a typical settling time of 5 ms on channel change.

The RF signal from the antenna is amplified in a low-noise amplifier (LNA) and mixed with the LO in an image-rejection mixer to produce a signal at an intermediate frequency (IF). Several stages of automatic gain control are used to minimize non-linear distortion and maximize signal-to-noise ratio of the signal. The signal passes through one tracking RF and two IF band-pass filters with software-selectable bandwidth. The final stage is a 10th order elliptic anti-aliasing filter with two settings: 500 kHz and 1000 kHz, corresponding to 1 Msample/s and 2 Msample/s sampling rates. After the anti-aliasing filter, the signal is routed to the VESNA sensor node core to be sampled by an analog-to-digital converter (ADC).

In addition to the main signal path, the SNE-ESHTER front-end board also includes additional analog energy detection blocks. Two logarithmic signal level detectors can be calibrated for accurate measurement of absolute signal power. An analog trigger circuit can also provide an interrupt to the CPU when the signal level in the tuned channel reaches a defined threshold. These functionalities are currently unused when SNE-ESHTER is used through the Transceiver Facility.

The SNE-ESHTER design allows for hosting two analog front-end boards on a single sensor node core. This allows simultaneous sensing of two different frequency channels or reception on a single channel using two antennas. Using multiple front-ends is not supported in the current implementation of the Transceiver Facility.

2.2. VESNA Sensor Node Core

VESNA SNC contains an integrated microcontroller with an ARM Cortex M3 CPU core with a 56 MHz clock and 64 KB of SRAM. The SNC also contains a RS-232 interface to the host PC with a 576 kbit/s maximum bitrate. An optional Ethernet interface can be installed in case the sensor is remotely installed.

```
 1 -> select channel 650000:1:650001 config 0,2     # Host PC instructs the sensor to tune to
                                                    # 700 MHz and hardware configuration 2
                                                    # (defining filter bandwidth and sampling rate)
 2 <- ok                                            # Sensor confirms the command
 3 -> samples 1024                                  # Host PC sets sampling buffer length to 1024
 4 <- ok                                            # Sensor confirms the command
 5 -> sample on                                     # Host PC instructs the sensor to start sensing
 6 <- TS 0.001 CH 650000 DS 2042 2045 2053 ...      # Sensor sends first full sampling buffer
                                                    # containing 1024 samples and a timestamp.
 7 <- TS 0.136 CH 650000 DS 2053 2056 2042 ...      # Sensor continues to send reports until
 8 <- ...                                           # commanded to stop
 9 -> sample off                                    # Host PC instructs the sensor to stop sensing
10 <- ok                                            # Sensor confirms the command
```

Figure 3: Example conversation between the SNE-ESHTER spectrum sensor and the host PC using the native serial protocol.

The SNC includes three 12-bit successive approximation ADCs with up to 2 Msample/s sample rates. ADCs are driven by a DMA controller and store samples directly into a sample buffer in SRAM without any intervention from the CPU. The sample buffer has space for up to 25000 samples (up to 12.5 ms at 2 Msample/s sample rate). Collected signal samples are read by the CPU. They can be either processed on-board or sent in a raw form over the RS-232 or Ethernet interface to the host PC. For example, the firmware currently implements calculating a sample covariance vector on-board. This significantly reduces the amount of data that needs to be sent from the sensor when sensing algorithms like covariance or Eigenvalue detection are employed.

The CPU is unable to either process or forward the samples to the host PC at the rates provided by the ADC. Different bottlenecks preventing this are shown in Figure 2. Hence the device typically operates in a sample-process-report cycle which includes significant blind time. For simple operations, like the covariance vector calculation, the signal processing capability of the CPU exceeds the bandwidth of interfaces to the host PC. Therefore, using on-board processing typically reduces the blind time.

The native serial interface between the firmware running on the VESNA SNE-ESHTER CPU and the host PC is an ASCII based protocol. In this interface, details of ADC and most analog front-end settings, like filter and AGC settings, are abstracted in the form of a low number of discrete hardware configurations identified by numerical identifiers. Configurations 2 (2 MHz sampling frequency/ 1 MHz anti-aliasing filter bandwidth) and 3 (1 MHz sampling frequency/500 kHz anti-aliasing filter bandwidth) allow for sampling of the IF signal and are the only two hardware configurations currently used by the Transceiver Facility implementation.

