Cognitive Radio Experimentation World

Project Deliverable D3.2
Optimized operational federated platform

Contractual date of delivery:	30-09-12
Actual date of delivery:	30-09-12
Beneficiaries:	IBBT – IMEC – TCD – TUB – TUD – TCS – EADS – JSI
Lead beneficiary:	TCD
Authors:	Danny Finn (TCD), Justin Tallon (TCD), Joao Paulo Cruz Lopes Miranda (TCD), Luiz DaSilva (TCD), Stefan Bouckaert (IBBT), Ingrid Moerman (IBBT), Christoph Heller (EADS), Somsaï Thao (TCS), Alejandro Sanchez (TCS), David Depierre (TCS), Sofie Pollin (IMEC), Peter VanWesemael (IMEC), Mattias Desmet (IMEC), Jan Hauer (TUB), Mikolaj Chwalisz (TUB), Nicola Michailow (TUD), Zoltan Padrah (JSI), Tomaž Šolc (JSI), Matevz Vucnik (JSI), Mihael Mohorcic (JSI), Tomaz Javornik (JSI), Miha Smolnikar (JSI)
Reviewers:	Carolina Fortuna (JSI), Stefan Bouckaert (IBBT)
Workpackage:	WP3 – Creating the Federation
Estimated person months:	53
Nature:	R
Dissemination level:	PU
Version	1.0

Abstract: This public document gives a detailed description of the functionality of the optimised operational federated platform. This platform will include a more advanced version of the PORTAL and supports cross-country component combinations and advanced data collection. This deliverable reports on the activities performed in all tasks of this work package.

Keywords: network testbeds, federation, wireless networks, cognitive radio, cognitive network, functionalities, capabilities, components, combination, interface, data format, portal, guidelines.

Executive Summary

This document provides a report on how well the CREW project has performed in the 4 core tasks of WP3 “Creating the Federation”. These are Task 3.1 “Baseline Analysis and Federated Portal Establishment”, Task 3.2 “Component Compatibility Analysis and Interface Design & Specification”, Task 3.3 “Common Data Collection/Storage Methodology Design” and Task 3.4 “ Interoperability Testing”. This document builds on the information provided in our previous document D3.1: “Basic Operational platform” and provide up-to-date information on the capabilities and functionalities of the CREW federated testbed. This document incorporates also the Logatec and Ljubljana testbeds as well as the TCS multi-antenna LTE sensing platform which were not present in the D3.1; however, discussion of the basic operational platform of the VENSA-based testbed (Logatec and Ljubljana) is provided in D3.3.

We start by reporting, the status of each of the testbeds and platforms in terms of the federation functionalities which they have implemented. The report demonstrates that the CREW federated testbeds and platforms now have most, if not all, of CREW core functionalities implemented. These include the sharing of baseline functionality information/testbed description, incorporation and sharing of hardware (/software) between testbeds, advanced sensing functionality, use of the CREW common data format, definition of benchmarked scenarios for each of the federation testbeds, the presence of access and usage information on the CREW Portal, remote open access for experiments that are performed in the context of the CREW project, and usage of the testbed/platform functionalities by other CREW partners, external users and open call experimenters.

Within the last year, a number of testbed optimisations have taken place in order to provide these CREW core functionalities as well as to enable additional others. In order to provide updated baseline analysis a number of these optimisations to the testbed functionalities are discussed in this document. Among these are:

- Increased testbed network sizes and ease of remote interaction (e.g. use of OMF (cOntrol and Management Framework)) to accommodate more cognitive network based experiments,
- Additional spectrum licensing (TV band and LTE 2.1GHz band transmissions),
- Improvements to better facilitate mix and match testbed usage and benchmarked experimentation.
- Additional spectrum sensing capabilities for a wider range of scenarios.
- Integration of hardware from other CREW partners into different testbeds, facilitating nomadic use of testbed hardware and tools

This document introduces the CREW repository, for sharing of experimental data, scripts and with the greater research community, in line with the common data collection and storage methodologies outlined in the CREW project description of work, as well as the. The repository has been included into as part of CREW Portal and will be used for sharing full experiment descriptions, traces, background environments, processing scripts, performance metrics and benchmarking scores.

Within the last year it was decided that, as part of the optimisation process, the common data format would move from a JSON specification, to one through XML. To facilitate this change an XML common data format validation tool has been included on the CREW Portal which is also discussed within this document.

Due to the extensiveness of the mix and match combination work performed as part of the interoperability testing within the CREW project (all partners, including open call partners, having taken part in combination experiments involving at least two other partners) we have included a table (Table 4) summarising all of the combinations which have been performed. These are divided into side-by-side combinations where recordings are performed in parallel on multiple devices and then processed together to provide a combined solution, software combinations where software developed in one testbed is used to operate hardware of another testbed, and hardware combination where a hardware coupling of two components from different testbeds is performed to avail of the benefits provided by each of the platforms individually. The table includes both intra-country and cross-country component combinations.

Detailed descriptions of the TCS Transceiver API is provided including set-up details, an architectural overview and code breakdown, and interfacing details for the VESNA-based testbeds are provided outlining how it can be controlled via an HTTP-like interface, a C interface or a custom firmware image. This forms part of the interface design and specification work of CREW.

As a whole, this document details the way in which the CREW federation has changed from a collection of island testbeds to a strongly interconnected federated platform which is capable of facilitating complex networks, building on the strengths of each of the individual testbeds and combining them to provide advanced federated experimental functionality.

List of Acronyms and Abbreviations

3G	Third Generation
ADC	Analogue to Dgital Converter
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BEE2	Berkeley Emulation Engine 2
BTS	Base Station Transceiver
CDF	Common Data Format
ComReg	Irish Commission for Communications Regulation
CR	Cognitive Radio
CRC	Cyclic Redundancy Check
CREW	Cognitive Radio Experimentation World
Dx.x	CREW Deliverable x.x
DAC	Digital to Analogue Converter
DEM	Digital Elevation map
DIFFS	Digital Front-end For Sensing
DNS	Domain Name System
DSF	Debugger Services Framework
DVB-T	Digital Video Broadcast Terrestrial
eNB	Enhanced Node B
FDD	Frequency Division Duplex
FFT	Fast Fourier Transform
GRASS	Geographical Resources Analysis Support System
GFMD	Generalized Frequency Division Multiplexing
GPS	Global Positioning System
GUI	Graphical User Interface
HaLo	Hardware-in-the-Loop
HTTP	Hypertext Transfer Protocol
ID	Identification
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
I/Q	In-phase/Quadrature
IP	Internet Protocol
ISM	Industrial Scientific Medical
JSON	JavaScript Object Notation
LSB	Least Significant Bit

LTE	Long Term Evolution
MAC	Medium Access Control
MSB	Most Significant Bit
NFS	Network File System
OFDM	Orthogonal Frequency Division Multiplex
OMF	cOntrol and Management Framework
OS	Operating System
PC	Personal Computer
PHY	Physical Layer
PXE	Pre-boot eXecution Environment
RC	Resource Controller
RF	Radio Frequency
REST	REpresentational State Transfer
RFIC	Radio Frequency Integrated Circuit
Rx	Receive
SCALDIO	Scalable Radio
SCPI	Standard Commands Programmable Interface
SD	Secure Digital
SDK	Software Development Kit
SDR	Software Defined Radio
SNC	Sensor Node Core
SPI	Serial Peripheral Interface bus
SSL	Secure Sockets Layer
TCP	Transfer Control Protocol
TDD	Time Division Duplex
TKN	Telecommunications Network Group at TU Berlin
TWIST	TKN Wireless Indoor Sensor Network Testbed
Tx	Transmit
UART	Universal Asynchronous Receiver/Transmitter
UE	User Equipment
UHF	Ultra High Frequency
UL	Up Link
URL	Uniform Resource Locator
USx	CREW Usage Scenario x
USB	Universal Serial Bus
USRP	Universal Software Radio Peripheral
VESNA	VErsatile platform for Sensor Network Applications

VNC	Virtual Network Computing
VSN	Virtual Sensor Network
WARP	Wireless open Access Research Platform
WLAN	Wireless Local Area Network
XCVR	Transceiver
WPx	CREW Work Package x
WSN	Wireless Sensor Network
XML	Extensible Markup Language

Table of contents

1 Introduction

1.1 Overview

This public document gives a detailed description of the functionality of the optimised operational federated CREW platform and builds upon the descriptions given in D3.1: “Basic Operational platform”. It reports on the activities performed in all tasks of work package 3 “Creating the Federation”.

Error! Reference source not found. below summarises the current status of each of the CREW testbeds and sensing platforms. The table can be interpreted as follows:

Baseline Analysis Provided indicates that either an on-site testbed demonstration and overview, or else a detailed presentation and decription have been provided of the testbed to the other CREW partners *and* also have been included in CREW deliverables D3.1, D3.2 and/or D3.3.

Hardware from other partners Incorporated into Testbed indicates where hardware from one federation partner has been incorporated into one of the other federation testbeds and can be easily used in experiments performed within that testbed. This also includes (although not specifically hardware) the incorporation of Iris SDR node into other federation testbeds.

Advanced sensing functionality indicates that this equipment is flexible enough to accommodate a wide range of advanced spectrum sensing algorithms and different frequency ranges.

Common data format indicates that experimental outputs from this testbed have been expressed in the CREW common data format.

Benchmarking Framework indicates that benchmarked scenarios, following the CREW benchmarking framework presented in D2.2 Section 3.5 have been defined for use in this testbed.

Portal indicates for each testbed whether it is currently present on the CREW portal.

Open Access indicates whether remote open access of this testbed is available for experiments that are performed in the context of the CREW project. Testbeds and platforms, which are not open access can, however, still be made available for specific experiments.

Used by CREW partners/ external users/ open call specifies by whom the testbed has been used in experimentation to date. This is denoted as C if the testbed has been used by other CREW partners, E if used by external users and OC1 if used by the participants of the first open call.

	Iris	IBBT	TWIST	LTE-TUD	VESNA	IMEC	TCS API	TCS multi-antenna senisng	EADS airplane cabin mock-up
Baseline Analysis Provided	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Hardware /software from other partners Incorporated into Testbed	X	Yes	Yes (not permanent)	Yes (not permanent)	X	N/A	N/A	N/A	N/A
Advanced sensing functionality	Yes	Yes	Yes	Yes	Yes	Yes	N/A	Yes	N/A
Common data format	Yes	Yes	Yes	X	Yes	Yes	N/A	Yes	Yes
Benchmarking Framework	Yes	Yes	Yes	Yes	Yes	Yes (part of IBBT	X	X	Yes

						w-iLab.t)			
Portal	Yes	Yes	Yes	Yes	Yes	Yes	Yes	X	X
Open Access	Yes	Yes	Yes	Yes	Yes	Yes (through IBBT w-iLab.t)	Yes	X	X
Used by CREW partners/ external users/ open call	C,E,OC1	C,E,OC1	E, OC1	C,E	C	C,E,OC1	E,OC1	C	C

Table 1: Federation status of testbeds and sensing platforms

As can be seen also from the table extensive development of the federation testbeds and has occurred during the first two years of the project to the extent that almost all partners have either performed, or else been involved in, each of the core CREW federation functionalities outlined in Table 1.

Within the last year the final steps of baseline analysis, to include testbed demonstrations and/or detailed descriptions of device operation and interfacing, for each of the core federation partners, was provided. These included site visits to the IMEC labs in Leuven, the EADS airplane mock-up environment in Munich, the Ljubljana and Logatec testbeds in Slovenia, and presentations on the TCS API and multi-antenna sensing platform. In each location visited on-site mix and match experiments were performed as well as interoperability testing to allow external experimenters to quickly assess feasibility of desired experiments. Further information of this is provided in Section 5.1.

In addition to baseline analysis information of the core partner experimental capabilities, baseline analysis of each the open call partner capabilities were presented in the Munich plenary meeting.

In the third row of the table we see that there has been a number of hardware exchanges between partners, which has resulted into full incorporation of that hardware into the other partner testbeds. While a number of the partners here may not have received hardware, that does not mean that they have not provided it, for example the IMEC sensing platform and Iris SDR nodes have both been incorporated into the w-iLab.t testbeds in IBBT and can both be accessed and used in experiments remotely via the w-iLab.t online interface. As a result of these hardware incorporations many testbeds, which would previously not have been capable of advanced sensing functionality, now are.

Section 2 provides updated baseline functionality information for each of the testbeds and platforms, as well as discussing hardware incorporations from one federated testbed to another, the benfits of these incorporations and we also introduce the nomadic use setup for testbed hardware and tools.

As can also be seen in the table, the CREW common data format has been extensively used by the vast majority of CREW partners, this in combination with the definitions of benchmark scenarios for each of the CREW testbeds enables reproducibility and transferability of results within the CREW federation.

Section 0 presents additions and changes to the CREW common data collection and storage methodology. With the introduction of the CREW repository [1] CREW experimental data and environment definition can now be shared easily, making use of the common data format. Within the last year also the common data format has transitioned from JSON specification to XML in order to improve ease of use. In order to facilitate this transition an XML common data format validation tool has been included on the CREW portal, which is also discussed in Section 0.

Currently information on all of the open access testbeds and sensing platforms is available through the CREW Portal, while information on the TCS multi- antenna sensing platform and the EADS airplane cabin mock-up (which are not open access but may, however, still be made available for specific experiments) are provided within this document and D6.2.

Section 4 provides discussion of new additions to the common Portal made within the second year of the project as well as a discussion of portal usage to date.

The final row of Table 1 demonstrates the extensiveness of the use of the CREW federated testbeds by other CREW partners, external users and open call experimenters. Not shown in this table is how different components and platforms were combined within collaborative experiments.

At the start of Section 5 Table 4 shows how combinations between the different platforms and testbeds were performed, a list of the experiments performed and details on where to look for further information on each is also provided. This makes up a large part of the interoperability testing of the CREW project and demonstrates to external users examples of the large number of component combination experiments made possible by the CREW federation. Also included in this section as part of the interface design and specification work of CREW we provide detailed interface information for both the VESNA outdoor testbeds and the TCS transceiver API, parts of which are included in Annexes.

The document is then rounded up and concluded in Section 6.

1.2 Document Purpose and intended audience

This document is intended to provide a main reference to anyone interested in the usage of the CREW Federation, which is up to date at the end of year 2 of the CREW project. It should provide enough information to clearly grasp the capabilities of the Federation in terms of available functionalities so a potential external user may be able to make an assessment on the feasibility of the experiment he or she could have in mind.

1.3 References and other work package deliverables

This document builds upon, and ties in with, information provided in a number of previous documents. The main one of these is D3.1: “Basic Operational platform”, for ease of reference the two documents are structured in the same way.

2 CREW federation optimised functionalities

This section provides an updated baseline analysis of the new and optimised federation functionalities of each of the CREW federation testbeds, as well as, the IMEC sensing platform and the TCS multi-antenna LTE sensing platform. This document serves as an update on the information provided in D3.1 and for this reason the details provided in D3.1 Section 2 are not repeated here, with the exception of those that have changed..

Figure 1 shows the federation as it is currently, going into the second open call.

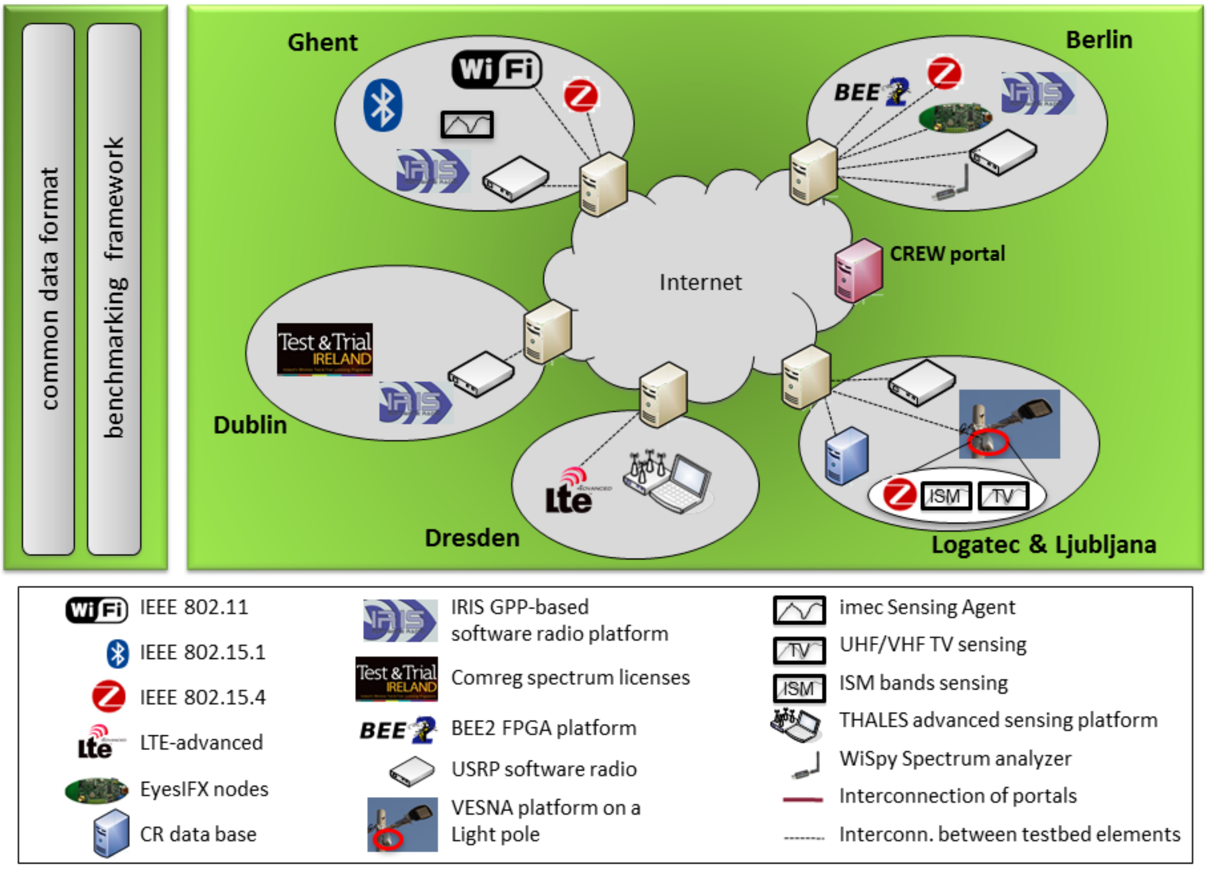


Figure 1: CREW federation at year 2 of the project

While in this figure the federated testbed is displayed as a number of separate islands, in reality strong interconnection between the hardware, software and control of the testbeds has developed in the course of the second year of the project.

The main additions include the incorporation of the Logatec and Ljubljana testbeds as well as the TCS multi-antenna LTE sensing platform (currently in experimental use in the Dresden island). As is also shown in the figure there have been numerous cases of hardware from one testbed being incorporated into other testbeds, for example: the addition of Iris to both the Ghent and Berlin islands. All testbeds are also currently connected through the CREW Portal [2] and access information for each is provided there.

This section provides testbed-by-testbed descriptions of each of the new and optimised federation functionalities.

2.1 TCD Iris testbed optimised functionalities

2.1.1 Iris Testbed equipment

In order to enable more network oriented experimentation, the Iris testbed has expanded from 4 remote access nodes to 8, as well as 2 local-only access nodes. The layout is shown in Figure 1.

As stated in D5.2 Section 2.3.2, the four new nodes are DELL T1500 desktops with Intel Xeon 3.4GHz Quad core processors, each equipped with a USRP N210. Each of these nodes is connected to the gateway node in the same fashion as the previous nodes to allow remote access. As with the others, the use of the nodes is managed by a calendar booking system. All machines in the testbed have been updated and are now running Ubuntu 12.04. Each also has a separate Clonezilla installation for reimaging and a grub option to restore the main partition from the master image on ctvr-gateway (admin password needed).

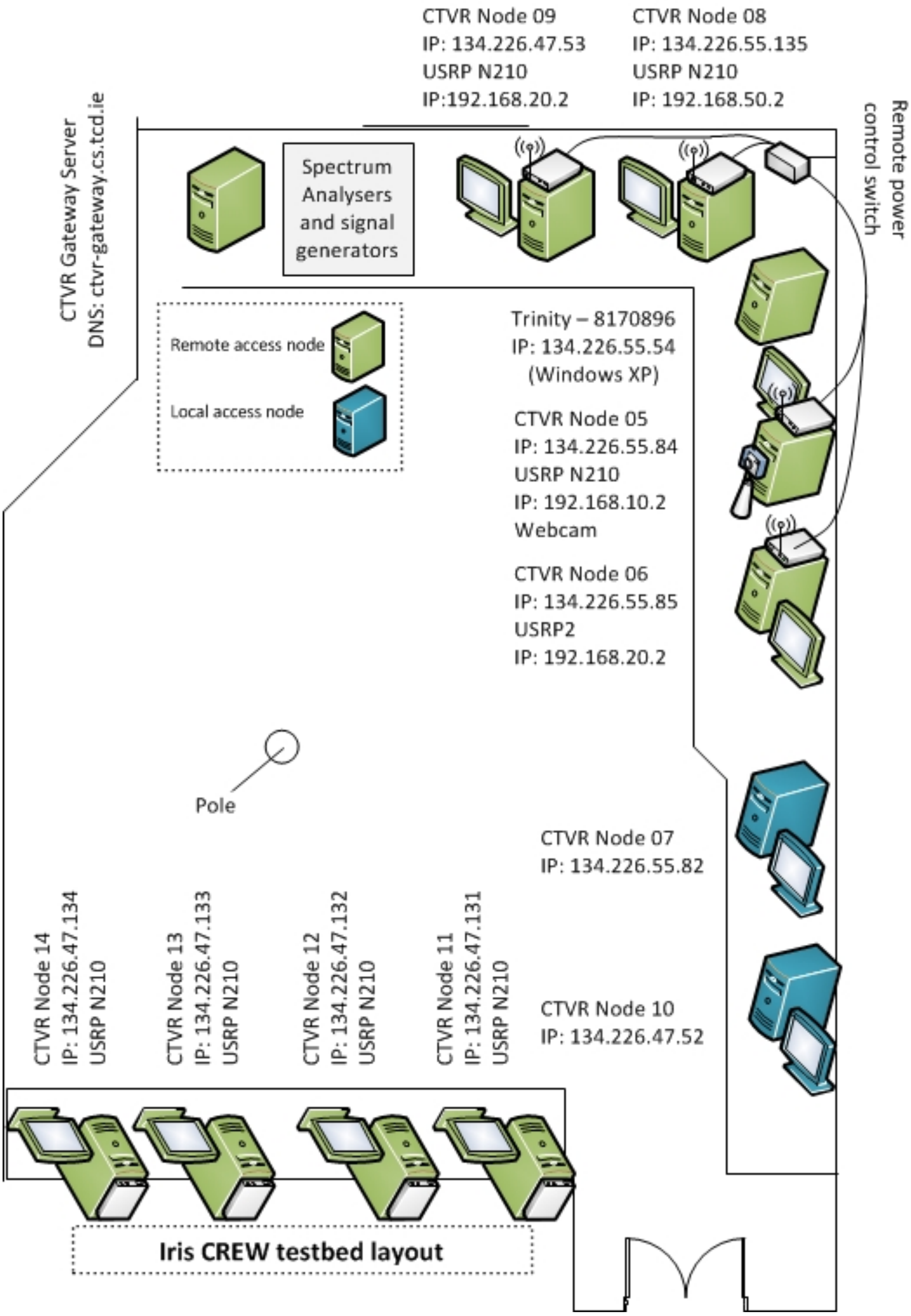


Figure 2: Testbed layout

Information on access to the testbeds is presented in D3.1 Section 2.1 as well as on the CREW Portal [2].

A camera has been installed next to testbed node 05 which allows experimenters to see what's going on in the testbed at the time of the experiment. It can be viewed by connecting via VNC to node 05 and using Cheese Photo Booth as a viewer.

2.1.2 Test and Trial Licence

Currently we have a TV-band Test and Trial licence from the Irish telecommunications regulator ComReg, starting on 10/08/2012 and valid for one year. The license is for 694-718 MHz, with channel bandwidth of 4 MHz and maximum ERP of -10 dbW.

Additional licences can also be obtained if required.

In the testbed we also have a discone antenna for use in the TV bands of the following specifications:

Frequency range: RX band: 25-1300 MHz TX band (@ SWR <=2): 49.5-50.5, 120-180, 215-300, 415-465, 610-650, 710-1000, 1130-1300 MHz

Impedance: 50 Ohms

Radiation (H-plane): 360° omnidirectional

Materials: Stainless Steel, Nylon

Wind load / resistance: 66 N @ 150 Km/h / 130 Km/h

Wind surface: 0.04 m²

Polarization: linear vertical Gain: 0 dBd

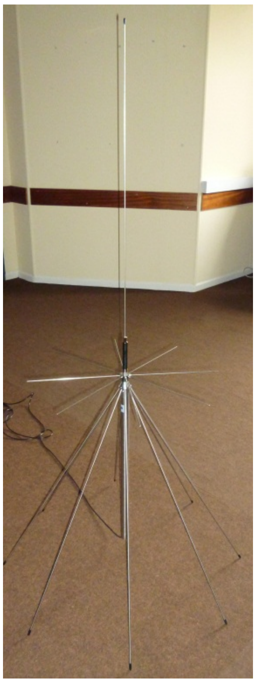


Figure 3: Discone wideband antenna

In addition to this each of the testbed nodes can be temporarily equipped with an Ettus WBX daughterboard [3] in order for TV-band Iris experiments to be performed.

2.2 TUB testbed optimised functionalities

The TUB testbed is a mature infrastructure for experimenting with sensor networks and cognitive radio equipment. It gives full flexibility in sensor node application programming. Until recently, however, it was hard to add new, non-wireless sensor related hardware to the testbed infrastructure. The CREW project has shown that it is necessary to add new functionalities as well as diverse hardware in parallel to the existing TWIST testbed, for example; to extend the infrastructure with further cognitive radio equipment and to enable the CREW mix and match concept (allowing experimenters to bring their own hardware). Furthermore we identified the need to extend the existing TWIST sensor network testbed by a mechanism that allows us to monitor the RF interference environment (and thus validate the experimental conditions) during or before an experiment. In the following, we first describe the extension of the TUB testbed with new hardware components and network infrastructure from the CREW project; afterwards we report on the extensions we made to the existing TWIST sensor network testbed.

2.2.1 CREW Segment in the TUB Testbed

In order to support the growing amount of new hardware used in the TUB testbed, we have added additional network infrastructure and a gateway server, which is dedicated to further testbed extensions. This solution is flexible enough to support any new network attached solution. The new server is an access point to all new hardware and makes it easy to couple together with the existing TWIST wireless sensor network testbed. An architectural overview of the extended infrastructure can be seen in Figure 4.

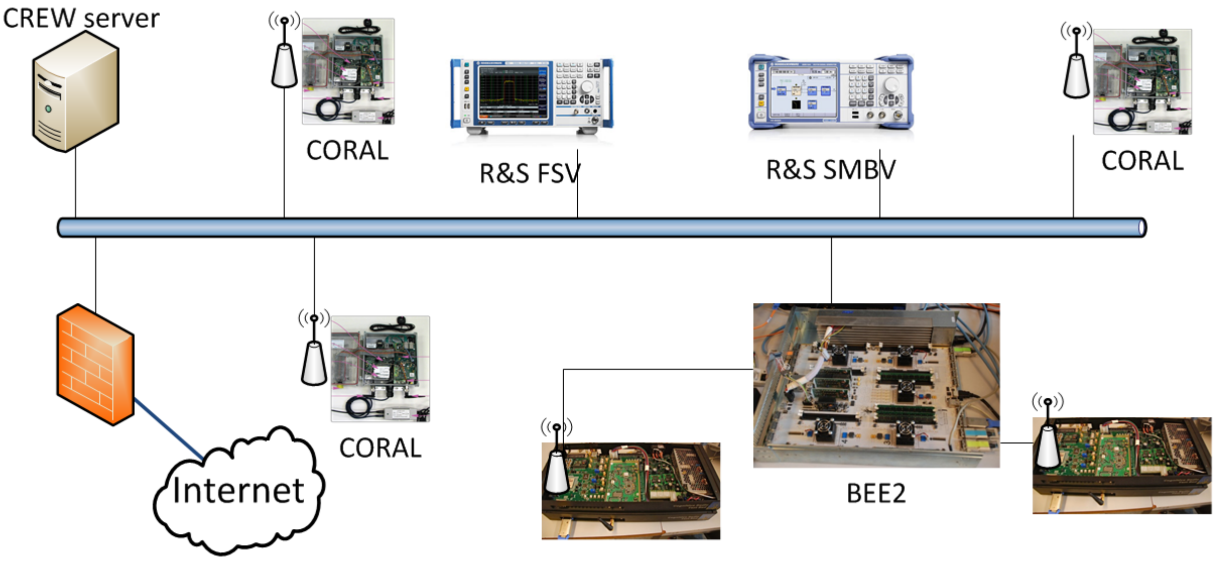


Figure 4: The dedicated CREW server and subnet created during Year 2

To control the CREW segment of the TUB testbed a generic approach is very useful to provide flexibility between the different infrastructures of certain testbeds within the federation. Such a control framework is OMF (cOntrol and Management Framework[4][5]). It allows entities to control the testbed, perform measurements and manage the results. OMF runs in a distributed manner across multiple different components of the testbed. The current status of OMF at the TUB testbed is as follows: the Aggregate Manager is installed at the CREW server and ready to run. Also an Experiment Controller is provided by the server. At the moment the Experiment Controller cannot yet be used to control the testbed, because of how the Resource Controller (RC) is typically used and that OMF needs to install an image via PXE. But none of TUB testbed nodes are using a pure Linux OS, which can be booted via PXE. The planned solution is to run the RC in a virtual machine or host which controls the components via a wrapper interface (planned to be investigated within year 3).