Figure 3 shows an example of a serial line session that includes all native commands, relevant for the current implementation of the Transceiver Facility interface. It can be seen that this interface itself does not allow for accurate scheduling of receive start and stop time or synchronization of the signal samples. It does however provide information on relative timing of individual samplings based on the sensing start time, which is derived from the quartz oscillator on the SNC. This provides information on how much of the signal has been lost. For example, the interval between two sample buffers sent to the host PC on lines 6 and 7 in Figure 3 is 135.0 ms, while a buffer with its 1024 samples sampled at 2 Msample/s only covers 0.5 ms of that time.

3. IMPLEMENTATION

We implemented the Receive Channel of the Transceiver Facility for SNE-ESHTER in the form of a C++ library, targeting the GNU/Linux operating system. The library exports the same user-facing interface as other Transceiver Facility implementations used in FP7 CREW (similar libraries exist for USRP devices and the Imec Sensing Engine). An experimenter that wants to use a receiver for spectrum sensing through the Transceiver Facility writes a C++ program and links it against one of these libraries, depending on which receiver they want to use. Except for some initialization parameters, the experimenter's code does not need any adaptations when switching from one receiver hardware to the other.

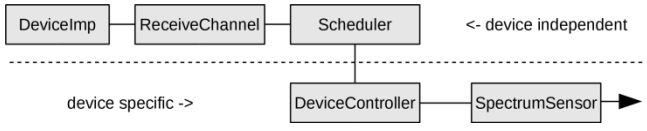

Figure 4: Main classes in SNE-ESHTER Transceiver Facility implementation

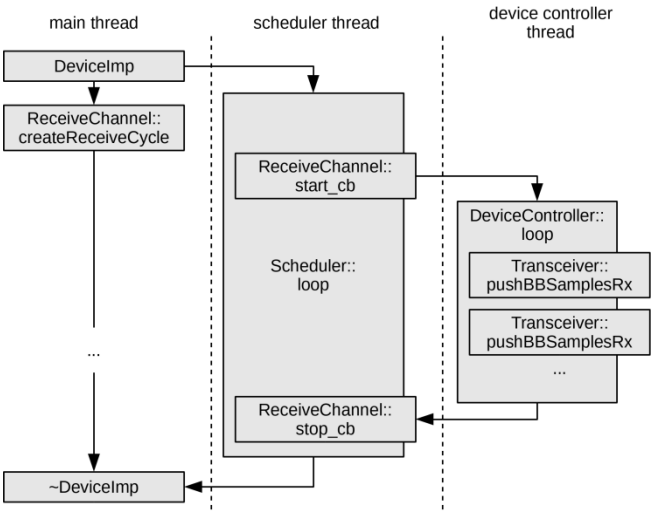

Figure 5: A diagram of threads of execution and their most important method calls in the SNE-ESHTER Transceiver Facility.

Experimenter's code using the Transceiver Facility interface runs on the host PC (and not on the CPU in the SNE-ESHTER device itself). This provides the benefit of not needing to adapt the device's firmware for each experiment, while on the other hand bounds the Transceiver Facility to the limitations of the serial interface. This same approach has been used in all other Transceiver Facility implementations in the CREW project.

Our library consists of 5 main object classes that are shown in Figure 4 and can be divided into two parts: a) a hardware independent event scheduler and the user-facing Transceiver Facility interface and b) the adaptor for the serial interface to the device. The library uses three threads of execution, illustrated in Figure 5. The source code is available on-line at https://github.com/sensorlab/xcvr-eshter.

3.1. Scheduler

The main task of the event scheduler is to translate between the asynchronous, event-based interface specified by the Transceiver Facility and the synchronous native serial interface to the spectrum sensor.

The user controls the Transceiver through the *DeviceImp* ("device implementation") class. This class contains the *ReceiveChannel* object that conforms to the Transceiver Facility specification and forms the user-facing part of the library. *TransmitChannel* is unimplemented, as the sensor is a receive-only device.

Construction of a *DeviceImp* object instance is the only step that is device specific. The user must supply the constructor with an instance of the *SpectrumSensor* object. *SpectrumSensor* constructor necessarily requires knowledge of the underlying hardware. For VESNA SNE-ESHTER device, the constructor requires a path to the Unix device file (e.g. "/dev/ttyUSB0") that is used to communicate with the sensor.