To enable more advanced spectrum sensing experiments we have added both a high end Rohde & Schwarz SMBV100A Signal Generator and an FSV Spectrum Analyzer. Both devices are attached to the new network infrastructure and can be remotely controlled, either by their native remote desktop connections or by scripts. Further integration of the scripts will allow usage of both of these devices to act as nodes in the OMF experiment control framework. Attaching these devices to the network enables us to relocate them, on demand, to anywhere in the building.

In the following we give an overview of the hardware components that have been added to the CREW segment and how they are supported by OMF.

Signal Generator: Here a wrapper is implemented between the Standard Commands for Programmable Instruments (SCPI) interface and OMF. This wrapper is written in the programming language ruby and is ready to run. For communication raw TCP is used. This code example shows the main implementation of running an 802.11 waveform file in 2.484GHz for one second.

```
  smbv = RsSMBV.new(options[:host])
  smbv.connect
  smbv.command("*RST;")
  smbv.freq("2kk")
  smbv.freq("2.484GHz") #Channel 14
  smbv.command(":SOURce:POWer:LEVel:IMMediate:AMPLitude -5dBm")
  smbv.command("BB:ARB:WAV:SEL '/hdd/waveforms/802.11a.wv'")
  smbv.command("BB:ARB:WAV:DATA? \
'/hdd/waveforms/802.11a.wv','comment'")
  smbv.command(":BB:ARB:STATe on")
  smbv.command(":OUTP:STATe on")
  smbv.command(":OUTP:STATe?")
  sleep 1
  smbv.command(":OUTP:STATe off")
  smbv.command(":OUTP:STATe?")
  smbv.disconnect()
```

Spectrum Analyzer: Here a wrapper is implemented between the SCPI interface and OMF as well. This wrapper is using the programming language ruby and is ready to run, too.

TWIST: As the TWIST testbeds contains embedded nodes (telosb) it is not possible to run controller software directly on the nodes. An interface to access and control the nodes is implemented by the REpresentational State Transfer (REST) API. To provide the bridge between OMF and TWIST a wrapper interface between REST and OMF is needed.

BEE2: The BEE2 platform is attached to CREW server and can be accessed from there over native interfaces. However, there are still problems with the configuration of the BEE2 hardware and until these have been solved an integration into OMF is being deferred.

CORAL: The CORAL platform is in the process of integration. Its OpenWRT OS makes it a candidate for implementation and controllability by OMF. An example implementation of running OMF within OpenWRT is given in [6].

IMEC Sensing Agent: We have borrowed the IMEC Sensing Agent to integrate non-permanently it into the TWIST extended infrastructure. The goal was to make the sensing agent hardware plug and play device within the testbed. As the Imec Sensing Agent is already working at the IBBT testbed further integration into the TWIST extended infrastructure will be based on the IBBT OMF implementation. With the new CREW network infrastructure at TUB it is now possible to place new devices anywhere in the building and connect them to the common control network. The building is an office building where the testbed covers 3 floors with approximately 40 rooms. The recent extensions of the CREW project now enable experimenters to place their equipment in any of the 40 rooms and connect to the CREW infrastructure via Ethernet (each room has a dedicated CREW Ethernet socket).This enables experimenters to easily access the measurement data and to control any connected device remotely.

2.2.2 Extensions of the TWIST testbed

During several experiments we had identified that it would be helpful to monitor the very crowded 2.4GHz ISM band during an experiment for uncontrolled RF interference. To this end we have added a set of commercial, low-cost USB spectrum analyzers WiSpy-2.4x to the TWIST. As explained in Deliverable 4.2 the spectrum analyzers can be used before an experiment to check RF interference conditions and decide if and where (what frequency range) to start the experiment; or during an experiment to validate the experiment conditions (RF interference environment). To this end we have connected WiSpy devices via the USB interface to the "supernodes" in the TWIST testbed. The TWIST supernodes are network-attached storage devices that are already placed in every room of the office building to provide access to the TWIST sensor nodes (reprogramming, interaction during experiments over a serial channel). Therefore it was more convenient to add the WiSpy devices to the existing TWIST infrastructure than to the new CREW segment described above. A WiSpy device sweeps the spectrum starting at 2.4 to 2.5 GHz by increasing the center frequency of a radio receiver by a given step size for a given number of times. The center frequency is held constant over a configurable time interval. Filtering of the receivers output is done with an adjustable filter bandwidth. This way the user obtains the signal power for individual frequency ranges with equal width. The WiSpy is capable of sweeping a range with some thousand steps with a minimal step size of 23.5 kHz and the filter bandwidth needs to have a minimal value of 53.6 kHz. The dwell time ranges from 10 µs to 2.55 ms in increments of 10 µs.

The extension of the TWIST testbed required considerable software development efforts: first the software component for accessing the WiSpy on the supernodes was developed. This component, called "spectrumserver" adds the ability to set and read the parameters of a connected WiSpy and to pass the retrieved spectrum sweeps on to the TWIST server as well. Spectrumserver consists of several parts written in C++ of which the device driver, the control channel and the data channel are the most important ones. Each of them is polled within a control loop and is continuously checked for errors.

Furthermore, the TWIST server was extended to interact with the spectrumserver processes running on the suppernodes. The implementation of the new TraceServer involved introducing a new dump manager which takes care of job instances which in turn handles the control routines for each spectrumserver. All data is dumped into a NFS replicated directory to provide the web server with the dump files. Finally, the user interface and webserver was extended, to incorporate the controls for spectrum sensing and dump file retrieval. Users have the possibility to start tracing RF interference measurements during an experiment and download the data (in a file) during or after the experiment has finished. Note that since the CREW segment is co-located with TWIST (using the same rooms in the same building) the WiSpy devices can naturally be used in experiments conducted on the CREW segment as well. More information on the integration of the WiSpy devices is provided in Deliverable D5.2.

2.3 TUD LTE+ Testbed optimised functionalities

2.3.1 Spectrum License for 2.1 GHz Band

The LTE testbed at TUD has been upgraded with a new spectrum license in the 2.1 GHz band (1980 MHz to 2000 MHz and 2170 MHz to 2190 MHz). This step was necessary to ensure the continuous operation of the LTE testbed when the spectrum license for 2.6 GHz is withdrawn due to commercial use of the corresponding frequencies in Germany. Along with the license, several nodes have been equipped with 2.1 GHz frontends. Note that only the RF parts have been replaced, while LTE eNB and UE baseband processing remains unaffected due to the modular structure of the equipment. Further note that as long as the 2.6 GHz license is not withdrawn, those frequencies are still available for experimentation.

The operation of the new equipment has been successfully tested. The internal US5 experiment "LTE Multi-Antenna Sensing" has been conducted in the 2.1 GHz frequency range.

2.3.2 Secondary System Implementation on Signalion HaLo Platform

The Signalion Hardware-in-the-Loop (HaLo) is a platform designated to simplifying the transition from simulation to implementation. To support cognitive radio setups that consider a primary/secondary user configuration, the LTE testbed has been extended by a HaLo node that can take the role of the secondary user. On the HaLo device, a novel modulation scheme called Generalized Frequency Division Multiplexing (GFDM) [7] is now available.

This enables experimenters to consider experiment setups in LTE testbed, where the LTE system can act as a monitored primary system, while the GFDM system can run as an interfering secondary system.

2.3.2.1 HaLo Concept

The HaLo consists of a wireless transceiver that can operate in the 2.6 GHz frequency band. The concept is such, that complex valued data samples are transmitted to the device's memory via USB from a control computer. The samples can be either read from a previously recorded file or generated on the fly e.g. by a Matlab script. The signal is transmitted over the air and received in a similar way. The device digitizes the signal and stores complex valued samples to an internal memory before they are fed back via USB to the control computer.

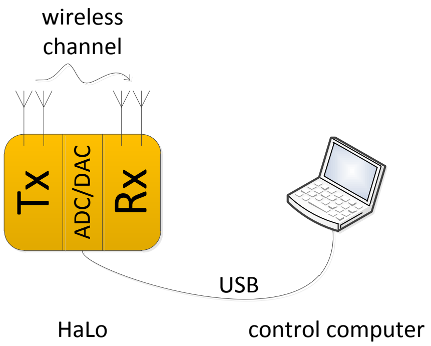


Figure 5: The HaLo setup

Note that due to limitations in the HaLo's internal memory real-time operation is not possible.

2.3.2.2 GFDM Theory

The transmission scheme chosen to be implemented on the HaLo device to act as a secondary system in the testbed is a novel, non-orthogonal and flexible modulation scheme called GFDM. The concept is such that a multicarrier signal is transmitted, quite similar to the well know and established OFDM scheme, however one of the differences is in the pulse shaping of the individual subcarriers. This step allows shaping of transmissions and produces a signal with particularly low out-of-band radiation, which is a very desirable property in cognitive radio. For further details please refer to [7].

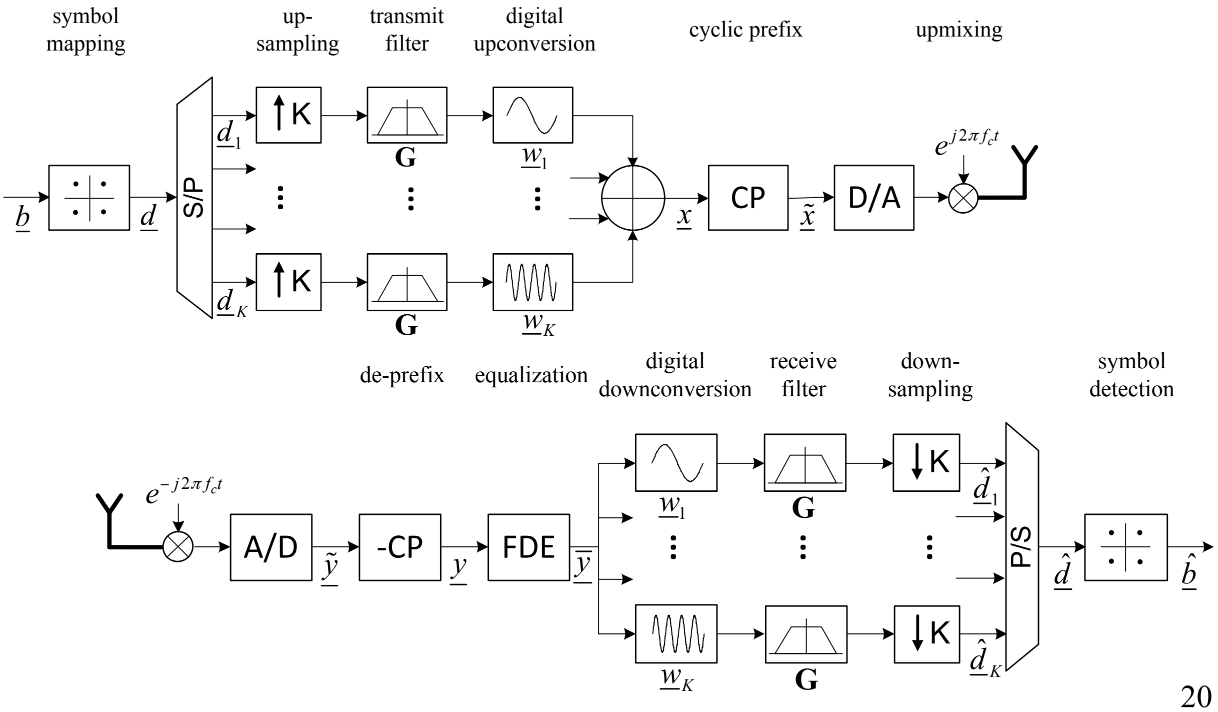


Figure 6: GFDM transmitter and receiver block diagram

2.4 IMEC sensing platform optimised functionalities

In deliverable D3.1 a high-level overview was given on the IMEC sensing engine and the software integration. A full description of the functions and implementation is out of the scope of this document, however this information is available on the CREW portal [8]. An overview of the configurations of the sensing platform is provided below, this describes the modes of operation currently available with the sensing engine, which is integrated in the w-iLab.t of the IBBT in Zwijnaarde (Belgium). Two variants of the IMEC sensing engine are available, both use the in house developed DIFFS [9] chip, in one instance this is connected to the commercially available WARP radio board [10] and the in the second instance connected to the in house developed SCALDIO2B SDR RFIC [11].

Below is the list of available configurations of the sensing engine and the output of each:

2.4.1 FFT_SWEEP

short description: sweep a part of the spectrum and perform an FFT calculation on each subband. There are 291 subbands available for Scaldio2b, numbered from 1 to 291; there are 28 subbands available for WARP, numbered from 1 to 28. Each subband is 20 MHz wide, spread around the channel-frequency. ADC sampling speed is 40 MHz for both Scaldio2b and WARP.

fe_gain: gain setting of the analog front-end; between 27 and 100 for Scaldio2b, between 5 and 100 for WARP (100 being maximum gain).

first_channel and last_channel: the first and last channel to sweep. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel.

Scaldio2b: the channel-frequency equals 200 MHz + 20 MHz * (channel number-1). Must be an integer number between 1 and 291.

WARP: the channel-frequency equals

- 2.412 GHz + 20 MHz * (channel number - 1) for channel 1 to 4;
- 2.84 GHz for channel 5;
- 5.18 GHz + 20 MHz * (channel number - 6) for channel 6 to 13;
- 5.5 GHz + 20 MHz * (channel number - 14) for channel 14 to 24;
- 5.745 GHz + 20 MHz * (channel number - 25) for channel 25 to 28.

bandwidth: not applicable. Fixed at 10 MHz for Scaldio2b and 10.45 MHz for WARP.

fft_points: number of bins in the FFT-calculation. Must be 16, 32, 64 or 128. Currently only 128 supported.

dvb_nr_carriers: not applicable in this mode.

dvb_guard_interval: not applicable in this mode.

threshold_power: not applicable in this mode.

output: each sweep yields an array with the number of channels times the number of FFT-points values. Every value is a logarithmic power value for the corresponding FFT-bin. Note: take into account an 8 MHz overlap between channel 4 and 5 in case the WARP analog FE is used.

2.4.2 WLAN_G

short description: determine the instantaneous power in a number of IEEE 802.11g channels. There are 14 channels available, numbered from 1 to 14, following the IEEE 802.11g standard. Each channel is 5 MHz wide, spread around the channel-frequency. ADC sampling speed is 40 MHz for both Scaldio2b and WARP.

fe_gain: gain setting of the analog front-end; between 27 and 100 for Scaldio2b, between 5 and 100 for WARP (100 being maximum gain).

first_channel and last_channel: the first and last channel to take into account. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel. The channel frequency equals 2.412 GHz + 5 MHz * (channel number - 1) for channel 1 to 13 and equals 2.484 GHz for channel 14.

bandwidth: not applicable. Fixed at 10 MHz for Scaldio2b and 10.45 MHz for WARP.

fft_points: not applicable in this mode.

dvb_nr_carriers: not applicable in this mode.

dvb_guard_interval: not applicable in this mode.

threshold_power: threshold to compare the instantaneous power to.

output: each run yields an array with the number of channels times two. Every first value is a logarithmic power value for the corresponding channel; every second value is 0 in case the power value is lower than the threshold or 1 otherwise.

2.4.3 WLAN_A

short description: determine the instantaneous power in a number of IEEE 802.11a channels. There are 23 channels available, numbered from 36 to 64 in steps of 4, from 100 to 140 in steps of 4 and from 149 to 161 in steps of 4, following the IEEE 802.11a standard. Each channel is 20 MHz wide, spread around the channel-frequency. ADC sampling speed is 40 MHz for both Scaldio2b and WARP.

fe_gain: gain setting of the analog front-end; between 27 and 100 for Scaldio2b, between 5 and 100 for WARP (100 being maximum gain).

first_channel and last_channel: the first and last channel to take into account. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel. The channel frequency equals 5 GHz + 5 MHz * channel number.

bandwidth: not applicable. Fixed at 10 MHz for Scaldio2b and 10.45 MHz for WARP.

fft_points: not applicable in this mode.

dvb_nr_carriers: not applicable in this mode.

dvb_guard_interval: not applicable in this mode.

threshold_power: threshold to compare the instantaneous power to.

output: each run yields an array with the number of channels times two. Every first value is a logarithmic power value for the corresponding channel; every second value is 0 in case the power value is lower than the threshold or 1 otherwise.

2.4.4 ZIGBEE

short description: determine the instantaneous power in a number of IEEE 802.15.4 Zigbee channels. There are 16 channels available, numbered from 11 to 26, following the

IEEE 802.15.4 standard. Each channel is 2 MHz wide, spread around the channel-frequency. ADC sampling speed is 40 MHz for both Scaldio2b and WARP.

fe_gain: gain setting of the analog front-end; between 27 and 100 for Scaldio2b, between 5 and 100 for WARP (100 being maximum gain).

first_channel and last_channel: the first and last channel to take into account. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel. The channel frequency equals 2.35 GHz + 5 MHz * channel number.

bandwidth: not applicable. Fixed at 10 MHz for Scaldio2b and automatically calculated for WARP.

fft_points: not applicable in this mode.

dvb_nr_carriers: not applicable in this mode.

dvb_guard_interval: not applicable in this mode.

threshold_power: threshold to compare the instantaneous power to.

output: each run yields an array with the number of channels times two. Every first value is a logarithmic power value for the corresponding channel; every second value is 0 in case the power value is lower than the threshold or 1 otherwise.

2.4.5 DVB_T

short description: cyclostationary detection for channels of the DVB-T standard. There are 51 channels available, numbered from 16 to 66. Each channel is 8 MHz wide, spread around the channel-frequency. ADC sampling speed is 40 MHz for Scaldio2b. This mode is not available for WARP front-ends.

fe_gain: gain setting of the analog front-end; between 27 and 100 (100 being maximum gain).

first_channel and last_channel: the first and last channel to take into account. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel. The channel frequency equals 434 MHz + 8 MHz * (channel number - 16).

bandwidth: not applicable. Fixed at 10 MHz.

fft_points: not applicable in this mode.

dvb_nr_carriers: the number of carriers per symbol. Set to 2048 for 2k mode or 8192 for 8k mode.

dvb_guard_interval: portion of the symbol that is copied in front of the signal, in order to create a cyclic signal and hence avoid ISI. Floating point value which must be set to 1/4, 1/8, 1/16 or 1/32.

threshold_power: threshold to compare the instantaneous power to.

output: each run yields an array with the number of channels times two. Every first value is a qualifier for the cyclostationary property of the received signal for the corresponding channel; every second value is 0 if the first value is lower than the threshold or 1 otherwise.

2.4.6 ISM_POWER_DETECT

short description: determine the instantaneous power in a number of ISM band channels. There are 89 channels available, numbered from 1 to 89. Each channel is 2 or 4 MHz wide, spread around the channel-frequency. ADC sampling speed is 40 MHz for both Scaldio2b and WARP.

fe_gain: gain setting of the analog front-end; between 27 and 100 for Scaldio2b, between 5 and 100 for WARP (100 being maximum gain).

first_channel and last_channel: the first and last channel must be integer numbers and last_channel must always be equal to first_channel. The channel frequency equals 2.404 GHz + 1 MHz * (channel number - 1).

bandwidth: 1 MHz or 2 MHz, depending on the width of the channel. Enter a value in Hz.

fft_points: not applicable in this mode.

dvb_nr_carriers: not applicable in this mode.

dvb_guard_interval: not applicable in this mode.

threshold_power: threshold to compare the instantaneous power to.

output: each run yields an array with the number of channels times two. Every first value is a logarithmic power value for the corresponding channel; every second value is 0 in case the power value is lower than the threshold or 1 otherwise.

2.4.7 ADC_LOG1

short description: log the time-domain samples coming out of the ADC of Scaldio2b. There are 901 channels available, numbered from 1 to 901, which correspond to every possible Scaldio2b carrier frequency.

fe_gain: gain setting of the analog front-end; between 27 and 100 for Scaldio2b (100 being maximum gain).

first_channel and last_channel: the first and last channel to take into account. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel. The channel frequency equals

- 93.75 MHz + 625 kHz * (channel number - 1) for channel 1 to 151;
- 187.5 MHz + 1.25 MHz * (channel number - 151) for channel 151 to 301;
- 375 Hz + 2.5 MHz * (channel number - 301) for channel 301 to 451;
- 750 MHz + 5 MHz * (channel number - 451) for channel 451 to 601;
- 1.5 GHz + 10 MHz * (channel number - 601) for channel 601 to 751;
- 3 GHz + 20 MHz * (channel number - 751) for channel 751 to 901;

bandwidth: not applicable. Fixed at 10 MHz.

fft_points: not applicable in this mode.

dvb_nr_carriers: not applicable in this mode.

dvb_guard_interval: not applicable in this mode.

threshold_power: not applicable in this mode.

output: each run yields an array of samples with the number of channels times two times 1023 floats. Every first value is the real part of the sample, every second value the imaginary part. Only the number of samples indicated by the function call that fetches the result is valid.

2.4.8 ADC_LOG2

short description: log the time-domain samples coming out of the ADC of WARP. There are 37 channels available, numbered from 1 to 37, which correspond to every possible WARP carrier frequency.

fe_gain: gain setting of the analog front-end; between 5 and 100 for WARP (100 being maximum gain).

first_channel and last_channel: the first and last channel to take into account. Both must be integer numbers and last_channel must always be bigger than or equal to first_channel. The channel frequency equals

- 2.412 GHz + 5 MHz * (channel number - 1) for channel 1 to 13;
- 2.484 GHz for channel 14;
- 5.18 GHz + 20 MHz * (channel number - 15) for channel 15 to 22;
- 5.5 GHz + 20 MHz * (channel number - 23) for channel 23 to 33;
- 5.745 GHz + 20 MHz * (channel number - 34) for channel 34 to 37;

bandwidth: selectable from following values: 6.75 MHz, 7.125 MHz, 7.5 MHz, 7.875 MHz, 8.25 MHz, 8.55 MHz, 9.025 MHz, 9.5 MHz, 9.975 MHz, 10.45 MHz, 12.6 MHz, 13.3 MHz, 14 MHz, 14.7 MHz, 15.4 MHz, 16.2 MHz, 17.1 MHz, 18 MHz, 18.9 MHz and 19.8 MHz. Enter a value in Hz.

fft_points: not applicable in this mode.

dvb_nr_carriers: not applicable in this mode.

dvb_guard_interval: not applicable in this mode.

threshold_power: not applicable in this mode.

output: each run yields an array of samples with the number of channels times two times 24575 floats. Every first value is the real part of the sample, every second value the imaginary part. Only the number of samples indicated by the function call that fetches the result is valid.

2.5 IBBT w-iLab.t testbed optimised functionalities

In deliverable D3.1, it was reported that an extension to the w-iLab.t facilities was being set up in Zwijnaarde (Belgium), a town not far from the initial w-iLab.t deployment which is located in an office building in Ghent, Belgium. In the meantime, this extension to w-iLab.t became operational and is known under the name "w-iLab.t Zwijnaarde". The w-iLab.t Zwijnaarde testbed is located on top of a clean room, in an unmanned utility room.

2.5.1 Zwijnaarde testbed

Although it should be noted that the CREW project is certainly not the only reason for which the extension to the testbed was made, and that the work involved in setting up the foundations of this new testbed location is also not a CREW effort, there are several clear CREW accents and CREW efforts in the Zwijnaarde testbed; While also true for wireless experiments in general, when performing cognitive radio and cognitive networking experiments, the amount of (uncontrollable) external interference caused by wireless transmitters external to the experiment, should be as low as possible. In w-iLab.t Zwijnaarde, the **amount of external interference is kept at a minimum** for two reasons: the absence of people and the application of a copper shielding foil on ducts that go to those parts of the building where there might be wireless activity because of human activities.

2.5.2 Extended and easier mix-and-match (1).

Deliverable D3.1 already discussed the *possibility* of installing the IRIS software platform on USRP hardware located in w-iLab.t. During the second year of CREW, the IRIS platform was often used on top of the w-iLab.t USRP hardware, either to generate interference, or to scan the spectrum during experiments. To make it also simple for end- users to use the IRIS platform remotely in the w-iLab.t

Zwijnaarde, a server image was made available to experimenters, already containing an installation of IRIS. Additionally, the use of the USRPs with IRIS was documented on the CREW portal [12]. An example IRIS configuration, configuring the USRPs as spectrum analyser, is made available on the w-iLab.t servers. Note that the use of IRIS is not compulsory: the experimenters may also decide to opt for other frameworks, such as GNU radio.

2.5.3 Extended and easier mix-and-match (2).

In addition, the integration of the IMEC spectrum sensing agents in the w-iLab.t Zwijnaarde was improved and remote interaction was simplified. Details are available on the CREW portal.

2.5.4 Improved benchmarking compatibility and transparency.

Mainly in the scope of the OpenLAB project (www.ict-openlab.eu/*)*, the choice was made to operate the w-iLab.t Zwijnaarde testbed using the OMF tools [4], [5], that seem to become the de-facto standard for experiment control in testbeds worldwide. As the benchmarking framework implementation for the w-iLab.t was being reworked (see deliverable D4.2) to make it less testbed dependent and more transparent to the user, the benchmarking framework was now also rewritten using OMF APIs. As a result, using the benchmarking framework has become more user-friendly. Additionally, several w-iLab.t "interference background environments" were uploaded to the CREW repository (see Section 4), which is part of the CREW portal. These environments can be reused and/or modified by experimenters.

2.5.5 Improved information on the portal.

After the initial deployment of the CREW portal in the first year of CREW, the portal has proven to be a very useful source of information, both for existing w-iLab.t users and for new experimenters. To guarantee the continued relevance of the CREW portal, the w-iLab.t content on the CREW portal was updated and extended several times.

2.5.6 Nomadic use of testbed hardware and tools.

In order to acquire measurements and generate interference in an easy and controlled way outside the IBBT testbed environment, a nomadic w-iLab.t testbed set-up was constructed and used during a common CREW measurement session in an airplane mock-up at EADS. The mobile testbed set-up consists of a small form factor PC (as also used in the w-iLab.t Zwijnaarde testbed environment) containing a minimum set of testbed management tools, together with a number of Ethernet-connected embedded PCs (of the type used in the w-iLab.t Office testbed), which serve as proxy devices to which different sensing devices can be collected. Figure 7 shows such proxy embedded PC (left), which is used as a platform to gather timestamped measurements from the IMEC sensing agent (center) and a TelosB sensor node (right, labelled 'R6').

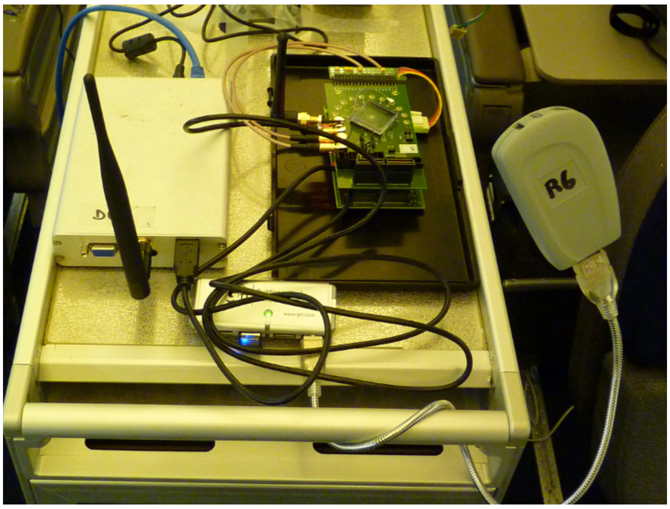


Figure 7: Mobile testbed set-up as used during the EADS plenary meeting

As indicated in the introduction of this section, please note that the above information only indicates the changes to the testbed functionality, compared to D3.1. For a full overview of functionalities, the reader can consult CREW deliverable D3.1, or the CREW portal at http://www.crew-project.eu/portal/wilabdoc.

2.6 TCS Multi-antenna LTE detection platform

2.6.1 Hardware description

2.6.1.1 Overall description

The multi-antenna LTE sensing platform allows us to acquire LTE (I, Q) data and to process using advanced antenna processing algorithms. As described in Figure 8, the multi-antenna platform is made of:

- A set of 4 antennas,
- A 4-channel receiver,
- A 4-channel acquisition board,
- A GPS system for positioning,
- A laptop for data storage and off-line multi-antenna LTE signal processing.