Most of the user's interaction with the transceiver happens through the *createReceiveCycleProfile()* method of the *ReceiveChannel* object. It allows the user to specify when the sensor starts and stops recording signal samples and other details of reception. *createReceiveCycleProfile()* schedules *ReceiveStartTime* and *ReceiveStopTime* events with the scheduler. In case *undefinedDiscriminator* has been used for *requestedReceiveStopTime*, the *ReceiveStopTime* event remains unscheduled. This allows for signal reception of undefined length. In that case, the *setReceiveStopTime()* method can be used to schedule the *ReceiveStopTime* event at a later time.

In the current implementation, *setReceiveStopTime()* cannot be used once a *ReceiveStopTime* event has been scheduled, as that would require cancelling an existing event. This operation is not currently supported by the underlying scheduler. For a similar reason, *configureReceiveCycle()* method has not been implemented.

The *Scheduler* class is hidden from the user of the library. It performs all asynchronous event scheduling using an event loop in a separate thread. The event loop uses the Boost.Asio library [8] using the system clock of the host PC as the reference. All discriminators, including the *eventBased* discriminator, have been implemented.

The *eventBased* discriminator supports selection of event count origin, event count and time offset. However, only up to one event in the past can be used as the origin. This for example, allows the use of *eventCountOrigin=Previous*, *eventCount=0* setting, which is a common pattern. On the other hand, the Transceiver Facility specification appears to allow for selecting an arbitrary past event as the reference for the *eventBased* discriminator. Implementing this functionality would require the scheduler to keep a log of timestamps for all past events. This was considered an unnecessary complication considering the limited use of such discriminators.

3.2. Device controller

The only hardware specific parts of the code are the *DeviceController* and *SpectrumSensor* classes.

DeviceController implements a thin asynchronous wrapper around the device-specific *SpectrumSensor* class. It provides only two methods: *start()* and *stop()*. These two methods start and stop *DeviceController*'s thread which runs its own event loop. *DeviceController* configures the hardware through the *SpectrumSensor* class before starting the reception. *DeviceController*'s event loop calls back to the user's code through the *pushBBSamplesRx()* method every time the sensor sends a packet of signal samples to the host PC. Hence the *pushBBSamplesRx()* is called

asynchronously from the perspective of the library user's code. *start()* and *stop()* methods are called from *ReceiveStart* and *ReceiveStop* event callbacks that were scheduled by the user's initial call to *createReceiveCycle()*.

Transceiver facility specifies that complex signal samples are pushed by the transceiver to the user code in the *BBPacket.packet* structure. However SNE-ESHTER is using low-IF sampling and provides only real-valued samples. To accommodate for that, *DeviceController* writes the actual signal samples in I field and fills Q field in *BBPacket.packet* structure with zeros.

The *SpectrumSensor* class provides a synchronous interface to the native serial ASCII protocol. This class is a straightforward C++ port of the Python *SpectrumSensor* class that is usually used to control SNE-ESHTER from a host PC [9]. C++ Serial library [10] was used, since it provides a similar interface to the Python Serial library and simplified the porting procedure. Numerical tuning profile identifiers from the Transceiver Facility are directly translated into SNE-ESHTER hardware configurations that are passed to the device over the native serial protocol. Hence only tuning profiles 2 and 3 can currently be used. *PacketSize* parameter on the Transceiver Facility side is directly used as the sample buffer length in the native serial protocol.

3.3. Test driven development

A test-driven development methodology [11] was employed when developing the Transceiver Facility implementation. Individual components were designed with minimal implicit dependencies to enable testing of each component separately. Whenever possible, dependencies between classes are injected explicitly through constructor parameters. This is, for example, the reason why *DeviceImp* constructor requires the user to provide a *SpectrumSensor* instance instead of the *SpectrumSensor* object being instantiated by the constructor implicitly. Tests were developed using the cpputest framework [12].

The component that benefited most from test driven development was the *Scheduler* class. Transceiver Facility specifies a relatively broad range of possibilities for event scheduling. Test driven development proved to be a very efficient way of deriving a reliable implementation.

Where dependency on other classes could not be avoided in tests, mock classes were created and used instead of real classes via dependency injection. For example, several tests of the *DeviceImp* class use a *SpectrumSensor* implementation that does not communicate with hardware. This approach simplified development, since it was possible to develop software without having a sensing device constantly connected. It excluded the possibility that a failed test would be caused by a malfunctioning device or a bug in the device's firmware. Tests with a mocked sensor were also faster to execute.

After individual components were developed, a suite of system tests was also created that tested the whole Transceiver Facility implementation. This suite uses the same cpputest framework, but is compiled separately. This allows the developer to choose between running tests that do not require a connected sensor and tests that do.

4. BENCHMARKS

The foremost concern with our implementation of the Transceiver Facility was the latency between the event scheduler and the sensing device. The latency is the time difference between when the user scheduled the signal reception to start and when the signal reception actually occurred. The Transceiver Facility specifies nanosecond precision. However such level of precision is optimistic even for USRP devices, which are much lower latency devices than our sensors.

To measure the latency when using Transceiver Facility with SNE-ESHTER, an instrumented version of the Transceiver Facility library was developed that logged function calls and serial line traffic together with current host PC system time. Additionally, a digital storage oscilloscope was setup to capture the waveform on the RS-232 line connecting the host PC and the SNE-ESHTER sensor. This setup was then used to observe the series of events that happen when a user of the Transceiver Facility calls the *createReceiveCycle()* method on the *ReceiveChain* object. *requestedReceiveStartTime* was set to immediate. SNE-ESHTER was configured to use 2 Msample/s sample rate and 2048 samples per packet.

A measurement result is shown in Figure 6 in form of an oscilloscope screenshot. Data from the instrumented C++ library is shown overlaid in form of text annotations. It can be seen that the first *pushBBPacketRx()* callback from the Transceiver Facility to the user's code happens 570 ms after the *createReceiveCycle()* call and the *ReceiveStart* scheduler event. Further *pushBBPacketRX()* calls follow each 270 ms. For each *pushBBPacketRX()* call, 10264 bytes of data have been transferred over the serial line.

Configuration parameters are sent to the device immediately after the *createReceiveCycle()* call. This involves two round-trips on the serial line and concludes with the »sample on« command. This measurement shows that this contributes a negligible amount to the overall latency.

Channel sampling time also takes a very small part of the interval between two *pushBBPacketRX()* calls:

$$t_{sampling} = \frac{2048 \text{ samples}}{2\frac{\text{Msample}}{\text{s}}} \cong 1 \text{ ms}$$

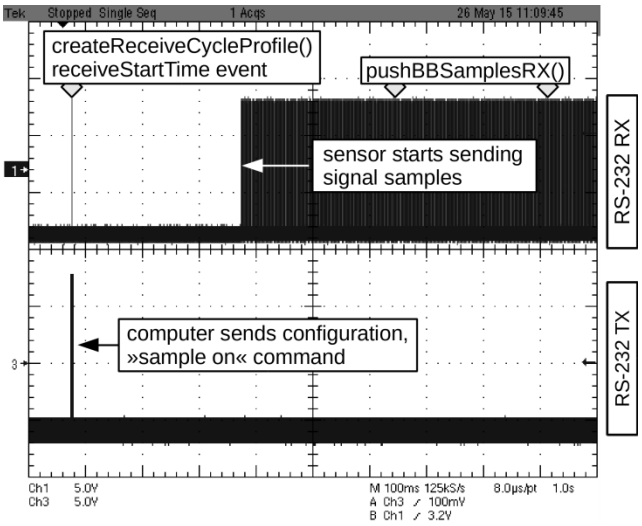

Figure 6: Timeline of events following a *createReceiveCycle()* call

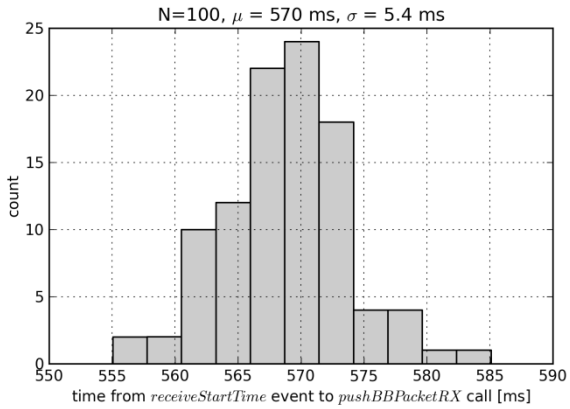

Figure 7: A histogram of latency measurements

Our experiment didn't provide information on when exactly the signal sampling occurred on the SNE-ESHTER device. From the firmware running on the device, we can infer that first sampling happened immediately before the sensor started sending signal samples over the serial line.