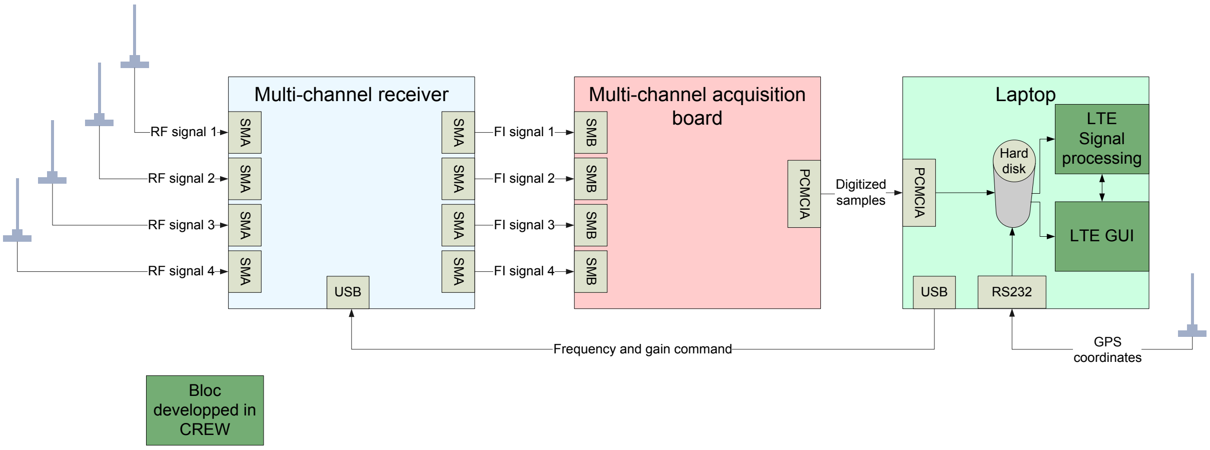


Figure 8: Multi-antenna sensing platform block diagram

A picture of the platform can be seen in Figure 9**Error! Reference source not found.**.

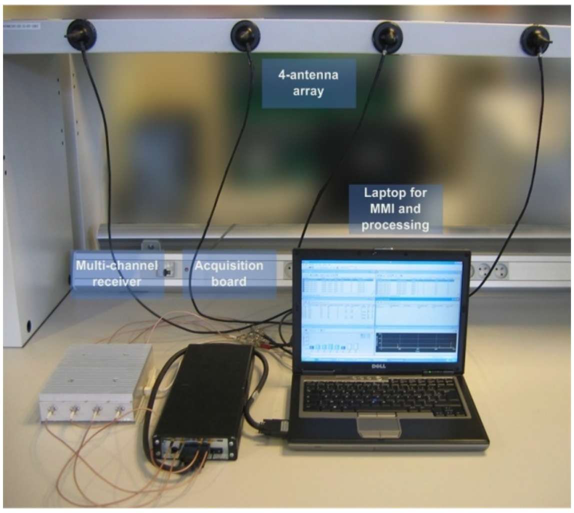


Figure 9: Multi-antenna sensing platform

2.6.1.2 Multi-channel receiver

The main characteristics of the multi-channel receiver are summarized in Table 2.

Table 2 : Multi-channel receiver main characteristics

Frequency bands	1920-1980 MHz / 2110-2170 MHz
Bandwidth	5 MHz
Output intermediate frequency	19.2 MHz
Noise factor (at maximal gain)	<7 dB
Rx gain	0 to 30dB (1 dB step)
Frequency step	200 kHz
Number of Rx channels	4
Gain dispersion	<1 dB
Phase dispersion	<6°
Frequency stability	$<10^{-7}$
Selectivity at ±5 MHz	>50 dB

2.6.1.3 Multi-channel acquisition board

The main characteristics of the multi-channel acquisition are summarized in Table 3.

Table 3 : Multi-channel acquisition board main characteristics

Resolution	12 bit
Internal quartz clock	15.36 MHz
Number of channels	4
-3 dB bandwidth	>25 MHz
Memory	8MSamples (i.e. 2MSamples per channel)

The four channels of the acquisition board are synchronized.

2.6.2 Brief software description

SMARTAIR 3G TCS software was extended to take into the LTE mode. It has three main operating modes:

- Antenna mode
- Replay mode
- Analysis mode

Antenna mode enables display and acquisition of the signals present on the antennas.

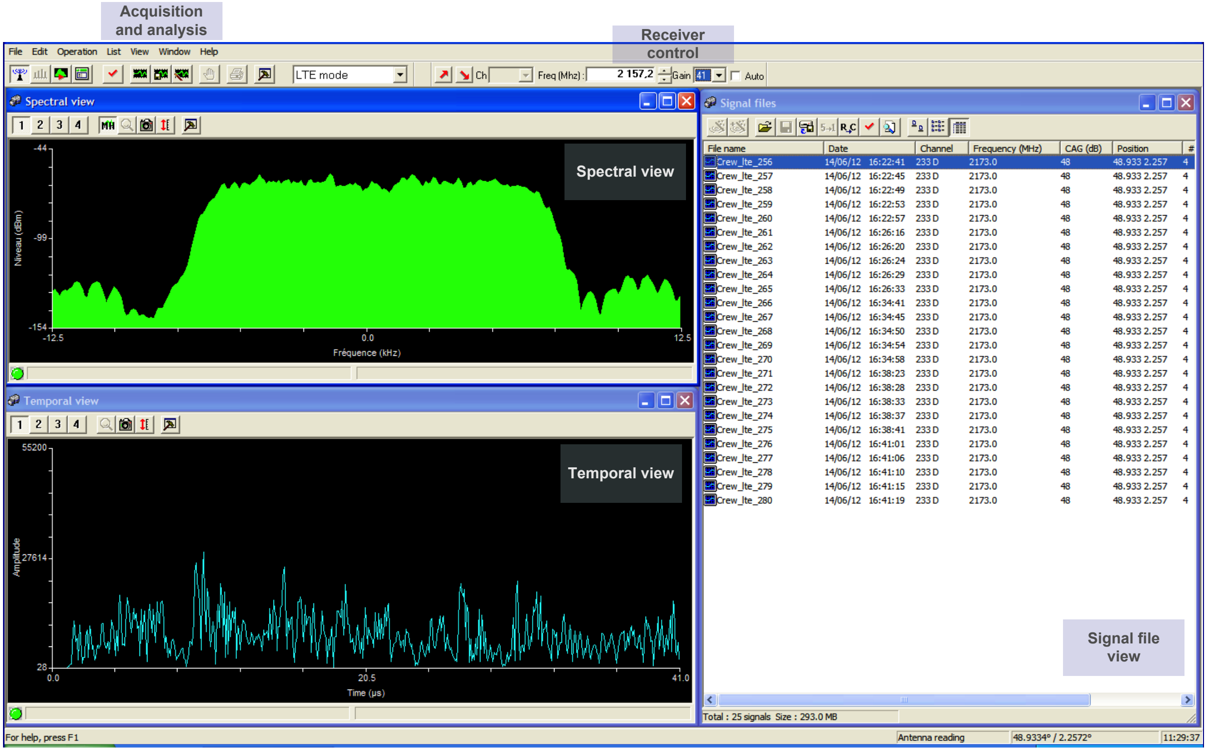
Replay mode enables off-line display of signals previously acquired on antennas.

Analysis mode enables display of results obtained from interference analysis.

2.6.2.1 Antenna mode

A screenshot of the antenna mode can be seen in. This mode enables display and acquisition of the signals present on the antennas. It includes the following functionalities:

- The *Spectral view* window displays in real-time of the spectral signal received on each antenna.
- The *Temporal view* window displays in real-time of the temporal signal received on each antenna.
- The *Signal file view* window displays the list of the acquired files.
- The *Receiver control* toolbar allows the control of the frequency and the gain of the receiver
- The *Acquisition and analysis* toolbar allows acquiring and/or saving and/or analyzing of the received signal.



LTE mode

Freq (Mhz) : 2 157,2

Gain 41

□ Auto

File name	Date	Channel	Frequency (MHz)	CAG (dB)	Position	#
Crew_lte_256	14/06/12 16:22:41	233 D	2173.0	48	48.933 2.257	4
Crew_lte_257	14/06/12 16:22:45	233 D	2173.0	48	48.933 2.257	4
Crew_lte_258	14/06/12 16:22:49	233 D	2173.0	48	48.933 2.257	4
Crew_lte_259	14/06/12 16:22:53	233 D	2173.0	48	48.933 2.257	4
Crew_lte_260	14/06/12 16:22:57	233 D	2173.0	48	48.933 2.257	4
Crew_lte_261	14/06/12 16:26:16	233 D	2173.0	48	48.933 2.257	4
Crew_lte_262	14/06/12 16:26:20	233 D	2173.0	48	48.933 2.257	4
Crew_lte_263	14/06/12 16:26:24	233 D	2173.0	48	48.933 2.257	4
Crew_lte_264	14/06/12 16:26:29	233 D	2173.0	48	48.933 2.257	4
Crew_lte_265	14/06/12 16:26:33	233 D	2173.0	48	48.933 2.257	4
Crew_lte_266	14/06/12 16:34:41	233 D	2173.0	48	48.933 2.257	4
Crew_lte_267	14/06/12 16:34:45	233 D	2173.0	48	48.933 2.257	4
Crew_lte_268	14/06/12 16:34:50	233 D	2173.0	48	48.933 2.257	4
Crew_lte_269	14/06/12 16:34:54	233 D	2173.0	48	48.933 2.257	4
Crew_lte_270	14/06/12 16:34:58	233 D	2173.0	48	48.933 2.257	4
Crew_lte_271	14/06/12 16:38:23	233 D	2173.0	48	48.933 2.257	4
Crew_lte_272	14/06/12 16:38:28	233 D	2173.0	48	48.933 2.257	4
Crew_lte_273	14/06/12 16:38:33	233 D	2173.0	48	48.933 2.257	4
Crew_lte_274	14/06/12 16:38:37	233 D	2173.0	48	48.933 2.257	4
Crew_lte_275	14/06/12 16:38:41	233 D	2173.0	48	48.933 2.257	4
Crew_lte_276	14/06/12 16:41:01	233 D	2173.0	48	48.933 2.257	4
Crew_lte_277	14/06/12 16:41:06	233 D	2173.0	48	48.933 2.257	4
Crew_lte_278	14/06/12 16:41:10	233 D	2173.0	48	48.933 2.257	4
Crew_lte_279	14/06/12 16:41:15	233 D	2173.0	48	48.933 2.257	4
Crew_lte_280	14/06/12 16:41:19	233 D	2173.0	48	48.933 2.257	4

Total : 25 signals Size : 293.0 MB

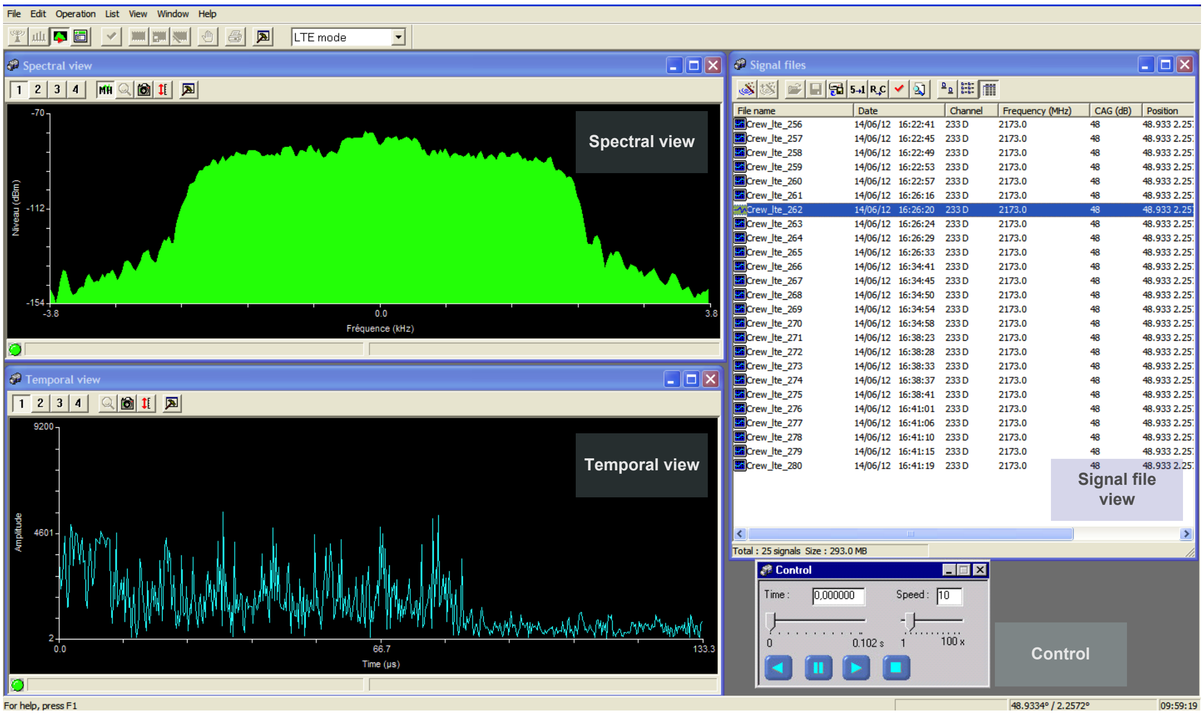
48.9334° / 2.2572° 11:29:37

Figure 10 : Antenna mode screenshot

2.6.2.2 Replay mode

A screenshot of the antenna mode can be seen in Figure 11. This mode enables off-line display of previously acquired signals. It includes the following functionalities:

- The *Spectral view* window displays the spectral signal received on each antenna of a previously saved file.
- The *Temporal view* window displays the temporal signal received on each antenna of a previously saved file.
- The *Signal file view* window displays the list of the acquired signal files.
- The *Control* window allows moving into the file.



LTE mode

File name	Date	Channel	Frequency (MHz)	CAG (dB)	Position
Crew_lte_256	14/06/12 16:22:41	233 D	2173.0	48	48.933 2.25
Crew_lte_257	14/06/12 16:22:45	233 D	2173.0	48	48.933 2.25
Crew_lte_258	14/06/12 16:22:49	233 D	2173.0	48	48.933 2.25
Crew_lte_259	14/06/12 16:22:53	233 D	2173.0	48	48.933 2.25
Crew_lte_260	14/06/12 16:22:57	233 D	2173.0	48	48.933 2.25
Crew_lte_261	14/06/12 16:26:16	233 D	2173.0	48	48.933 2.25
Crew_lte_262	14/06/12 16:26:20	233 D	2173.0	48	48.933 2.25
Crew_lte_263	14/06/12 16:26:24	233 D	2173.0	48	48.933 2.25
Crew_lte_264	14/06/12 16:26:29	233 D	2173.0	48	48.933 2.25
Crew_lte_265	14/06/12 16:26:33	233 D	2173.0	48	48.933 2.25
Crew_lte_266	14/06/12 16:34:41	233 D	2173.0	48	48.933 2.25
Crew_lte_267	14/06/12 16:34:45	233 D	2173.0	48	48.933 2.25
Crew_lte_268	14/06/12 16:34:50	233 D	2173.0	48	48.933 2.25
Crew_lte_269	14/06/12 16:34:54	233 D	2173.0	48	48.933 2.25
Crew_lte_270	14/06/12 16:34:58	233 D	2173.0	48	48.933 2.25
Crew_lte_271	14/06/12 16:38:23	233 D	2173.0	48	48.933 2.25
Crew_lte_272	14/06/12 16:38:28	233 D	2173.0	48	48.933 2.25
Crew_lte_273	14/06/12 16:38:33	233 D	2173.0	48	48.933 2.25
Crew_lte_274	14/06/12 16:38:37	233 D	2173.0	48	48.933 2.25
Crew_lte_275	14/06/12 16:38:41	233 D	2173.0	48	48.933 2.25
Crew_lte_276	14/06/12 16:41:01	233 D	2173.0	48	48.933 2.25
Crew_lte_277	14/06/12 16:41:06	233 D	2173.0	48	48.933 2.25
Crew_lte_278	14/06/12 16:41:10	233 D	2173.0	48	48.933 2.25
Crew_lte_279	14/06/12 16:41:15	233 D	2173.0	48	48.933 2.25
Crew_lte_280	14/06/12 16:41:19	233 D	2173.0	48	48.933 2.25

Total : 25 signals Size : 293.0 MB

Time : 0,000000

Speed : 10

48.9334° / 2.2572° 09:59:19

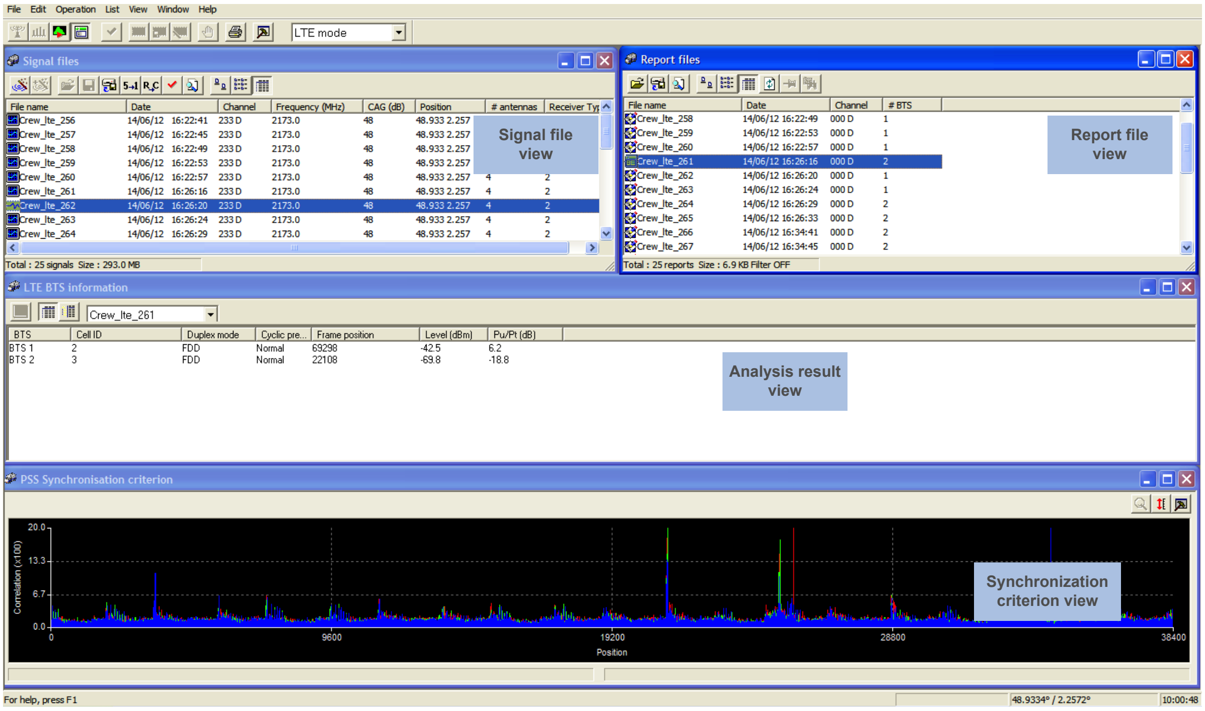
Figure 11 : Replay mode screenshot

2.6.2.3 Analysis mode

A screenshot of the analysis mode can be seen in Figure 12. This mode is used to display results obtained from interference analysis. It includes the following functionalities:

- The *Signal file view* window displays the list of the acquired signal files.
- The *Report file view* window displays the list of the report files from previous analysis.
- The *Analysis result view* window displays the list of the detected BTS with their characteristics (Cell ID, duplex mode, cyclic prefix length, frame position, level in dBm).
- The *Synchronization criterion view* window displays the primary synchronization criterion for the three possible sequences (one color per sequence).

The LTE multi-antenna processing is described in section 2.6.3.



File name	Date	Channel	Frequency (MHz)	CAG (dB)	Position	# antennas	Receiver Typ
Crew_lte_256	14/06/12 16:22:41	233 D	2173.0	48	48.933 2.257		
Crew_lte_257	14/06/12 16:22:45	233 D	2173.0	48	48.933 2.257		
Crew_lte_258	14/06/12 16:22:49	233 D	2173.0	48	48.933 2.257		
Crew_lte_259	14/06/12 16:22:53	233 D	2173.0	48	48.933 2.257		
Crew_lte_260	14/06/12 16:22:57	233 D	2173.0	48	48.933 2.257	4	2
Crew_lte_261	14/06/12 16:26:16	233 D	2173.0	48	48.933 2.257	4	2
Crew_lte_262	14/06/12 16:26:20	233 D	2173.0	48	48.933 2.257	4	2
Crew_lte_263	14/06/12 16:26:24	233 D	2173.0	48	48.933 2.257	4	2
Crew_lte_264	14/06/12 16:26:29	233 D	2173.0	48	48.933 2.257	4	2

Total : 25 signals Size : 293.0 MB

File name	Date	Channel	# BTS
Crew_lte_258	14/06/12 16:22:49	000 D	1
Crew_lte_259	14/06/12 16:22:53	000 D	1
Crew_lte_260	14/06/12 16:22:57	000 D	1
Crew_lte_261	14/06/12 16:26:16	000 D	2
Crew_lte_262	14/06/12 16:26:20	000 D	1
Crew_lte_263	14/06/12 16:26:24	000 D	1
Crew_lte_264	14/06/12 16:26:29	000 D	2
Crew_lte_265	14/06/12 16:26:33	000 D	2
Crew_lte_266	14/06/12 16:34:41	000 D	2
Crew_lte_267	14/06/12 16:34:45	000 D	2

Total : 25 reports Size : 6.9 KB Filter OFF

BTS	Cell ID	Duplex mode	Cyclic pre...	Frame position	Level (dBm)	Pu/Pt (dB)
BTS 1	2	FDD	Normal	69298	-42.5	6.2
BTS 2	3	FDD	Normal	22108	-69.8	-18.8

Figure 12 : Analysis mode screenshot

2.6.3 Multi-antenna LTE signal processing

The multi-antenna LTE detection procedure is described in D3.1. It is integrated in the SMARTAIR 3G software and can be run off-line on previously acquired signals, in the analysis mode.

This procedure allows detecting all the LTE BTS emitting on a frequency channel. It is made of two steps, the primary synchronization and the secondary synchronization. At the end of these two steps, for each detected BTS, the following information is known:

- The frame synchronization position
- The duplex mode (FDD or TDD)
- The length of the cyclic prefix (Normal or extended)
- The physical layer identity

The performance of this detection is described in D6.1 for two CREW usage scenarios (US1 and US5). In this deliverable the performance was presented only for the first part of the detection procedure that is the primary synchronization. Since then, additional simulations were run in order to evaluate the overall detection performance. This global performance is exactly the same as for primary synchronization only. It is due to the fact that the length, the periodicity and the autocorrelation properties of the secondary synchronization sequences are the same as the ones of the primary synchronization sequences.

Moreover, the level in dBm of the detected BTS is also estimated from the channel impulse response estimates.

2.7 JSI Outdoor VESNA based testbed supported functionalities

The JSI outdoor testbed LOG-a-TEC is based on a custom-developed low-cost Wireless Sensor Network (WSN) platform VESNA supporting spectrum sensing in UHF bands and spectrum sensing plus transmission in ISM band, complemented with USRP software radios, and was introduced in CREW deliverable D2.4, Section 3.1.

The core of the LOG-a-TEC testbed consists of 50 VESNA sensor nodes mounted on public lighting infrastructure in the city of Logatec in two distinct clusters and 10 VESNA sensor nodes deployed in the JSI campus indoor testbed. Sensor nodes are connected to the gateway sensor node via a ZigBee network, and then connected via Internet to the server hosted at JSI. The hardware setup of VESNA-ased testbeds is depicted in Figure 13 and is based on a mix of nodes capable of spectrum sensing in UHF, ISM 868 MHz and ISM 2.4 GHz bands, each equipped with a 2GB microSD card for storing predefined measurement configurations as well as measurement results. Each node is directly accessibile via web server with custom java application.

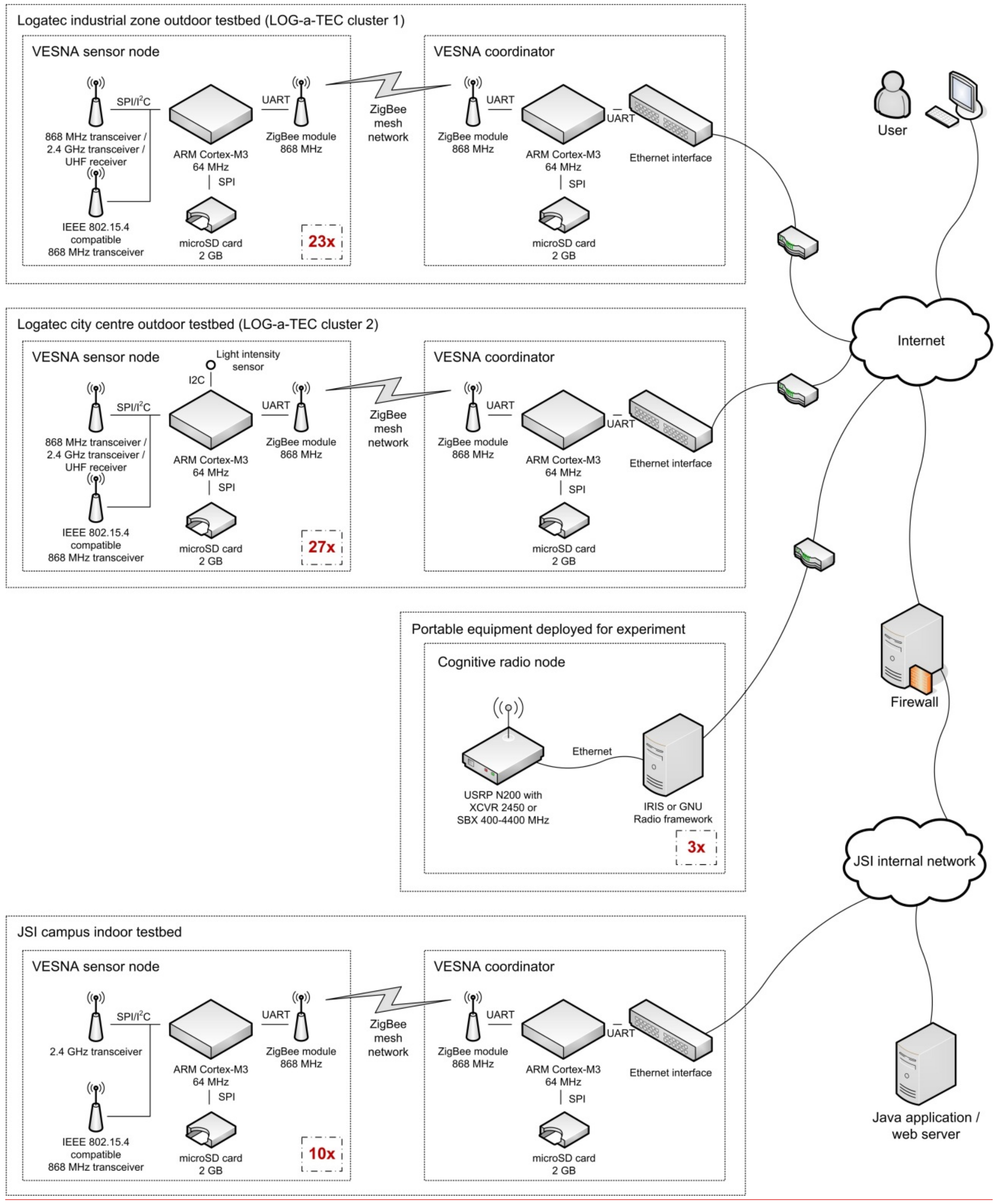


Figure 13 : Overview of hardware available to experimenters in the VESNA-based testbeds

An overview of the functionalities of VESNA-based testbeds is depicted in Figure 14. The baseline functionality and its integration with the CREW federated platform were described in CREW deliverable D3.3, which amends the deliverable D3.1, while in the following we describe the additional functionalities supporting planning, simulation and scheduling of experiments.