Approximately half of the time interval (270 ms) before the first *pushBBPacketRX()* call is taken by the transmission of the packet of signal samples over the serial interface ($t_{transmission}$). The throughput is less than the theoretical 576 kbps of the serial line (RS-232 line uses 1 start bit and 1 stop bit, hence 10 transferred bits per byte):

$$B = \frac{10264 \text{ bytes} \cdot 10 \frac{\text{bits}}{\text{byte}}}{270 \text{ ms}} \cong 380 \text{ kbps}$$

This result suggests that even the throughput of unprocessed signal samples is in part limited by the CPU on the sensor. It also shows that the latency is heavily dependent on the size of the sampling buffer. Doubling the size of the sample buffer would double the time spent in transmission.

The rest of the time between the "sample on" command and first signal samples appearing on the serial line is spent in power up sequence for the analog front-end (t_{start}).

Since the information on the actual instant of signal sampling is not available to the user of Transceiver Facility, we can define worst-case latency for the purpose of this benchmark as the time between the *createReceiveCycle()* method call and the *pushBBPacketRX()* method call. A histogram of 100 such latency measurements is shown in Figure 7. Since the host PC system time was used to perform these measurements, the variance in measurements is most likely caused by the granularity of task switching in the Linux kernel on the host PC rather than by the device itself.

5. FUTURE IMPROVEMENTS

5.1. Improving latency

The simplest way to improve the latency would be to adapt the native serial protocol to use binary instead of ASCII representation when sending signal samples to the host PC. This would significantly reduce the amount of data sent over the serial line per each sampling. Our benchmark has shown that current ASCII encoding requires 10264 bytes to transmit 2048 signal samples. With 12 significant ADC bits per sample, that is 3072 bytes of information and encoding overhead of approximately 300%. Overhead of an efficient binary encoding would likely be significantly lower. It would also likely reduce the CPU load on the sensor core, since sample data would no longer need to pass through an ASCII formatter.

The best theoretical transmission time achievable with this setup would be:

$$t_{transmission} = \frac{3072 \text{ bytes} \cdot 10 \frac{\text{bits}}{\text{byte}}}{576 \text{ kbps}} \cong 53 \text{ ms}$$

Another relatively simple way of significantly improving the latency would be to optimize t_{start}. The current SNE-ESHTER firmware puts the analog hardware into a complete power down mode between receive cycles. Power up requires a lengthy setup and calibration sequence. A modified firmware that would leave the receiver turned on would hence reduce the latency by approximately 300 ms.

A more complex approach would be to implement the Transceiver Facility on the VESNA SNE-ESHTER CPU. In theory such an approach would enable the Transceiver Facility to operate with minimum latency, since then it

would no longer be acting merely as a façade over the existing firmware. This would however require users to reprogram the device firmware, which would significantly raise the barrier to entry for testbed users. Running the experiment on the sensor node would also put a much stricter limit to the complexity of the software due to the limited amount of RAM and ROM storage on the sensor node. Due to the asynchronous nature of the Transceiver Facility interface, a multi-tasking real-time operating system would be required on the device (current firmware does not use an operating system), further reducing storage available for experimenter's code.

5.2. Missing timing information

Transceiver Facility is specified for devices capable of supplying a continuous stream of signal samples to the application. While the Facility enables the operation of a transceiver in burst mode, it assumes that the burst length and timing is defined by software (by scheduling multiple receive cycles) and that hardware can accommodate arbitrary timing of bursts. As has been discussed in previous sections, SNE-ESHTER can only provide a packet of continuous samples up to maximum sampling buffer size of 25000 samples with some mandatory amount of blind time between packets. This creates two problems: a) the Transceiver Facility provides no means of communicating such a hardware limitation to the software and b) there is no means of communicating the length of time or number of dropped samples between two consecutive calls of *pushBBSamplesRx()*. The latter problem could be solved in a backwards compatible way by adding a timestamp field to the *BBPacket* structure.

5.3. Exploiting other capabilities of the hardware

As discussed in Section 2, our spectrum sensors are optimized for the use case where some signal processing occurs on the device itself. At the moment, all this functionality is disabled when the sensor is used through the Transceiver Facility interface. The Transceiver Facility could be expanded in a way to allow the user to specify certain pre-processing functions to be applied in the hardware before data is passed to the waveform application.