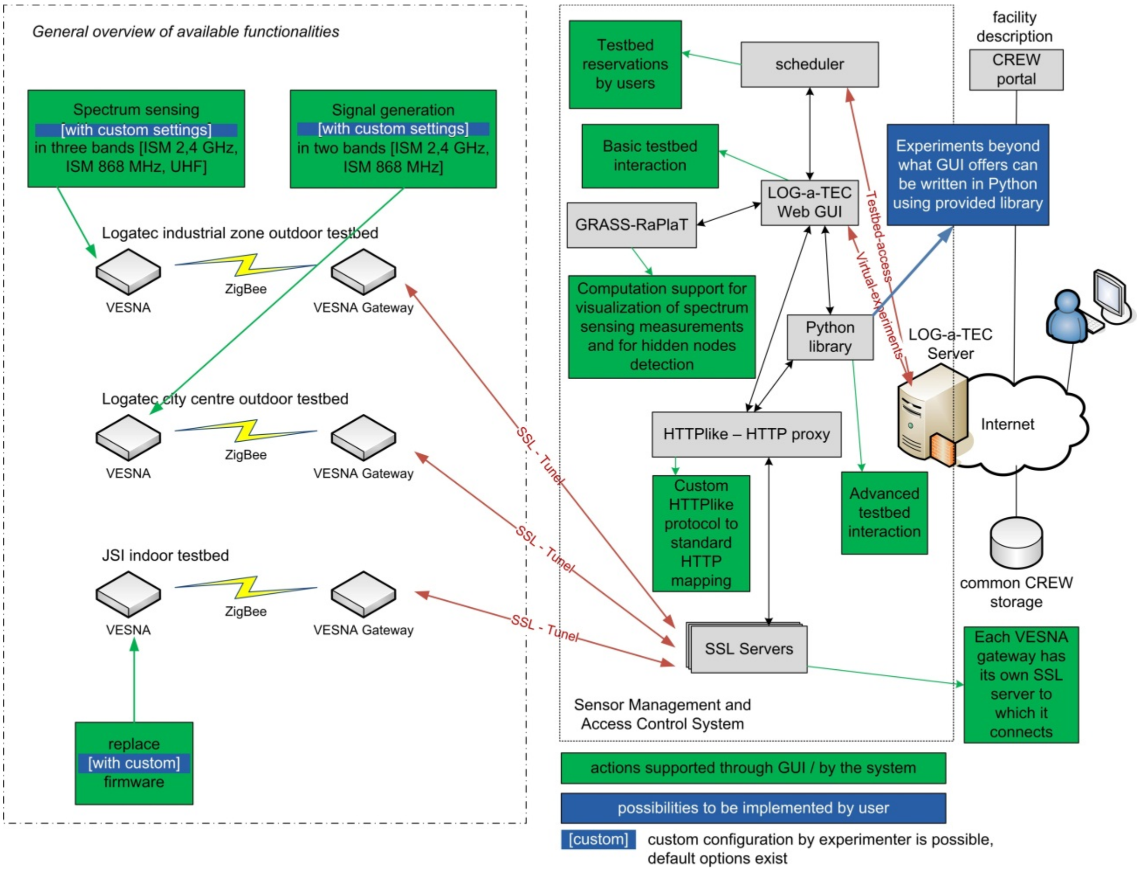

General overview of available functionalities
Spectrum sensing
[with custom settings]
in three bands [ISM 2,4 GHz, ISM 868 MHz, UHF]
Signal generation
[with custom settings]
in two bands [ISM 2,4 GHz, ISM 868 MHz]
Logatec industrial zone outdoor testbed
VESNA
ZigBee
VESNA Gateway
Logatec city centre outdoor testbed
VESNA
ZigBee
VESNA Gateway
JSI indoor testbed
VESNA
ZigBee
VESNA Gateway
replace
[with custom]
firmware
SSL - Tunel
SSL - Tunel
SSL - Tunel
Testbed reservations by users
scheduler
Basic testbed interaction
GRASS-RaPlaT
LOG-a-TEC Web GUI
Testbed-access
Virtual-experiments
Computation support for visualization of spectrum sensing measurements and for hidden nodes detection
Python library
HTTPlike – HTTP proxy
Custom HTTPlike protocol to standard HTTP mapping
Advanced testbed interaction
SSL Servers
Sensor Management and Access Control System
facility description
CREW portal
Experiments beyond what GUI offers can be written in Python using provided library
LOG-a-TEC Server
Internet
common CREW storage
Each VESNA gateway has its own SSL server to which it connects
actions supported through GUI / by the system
possibilities to be implemented by user
[custom] custom configuration by experimenter is possible, default options exist

Figure 14 : Overview of the VESNA-based testbeds functionality

As depicted in Figure 14 the default central point of access to the VESNA-based testbed is a LOG-a-TEC portal hosting a web based Graphical User Interface (GUI), and allowing access to (i) the scheduler responsible for reserving testbed resources for consecutive execution of experiments, (ii) the GRASS-RaPlaT for virtual experiment planning and simulation, and (iii) the sensor network via an HTTP-like protocol and SSL servers using `GET` and `POST` requests. For advanced users the experiment can also be configured and controlled directly, for which, a library of Python scripts are provided.

2.7.1 GRASS RaPlaT virtual experiment planning

The role of the GRASS-RaPlaT in LOG-a-TEC testbed is (i) to provide the virtual experiment planning via simulation in order to ascertain the best setup before the actual execution in the testbed as well as (ii) to support the postprocessing and visualization of experimentation results. No direct access to GRASS-RaPlaT or the data storage/database is implemented or planned. An experiment is configured via a web interface and the results of the experiment are also accessible via the web interface. GRASS-RaPlaT is installed on the same computer as a web server, which simplifies their interworking. A web server requests the required computation (e.g. radio coverage computation) by simply issuing a Linux command with corresponding parameters (an example of coverage computation is given below). This command, which is actually a script (Bash or Phyton), establishes the necessary GRASS environment and executes callbacks of GRASS-RaPlaT or other commands. The GRASS-RaPlaT allows to configure the nodes of an experiment either as a transmitters or receivers. If a particular node is not configured as transmitter or receiver, it is assumed as non-active for the experiment. The main variable parameters for the transmitter are transmission power and frequency, though the tool also allows to vary other parameters such as antenna tilt, azimuth or antenna radiation pattern.

If a number of web users want to use GRASS service at the same time, the web server manages a waiting queue and issues individual GRASS requests in sequence.

The described single-server approach in no way limits the possibility of moving GRASS-RaPlaT to a separate machine in the future if so required. In such a case, suitable inter-machine communication mechanisms should be established for remote command execution and data (file) access.

In the following a set of virtual experiments supported by GRASS-RaPlaT are briefly defined.

2.7.1.1 Transmission radio coverage estimation

By this virtual experiment, with the flow diagram depicted in Figure 15, the radio coverage of the specified node is calculated.

User input parameters are:

- The set of nodes involved in the experiment,
- The transmit power in dBm,
- The carrier frequency in MHz,
- The receiver sensitivity or the interference threshold level in dBm.

The virtual experiment output is:

- Map of signal level in dBm.

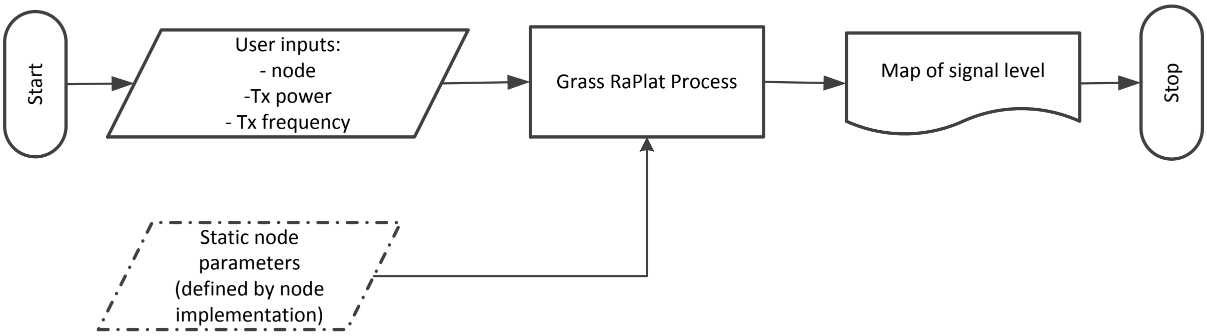


Figure 15 : Flow diagram for radio coverage estimation

2.7.1.2 Transmission range (interference area) of the nodes

By this virtual experiment, the interference area of the selected node is calculated, as graphically shown in Figure 16.

User input parameters are:

- The set of nodes involved in the experiment,
- The transmit power in dBm,
- The carrier frequency in MHz,

- The receiver sensitivity or the interference threshold level in dBm.

The virtual experiment output is:

- The map with transmission range (interference area) of nodes.

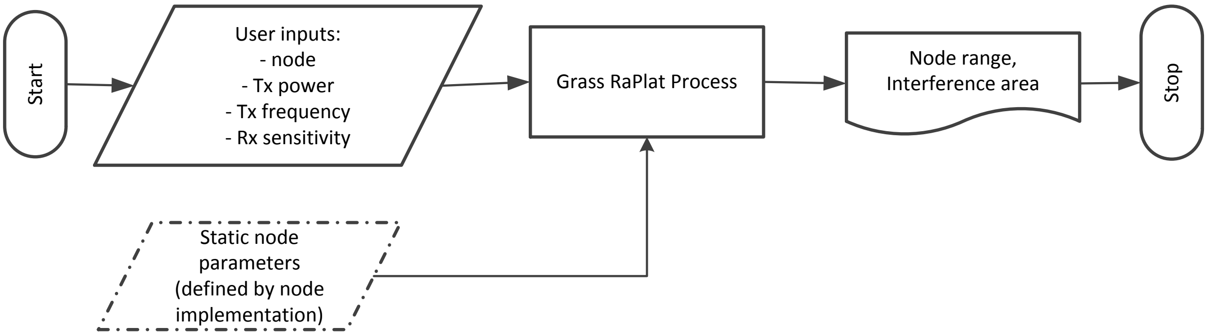


Figure 16 : Flow diagram for transmission range (interference area) of the nodes

2.7.1.3 Estimation of the signal level at the specified receiver locations for all active transmitters

The aim of this virtual experiment with flow diagram given in Figure 17 is to find signal levels of all active transmitters at the point of nodes specified as receivers and find the best server, i.e. transmitter with the highest signal level.

User input parameters are:

- The set of involved in the experiment and their transmit powers in dBm.
- The set of receiver locations,
- The carrier frequency in MHz.

The virtual experiment output is:

- The signal levels of all active transmitters in specified receiver locations.

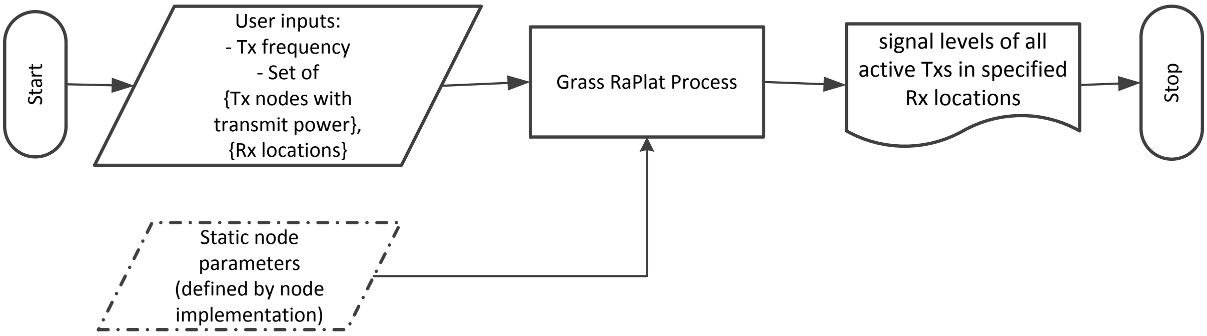


Figure 17 : Flow diagram for estimation of signal level at receiver positions for all active transmitters

2.7.1.4 Estimation of the coverage area of all active transmitters

The aim of this virtual experiment is to show the coverage area of the testbed. The flow diagram of the experiment is shown in Figure 18.

User input parameters are:

- The set of nodes involved in the experiment and their transmit powers in dBm,
- The carrier frequency in MHz.

The virtual experiment output is:

- The coverage area of testbed.

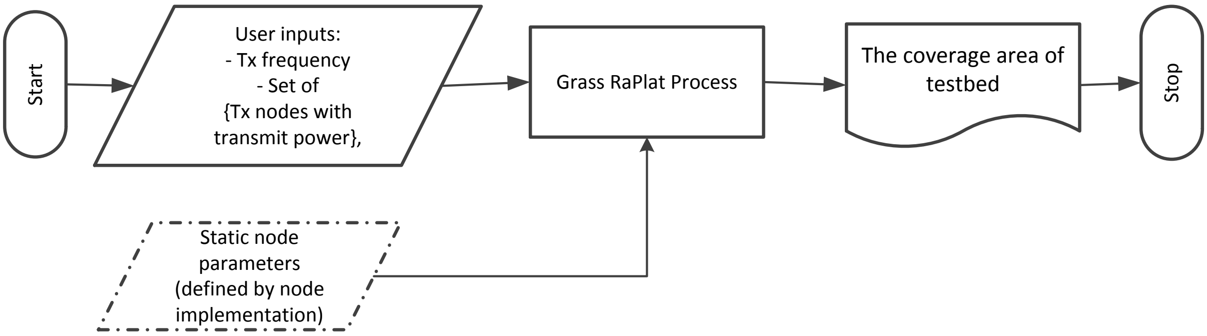


Figure 18 : Flow diagram for estimation of the coverage area of all active transmitters

2.7.1.5 Hidden node detection virtual experiment

The aim of this virtual experiment is to determine the location of the hidden node, and to calculate the hidden node interference area. In this respect the user has to specify the received signal level at the selected receiver node. The experimenter can do this by selecting a random value for received signal level, however the proposed procedure is to determine these values based on the experiment "Transmission radio coverage estimation", assuming one of the transmitter nodes as hidden node. Based on these measurements GRASS-RaPlaT calculates hidden node position and transmitter power, which is the input for calculation of the interference areas, as outlined in **Figure 19**.

Input parameters required are:

- The set of nodes involved in the experiment.
- The carrier frequency in MHz.
- Receiver sensitivity in dBm.

Intermediate result:

- Position of the hidden node.
- Transmit power of hidden node.

Final result:

- Map of interference regions for hidden node.

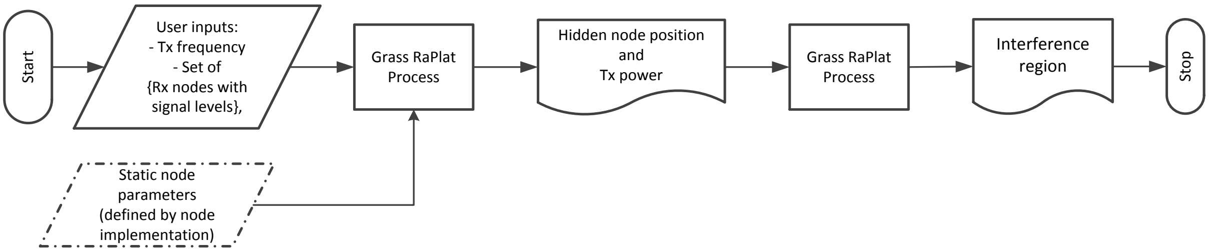


Figure 19 : Flow diagram for hidden node detection virtual experiment

2.7.1.6 Hidden node detection experiment in the testbed

This experiment aims at determining the location of the hidden node based on measurements obtained from the testbed and calculating the hidden node interference area. In this respect the user has to configure the LOG-a-TEC testbed and specify a set of nodes as receivers and one node as a hidden node, i.e. transmitter and its power. The receiver nodes measure the signal level and send these data via the web interface to the GRASS-RaPlaT, which calculates the hidden node position and its transmit power. Based on these results the interference region is calculated and displayed on the web based GUI. The flow diagram of this experiment is shown in **Figure 20**.

Inputs required by this experiment include:

- Configuration of the LOG-a-TEC testbed with
  - a set of nodes acting as receivers and
  - one node acting as transmitter with transmit power and carrier frequency.
- Raw measurement results from the LOG-a-TEC testbed in terms of
  - estimated levels of received signal in dBm and
  - the carrier frequency in MHz.

Intermediate result:

- Position of hidden node.
- Transmit power of hidden node.

Final result:

- Map of interference regions for hidden node.

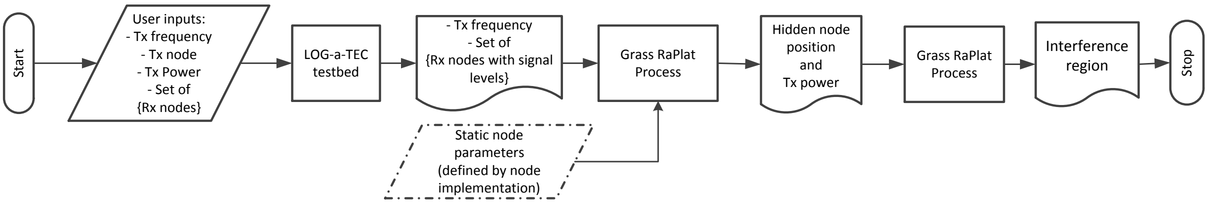


Figure 20 : Flow diagram for hidden node detection experiment in the testbed

2.7.2 WEB server – GRASS RaPlaT communication for coverage computation

The command and data exchange between the web server and GRASS-RaPlaT is illustrated on an example command for transmission radio coverage estimation, `wsn.coverage`.

The WEB server requests coverage computation by issuing the following Linux command:

```
wsn.coverage --input=<input_file> --output=<output_file> --resolution=<resolution_in_m>
```

or in short form:

```
wsn.coverage -i <input_file> -o <output_file>  -r <resolution_in_m>
```

The command returns 0 upon successful completion, or an error code otherwise. It may also print some descriptive information on `stdout/stderr`, but the web servers can safely ignore it.

The *input* and *output* parameters contain the paths to the input and output files. Each path can be either an absolute one, or relative to the working directory. Both files are text files and are described below.

The *resolution* parameter defines the computation resolution independently of the resolution of the digital elevation map (DEM) and clutter maps (which can be interpolated by GRASS if necessary).

2.7.2.1 Input file format

The input file (specified by the *input* parameter) consists of a number of text lines. The first line is the header line containing a semicolon-separated list of names of data contained in the subsequent lines. Each following line contains the corresponding data values for one sensor node. Values can be either floating point numbers or character strings. Spaces are ignored. Each line contains the following data about a node (the sequence of data is fixed):

- `n`, `e` - geographic coordinates, WSG 84 Latitude/Longitude coordinate system, north and east in degrees,
- `h` - antenna height above the ground in meters,
- `tx_frequency` - transmission frequency in MHz (all nodes included in a coverage computation must use the same transmission frequency, otherwise `wsn.coverage` returns an error),
- `tx_power` - transmission power in dBm,
- `antenna_gain` - antenna gain in dB,
- `antena_type` - antena type, which is used to define the antenna radiation pattern (antenna radiation patterns are described in corresponding .MSI files),
- `antenna_direction` - horizontal antenna direction in degrees (northwards is 0 degrees, clockwise is positive),
- `antenna_tilt` - vertical antenna tilt in degrees (downwards is positive).

The last three data are optional and facilitate the use of directed antennas. Even an “undirected” vertical dipole antenna is actually directed in the vertical plane, but this can often be neglected. A combination of an undirected antenna and a lighting pole on which it is mounted can result in a horizontally directed pattern, which can be modeled as a directed antenna.

The header line of the input file is used to check/identify the data format, and looks as follows (spaces are redundant and used here for better readability):

```
n ; e ; h ; tx_frequency ; tx_power ; antenna_gain ; antena_type ; antenna_direction ; antenna_tilt
```

A short form can also be used, if antenna directivity is not important:

```
n ; e ; h ; tx_frequency ; tx_power ; antenna_gain
```

An example of a data line for a sensor node is as follows:

```
45.91175 ; 14.206009 ; 10.0 ; 868.0 ; 15.0 ; 1.0 ; undirected ; 0.0 ; 0.0
```

Here, the long form is used for an undirected antenna along with a special antenna type “undirected”. The node is located in Logatec, Slovenia at the height of 10 m, its radio frequency is 868 MHz, transmission power 15 dBm (32 mW), and antenna gain 1 dB.

2.7.2.2 Output file format

The output file (specified by the *output* parameter) consists of a number of text lines. The first line is the header line containing a semicolon-separated list of names of data contained in the subsequent lines. Each following line contains the corresponding data values for one raster point of the calculated coverage map. These data are the raster point’s geographic coordinates, followed by a list of radio signal strengths received from each of the nodes (the sequence of the nodes is the same as in the previously described input file, which defines them). The number of raster points is defined with the size of the DEM and clutter maps (fixed within `wsn.coverage`) and the selected computation resolution (defined by the `ws.coverage` parameter `resolution`).

The header line of the input file is used to check/identify the data format and looks as follows (spaces are redundant and used here for better readability):

- `n`, `e` - geographic coordinates, WSG 84 Latitude/Longitude coordinate system, north and east in degrees,
- `node1` - received power from node 1 at this geographic point in dBm,
  ...
- `nodeN` - received power from node N at this geographic point in dBm.

In case of 5 nodes, the header line would be (spaces are included for better readability):

```
  n ; e ; node1 ; node2 ; node3 ; node4 ; node5
```

A data line for a point about 300 m SW form the node in Logatec as defined above would be (the received power values displayed here are fictional):

```
45.910033 ; 14.203348 ; -54.3 ; -43.2 ; -32.1 ; -45.6 ; -78.9
```

2.7.3 Scheduler for LOG-a-TEC Testbed

In order to allow the access to the LOG-a-TEC testbed and to sequentially execute planned experiments we decided to develop a scheduler which should support:

- User registration,
- Time slot reservation,
- LOG-a-TEC access control,
- Administration interface,
- User access control,
- Overall control,

2.7.3.1 Implementation of the scheduler

Technologies used for LOG-a-TEC scheduler implementation include MySQL, PHP, jQuery & JavaScript. We have started the implementation of the LOG-a-TEC scheduler by designing the MySQL database consisting of three tables (see Figure 21). The main table is the “users” table. Each registered user will have an entry in this table, while tables “session” and “scheduler” support the functionality of the scheduler system.

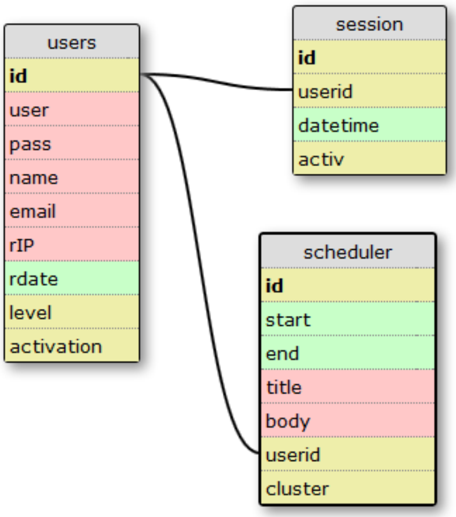


Figure 21 : LOG-a-TEC scheduler database design

The next step was to make connections in the database. For this PHP was used as a backend technology and jQuery and JavaScript as frontend technologies. The frontend is designed to be easy to use and that the user can easily create an account and reserve the testbed in just a few clicks.

Additionally the user interface includes the administrator access which has full control over all the regular users making testbed reservations. The described system architecture is depicted in Figure 22.

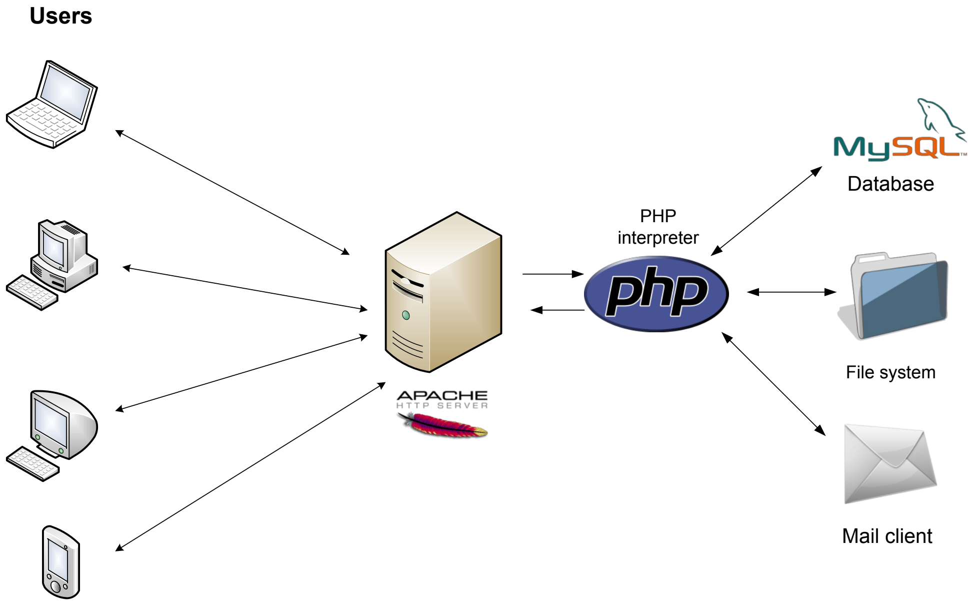
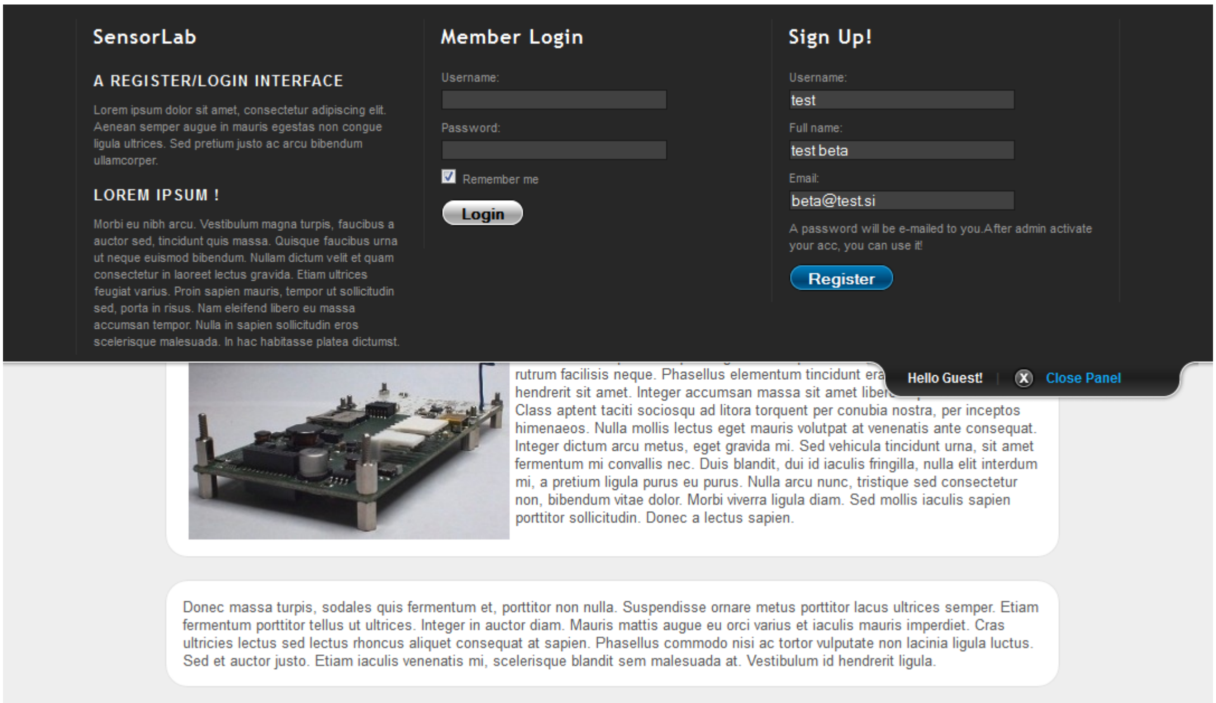


Figure 22 : LOG-a-TEC scheduler system design

2.7.3.2 Usage of the LOG-a-TEC scheduler

Upon entering the system the web page displays the information about the project and some standard information such as contact details, pictures, etc. Furthermore, the user has an option to open a panel in header of the page to make registration or just sign in with an existing account. As shown in the , the same interface is used for users and administrators.



SensorLab

A REGISTER/LOGIN INTERFACE

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean semper augue in mauris egestas non congue ligula ultrices. Sed pretium justo ac arcu bibendum ullamcorper.

LOREM IPSUM !

Morbi eu nibh arcu. Vestibulum magna turpis, faucibus a auctor sed, tincidunt quis massa. Quisque faucibus urna ut neque euismod bibendum. Nullam dictum velit et quam consectetur in laoreet lectus gravida. Etiam ultrices feugiat varius. Proin sapien mauris, tempor ut sollicitudin sed, porta in risus. Nam eleifend libero eu massa accumsan tempor. Nulla in sapien sollicitudin eros scelerisque malesuada. In hac habitasse platea dictumst.

Member Login

Username:

Password:

☑ Remember me

Login

Sign Up!

Username: test

Full name: test beta

Email: beta@test.si

A password will be e-mailed to you.After admin activate your acc, you can use it!

Register

Hello Guest! | ⓧ Close Panel

rutrum facilisis neque. Phasellus elementum tincidunt era hendrerit sit amet. Integer accumsan massa sit amet liber Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla mollis lectus eget mauris volutpat at venenatis ante consequat. Integer dictum arcu metus, eget gravida mi. Sed vehicula tincidunt urna, sit amet fermentum mi convallis nec. Duis blandit, dui id iaculis fringilla, nulla elit interdum mi, a pretium ligula purus eu purus. Nulla arcu nunc, tristique sed consectetur non, bibendum vitae dolor. Morbi viverra ligula diam. Sed mollis iaculis sapien porttitor sollicitudin. Donec a lectus sapien.

Donec massa turpis, sodales quis fermentum et, porttitor non nulla. Suspendisse ornare metus porttitor lacus ultrices semper. Etiam fermentum porttitor tellus ut ultrices. Integer in auctor diam. Mauris mattis augue eu orci varius et iaculis mauris imperdiet. Cras ultricies lectus sed lectus rhoncus aliquet consequat at sapien. Phasellus commodo nisi ac tortor vulputate non lacinia ligula luctus. Sed et auctor justo. Etiam iaculis venenatis mi, scelerisque blandit sem malesuada at. Vestibulum id hendrerit ligula.

Figure 23 : Header panel

2.7.3.2.1 User interface

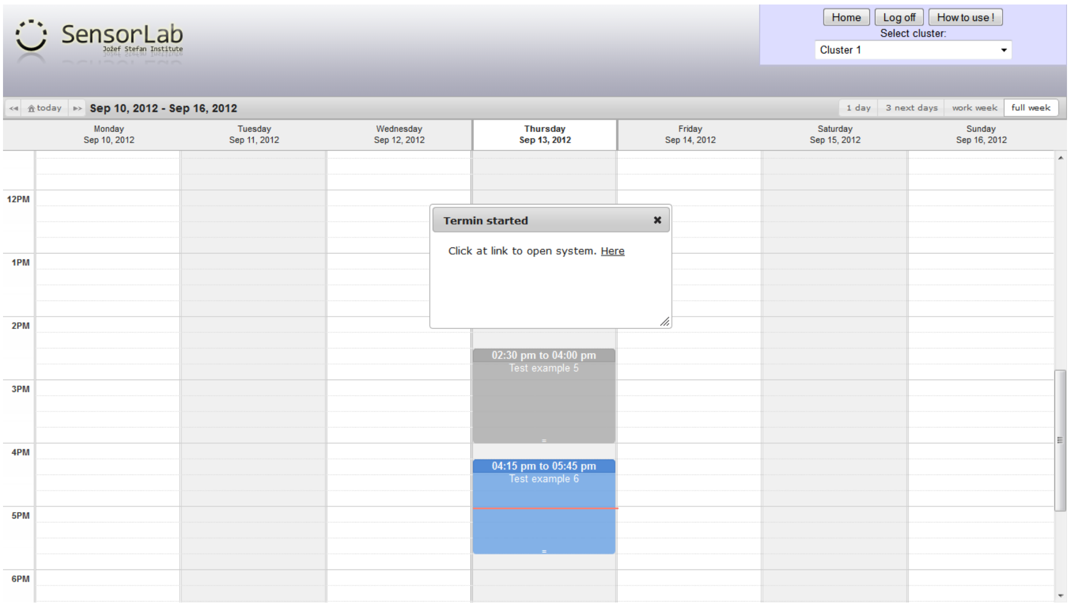
When accessing the portal the user is served with a form to create an account or log in with an existing account. Upon the creation of an account the user receives an email with a randomly generated password. After admin approves a new account the user gets access to LOG-a-TEC scheduling system, depicted in Figure 24 .

After a successful login the user is able to change the automatically generated password. When logged in, he/she is able to access the calendar and make a reservation of the testbed facilities based on hourly timeslots. The reservation process can also define a timeslot consisting of several hours. Since each of the three WSN clusters has its own calendar it is possible to reserve one cluster, two clusters or all three in the same time slot or in separate time slots. An existing reservation can also be at any subsequent occasion edited or removed. Furthermore, the user will be able to see reservations of other users but will not be able to change them.

The user interface is made using jQuery library, so the page does not need to be refreshed to get new information, i.e. while looking the calendar it will be possible to see how new reservations are created.

Figure 24 depicts the calendar interface. There are two events visible, one gray and one blue. Both events were made by test user. The gray event occurred in the past and is thus not editable. It will go in the history of passed reservations. The blue timeslot is editable, since this event has not started yet. The current time is depicted on the calendar using a red line, and when the line reaches a reservation it activates it.

Figure 24 depicts the event when the reservation becomes active. The event is not editable anymore additionally a popup allowing the LOG-a-TEC access appears.



Select cluster: Cluster 1

Sep 10, 2012 - Sep 16, 2012

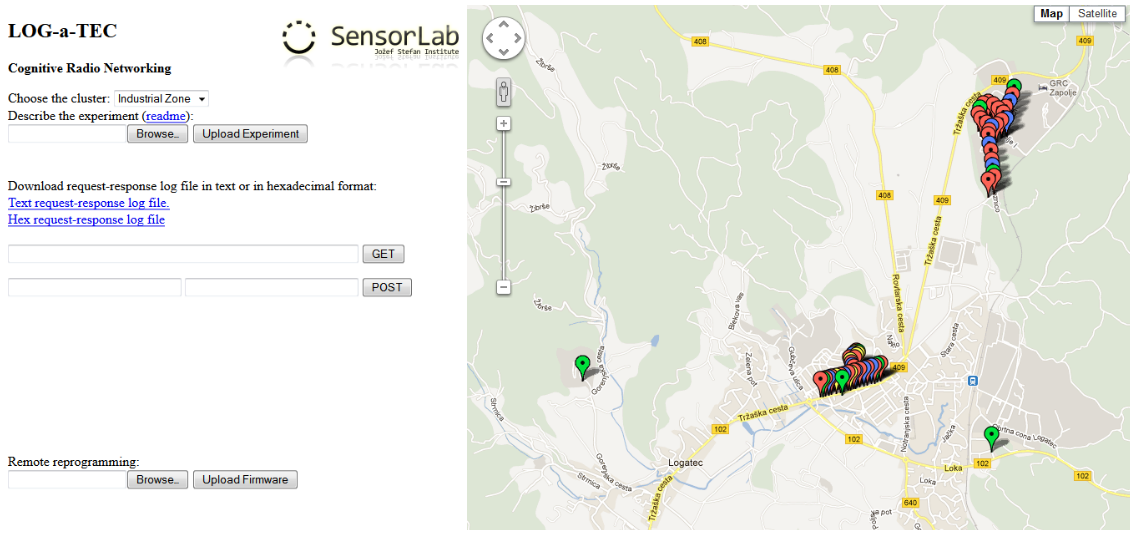
Monday Sep 10, 2012	Tuesday Sep 11, 2012	Wednesday Sep 12, 2012	Thursday Sep 13, 2012	Friday Sep 14, 2012	Saturday Sep 15, 2012	Sunday Sep 16, 2012
			02:30 pm to 04:00 pm Test example 5			
			04:15 pm to 05:45 pm Test example 6			

Termin started

Click at link to open system. Here

Figure 24 : Calendar interface: example of activated timeslot

When the session time for a given user is up he/she will automatically be disconnected from the portal (see Figure 25) and will have to make a new reservation in order to gain access again.



LOG-a-TEC

Cognitive Radio Networking

Choose the cluster: Industrial Zone

Describe the experiment (readme):

Download request-response log file in text or in hexadecimal format:
Text request-response log file.
Hex request-response log file

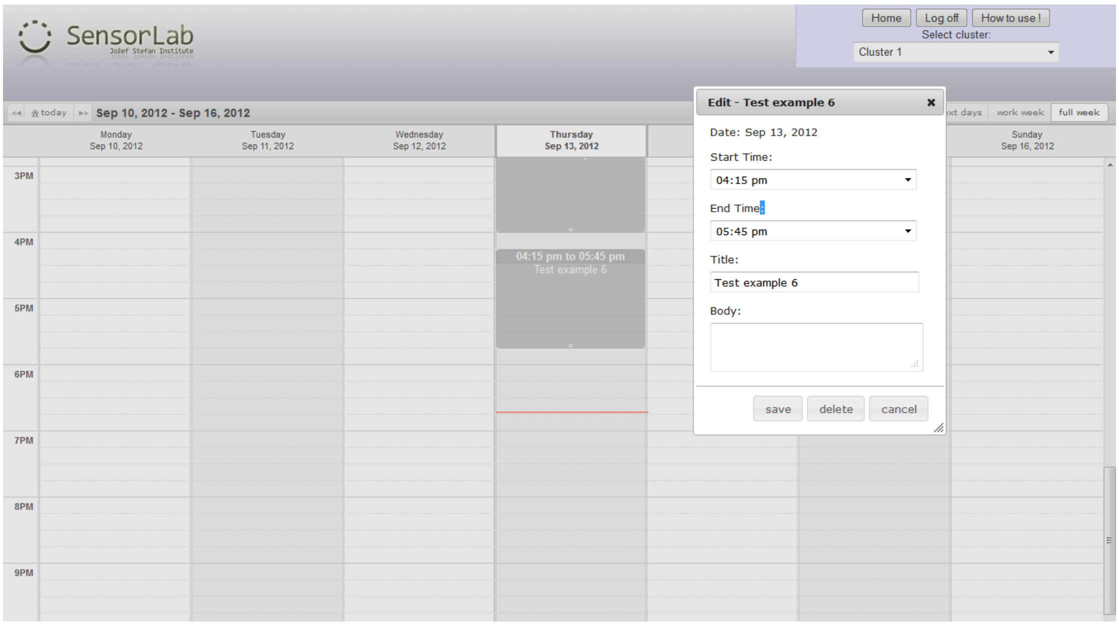
Remote reprogramming:

Figure 25: LOG-a-TEC interface

2.7.3.2.2 Administrator interface

The administrator account with user interface depicted in Figure 26 allows for:

- Approving registrations
- Changing user access level admin/normal
- Making reservations
- Editing every reservation
- Monitoring the current use of the system
- Controlling activated timeslots



Select cluster: Cluster 1

Edit - Test example 6

Date: Sep 13, 2012

Start Time: 04:15 pm

End Time: 05:45 pm

Title: Test example 6

Body:

Figure 26 : Example of altering timeslot of another user

2.7.3.3 LOG-a-TEC portal integration

The integration of LOG-a-TEC portal and scheduler was done by generating user specific hashes, which are stored in the database for the duration of the user session. After the session the hash in the database is destroyed and the user does not have permission to access the portal anymore. Additionally, the administrator has the power to destroy the hash and to cancel a users' session if needed.

3 Common Data Collection/Storage Methodology Design

CREW Deliverable 3.1 introduced the common data collection and storage methodology design. In this deliverable we describe the extensions realized in CREW Year 2. This includes a public server for hosting experimental data, a mechanism for automatically validating the formal correctness of the user's experiment specification, extensions to the format and conversion to XML, as well as MATLAB specification for traces and a set of public scripts to process / convert the traces.

3.1 Public server concept for publishing experiment data

In order to enable better use of the common data and experiment description format we have started the CREW repository [1] with several types of data, which are useful for sharing with other experimenters. We have divided the repository to five sections:

- **full experiment descriptions:** when reading reports containing experimental results, it is often difficult or impossible to verify the claims of the author or to use the experiment as a base for further research. Full experiment descriptions contain all information needed to run a particular experiment. By publishing full experiment descriptions, we aim for more transparent and thus more valuable experimentation results. Furthermore, the possibility of reusing experiment descriptions enables fair comparison (benchmarking) of wireless solutions.
- **traces:** traces are files describing wireless traffic of heterogeneous technologies, either at packet level (e.g. a timed sequence of Bluetooth packets, a certain profile of Wi-Fi use, the activity of a primary LTE user), or data recorded at spectrum level (e.g. recorded spectrum use in a certain location over time, artificially created patterns containing reference interference on a particular frequency).
- **background environments:** background environments are configurations that can be used to create repeatable wireless background environments and are tied to a particular testbed. The configuration files of these background environments link particular traces (see above) to particular nodes at particular times, inside a particular testbed.
- **processing scripts:** useful scripts (in multiple scripting languages, matlab, java,...) that, for example, convert the output of a certain type of commercially available tool or CREW component to the CREW common data format.
- **performance metrics and benchmarking scores**: detailed descriptions of performance metrics and measurement methodologies. Benchmarking scores are abstractions derived by combining different performance metrics. These scores may be useful when comparing a large set of solutions or for automated decision making during an automated benchmarking process.

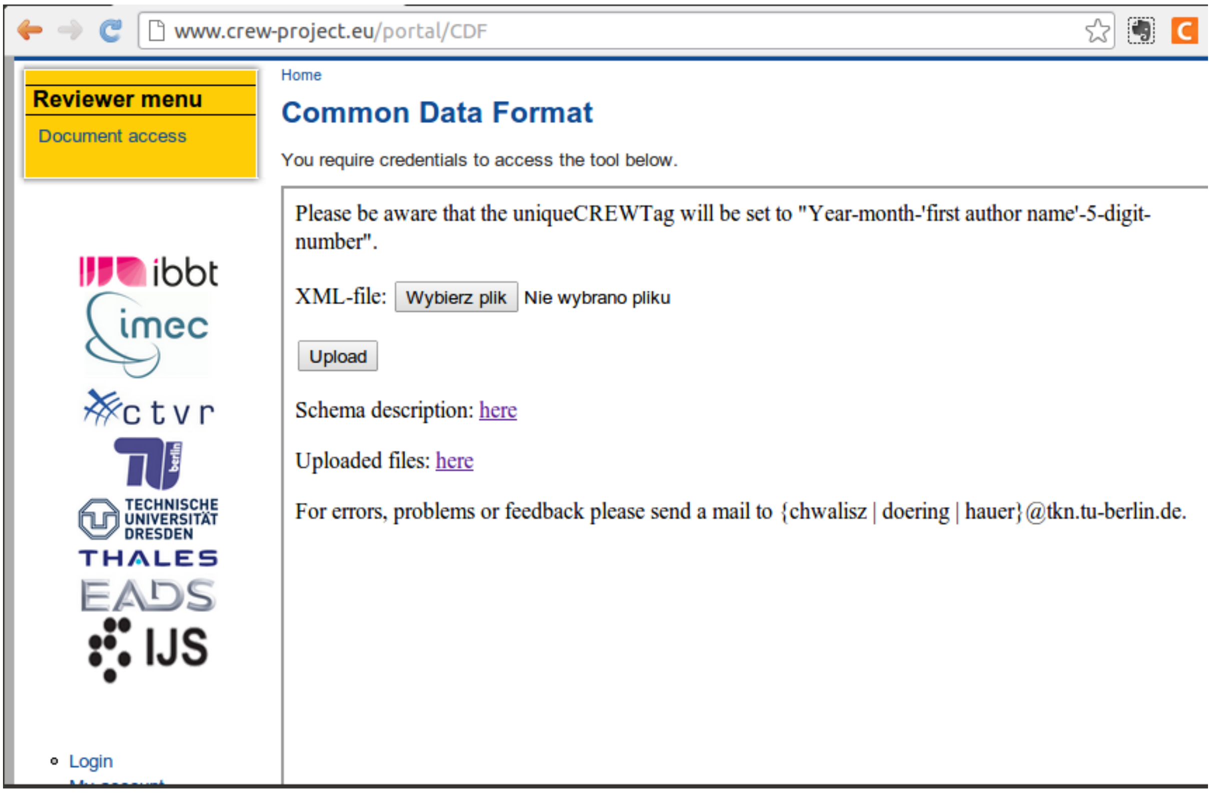
3.2 Experiment Description Validation

Full experiment descriptions are a very important part of the common data storage methodology. In Deliverable 3.1 the way in which CREW experiments are described (specified) was defined in detail. An example was given how the experiment specification can be encoded in JSON (JavaScript Object Notation), which is a text-based open standard designed for human-readable data interchange similar to XML. JSON was selected over XML, mainly because it is better human-readable, but there are no conceptual differences between the two formats. Also, libraries and conversion tools exist that can convert from one version to the other.

During Year 2 we identified that XML has an advantage over JSON, namely XML is used within the OMF framework, which is already employed in several testbeds. Therefore typically these testbeds already have the necessary software libraries installed for working with XML encoded documents. To minimize software dependencies, in Year 2 we therefore selected XML as the primary choice for encoding experiment specification. Since JSON can be converted to XML the specification (metadata) defined in Deliverable 3.1 remains unchanged.

An experimenter using the CREW experiment specification to document his/her experiment may (by accident) introduce syntactic or semantic errors into the document. In order to automatically check a CREW experiment specification for syntactic (and some semantic) errors, we are using a syntax parser and have defined the XML schema for our experiment specification as shown in 3.2.1. An XML schema defines the logical structure of all fields in a XML and there exist tools that validate the correctness an XML document based on a given schema. In addition we also added a mechanism to create a unique CREW tag, which is used to automatically uniquely define an individual experiment specification (e.g. for reference in a paper).

We have implemented a small software framework that integrates the above techniques into a validation tool for CREW experiment specifications. The service is provided through a web-interface, i.e. an experimenter my use the service (and upload the validated experiment description) remotely over the Internet. An excerpt of the web-interface can be seen in Figure 27.



Common Data Format

You require credentials to access the tool below.

Please be aware that the uniqueCREWTag will be set to "Year-month-'first author name'-5-digit-number".

XML-file: Wybierz plik Nie wybrano pliku

Schema description: here

Uploaded files: here

For errors, problems or feedback please send a mail to {chwalisz | doering | hauer}@tkn.tu-berlin.de.

Figure 27: XML validator form

The validation tool for CREW experiment specifications implements the following three step approach:

1. Parsing the XML file for syntax problems and throw related errors to help the file creator
2. Compare the XML file with the defined schema. All fields of the XML file are manifested in the common data format schema file. If the comparison process will find a undefined field or e.g., spelling errors the user will be informed and the file rejected

3. The unique CREW tag is modified for database storage. The tag is defined by "Year-month-'first author name'-5-digit-number". All information is gathered from the XML file, as well as from the storage.

All uploaded files are stored in the same folder and available to the public. The XML parser is available online under http://www.crew-project.eu/portal/CDF. Further we provide at the upload form (Figure 27) a schema description with documentation to explain the meaning of the subfields.

3.2.1 XML Instance Representation

```
<experimentDescription>
      <experimentAbstract> [1] ?
            <title> xs:string </title> [1]
            <uniqueCREWTag> xs:string </uniqueCREWTag> [1]
            <author> [1..*]
                  <name> xs:string </name> [1]
                  <email> xs:string </email> [1]
                  <address> xs:string </address> [1]
                  <phone> xs:string </phone> [1]
            </author>
            <releaseDate> xs:date </releaseDate> [1]
            <experimentSummary> xs:string </experimentSummary> [1] ?
            <collectionMethodology> xs:string </collectionMethodology> [1]
            <furtherDocumentation> [1]
                  <description> xs:string </description> [1]
                  <bibtex> xs:string </bibtex> [1]
            </furtherDocumentation>
            <relatedExperiments> xs:string </relatedExperiments> [1] ?
            <notes> xs:string </notes> [1]
      </experimentAbstract>
      <metaInformation> [1] ?
            <device> [0..*]
                  <name> xs:string </name> [1]
                  <description> xs:string </description> [1]
                  <datasheet> xs:string </datasheet> [0..*]
                  <software> [1]
                        <description> xs:string </description> [1]
                        <operatingSystem> xs:string </operatingSystem> [1]
                        <driver> xs:string </driver> [1]
                        <applicationName> xs:string </applicationName>
                  [0..*]
                        <sourcecode> xs:string </sourcecode> [1]
                  </software>
            </device>
            <location> [1]
                  <layout> xs:string </layout> [1]
                  <mobility> xs:string </mobility> [1]
            </location>
            <time> xs:string </time> [1]
            <radioFrequency> [1]
                  <operatingRange> xs:string </operatingRange> [1]
                  <interferenceSources> xs:string </interferenceSources>
            [1]
            </radioFrequency>
            <traceDescription> [1]
                  <collectedMetrics> [1]
```

```
                        <name> xs:string </name> [1]
                        <unitOfMeasurements> xs:string
                  </unitOfMeasurements> [1]
                  <accuracy> xs:string </accuracy> [1]
            </collectedMetrics>
            <format> xs:string </format> [1]
            <processingTools> xs:string </processingTools> [1]
            <signalGeneration> [1]
                  <description> xs:string </description> [1]
                  <trace> xs:string </trace> [1]
            </signalGeneration>
      </traceDescription>
</metaInformation>
<experimentIteration> [0..*] ?
      <description> xs:string </description> [1]
      <time> [1]
            <starttime> xs:dateTime </starttime> [1]
            <endtime> xs:dateTime </endtime> [1]
      </time>
      <parameters> [1]
            <name> xs:string </name> [1]
            <value> xs:byte </value> [1]
      </parameters>
      <traceFile> xs:string </traceFile> [0..*]
</experimentIteration>
</experimentDescription>
```

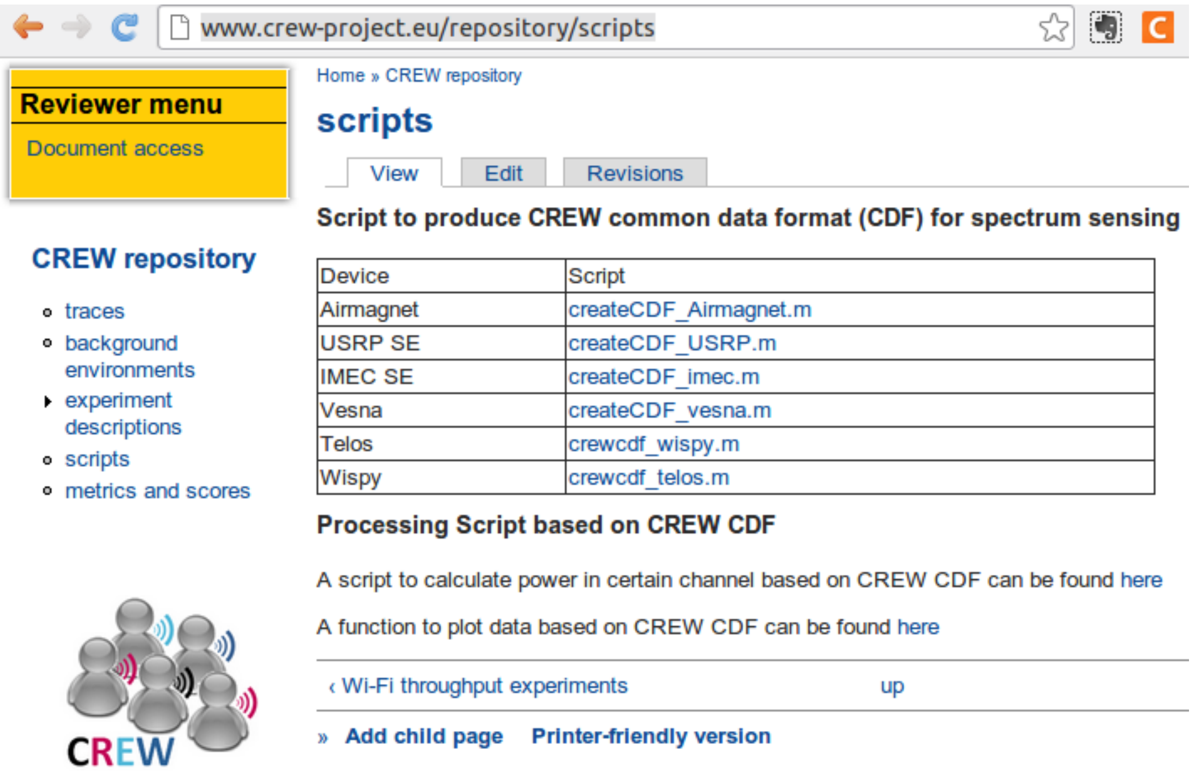
3.3 CREW trace format

During the experiments performed in the CREW project we have identified that it is very hard to work with heterogonous hardware devices. Almost all devices produce output in the different format. This is even the case in the simple energy detection based spectrum sensing, e.g. different devices have different resolution bandwidths. In Year 1 we defined that traces (traces are the actual measurement results/samples, whereas experiment specification defines the modalities of the experiments) may be stored in different formats as long as it is documented sufficiently well how to extract the data.

In Year 2 we have developed the common structure for such traces in MATLAB, because MATLAB seems to be a preferred choice for storing experiment outcome. It has the following fields:

```
p = common data format structure
p.Name         = Unique identifier of the sensing device
p.Location     = Location of the sensing device (m) e.g [x,y,z]
p.CenterFreq   = Array defining center frequencies
                   of the columns of power the matrix (Hz)
p.BW           = Bandwidth around each center frequency (Hz)
p.Tstart       = Start time of the measurement in datestr format
                   e.g. '24-Jan-2003 11:58:15'
p.SampleTime   = Timestamp relative to Tstart (s)
p.Power        = Matrix containing power measurements (dBm)
                   row contains all frequencies for one timestamp
```

Under http://www.crew-project.eu/repository/scripts (**Figure 28**) we provide the scripts to convert data from most devices that are able to perform measurements in 2.4GHz ISM band in CREW project. We also provide the plotting function that takes advantage of all fields from common data format.



scripts

Script to produce CREW common data format (CDF) for spectrum sensing

Device	Script
Airmagnet	createCDF_Airmagnet.m
USRP SE	createCDF_USRP.m
IMEC SE	createCDF_imec.m
Vesna	createCDF_vesna.m
Telos	crewcdf_wispy.m
Wispy	crewcdf_telos.m

Processing Script based on CREW CDF

A script to calculate power in certain channel based on CREW CDF can be found here

A function to plot data based on CREW CDF can be found here

Figure 28 Matlab scripts

4 Common portal

4.1 Introductory comments on the portal

As already explained in D3.1, the common portal is a public website, containing all information needed by experimenters to be able to use the CREW platform. This includes but is not limited to information on the available hardware, information on how to use the hardware, information on who can apply for an account and how to do this, and tutorials to get people started.

The CREW portal is located at www.crew-project.eu/portal.

As in the other sections of this deliverable, the content below only discusses updates to the common CREW portal, and not the general concepts of the portal.

4.2 Updates to the portal

From an implementation point of view, no changes were made to the portal. The biggest changes to the portal are:

4.2.1 Addition of JSI sections.

Partner JSI joined the CREW project in July 2010, complementing the CREW platform with an outdoor wireless sensor network. The portal was extended so that the JSI testbed is now also described according to the same template as the other CREW testbeds.

4.2.2 Updates and extensions to the content.

Where needed, the portal information was extended or updated. It now also covers the functionality and APIs that were added during the second year of CREW, and explains the old functionalities in a more clear way.

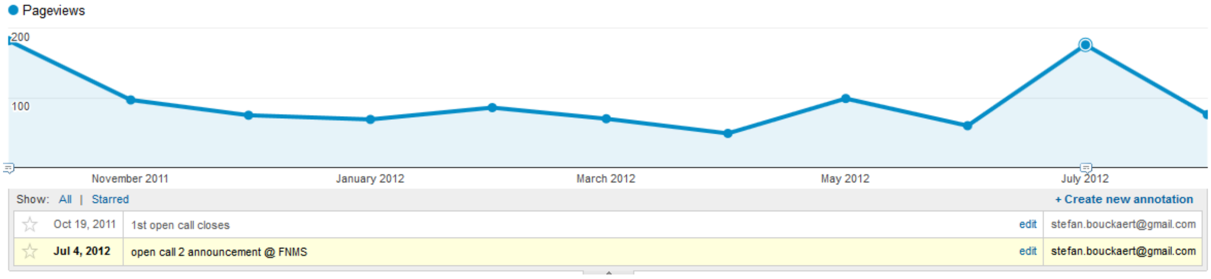
4.2.3 The CREW repository.

Last but not least, as already introduced in Section 3.1, a repository section was added to the CREW portal **[1]**. The CREW repository contains a collection of data of different types that are believed to be of use to experimenters in the field of cognitive radio and cognitive networking.

4.3 Portal Statistics

At the moment of writing this deliverable, it can be seen from the public website visitors monitoring that –so far- in year 2 of CREW (October 2011 – end of August 2012), over 652 unique (1039 in total) page views of the www.crew-portal.eu/portal page were recorded. On average, visitors spend 1 min 8 seconds on this entry page, indicating that most of the people reaching this page are really interested in the portal content and do not arrive there by accident. Moreover, over 85% of these visitors continues browsing the CREW portal (or CREW website) after reaching the CREW portal welcome page.

Not surprisingly, there is particular interest in the CREW portal when information on the open call is disseminated. For example, as can be seen from Figure 29, after the announcement of the second open call in July 2012, the number of visitors roughly doubled.

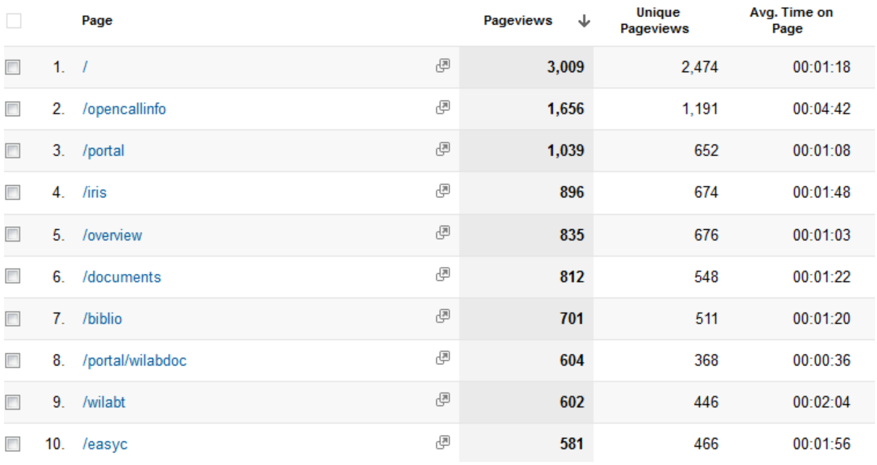


☆	Oct 19, 2011	1st open call closes	edit	stefan.bouckaert@gmail.com
☆	**Jul 4, 2012**	open call 2 announcement @ FNMS	edit	stefan.bouckaert@gmail.com

Figure 29 - Y2 (11 months) page views for the entry page of the CREW portal, totals per month

With 652 unique views during the same 11-months time span, the portal entry page is the third most popular page of the public CREW website; Only the general entry page (www.crew-project.eu – 2474 unique views – average time on page 1 min 18 sec) and the open call information (www.crew-project.eu/opencallinfo - 1039 unique views – average time on page 4 min 42 seconds) are more popular. The interested reader can find the complete top 10 in Figure 30.

As such, these statistics show the continued relevance of the CREW portal.