For instance, an expanded *createReceiveCycle()* method could take an additional parameter specifying such a function. The transceiver could define a list of supported preprocessing functions. For example, specifying a constant *nullFunction* would pass unprocessed signal samples to the waveform application, as per existing specification. Specifying *covarianceFunction* would pass elements of the covariance vector to the waveform application. Specifying *fftFunction* would pass the result of the discrete Fourier transform and so on. This would make it simple for Facility implementations to off-load such functions to hardware, if the hardware supports it. In CREW testbeds where portability of experiments between testbeds is important, those Transceiver Facility implementations that do not have hardware support for a certain function could implement it in the C++ library on the host PC.

6. CONCLUSIONS

We have described an implementation of the Wireless Innovation Forum Transceiver Facility for VESNA SNE-ESHTER spectrum sensor devices deployed in wireless testbeds of the FP7 CREW project. While this usage of the Transceiver Facility is likely not "in the spirit" of the specification, it currently appears useful in our case for providing a unified interface to a broader set of testbed hardware. It enables for example a spectrum sensing algorithm to be developed once and used on USRP-equipped testbed as well as on SNE-ESHTER and Imec Sensing Engine devices with minimal adaptations.

Our measurements show that typical latency that can be expected using the SNE-ESHTER device is 570 ms for a packet length of 2048 samples. Latency increases with higher packet lengths. Most of the latency is due to the delay in communication between the host PC and the device and due to analog front-end setup. We provided some suggestions how the latency could be reduced by improving spectrum sensor's firmware. The high latency currently makes this implementation unsuitable for use cases requiring fast or well-synchronized changes to radio configuration.

Current implementation leaves several functions of the SNE-ESHTER device inaccessible due to the limitations of the current Transceiver Facility specification. The specific limitations on the size and timing of sample packets also remains undescribed within the framework of the specification. We provided some suggestions on how these drawbacks could be mitigated through changes to the Transceiver Facility specification.

Whether the high latency and limitations regarding the particular properties of spectrum sensors will make this Transceiver Facility implementation usable in practical experiments on CREW testbeds remains to be seen.

7. ACKNOWLEDGEMENTS

This work has been partially funded by the European Community through the 7th Framework Programme project CREW (FP7-ICT-2009-258301).

8. REFERENCES

[1] E. Nicollet, S. Pothin and A. Sanchez, *Transceiver Facility Specification*, Wireless Innovation Forum, 28 January 2009. Available from: http://groups.winnforum.org/p/cm/ld/fid=85

[2] *CREW project – project overview*, CREW consortium, 2015 [viewed 3 June 2015]. Available from: http://www.crew-project.eu/

[3] M. Mohorčič, M. Smolnikar and T. Javornik, "Wireless Sensor Network Based Infrastructure for Experimentally Driven Research," *The Tenth International Symposium on Wireless Communication Systems (ISWCS)*, Ilmenau, Germany, August 2013.

[4] T. Šolc, "SNE-ESHTER: A low-cost, compact receiver for advanced spectrum sensing in TV White Spaces," *ETSI workshop on Reconfigurable Radio Systems*, Sophia Antipolis, France, December 3-4 2014.

[5] S. Pollin, et al, "An integrated reconfigurable engine for multi-purpose sensing up to 6 GHz," *IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN),* pp. 656-657, IEEE, May 2011.

[6] S. Thao, "Transceiver Facility / API - SDR applications Fast-prototyping on Ettus Research USRP platform," *CREW Training Days*, Brussels, Belgium, February 19-20 2013.

[7] M. Smolnikar, et al, "Wireless Sensor Network Testbed on Public Lighting Infrastructure," *The Second International Workshop on Sensing Technologies in Agriculture, Forestry and Environment*, pp. 6-7, 2011.

[8] C. Kohlhoff, *Boost.Asio*, 2008 [viewed 3 June 2015]. Available from: http://www.boost.org/doc/libs/1_37_0/doc/html/boost_asio.html

[9] *Spectrum sensing application for the VESNA platform,* 2015 [viewed 3 June 2015]. Available from: https://github.com/sensorlab/vesna-spectrum-sensor

[10] W. Woodall, *Serial Communication Library*, 2015 [viewed 3 June 2015]. Available from: https://github.com/wjwwood/serial

[11] J. W. Grenning, *Test Driven Development for Embedded C*, Pragmatic Bookshelf, 2011.

[12] *Cpputest* [viewed 3 June 2015]. Available from: http://cpputest.github.io