□		Page	Pageviews ↓	Unique Pageviews	Avg. Time on Page
□	1.	/	**3,009**	2,474	00:01:18
□	2.	/opencallinfo	**1,656**	1,191	00:04:42
□	3.	/portal	**1,039**	652	00:01:08
□	4.	/iris	**896**	674	00:01:48
□	5.	/overview	**835**	676	00:01:03
□	6.	/documents	**812**	548	00:01:22
□	7.	/biblio	**701**	511	00:01:20
□	8.	/portal/wilabdoc	**604**	368	00:00:36
□	9.	/wilabt	**602**	446	00:02:04
□	10.	/easyc	**581**	466	00:01:56

Figure 30 - Y2 (11 months) top 10 most popular

5 Testbeds components and combinations

This section starts by discussing the mix and match combinations performed within the project, to date, as part of the interoperability testing work within the CREW project. A table is presented categorising and accounting for the component combination performed. For each component combination a description or reference (within this or another deliverable, peer reviewed publication or both) to further information is provided.

As many of the mix and match combination are highly reliant on efficient device interfacing, the mix and match discussion is followed by interface information for both the TCS Transceiver API and the VESNA-based testbeds (Ljubljana and Logatec). The Transceiver API description includes set-up details, an architectural overview and code breakdown, while the VESNA-based testbed descriptions outline how the testbeds can be controlled via HTTP-like interface, C interface or via custom firmware image. This forms part of the interface design and specification work of CREW.

5.1 Mix and match

The following table presents a summary of mix and match combinations performed over the first two years of the project between CREW partners and members of the first open call. These have been categorized into hardware (H), software(S) and/or side-by-side(S-B-S) combinations, those marked in in bold signify cross-country combinations whereas those not in bold signify intra-country combinations.

Hardware combinations are those where either a hardware coupling of two components from different testbeds is performed to avail of the benefits provided by each of the platforms individually or else where hardware is taken from one of the testbeds and physically integrated into another. Software combinations are where software developed in one testbed is used with hardware of another testbed, this can be either a full integration full use in the testbed nodes or else performed just for a specific experiment. Finally, side-by-side combinations are where recordings are performed in parallel on multiple devices located side-by-side and then processed together to provide a combined solution.

	Iris	IBBT	Imec	TWIST	LTE-TUD	TCS	EADS	VESNA	Durham	Ilmenau	Tecnalia
Iris	ctvr	**S, S-B-S**	**H, S, S-B-S**	**S-B-S**		**S-B-S**		**S-B-S, planned H**	**Planned**	**H, S**	**S**
IBBT	**S, S-B-S**	ibbt	H, S-B-S				**S-B-S**	**S-B-S**	**Planned**		
Imec	**H, S, S-B-S**	H, S-B-S	imec	**H, S, S-B-S**	**S-B-S**		S-B-S	S-B-S	**S-B-S**	**H, S**	
TWIST	**S-B-S**		**H, S, S-B-S**	berlin			S-B-S	**S-B-S**	**S-B-S**		
LTE-TUD			**S-B-S**		TECHNISCHE UNIVERSITÄT DRESDEN	**S-B-S**					
TCS	**S-B-S**				**S-B-S**	THALES					**S**
EADS		**S-B-S**	**S-B-S**	S-B-S			EADS	**S-B-S**	**S-B-S**		
VESNA	**S-B-S, Planned H**	**S-B-S**	**S-B-S**	**S-B-S**			**S-B-S**	IJS			
Durham	**Planned**	**Planned**	**S-B-S**	**S-B-S**			**S-B-S**		Durham University		
Ilmenau	**H, S**		**H, S**							TECHNISCHE UNIVERSITÄT ILMENAU	
Tecnalia	**S**					**S**					tecnalia

Table 4: Mix and match performed combinations. Categorised as hardware(H), software(S) and/or side-by-side(S-B-S) combinations, those marked in in bold signify cross-country combinations whereas those not in bold signify intra-country combinations.

As can be seen from the table, a large number of different combinations have been performed involving all CREW federation and open call partners, including both intra-country and a large number of cross-country combinations. As no such table was provided in D3.1, below we categorize these mix and match combinations into those performed in the first year and those performed in the second year of the project. Many of the combination experiments performed in year two build on and extend the combinations performed in year one of the project. As the main focus of CREW is to facilitate experimentally-driven research in cognitive radio, cognitive networks and advanced spectrum sensing these combinations serve as examples of combinations which can be easily performed within the CREW federation.

5.1.1 Year 2 mix and match combinations

5.1.1.1 Integration of Iris into IBBT Testbed

Type: Software

Participants: Iris, IBBT

Description: Description given in Section 2.5.

5.1.1.2 Context Awareness in the ISM Band – cafeteria environment, IMEC

Type: Side-by-side

Participants: Iris, IBBT, IMEC, TWIST, VESNA

Description: Description given in D6.2, Section 2.1.3 and [13].

5.1.1.3 2.1.2 Context Awareness in the TVWS – Experiment in the Logatec, JSI

Type: Side-by-side

Participants: IBBT, IMEC, TWIST, VESNA

Description: Measurements were performed from a mobile vehicle in the area of LOG-a-TEC testbed in the city of Logatec, Slovenia, in a similar way to as done in D6.2 Section 2.1.2. Sensing using heterogeneous devices was performed of both DVB-T and wireless microphone signals in order to verify pathloss estimates.

5.1.1.4 Multi-antenna sensing of LTE devices

Type: Side-by-side

Participants: LTE-TUD, TCS

Description: Description given in D6.2, Section 2.5.1.

5.1.1.5 Integration of TCS Transceiver API and Iris into Tecnalia Experiment

Type: Software

Participants: Iris, TCS, Tecnalia

Description: Description given in Tecnalia open call deliverable D7.3.1.

5.1.1.6 Integration of Imec Sensing agent into TWIST Testbed

Type: Software, Hardware

Participants: Imec, TUB

Description: Description given in Section 2.2.

5.1.1.7 Use of nomadic testbed for channel sounding measurements in EADS plane mock-up

Type: Other; combination of software and Side-by-side

Participants: IBBT, IMEC TWIST, EADS, VESNA, Durham

Description: Short description of given in Section 2.5 (nomadic testbed), also discussed in D6.2 Section 2.2.2 (channel sounding and communication system design).

5.1.1.8 Sensing experiments with open call partner Durham

Type: side-by-side

Participants: IBBT, IMEC, TWIST, EADS, Durham

Description: As also reported in D5.2, among other things, channel sounder measurements have been performed in the TWIST testbed and at EADS, and are planned in the IBBT testbed. Detailed results will be reported in the open call deliverables of Open Call partner Durham.

5.1.1.9 Coupling of Iris and the IMEC sensing platform with open call partner Ilmenau

Type: Hardware, software

Participants: Iris, IMEC, Ilmenau

Description: Both hardware and software coupling of the IMEC sensing platform and Iris for use in contention based MAC protocols. Description given in D7.2.1.

5.1.2 Year 1 mix and match combinations

5.1.2.1 Integration of IMEC sensing nodes into IBBT testbed

Type: Hardware

Participants: IBBT, IMEC

Description: Description given in D3.1, Section 5.3.

5.1.2.2 Combination of IMEC sensing platform and Iris transceiver link

Type: Hardware

Participants: Iris, IMEC

Description: Description given in D6.1, Section 2.4.1.

5.1.2.3 Comparison of devices – office environment, TCD

Type: Side-by-side

Participants: Iris, IBBT, IMEC, TWIST

Description: Description given in D6.1, Section 2.1.1 and [14], [15].

5.1.2.4 Comparison of devices – office environment, TUD

Type: Side-by-side

Participants: Iris, IBBT, IMEC, TWIST, LTE-TUD

Description: Description given in D6.1, Section 2.1.1 and [14], [15].

5.1.2.5 Comparison of devices – office environment, TUB

Type: Side-by-side

Participants: Iris, IBBT, IMEC, TWIST, EasyC

Description: Discussed briefly in D6.2 Section 2.1.1. Formed a basis for 5.1.1.2 above.

5.2 Transceiver API

5.2.1 Introduction

The TCS Transceiver API is an interface specification [16]. For industrial standardization requirements and to master architectures, the Transceiver API was specified in the Wireless Innovation Forum (WInnF) by TCS Communications & Security. Thus, while the Transceiver API doesn't provide any additional experimental functionality, it standardizes the interface between the Modem (any WaveForm application) and the Transceiver Subsystem (any transceiver device, as was done as part of the TECNALIA experiment in Task 7.3 of Work package 7 for example, with several Ettus Research USRP2 boards used as sensing nodes).

5.2.1.1 Goal

The goal of this section is to provide a short guide to get started with the transceiver implementation on USRP board. It provides:

- A description of the versioning of the Ettus Research USRP2 board environment
- A quick architectural and design overview of the solution
- The salient features of the implementation
- A files list and configuration variables description

5.2.1.2 System overview

The *Transceiver Facility* is a specification that provides a reference API addressing the common programming needs of radio transceivers [16]. It is built upon core basic concepts like the transmitter and receiver channel, the radio programming by a *tuning profile* and *time profile* for radio bursts and simple interface for I&Q baseband samples exchange.

The implementation of this interface on USRP2, also called *"Façade"* could be summarized as depicted in the figure below. It is worth noting that the implementation described in this section refers to the current version of the specification (version 1 of the document, the only openly published document at the present time), therefore the implementation suffer from the ambiguities that remain on the official document, like the stop time and burst size inconsistencies or *Undefined time* mode usage.

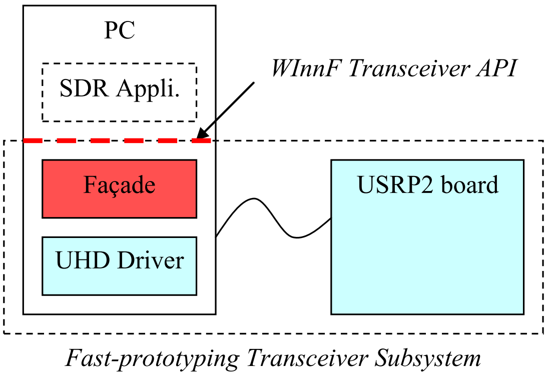


Figure 31: Transceiver implementation "Façade" in the overall system.

5.2.1.3 Intended audience

This section describes the implementation of the *Transceiver Facility* Interface on the USRP2 board. The aim of the next chapters is not to be exhaustive on the description of the software development or implementation but to provide a quick reference for a better understanding of the code. This section is typically aiming at developers wanting to enhance the existing code by adding new functionalities ("transceiver developers") or waveform designers wanting to use the current implementation of the transceiver for waveforms running on top of USRP2.

The section 5.2 makes the assumption that the reader is acquainted with the Transceiver Facility Specification document as available on the CREW portal, and the concepts it introduces. Otherwise the understanding of the next chapters could be difficult.

5.2.2 Set-up details

5.2.2.1 Installing

No specific installing steps are required for the transceiver implementation. The common USRP UHD driver installing procedure can be followed as depicted in USRP2 manufacturer website Ettus Research.

Please refer to [17] for more details.

5.2.2.2 IDE, development and linking

The code was developed using Eclipse IDE Platform SDK v3.5.2 with CDT Toolchain for Build and Debug (source included), CDT Utilities, Eclipse C/C++ Development Tooling Source, Eclipse C/C++ DSF gdb Debugger Integration (source included).

Together with UHD driver, the implemented code is using the Boost library and the Ommithread lib. These libraries need to be linked with the target binary.

The Eclipse project XCVR_USRP2_v1.0 file already contains the necessary settings.

For reference the table below compiles all the environment configuration and driver versioning information.

Table 5: Driver and environment versioning control.

Item	Version

Item	Version
Linux PC	Ubuntu version 11.04
UHD driver	003.004.002-128-g12f7a5c9
Eclipse IDE Platform SDK	v3.5.2
CDT GNU Toolchain Build Support	6.0.0.201002161416
CDT GNU Toolchain Build Support Source	6.0.0.201002161416
CDT GNU Toolchain Debug Support	6.0.0.201002161416
CDT GNU Toolchain Debug Support Source	6.0.0.201002161416
CDT Utilities	5.1.0.201002161416
Eclipse C/C++ Development Tooling Source	6.0.0.201002161416
Eclipse C/C++ DSF gdb Debugger Integration	2.0.0.201002161416
Eclipse C/C++ DSF gdb Debugger Integration Source	2.0.0.201002161416

5.2.2.3 Hardware

The implementation was carried out for the USRP2 board. Several daughter boards were available during development. Owing to the focus on time response functionalties implementation and validation, Basic TX and Basic RX daughter boards were used. A digital oscilloscope was used for validation of TX. A signal generator was used for the RX functionalities testing. **The current implementation version does not use RF features of the USRP2 hardware set-up.**

5.2.3 Overall architecture

5.2.3.1 Components and software architecture

The *"Façade"* implementation is primarily based on the UHD driver. However it relies as well on standard C++ libraries, especially on the boost library. Also the ommiThread library is used for threads creation and multitasking primitives like mutexes and semaphores. The Figure 32 depicts this simple layered architecture.

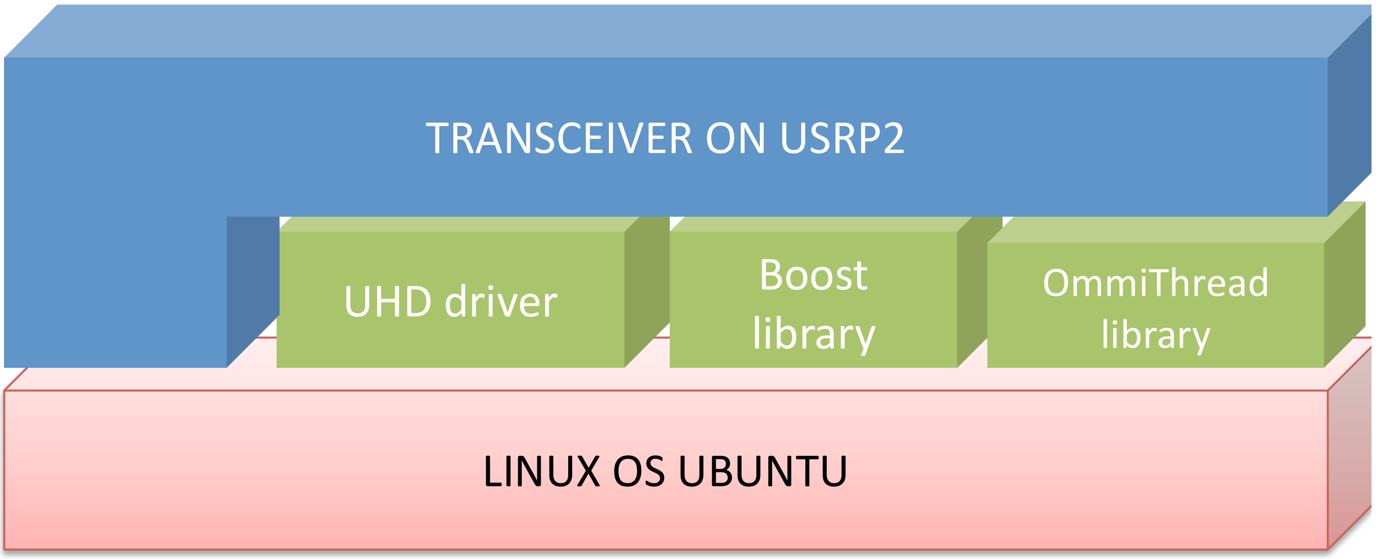


Figure 32: Transceiver on USRP2 simple layered architecture.

5.2.3.2 Logical architecture

The whole implementation is based on the concepts from the *Transceiver Facility* document.

This approach turns out into the implementation of a main class DeviceImp with two key attributes: a `TransmitChannel` and a `ReceiveChannel` classes. It is important to note that any transceiver device implementation instantiation creates both Transmit and Receive channels, whatever the modem waveform on the other side of the API needs are.

The next picture provides an overview of the transceiver overarching architecture.

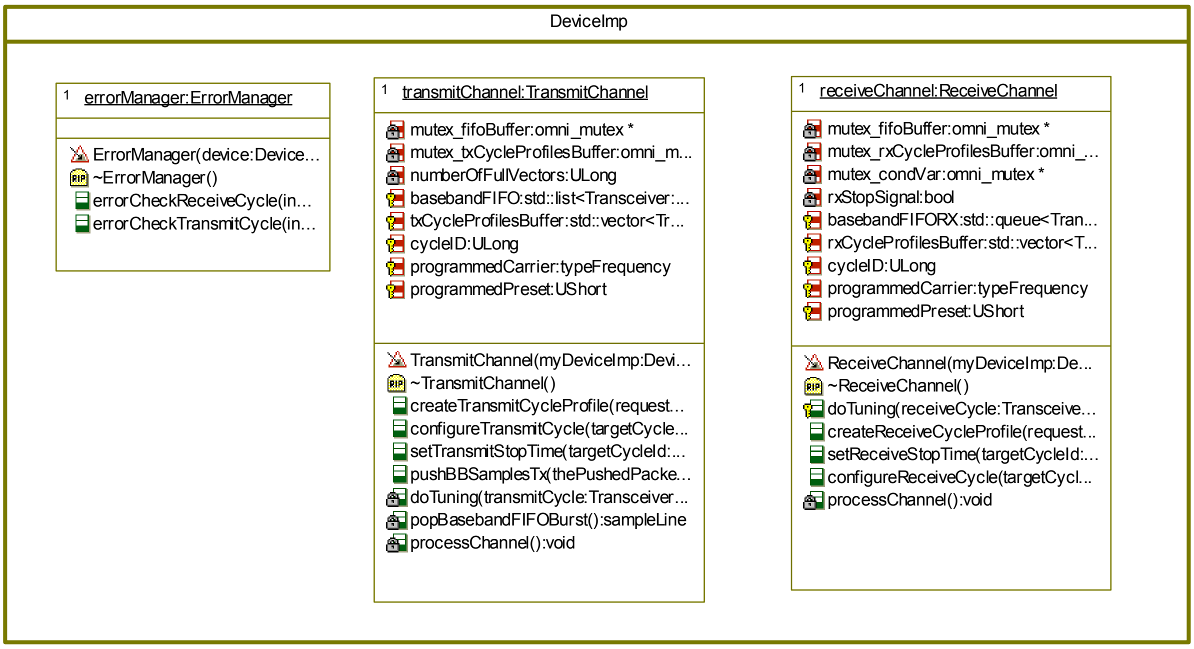

DeviceImp
1 errorManager:ErrorManager
ErrorManager(device:Device...
~ErrorManager()
errorCheckReceiveCycle(in...
errorCheckTransmitCycle(in...
1 transmitChannel:TransmitChannel
mutex_fifoBuffer:omni_mutex *
mutex_txCycleProfilesBuffer:omni_m...
numberOfFullVectors:ULong
basebandFIFO:std::list<Transceiver:...
txCycleProfilesBuffer:std::vector<Tr...
cycleID:ULong
programmedCarrier:typeFrequency
programmedPreset:UShort
TransmitChannel(myDeviceImp:Devi...
~TransmitChannel()
createTransmitCycleProfile(request...
configureTransmitCycle(targetCycle...
setTransmitStopTime(targetCycleId:...
pushBBSamplesTx(thePushedPacke...
doTuning(transmitCycle:Transceiver...
popBasebandFIFOBurst():sampleLine
processChannel():void
1 receiveChannel:ReceiveChannel
mutex_fifoBuffer:omni_mutex *
mutex_rxCycleProfilesBuffer:omni_...
mutex_condVar:omni_mutex *
rxStopSignal:bool
basebandFIFORX:std::queue<Tran...
rxCycleProfilesBuffer:std::vector<T...
cycleID:ULong
programmedCarrier:typeFrequency
programmedPreset:UShort
ReceiveChannel(myDeviceImp:De...
~ReceiveChannel()
doTuning(receiveCycle:Transceive...
createReceiveCycleProfile(request...
setReceiveStopTime(targetCycleId:...
configureReceiveCycle(targetCycl...
processChannel():void

Figure 33: DeviceImp class and attributes.

The only possible interactions between the transceiver and the modem waveform are those described on the Transceiver API document.

Thus for receive functionalities, waveform interfaces through:

```
Transceiver::ULong createReceiveCycleProfile(
                  Transceiver::Time              requestedReceiveStartTime,
                  Transceiver::Time              requestedReceiveStopTime,
                  Transceiver::ULong                   requestedPacketSize,
                  Transceiver::UShort            requestedPresetId,
                  Transceiver::Frequency         requestedCarrierFrequency);
void configureReceiveCycle(
                  Transceiver::ULong                   targetCycleId,
                  Transceiver::Time              requestedReceiveStartTime,
                  Transceiver::Time              requestedReceiveStopTime,
                  Transceiver::ULong                requestedPacketSize,
                  Transceiver::Frequency       requestedCarrierFrequency);
void setReceiveStopTime(
                  Transceiver::ULong                targetCycleId,
                  Transceiver::Time            requestedReceiveStopTime);
```

And for transmit functionalities, waveform interfaces through:

```
void pushBBSamplesTx(
                  Transceiver::BBPacket        *thePushedPacket,
                  Transceiver::Boolean         endOfBurst);
Transceiver::ULong createTransmitCycleProfile(
                  Transceiver::Time            requestedTransmitStartTime,
                  Transceiver::Time            requestedTransmitStopTime,
                  Transceiver::UShort          requestedPresetId,
                  Transceiver::Frequency       requestedCarrierFrequency,
                  Transceiver::AnaloguePower   requestedNominalRFPower);
void configureTransmitCycle(
                  Transceiver::ULong                targetCycleId,
                  Transceiver::Time            requestedTransmitStartTime,
                  Transceiver::Time            requestedTransmitStopTime,
                  Transceiver::Frequency       requestedCarrierFrequency,
                  Transceiver::AnaloguePower   requestedNominalRFPower);
void setTransmitStopTime(
                  Transceiver::ULong                targetCycleId,
                  Transceiver::Time            requestedTransmitStopTime);
```

For the data exchange, the interface for receive channel is

```
pushBBSamplesRx(
                  BBPacket                     *thePushedPacket,
                  Boolean                      endOfBurst) = 0;
```

But this operation (according to the *Transceiver Facility Specification*) is to be implemented by the waveform modem[1].

5.2.4 System wide design decisions

5.2.4.1 USRP2 constraints

One of the main issues of the USRP2 hardware is the time required for communicating with the board. Given the fact that data and command and control operations are conveyed through the Ethernet driver

[1] For a complete description of this operations please refer to the Transceiver Facility Specification.

a significant latency is introduced between the PC running the transceiver and waveform and the board embedded firmware.

This latency introduces two constraints:

- **Reduced time accuracy:** the transceiver software running on the PC is not able to know with enough accuracy the actual time of the board;
- **Reduced reactivity:** a significant anticipation time is necessary to make sure that the requests are issued in time with regards to the target internal time.

Following measurements performed on the board, this time is estimated to 3 ms. In other words the transceiver by default requires a minimum anticipation of 3 ms prior to any action on the board.

Example: if one command for a burst creation is targeting time T0+4 ms the corresponding command needs to be issued at T0+1 ms the latest.

5.2.4.2 The main transceiver task

The key design principle of the implementation is the inclusion of all the processing in a single task. During the development phase other alternatives were considered, typically creating two independent tasks for transmitting (belonging therefore to the `TransmitChannel` class) and for receiving (belonging to the `ReceiveChannel` class) however the existence of two separated tasks required more elaborated synchronization mechanisms and increased the complexity. The final decision to go for a single task simplifies significantly the design for a multithreading point of view and reduces latencies produced by task contention Linux OS mechanisms.

The main transceiver task follows a consumer-producer paradigm: the waveform modem produces transmission and receive burst requests with a *timing profile* and *tuning profile*. These requests are stored in a cycles buffer. The requests production depends entirely on the waveform modem execution. The transceiver main task consumes those cycles by performing the corresponding radio programming of the USRP2 board in accordance to the *tuning profile* of the burst and sends or receives samples in accordance to the *time profile*.

5.2.4.3 Cycles buffers

5.2.4.3.1 The main cycles buffer

The `cyclesBuffer` from the `DeviceImp` class stores the cycles requested by the modem. This buffer is the single entry point for the waveform modem to provide cycles to the transceiver.

It is this FIFO buffer that keeps track of the number of requested cycles and its type (either TX or RX). The details of the cycle i.e. *Tuning profile* and *Time profile* are stored in separated buffers within `transmitChannel` and `receiveChannel` classes. This main buffer is intended as a token buffer enabling the single transceiver task or `cyclesProcessingMainTask()` (consumer) to be synchronized through semaphores to the modem waveform task (producer).

`createTransmitCycleProfile()` and `createReceiveCycleProfile()` operations fill those buffers at each modem waveform invocation.

5.2.4.3.2 Transmit cycles buffer

This buffer is independent from the main cycles buffer from `DeviceImp` class. It is filled by the `createTransmitCycleProfile()` operation. The TX burst or cycle details are stored in this buffer. It stores elements of the type class `TransmitCycleProfile`.

5.2.4.3.3 Receive cycles buffer

Similar to the previous buffer, this buffer is independent from the main cycles buffer from `DeviceImp` class. It is filled by the `createReceiveCycleProfile()` operation. The RX burst or cycle details are stored in this buffer. It stores elements of the type class `ReceiveCycleProfile`.

5.2.4.4 The synchronization between modem and transceiver

Producer consumer synchronization is achieved taking advantage of the usage of the common buffer. A counting semaphore is allocated to the common buffer. The semaphore behaves as follows:

- The counting semaphore is initialized to a maximum value defined as constant in the constants file.
- The value is decremented by `createTransmitCycleProfile()` and `createReceiveCycleProfile()` operations.
- The value is incremented every time one cycle is processed by the main transceiver task.

The figure below provides an overview of the transceiver real-time architecture.

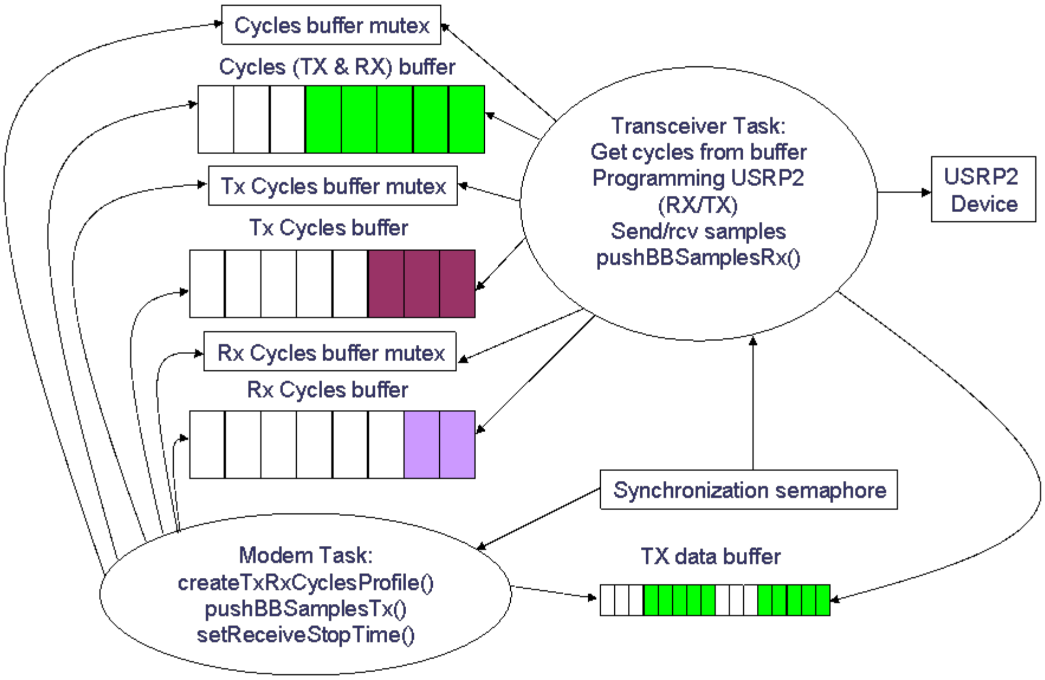


Figure 34: Main Transceiver task, modem and buffers synchronization.

5.2.4.5 Tuning presets

5.2.4.5.1 Tuning presets identification

Following the specification guidelines the radio parameters programming is encapsulated in the “Tuning Presets” concept. Tuning presets are defined as a set of variables corresponding to a specific radio profile. By definition and as stated in the specification the tuning preset contents, in terms of variables and their values ranges, is highly depending on the hardware. For the sake of simplicity the implementation covers a very reduced number of those parameters.

These parameters were deemed sufficient to characterize the USRP2 platform. Moreover the focus was rather on the dynamic behaviour and the feasibility of changing on the fly the values of some parameters from burst to burst (for example the sampling ratio).

In the present implementation only two arbitrary Tuning presets are defined in the configuration file.

It is important to highlight that Preset value is exchanged between waveform modem and transceiver by a single number. Both sides of the interface need to know in advance the numbering and Tuning preset contents. As described in the specification this definition is typically carried out during engineering phase.

5.2.4.5.2 Tuning presets management

The programming of the USRP2 through its driver has a significant cost in terms of delay. In order to avoid unnecessary programming, the `TransmitChannel` and the `ReceiveChannel` keep track of the previous programmed Preset and avoid reprogramming if the Preset remains the same as it is the case for some waveforms.

The attributes `programmedPreset` within the `TransmitChannel` and `ReceiveChannel` classes hold the last programmed preset.

5.2.4.6 Time management

The time management is the core feature of the implementation. As introduced above the USRP2 latency constraints call for a specific approach in which operations are requested well in advance. This same latency makes difficult accurate time tracking on the PC hosting the Transceiver Façade.

Indeed the *Façade* never knows the time with precision; it is actually timed thanks to the bursts (cycles creation). The first bursts either TX or RX are requested with `Immediate time` mode. This resets the internal board counter to zero and establishes a reference time, the first activation, to be used by further bursts using `EventBased time` mode with time shifts based on this very first activation. It is also possible, in order to avoid tracking the time from the very beginning, to use the previous burst activations (event sources `TransmitStartTime` and `ReceiveStartTime`). As far as `EventBased time` is used, the *Façade* is therefore able to timely create Bursts activations.

If `Immediate time` mode is used in ways other than the first activation, the transceiver cannot guarantee time tracking and synchronization between the *Façade* and the USRP2 board is lost. For these cases the Transceiver behaviour is considered as undefined.

One of the identified shortcomings of the current version of the Transceiver Facility Specification is the ambiguity regarding `Undefined time` mode. The Specification describes this time mode as any other time mode (Absolute, Event Based, Immediate) and thus it could be used for both start time and stop times. However, it usage as start time is clearly confusing, and overlapping with the Immediate one. In consequence, in the USRP2 implementation the Undefined mode is reserved for stop time only.

When requesting an Undefined stop time burst[2], the transceiver will receive samples in a continuous way until the `setReceiveStopTime()` operation is invoked by the modem waveform.

5.2.4.7 Supported functionalities

The current version of the implementation is supporting a limited number of features:

- A reduced sub-set of timing modes and combinations are implemented: Only Immediate and Event Based time modes are supported. Moreover Event Based mode is only supporting `TransmitStartTime` event source for the Transmit Channel and the `ReceiveStartTime` event source for the Receive Channel.

[2] For high sampling rates the waveform modem running on the PC and the Ethernet configuration shall make sure the samples are retrieved fast enough. Otherwise the UHD driver will signal buffer overflow error.

- Undefined time mode is only supported for the receive channel. Since reception during undefined and potentially long durations is a common use case for waveforms requesting typically a synchronization procedure, the implementation priority was given to the receive channel.
- Radio programming ("tuning profile") features are simplified: Only sampling ratio is actually implemented within the Tuning Preset.
- It is possible to send several packets (`pushBBSamplesTx()`) for a burst, however only one single packet per burst has been validated.
- Error manager extension is not implemented.

5.2.5 Reference diagrams

5.2.5.1 Transmit channel process channel diagram

The Transmit channel process channel diagram is given in Figure 35.

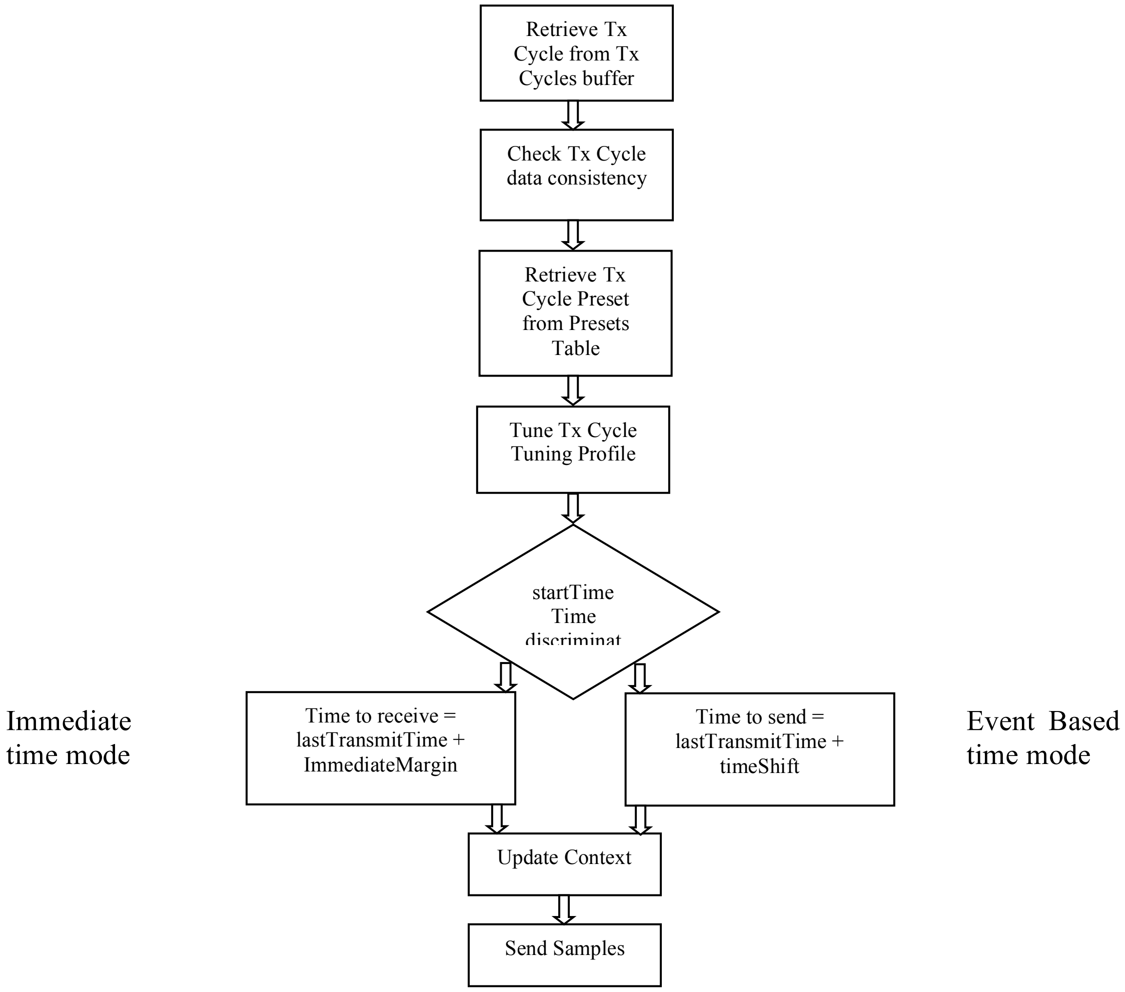

Retrieve Tx Cycle from Tx Cycles buffer
Check Tx Cycle data consistency
Retrieve Tx Cycle Preset from Presets Table
Tune Tx Cycle Tuning Profile
startTime Time discriminat
Immediate time mode
Time to receive = lastTransmitTime + ImmediateMargin
Time to send = lastTransmitTime + timeShift
Event Based time mode
Update Context
Send Samples

Figure 35: Transmit channel process channel diagram.

5.2.5.2 Receive channel process channel diagram

The Receive channel process channel diagram is given in Figure 36.

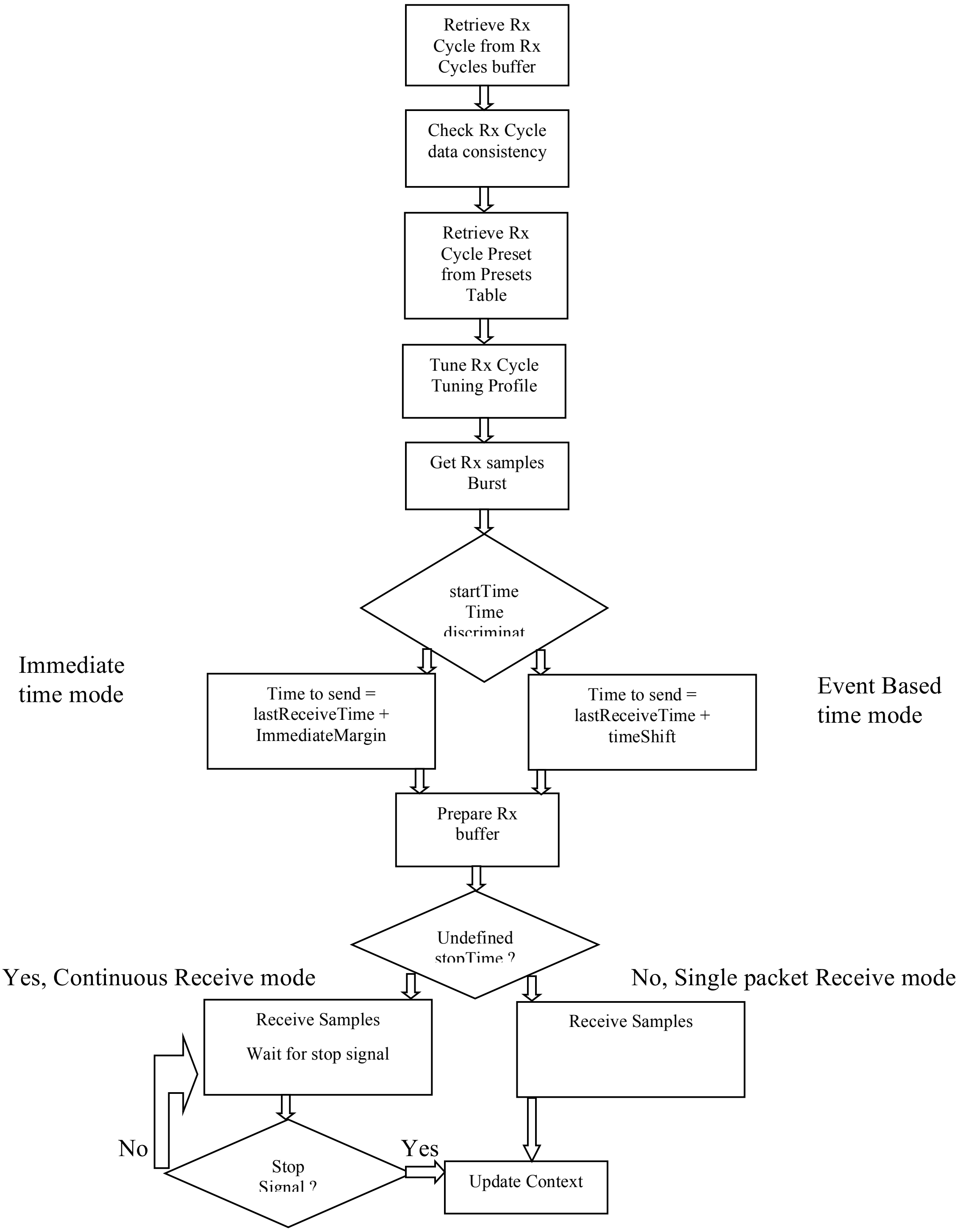

Retrieve Rx Cycle from Rx Cycles buffer
Check Rx Cycle data consistency
Retrieve Rx Cycle Preset from Presets Table
Tune Rx Cycle Tuning Profile
Get Rx samples Burst
startTime Time discriminat
Immediate time mode
Time to send = lastReceiveTime + ImmediateMargin
Time to send = lastReceiveTime + timeShift
Event Based time mode
Prepare Rx buffer
Undefined stopTime ?
Yes, Continuous Receive mode
No, Single packet Receive mode
Receive Samples
Wait for stop signal
Receive Samples
No
Stop Signal ?
Yes
Update Context

Figure 36: Receive channel process channel diagram.

5.2.6 Code breakdown

5.2.6.1 List of files

transceiver_implementation_usrp2.hpp:

This file contains the main classes definition and some additional structures and data types.

transceiver_implementation_usrp2_config.hpp:

This file contains all the constants for transceiver configuration (refer to next chapter for further details).

transceiver_implementation_usrp2.cpp:

This file contains the `DeviceImp` class constructor/destructor and Preset configuration operations.

transceiver_implementation_ursp2_rx.cpp:

This file contains the `ReceiveChannel` class constructor/destructor and operations.

transceiver_implementation_usrp2_tx.cpp:

This file contains the `TransmitChannel` class constructor/destructor and operations.

transceiver_implementation_main_task.cpp:

This file contains the transceiver cycles processing main task.

transceiver_implementation_common.cpp:

This file contains shared code.

transceiver_implementation_usrp2_error_man.cpp:

This file contains the error manager (not implemented).

5.2.6.2 Global configuration variables

The configuration header contains a number of global constants that define the USRP2 configuration and board settings.

Maximum number of cycles: the number of both TX and RX cycles that the transceiver can put in the FIFO buffer at a given time.

```
static const Transceiver::UShort MAXCYCLES = 10;
```

Maximum packet size: the maximum number of samples in a packet. This number is directly related to the Ethernet driver configuration. 363 is the maximum value to avoid fragmentation on the PC size and reconstruction in the board embedded firmware.

```
static const Transceiver::Long MAX_PACKET_SIZE = 363; // Current
ethernet transport conf. 363x2(I&Q)*16 = 1452 bytes
```

Default carrier frequency: the default RF carrier frequency value is in Hz.

```
static const Transceiver::Frequency DEFAULT_CARRIER_FREQ = 100e6; //
100 MHz
```

Default analogue power: the Default analogue power is in dBm (not used in the current version).

```
static const Transceiver::AnaloguePower DEFAULT_ANALOGUE_PW = 10;
```

Number of presets: is the predefined number of tuning presets supported by the implementation.

```
static const Transceiver::UShort NUMBER_OF_PRESETS = 2;
```

MHz constant: is the constant definition for MHz calculations.

```
static const Transceiver::ULong MHz = 1e6;
```

nanoseconds constant: is the constant definition of nanoseconds for calculations.

```
static const Transceiver::ULong nSec = 1e9;
```

Reactivity time margin: is the time guard for immediate time commands.

```
static const Transceiver::ULong TIMEMARGIN = 3e6; // 3 us
```

Maximum number of receive cycles: is the maximum number of receive cycles that the transceiver can store in the RX FIFO buffer at a given time.

```
static const Transceiver::UShort MAXRXCYCLES = 10;
```

Default RX sampling frequency: is the Default sampling frequency for the receive channel.

```
static const double DEFAULT_RX_SAMPLING_FREQUENCY = 100e6/64;
```

Time margin for immediate receive requests: is the time margin that is added to current time at any receive immediate time request.

```
static const Transceiver::AbsoluteTimeStruct
IMMEDIATE_MARGIN_RX(1,0);
```

Default RX antenna configuration: is the settings for the receive channel antenna (configuration options are depending upon the daughter board options).

```
static const std::string DEFAULT_RX_ANTENNA = "RX2";
```

Maximum number of transmit cycles: is the maximum number of transmit cycles that the transceiver can store in the TX FIFO buffer at a given time.

```
static const Transceiver::UShort MAXTXCYCLES = 2;
```

Default TX sampling frequency: is the Default sampling frequency for the transmit channel.

```
static const double DEFAULT_TX_SAMPLING_FREQUENCY = 100e6/16; //
6.25 Msps
```

Time margin for immediate transmit requests: is the time margin that is added to current time at any transmit immediate time request.

```
static const Transceiver::AbsoluteTimeStruct
IMMEDIATE_MARGIN_TX(0,5000000); // 5000000 nanoseconds = 5ms
```

Default TX antenna configuration: is the settings for the transmit channel antenna (configuration options are depending upon the daughter board options).

```
static const std::string DEFAULT_TX_ANTENNA = "TX/RX";
```

5.3 VESNA Interfaces

5.3.1 Overview

The VESNA based testbed uses a layered approach to the control of wireless sensor nodes and SNE-ISMTV radio hardware attached to them. Each interface layer provides increased flexibility at the expense of increased complexity and in turn increased time required to develop, test and verify the testbed configuration before the experiment can be performed. Since experiments supported by the testbed can vary greatly in complexity it is therefore up to the experimenter to choose the interface that represents the best compromise between flexibility and complexity. This section provides an overview of the available interfaces to aid in this choice.

Interface layers in the order of increasing complexity.

- Nodes running default testbed firmware, control through the HTTP-like resource access protocol
- Nodes running custom firmware, using C interface to the spectrum sensing and/or signal generation API
- Nodes running custom firmware, using register-level hardware access

Wireless sensor nodes in the VESNA based testbed communicate through a mesh network based on ZigBee with proprietary extensions. This back-bone network is used to set up and control the experiment and retrieve measurement results. It uses separate, dedicated radio hardware and is therefore independent of any radio operations that are part of the experiment itself. As nodes are not normally physically accessible, this back-bone connection is the only link between the experimenter and the hardware.

Each VESNA node runs a single application at a time and can store a large number of applications on the microSD card. Over-the-air reprogramming infrastructure allows uploading and execution of applications remotely through the back-bone network. The microSD card is also available to applications as a temporary non-volatile mass-storage for experimental results.

VESNA nodes in the CREW testbeds are normally running a default, multi-purpose application that exposes a very high-level interface to the radio hardware. This interface consists of a number of resources with read- and/or write- access through an application-layer, HTTP-like protocol. Each node exposes this protocol to the ZigBee mesh network. The ZigBee network coordinator in each testbed acts as a proxy that provides an encapsulation of this protocol through a TCP/IP connection. A web application running on a server at JSI accepts the connection from the coordinator and provides a proper HTTP REST interface to it. For development and debugging purposes in a laboratory the HTTP-like protocol can also be exposed on a serial line and connected directly to a desktop computer.

While using the testbed through this interface is sufficient for simple spectrum sensing and signal generation tasks its usability is limited because of the restricted bandwidth and inherent and unpredictable delays involved in sending commands to VESNA nodes through the ZigBee mesh network and the Internet. Experiments that have stricter synchronization requirements must remove the latency of the network communication and bring control closer to the hardware. In this case, the experimenter can develop a custom application that is uploaded and executed directly on VESNA nodes. Removing the need for back-bone network traffic during an experiment is also beneficial when such traffic might interfere with measurements (e.g. when experiments are done in the same frequency band as the one used for the back-bone network). VESNA software driver library exposes a spectrum sensing and signal generation API that abstracts radio hardware details for these two tasks, allowing for faster development.

SNE-ISMTV hardware on VESNA nodes contains highly configurable transceivers. While none of them features a true software-defined radio architecture, hardware components can be reconfigured by software to fit a large number of usage scenarios. The spectrum sensing and signal generation API provides a number of pre-set hardware profiles that cover most commonly used radio configurations. This was done to free the experiment developer from the task of configuring radio hardware, which can be a complicated and time consuming task. If required, new profiles can still be added by the experimenter though. This usually involves specialized software from the transceiver IC manufacturer (e.g. SmartRF studio from Texas Instruments for VESNA transceivers based on the Texas Instruments CC series). Due to implementation details radios may not perform to specification in certain combinations of settings. This means that each profile usually requires testing and calibration in a controlled laboratory environment before it can be used for experiments in the field.

Should an experiment need to perform radio operations other than energy detection spectrum sensing and signal generation, the experimenter can choose to communicate directly with the radio hardware through register-level access functions. This exposes the full range of features, but requires intimate knowledge of the hardware involved. Using register-level access functions is outside of the scope of this document. The last section gives a brief overview and gives references to relevant documentation.

5.3.2 HTTP-like interface

Each sensor node in the testbed acts as a server, accepting requests over the ZigBee backbone mesh network and sending responses back to the client. A protocol similar to HTTP is used in these transactions. The sensor node exposes a number of resources, each with its unique URL. A client can issue a GET request to read data associated with the resource, or a POST request, to write data to the resource. Each resource can also accept a number of parameters, appended to the URL, again similar to HTTP query parameters.

The HTTP-like resource access protocol itself and the means of accessing it have been described in more detail in D3.3. The following sections focus only on the resources that can be used to control SNE-ISM TV hardware.

5.3.2.1 Spectrum Sensing

Nodes can be programmed with spectrum sensing tasks in form of simple programs. Each program has a start time at which it is activated and an end time at which it is stopped. While the program is active, the specified spectrum sensing device continuously sweeps its tuner over a range of frequencies, recording input power at the receive antenna. Results of short spectrum sensing programs can be retrieved immediately over the network, while longer scans store the results on the SD card for later retrieval.

All radio tuners used on VESNA based testbeds divide their frequency range into channels. The translation between the channel number and the tuned frequency depends on the hardware configuration.

Each program is hence defined with the following parameters:

- Start time
- End time
- Device and its configuration
- First channel, channel step, last channel
- Destination for the measurement results (slot ID or real-time network output)

Storage of the measurements

Results of measurements are stored and transmitted in a binary format. Each program records multiple sweeps over the radio frequency spectrum:

```
+---------+---------+-----+---------+
| sweep 1 | sweep 2 | ... | sweep M |
+---------+---------+-----+---------+
```

Number of sweeps, M, depends on the length of the spectrum sensing activity and the sweep time of the chosen device.

Each sweep contains a timestamp of the start of the sweep and input power measurements:

```
+-----------+---------+---------+-----+---------+
| timestamp | value 1 | value 2 | ... | value P |
+-----------+---------+---------+-----+---------+
  int32_t     int16_t   int16_t         int16_t
```

Timestamp records the time in milliseconds since the program started and is encoded as an unsigned 32-bit integer.

Input power measurements are in 1/100 dBm (for example 100 => 1.00 dBm, 50 => 0.50 dBm) and are encoded as signed 16-bit integers.

Number of input power measurements per sweep depends on the first and last channel and channel step programmed for the spectrum sensing task.

Value N contains power measurement for channel M, where

M = channel_start + channel_step * (N – 1)

and

N is in the range from 1 to channel_num inclusive

Limitations on measurement length and data size

The most relevant limiting factor is the transfer speed of the ZigBee network. Its effective speed is about 1 kB / second.

The maximum size of an SD card slot is 1 MB. This is enough for approximately 40 minutes of scanning with CC-series based hardware and approximately 6 hours with the UHF receiver.

When the SD card slot in use is completely full, the scanning is automatically stopped.

Getting hardware information

The following resources allow querying for spectrum sensing hardware present on the node.

```
GET sensing/deviceList
```

```
dev #0, CC1101, 4 configs
```

Lists all devices present and number of pre-set configurations available for each device.

```
GET sensing/deviceConfigList
GET sensing/deviceConfigList?devNum=0
dev #0, CC1101, 4 configs:
  cfg #0: CC1101, f_c=868 MHz, BW=400 kHz:
    base: 868299866 Hz, spacing: 199951 Hz, bw: 406250 Hz,
    channels: 127, time: 20 ms
  cfg #1: CC1101, f_c=868 MHz, BW=800 kHz:
    base: 867999729 Hz, spacing: 199797 Hz, bw: 843681 Hz,
    channels: 63, time: 100 ms
  cfg #2: CC1101, f_c=906 MHz, BW=400 kHz:
    base: 905999991 Hz, spacing: 399595 Hz, bw: 421840 Hz,
    channels: 255, time: 20 ms
  cfg #3: CC1101, f_c=906 MHz, BW=800 kHz:
    base: 905999991 Hz, spacing: 399595 Hz, bw: 843681 Hz,
    channels: 255, time: 20 ms
```

Lists all configurations for a device. If no device number is given, then all configurations are listed for all devices.

Please consult Annex I for the full list of procedures on how to get the hardware information.

Spectrum sensing setup

Please consult Annex II for the full list of procedures on how to setup spectrum sensing hardware.

Retrieving the measurements

```
GET sensing/slotInformation?id=0
size=0
version=0
status=EMPTY
crc=0
```

Retrieves meta-data about spectrum sensing measurements stored in a SD card slot.

```
size=%u (1)
version=%u (2)
status=(EMPTY | INCOMPLETE | COMPLETE | UNDEFINED) (3)
crc=%u (4)
```

Fields (%u - unsigned integer number, represented in decimal notation):

1. Size of stored data, in bytes
2. Uptime of the node in seconds at the time of writing data to the SD card.
3. Status of the slot: EMPTY – the slot does not contain any data and is ready to be used in a new measurement, INCOMPLETE – data is currently being written to the slot, COMPLETE –

measurement results have been successfully written to the slot and are ready for retrieval, UNDEFINED – an unexpected value.

4. CRC of data in the slot.

Please consult Annex III for the full list of procedures on how to retrieve measurement information.

Freeing up data storage slots

For preventing data loss, nodes will refuse to overwrite slots that have data in them. Setting up a slot that already contains data as destination for measurements will give an error. Slots have to be explicitly cleared after the data in them has been downloaded

```
POST sensing/freeUpDataSlot?id=0
length=...
1
crc=...
```

5.3.2.2 Signal Generation

Nodes can be programmed with signal generation tasks in form of simple programs. Each program has a start time at which it is activated and an end time at which it is stopped. While the program is active, the specified signal generation device transmits a signal with the specified central frequency with the specified power. Transmitted waveform depends on the hardware configuration pre-set and is usually evident from the configuration's name.

All radio transmitters used on VESNA based testbeds divide their frequency range into channels. The translation between the channel number and the tuned frequency depends on the hardware configuration.

Each program is hence defined with the following parameters:

- start time
- end time
- device and its configuration
- transmission channel and power

Getting hardware information

The following resources allow querying for signal generation hardware present on the node.

```
GET generator/deviceList
dev #0, CC2500, 1 configs
```

Lists all devices present and number of pre-set configurations available for each device.

```
GET generator/deviceConfigList
GET generator/deviceConfigList?devNum=0
dev #0, CC2500, 1 configs:
  cfg #0: CC2500, 2.4 GHz, 200 kHz channels:
     base: 2399999908 Hz, spacing: 199814 Hz, bw: 210938 Hz,
     channels: 256, min power: -55 dBm, max power: 0 dBm,
     time: 5 ms
```

Lists all configurations of a device. If no device number is given, then all configurations are listed for all devices.

Format of each entry is as follows:

```
dev #%u, %s
    (1) (2)
  cfg #%u: %s
      (3) (4)
     base: %u Hz, spacing: %u Hz, bw: %u Hz,
           (5)             (6)        (7)
     channels: %u, min power: %d dBm, max power: %d dBm,
               (8)            (9)                (10)
     time: %u ms
           (11)
```

Fields (%u - unsigned integer number, represented in decimal notation, %d – signed integer number, represented in decimal notation, %s – free form string):

1. ID of the device, used when referring to it when programming a signal generation task
2. human-readable name of the device
3. ID of the device configuration pre-set, used when referring to it when programming a signal generation task
4. human-readable name of the configuration
5. central frequency of channel 0
6. difference in central frequency between two adjacent channels
7. transmit bandwidth
8. total number of channels
9. minimum transmission power
10. maximum transmission power
11. setup time in milliseconds

Channel numbers M range from 0 to channel_num – 1. To calculate central frequency for a channel use the following formula:

f_c = channel_base + M * channel_spacing

Signal generation setup

```
POST generator/program
length=...
in 10 sec for 10 sec with dev 0 conf 0 channel 0 power -10
in 30 sec for 10 sec with dev 0 conf 0 channel 0 power -20
crc=...
```

POST data contains one signal generation program per line, maximum 10 lines.

Format of each line is as follows:

```
in %u sec for %u sec with dev %u conf %u channel %u power %d
   (1)        (2)             (3)     (4)        (5)      (6)
```

Fields (%u - unsigned integer number, represented in decimal notation, %d – signed integer number, represented in decimal notation):

1. Number of seconds until the start of transmission; the start is relative to the moment of receiving the request.
2. Length of transmission, in seconds

3. ID of device used for transmission
4. ID of configuration used for transmission
5. Frequency channel used for transmission
6. Transmit power, in dBm

Programs must be sorted by increasing start time and their time intervals must not overlap. If any of the lines is invalid, the program is rejected.

```
GET generator/program
in 10 sec for 10 sec with dev 0 conf 0 channel 0 power -10
in 30 sec for 10 sec with dev 0 conf 0 channel 0 power -20
```

Issuing a GET request to **generator/program** resource returns the information in same format as used in the POST request.

The number of seconds until the start of transmission is re-calculated relative to the reception time of the GET request, hence the number of seconds until transmission is constantly decreasing. When a program is active, the number of seconds until the start is 0 and the length of the transmission is decreased accordingly.

5.3.3 C interface

This section describes the C interface to spectrum sensing and signal generation API for transceivers and receivers on VESNA SNE-ISMTV hardware. It is aimed at the experiment developer that wishes to develop custom firmware for wireless sensor nodes in VESNA based testbeds while still using the simple abstract interface to the experimental radio hardware.

Custom firmware can implement arbitrary functionality on the sensor node itself, limited only by the memory and processing capabilities of the VESNA sensor node core (SNC). However it is expected that majority of custom firmware configurations will build upon the default firmware and use the existing framework by integrating with the ZigBee mesh network, resource access protocol and other testbed infrastructure. For this purpose a VESNA software library ("vesna-drivers") is provided. The whole VESNA library is outside of the scope of this document and the following sections focus only on the spectrum and signal generation part.

The experimenter does not usually have physical access to wireless sensor nodes deployed on public lighting infrastructure in an outdoor VESNA based testbed. Therefore an error in custom firmware that makes a sensor node inaccessible through the backbone ZigBee network, and thus also the over-the-air reprogramming infrastructure, can be a serious issue. In such cases, time consuming and costly manual access and reprogramming is necessary to restore the sensor node to operation. While there are mechanisms in place to reduce the probability of such an occurrence (namely a hardware watchdog timer and an intelligent bootloader that reverts to a known-good configuration in case of a failing firmware) not all possibilities for such an error can be covered. Therefore any custom firmware must be thoroughly tested at JSI before being uploaded to one of the outdoor testbeds.

5.3.3.1 Spectrum sensing API

Below is a list of steps required to use the spectrum sensing API followed by a detailed description of data types and functions involved.

1. **Choose device and a configuration pre-set**. VESNA library already contains definitions for all devices present on SNE-ISMTV. Many configuration pre-sets are also available, covering most commonly used hardware configurations. Alternatively the developer may choose to define a new one.

```
const struct spectrum_dev *my_dev = ...;
const struct spectrum_dev_config *my_dev_config = ...;
```

2. **Define a frequency sweep configuration for the chosen device and configuration**. Sweep configuration contains for example the frequency range to be sensed.

```
const struct sweep_config my_config = {
      .dev_config = my_dev_config,
      ...
};
```

3. **Define a callback function**. This function will be called to process, store or transmit measurements for each finished frequency sweep.

```
int  my_cb(const  struct  spectrum_sweep_config  *sweep_config,  int
timestamp, const int16_t data_list[]) {
      ...
}
```

4. **Setup the hardware**. VESNA library automatically detects and registers the radio hardware connected to the wireless sensor node, so this step is not normally required unless initialization routines have been overridden.

```
spectrum_add_dev(my_dev);
spectrum_reset();
```

5. **Start spectrum sensing**.

```
spectrum_run(my_dev, &my_config);
```

Please consult Annex IV for the documentation related to the data structures, types and functions reported in this section.

5.3.3.2 Signal generation API

Below is a list of steps required to use the signal generation API followed by a detailed description of data types and functions involved.

1. **Choose device and a configuration pre-set**. VESNA library already contains definitions for all devices present on SNE-ISMTV. Many configuration pre-sets are also available, covering most commonly used hardware configurations. Alternatively the developer may choose to define a new one.

```
const struct vsnSignalGenerator_device *myDev = ...;
const struct vsnSignalGenerator_deviceConfig *myDeviceConfig = ...;
```

2. **Define a signal transmission for the chosen device and configuration**. Configuration contains for example the central frequency and power for the transmission.

```
const struct vsnSignalGenerator_txConfig myTxConfig = {
      .deviceConfig = myDeviceConfig,
      ...
};
```

3. **Setup the hardware**. VESNA library automatically detects and registers the radio hardware connected to the wireless sensor node, so this step is not normally required unless initialization routines have been overridden.

```
vsnSignalGenerator_addDevice(myDev);
vsnSignalGenerator_reset();
```

4. **Start transmission**.

```
vsnSignalGenerator_start(myDev, &myTxConfig);
```

5. **Stop transmission**.

```
vsnSignalGenerator_stop(myDev);
```

Please consult Annex V for the documentation related to the data structures, types and functions reported in this section.

5.3.4 Register level access

At the lowest level all radio hardware used in VESNA based testbeds abstract their configuration in the form of registers with read- and/or write-access via digital buses. These registers can be used to reconfigure radio hardware (central frequency, channel filter bandwidth, modulation and demodulation settings, signal generation, etc.), send data for transmission or retrieve received data, retrieve measurement results and so on. Additional signals (for example, one or more interrupt lines and power-down switches) may also exist, depending on radio hardware, but their use is usually optional.

VESNA software library provides low-level functions for each radio architecture used on SNE-ISMTV hardware. Following is a brief overview of the available functions.

CC-based radios (SNE-ISMTV-TI868, SNE-ISMTV-TI24)

Driver module: vsnccradio.c

Register access functions:

```
int  vsnCC_read (u8 addr)
```

Read a byte from a radio register.

```
int  vsnCC_write (u8 addr, u8 data)
```

Write a byte to a radio register.

```
int  vsnCC_readBurst (char addr, char *dataPtr, u16 dataCount)
```

Read many values from one radio register.

```
int  vsnCC_writeBurst (char addr, const char *dataPtr, u16 dataCount)
```

Write many values to one radio register.

Hardware reference documents:

Texas Instruments: CC1101, Low-power sub-1GHz RF transceiver

Texas Instruments: CC2500, Single chip low cost low power RF transceiver

UHF receiver (SNE-ISMTV-UHF)

Driver module: vsntda18219hn.c

Register access functions:

```
uint8_t    vsnTDA18219_readReg (uint8_t reg)
```

Read a TDA18219 register.

```
void vsnTDA18219_writeReg (uint8_t reg, uint8_t value)
```

Write a value to a TDA18219 register.

Hardware reference documents:

NXP Semiconductors: TDA18219HN, Silicon Tuner for terrestrial and cable digital TV reception, Product datasheet

6 Conclusion

This document demonstrates how the CREW project has been optimized in the second year of the project in terms of testbed functionality, common data collection and storage, the CREW Portal, mix and match combinations, use of the transceiver API and interfacing with the VESNA based testbeds.

Table 1 in Section 1, in particular, provides a breakdown of how the each of the CREW federated testbeds and platforms now have most, if not all, of CREW core functionalities. These include the sharing of baseline functionality information, incorporation and sharing of hardware (/software) between testbeds, advanced sensing functionality, use of the CREW common data format, definition of benchmarked scenarios for each of the federation testbeds, the presence of access and usage information on the CREW Portal, remote open access for experiments that are performed in the context of the CREW project and usage of other CREW partners, external users and open call experimenters of the testbed/platform functionalities. This table provides a status report detailing how each of the testbeds has performed in terms of federation functionality.

This document introduces both the CREW repository, for sharing of experimental data, scripts and with the greater research community, in line with the common data collection and storage methodologies outlined in the CREW project description of work.

Table 4, in Section 5 of the document provides an extensive breakdown of the mix and match combinations performed as part of the interoperability testing work of the CREW project. The table demonstrates that the amount of device combination was quite extensive (all partners, including open call partners, having taken part in combination experiments involving at least two other partners) and should provide users wishing to use the CREW federation with a fair understanding of the sort of experiments, which are likely to be feasible. The table includes both intra-country and cross-country component combinations.

Detailed descriptions of the interfaces for transceiver API and the VESNA based testbeds are also provided as part of the CREW interface design and specification work of CREW.

This document shows how the basic operational platform has developed into a strong CREW federation, capable of numerous inter- testbed and platform experimental operations and facilitating functionality far greater than the use of the individual testbeds/platforms independently. The information provided in this document should prove very useful to experimenters for the second CREW Open Call as well as other experimenters wishing to make use of the CREW federation's capabilities.

7 References

[1] "CREW Repository." [Online]. Available: http://www.crew-project.eu/repository/.

[2] "CREW project common portal." [Online]. Available: www.crew-project.eu/portal.

[3] "Ettus Research, RF Daughterboards." [Online]. Available: https://www.ettus.com/product/category/Daughterboards.

[4] "cOntrol and Management Framework." [Online]. Available: http://mytestbed.net/projects/omf.

[5] T. Rakotoarivelo, M. Ott, G. Jourjon, and I. Seskar, "OMF: A Control and Management Framework for Networking Testbeds," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, p. 54, Jan. 2010.

[6] "Example of OMF within OpenWRT." [Online]. Available: https://github.com/nathansamson/OMF-Openwrt.

[7] N. Michailow, S. Krone, M. Lentmaier, and G. Fettweis, "Bit Error Rate Performance of Generalized Frequency Division Multiplexing," in *76th IEEE Vehicular Technology Conference (VTC Fall'12), Québec City, Canada, 3.9.*, 2012.

[8] "IMEC Sensing Engine User Manual." [Online]. Available: http://www.crew-project.eu/sites/default/files/SensingEngine_UserManual.pdf.

[9] A. Dejonghe, S. Pollin, L. Hollevoet, F. Naessens, E. Lopez, P. Raghavan, A. Bourdoux, P. Van Wesemael, J. Ryckaert, J. Craninckx, and L. Van der Perre, "Versatile Spectrum Sensing on Mobile Devices?," in *2010 IEEE Symposium on New Frontiers in Dynamic Spectrum (DySPAN)*, 2010, pp. 1–6.

[10] "Rice University WARP Project." [Online]. Available: http://warp.rice.edu.

[11] M. Ingels, V. Giannini, J. Borremans, G. Mandal, B. Debaillie, P. Van Wesemael, T. Sano, T. Yamamoto, D. Hauspie, J. Van Driessche, and J. Craninckx, "A 5mm^2 40nm LP CMOS 0.1-to-3GHz multistandard transceiver," in *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*, 2010, pp. 458–459.

[12] "CREW Portal, w-ilab.t Zwijnaarde USRP Iris usage." [Online]. Available: http://www.crew-project.eu/content/usrp2-usage.

[13] P. Van Wesemael, W. Liu, M. Chwalisz, J. Tallon, D. Finn, Z. Padrah, S. Pollin, S. Bouckaert, I. Moerman, and D. Willkomm, "Robust distributed sensing with heterogeneous devices," in *Future Network & Mobile Summit*, 2012.

[14] J.-H. H. C. Heller, S. Bouckaert, I. Moermann, S. Pollin, P. v. Wesemael, D. Finn, D. Willkomm, "WInnComm2011 - A Performance Comparison of Different Spectrum Sensing Techniques." 2011.

[15] D. Finn, J. Tallon, L. Dasilva, S. Pollin, W. Liu, S. Bouckaert, and J. V. Gerwen, "Experimental Assessment of Tradeoffs among Spectrum Sensing Platforms," *The Sixth ACM*

International Workshop on Wireless Network Testbeds, Experimental evaluation and Characterization (WiNTECH '11), 2011.

[16] E. Nicollet, S. Pothin, and A. Sanchez, "Transceiver Facility Specification, Wireless Innovation Forum, 2 February 2009," *SDRF-08-S-0008-V1_0_0_Transceiver_Facility_Specification.pdf*. [Online]. Available: http://groups.winnforum.org/p/cm/ld/fid=85.

[17] "Ettus Research, UHD - USRP Hardware Driver, 3 July 2012." [Online]. Available: http://files.ettus.com/uhd_docs/manual/html/.

Annex I. Getting the spectrum sensing hardware information from VESNA

The following resources allow querying for spectrum sensing hardware present on the node.

```
GET sensing/deviceList
dev #0, CC1101, 4 configs
```

Lists all devices present and number of pre-set configurations available for each device.

```
GET sensing/deviceConfigList
GET sensing/deviceConfigList?devNum=0
dev #0, CC1101, 4 configs:
  cfg #0: CC1101, f_c=868 MHz, BW=400 kHz:
    base: 868299866 Hz, spacing: 199951 Hz, bw: 406250 Hz,
    channels: 127, time: 20 ms
  cfg #1: CC1101, f_c=868 MHz, BW=800 kHz:
    base: 867999729 Hz, spacing: 199797 Hz, bw: 843681 Hz,
    channels: 63, time: 100 ms
  cfg #2: CC1101, f_c=906 MHz, BW=400 kHz:
    base: 905999991 Hz, spacing: 399595 Hz, bw: 421840 Hz,
    channels: 255, time: 20 ms
  cfg #3: CC1101, f_c=906 MHz, BW=800 kHz:
    base: 905999991 Hz, spacing: 399595 Hz, bw: 843681 Hz,
    channels: 255, time: 20 ms
```

Lists all configurations for a device. If no device number is given, then all configurations are listed for all devices.

Format of each entry is as follows:

```
dev #%u, %s
    (1) (2)
  cfg #%u: %s
      (3) (4)
     base: %u Hz, spacing: %u Hz, bw: %u Hz,
           (5)             (6)         (7)
     channels: %u, time: %u ms
               (8)       (9)
```

Fields (%u - unsigned integer number, represented in decimal notation, %s – free form string):

1. ID of the device, used when referring to it when programming a spectrum sensing task
2. Human-readable name of the device
3. ID of the device configuration pre-set, used when referring to it when programming a spectrum sensing task
4. Human-readable name of the configuration
5. Central frequency of channel 0
6. Difference in central frequency between two adjacent channels
7. Channel filter bandwidth (note: might be different from channel spacing, meaning that some configurations have gaps or overlaps between channels)

8. Total number of channels
9. Tuner settle time in milliseconds

Channel numbers M range from 0 to channel_num – 1. To calculate central frequency for a channel the following formula can be used:

$$f\_c = \text{channel\_base} + M * \text{channel\_spacing}$$

```
GET sensing/deviceStatus
GET sensing/deviceStatus?devNum=0
IC    : CC2500
Part num   : 80
Version    : 03
```

Get the status of all devices, or of one device. The format of the status message is hardware dependent.

```
GET sensing/activeProgram
dev 0 conf 3 ch 0:1:42
```

Get the currently active scanning programs. Returns nothing if no scanning program is active at the time. See sensing/program resource for the description of the format.

```
GET uptime
1345.234
```

Returns the uptime of the node (time since last reset) in seconds. This information is useful for programming tasks in advance.

Annex II. VESNA spectrum sensing setup

```
POST sensing/program
length=...
in 13 sec for 60 sec with dev 0 conf 3 ch 0:1:46 to slot 3
in 90 sec for 600 sec with dev 1 conf 4 ch 4:14:250 to slot 4
crc=...
```

POST data contains one spectrum sensing program per line, maximum 10 lines.

Format of each line is as follows:

```
in %u sec for %u sec with dev %u conf %u ch %u:%u:%u to slot %u
   (1)        (2)             (3)     (4)   (5)(6)(7)       (8)
```

Fields (%u - unsigned integer number, represented in decimal notation):

1. Number of seconds until the start of scanning; the start is relative to the moment of receiving the request
2. Length of the scan, in seconds
3. ID of device used for scanning
4. ID of configuration used for scanning
5. First channel in the sweep
6. Channel increment
7. Last channel in the sweep
8. Slot on SD card to which the data will be saved (slot must be empty)

Programs must be sorted by increasing start time and their sensing intervals must not overlap. Each program must have a unique SD card slot number. If any of the lines is invalid, the program is rejected.

```
GET sensing/program
in 13 sec for 60 sec with dev 0 conf 3 ch 0:1:46 to slot 3
in 90 sec for 600 sec with dev 1 conf 4 ch 4:14:250 to slot 4
```

Issuing a GET request to sensing/program resource returns the information in same format as used in the POST request.

The number of seconds until the start of scanning is re-calculated relative to the reception time of the GET request, hence the number of seconds until scan is constantly decreasing. When a program is active, the number of seconds until the start is 0 and the length of the scan is decreased accordingly.

```
GET sensing/lastSweepBin
GET sensing/lastSweepText
```

If a sensing activity is currently active, then these resources return the results of the last completed sweep, in binary and in text form, respectively. The results are RSSI values, expressed in dBm.

The results are stored until a new sensing activity is started.

In the case that a new sensing started and the first set of results is not ready, then these resources return empty result set.

sensing/lastSweepBin returns the results as a 16-bit signed integer data type, with values representing 1/100 dBm values per channel.

sensing/lastSweepText returns the same data formatted as ASCII text, separated with whitespaces.

In the output, a CRC-32 is returned with the data. The CRC is calculated on the binary representation of the data, and has 4 bytes. In binary format, the CRC is returned in MSB -> LSB order, while in text format, a hexadecimal number is appended at the end of the line

```
POST sensing/quickSweepBin
POST sensing/quickSweepText
length=...
dev 1 conf 4 ch 4:14:250
crc=...
```

Perform a quick sweep with a given device, configuration and channels. If performing the sweep would require more than 1.5 seconds to completion an error is returned instead of results.

The difference between the binary and text version is the format of the returned results:

sensing/quickSweepBin returns the results as a 16-bit signed integer data type, with values representing 1/100 dBm values per channel.

sensing/lastSweepText returns the same data formatted as ASCII text, separated with whitespaces.

In the output, a CRC-32 is returned with the data. The CRC is calculated on the binary representation of the data, and has 4 bytes. In binary format, the CRC is returned in MSB -> LSB order, while in text format, a hexadecimal number is appended at the end of the line

Annex III. Retrieving the spectrum sensing measurements from VESNA

```
GET sensing/slotInformation?id=0
size=0
version=0
status=EMPTY
crc=0
```

Retrieves meta-data about spectrum sensing measurements stored in a SD card slot.

```
size=%u (1)
version=%u (2)
status=(EMPTY | INCOMPLETE | COMPLETE | UNDEFINED) (3)
crc=%u (4)
```

Fields (%u - unsigned integer number, represented in decimal notation):

1. Size of stored data, in bytes
2. Uptime of the node in seconds at the time of writing data to the SD card.
3. Status of the slot: EMPTY – the slot does not contain any data and is ready to be used in a new measurement, INCOMPLETE – data is currently being written to the slot, COMPLETE – measurement results have been successfully written to the slot and are ready for retrieval, UNDEFINED – an unexpected value.
4. CRC of data in the slot.

```
GET sensing/slotDataHeader?id=0
{     "sensingStart"   : "1325970241",
      "sensingEnd"     : "1325970242",
      "deviceNumber"   : "0",
      "deviceName"     : "CC2500",
      "configNumber"   : "0",
      "configName"     : "CC2500, f_c=2400 MHz, BW=400 kHz, 10
samples averaged",
      "channelStart"   : "0",
      "channelStep"    : "1",
      "channelStop"    : "10" }
```

Retrieves configuration used for spectrum sensing measurements stored in a SD card slot. String is formatted as a JSON document.

Fields:

- **sensingStart**: internal clock at the start of measurement (in seconds)
- **sensingEnd**: internal clock at the end of measurement (in seconds)
- **deviceNumber**: ID of device used for scanning
- **deviceName**: Name of device used for scanning
- **configNumber**: ID of configuration used for scanning
- **configName**: Name of configuration used for scanning
- **channelStart**: First channel in the sweep
- **channelStep**: Channel increment

- **channelStop**: Last channel in the sweep

```
GET sensing/slotDataBinary?id=0&start=45&size=200
```

Reads block of data from a slot.

Parameters:

- **id**: SD card slot to read from
- **start**: Offset of the block, in bytes from the start of the data
- **size**: length of the block to read, in bytes

Result is binary data in the format described above. The node will send at most 512 bytes of data in one request. Errors are signaled with “error:”; in this case no data is returned.

In the output, CRC-32 is returned with the data. The CRC is returned as a 4-byte value in MSB -> LSB order appended at the end

Annex IV. VESNA Spectrum Sensing C API documentation

DATA STRUCTURE DOCUMENTATION

spectrum_dev Struct Reference

Description of a spectrum sensing device.

```
#include <spectrum.h>
```

Data Fields

const char * **name**

Name of the device.

struct **spectrum_dev_config** *const * **dev_config_list**

List of configuration pre-sets supported by this device.

int **dev_config_num**

Length of dev_config_list.

spectrum_dev_reset_t dev_reset

Function to reset the device.

spectrum_dev_setup_t dev_setup

Function to setup a spectrum sensing sweep.

spectrum_dev_run_t dev_run

Function to start a spectrum sensing sweep.

spectrum_dev_status_t dev_status

Function to query the status of the device.

const void * **priv**

Opaque pointer to a device-specific data structure.

spectrum_dev_config Struct Reference

Configuration pre-set for a spectrum sensing device.

```
#include <spectrum.h>
```

Data Fields

const char * **name**

Name of the pre-set.

unsigned int **channel_base_hz**

Center frequency of the first channel, in Hz.

int **channel_spacing_hz**

Difference between center frequencies of two adjacent channels, in Hz.

int **channel_bw_hz**

Bandwidth of a channel, in Hz.

int **channel_num**

Number of channels.

int **channel_time_ms**

Time required for detection per channel, in milliseconds.

const void * **priv**

Opaque pointer to a device-specific data structure.

Detailed Description

Configuration pre-set for a spectrum sensing device.

Minimum possible central frequency:

f_cmin = channel_base

Maximum possible central frequency:

f_cmax = channel_base + (channel_num - 1) * channel_spacing

Resolution bandwidth:

resolution_bw = channel_bw

Full band sweep time:

sweep_time = channel_time * channel_num

Field Documentation

int spectrum_dev_config::channel_time_ms

Time required for detection per channel, in milliseconds.

Approximate number - accurate timestamps are returned with measurement results.

spectrum_sweep_config Struct Reference

Description of a frequency sweep.

```
#include <spectrum.h>
```

Data Fields

struct **spectrum_dev_config * dev_config**

Device configuration pre-set to use.

int **channel_start**

Channel of the first measurement.

int **channel_step**

Increment in channel number between two measurements.

int **channel_stop**

Channel of the one after the last measurement.

spectrum_cb_t cb

Callback function.

FILE DOCUMENTATION

VESNALib/inc/spectrum.h File Reference

Common API for various spectrum sensing hardware.

```
#include <string.h>
#include <stdint.h>
```

Data Structures

struct **spectrum_sweep_config**

Description of a frequency sweep.

struct **spectrum_dev_config**

Configuration pre-set for a spectrum sensing device. s

truct **spectrum_dev**

Description of a spectrum sensing device.

Defines

#define **E_SPECTRUM_STOP_SWEEP** 1

#define **E_SPECTRUM_OK** 0

#define **E_SPECTRUM_INVALID** -1

#define **E_SPECTRUM_TOOMANY** -2

#define **SPECTRUM_MAX_DEV** 10

Maximum number of spectrum sensing devices supported.

Typedefs

typedef int(* **spectrum_cb_t**)(const struct **spectrum_sweep_config** *sweep_config, int timestamp, const int16_t data_list[])

*Callback function called by **spectrum_run()** for each completed sweep.*

typedef int(* **spectrum_dev_reset_t**)(const void *priv)

Function to reset the device.

typedef int(* **spectrum_dev_setup_t**)(const void *priv, const struct **spectrum_sweep_config** *sweep_config)

Function to setup a spectrum sensing sweep.

typedef int(* **spectrum_dev_run_t**)(const void *priv, const **struct spectrum_sweep_config** *sweep_config)

Function to start a spectrum sensing sweep.

typedef int(* **spectrum_dev_status_t**)(const void *priv, char *buffer, size_t len)

Function to query the status of the device.

Functions

int **spectrum_add_dev** (const **struct spectrum_dev** *dev)

Register a new spectrum sensing device to the system.

int **spectrum_reset** (void)

Reset all spectrum sensing devices.

int **spectrum_sweep_channel_num** (const struct **spectrum_sweep_config** *sweep_config)

Return number of channels for a sweep config.

int **spectrum_run** (const struct **spectrum_dev** *dev, **struct spectrum_sweep_config** *sweep_config)

Start a spectrum sensing sweep on a device.

int **spectrum_status** (const **struct spectrum_dev** *dev, char *buffer, size_t len)

Variables

int **spectrum_dev_num**

Current number of registered spectrum sensing devices.

struct **spectrum_dev** * **spectrum_dev_list** []

Array of registered spectrum sensing devices.

TYPEDEF DOCUMENTATION

typedef int(* spectrum_cb_t)(const struct spectrum_sweep_config *sweep_config, int timestamp, const int16_t data_list[])

Callback function called by **spectrum_run**() for each completed sweep.

Interpretation of the data array:

data[n] = measurement for channel m, where

m = channel_start + channel_step * n

n = 0 .. channel_num - 1

Values: 1/100 dBm value, for ex. 100 = 1 dBm, 50 = 0.5 dBm

Parameters:

sweep_config Pointer to the sweep_config struct passed to **spectrum_run**()

timestamp Timestamp of the measurement in ms since **spectrum_run**() call

data_list Array of measurements

Returns:

E_SPECTRUM_OK to continue sweep, E_SPECTRUM_STOP_SWEEP to stop sweep and return from spectrum_run or any other value on error.

typedef int(* spectrum_dev_reset_t)(const void *priv)

Function to reset the device.

Parameters:

priv Pointer to a device-specific data structure

Returns:

E_SPECTRUM_OK on success

typedef int(* spectrum_dev_run_t)(const void *priv, const struct spectrum_sweep_config *sweep_config)

Function to start a spectrum sensing sweep.

Parameters:

priv Pointer to a device-specific data structure *sweep_config* Pointer to the frequency sweep description

Returns:

E_SPECTRUM_OK on success

typedef int(* spectrum_dev_setup_t)(const void *priv, const struct spectrum_sweep_config *sweep_config)

Function to setup a spectrum sensing sweep.

Parameters:

priv Pointer to a device-specific data structure

sweep_config Pointer to the frequency sweep description

Returns:

E_SPECTRUM_OK on success

typedef int(* spectrum_dev_status_t)(const void *priv, char *buffer, size_t len)

Function to query the status of the device.

This function fills a caller-allocated string buffer with status information about the device. No format for the string is specified.

Parameters:

priv Pointer to a device-specific data structure

buffer Pointer to a caller-allocated string.

len Size of the buffer in bytes (at most len bytes will be written to buffer, including the terminating 0)

Returns:

E_SPECTRUM_OK on success

FUNCTION DOCUMENTATION

int spectrum_add_dev (const struct spectrum_dev * dev)

Register a new spectrum sensing device to the system.

Parameters:

dev Pointer to the device structure to add

Returns:

E_SPECTRUM_OK on success

int spectrum_reset (void)

Reset all spectrum sensing devices.

Returns:

E_SPECTRUM_OK on success

int spectrum_run (const struct spectrum_dev * dev, struct spectrum_sweep_config * sweep_config)

Start a spectrum sensing sweep on a device.

Parameters:

dev Pointer to the device structure to be used for sweep

sweep_config Pointer to the frequency sweep description

Returns:

E_SPECTRUM_OK on success

int spectrum_status (const struct spectrum_dev * dev, char * buffer, size_t len)

Return status of the device

This function fills a caller-allocated string buffer with status information about the device. No format for the string is specified.

Parameters:

dev Pointer to the device structure for which to query status.

buffer Pointer to a caller-allocated string.

len Size of the buffer in bytes (at most len bytes will be written to buffer, including the terminating 0)

Returns:

E_SPECTRUM_OK on success

int spectrum_sweep_channel_num (const struct spectrum_sweep_config * sweep_config)

Return number of channels for a sweep config.

Parameters:

sweep_config Pointer to the frequency sweep description

Returns:

Number of channels in the sweep config

Annex V. VESNA Signal Generation C API documentation

DATA STRUCTURE DOCUMENTATION

vsnSignalGenerator_device Struct Reference

Description of a signal generation device.

`#include <vsnsignalgenerator.h>`

Data Fields

const char * **name**

Name of the device.

struct **vsnSignalGenerator_deviceConfig** *const * **configList**

List of configuration pre-sets supported by this device.

int **configNum**

Length of configList.

vsnSignalGenerator_resetFunc reset

Function to reset the device.

vsnSignalGenerator_setupFunc setup

Function to setup the transmission.

vsnSignalGenerator_startFunc start

Function to start the transmission.

vsnSignalGenerator_stopFunc stop

Function to stop the transmission.

const void * **priv**

Opaque pointer to an implementation-specific data structure.

vsnSignalGenerator_deviceConfig Struct Reference

Configuration pre-set for a signal generation device.

`#include <vsnsignalgenerator.h>`

Data Fields

const char * **name**

Name of the pre-set.

unsigned int **channelBase**

Center frequency of the first channel, in Hz.

unsigned int **channelSpacing**

Difference between center frequencies of two adjacent channels, in Hz.

unsigned int **channelBW**

Bandwidth of a channel, in Hz.

unsigned int **channelNum**

Number of channels.

int **txPowerMax**

Maximum transmission power, in dBm.

int **txPowerMin**

Minimum transmission power, in dBm.

unsigned int **channelSettleTime**

Time required for channel selection, in ms.

const void * **priv**

Opaque pointer to an implementation-specific data structure.

Detailed Description

Configuration pre-set for a signal generation device.

Minimum possible central frequency:

f_cmin = channelBase

Maximum possible central frequency:

f_cmax = channelBase + (channelNum - 1) * channelSpacing

vsnSignalGenerator_txConfig Struct Reference

Description of a signal transmission.

```
#include <vsnsignalgenerator.h>
```

Data Fields

struct **vsnSignalGenerator_deviceConfig * deviceConfig**

Device configuration pre-set to use.

unsigned int **channel**

Channel number for transmission.

int **power**

Transmit power, in dBm.

FILE DOCUMENTATION

VESNALib/inc/vsnsignalgenerator.h File Reference

Common API for various signal generation hardware.

Data Structures

struct **vsnSignalGenerator_deviceConfig**

Configuration pre-set for a signal generation device.

struct **vsnSignalGenerator_txConfig**

Description of a signal transmission.

struct **vsnSignalGenerator_device**

Description of a signal generation device.

Defines

#define **VSNSIGNALGENERATOR_OK** 0

#define **VSNSIGNALGENERATOR_ERROR** -1

#define **VSNSIGNALGENERATOR_MAX_DEV** 10

Maximum number of signal generation devices supported.

Typedefs

typedef int(* **vsnSignalGenerator_resetFunc**)(const void *priv)

Function to reset the device.

typedef int(* **vsnSignalGenerator_setupFunc**)(const void *priv, const struct **vsnSignalGenerator_txConfig** *txConfig)

Function to setup the transmission.

typedef int(* **vsnSignalGenerator_startFunc**)(const void *priv)

Function to start the transmission.

typedef int(* **vsnSignalGenerator_stopFunc**)(const void *priv)

Function to stop the transmission.

Functions

int **vsnSignalGenerator_addDevice** (const **struct vsnSignalGenerator_device** *device)

Register a new signal generation device to the system.

int **vsnSignalGenerator_reset** (void)

Reset all signal generation devices.

int **vsnSignalGenerator_start** (const struct **vsnSignalGenerator_device** *device, const struct **vsnSignalGenerator_txConfig** *txConfig)

Start transmission on a device.

int **vsnSignalGenerator_stop** (const struct **vsnSignalGenerator_device** *device)

Stop transmission on a device.

Variables

int **vsnSignalGenerator_deviceNum**

Current number of registered signal generation devices.

struct **vsnSignalGenerator_device * vsnSignalGenerator_deviceList []**

Array of registered signal generation devices.

TYPDEF DOCUMENTATION

typedef int(vsnSignalGenerator_resetFunc)(const void *priv)*

Function to reset the device.

Parameters:

priv Pointer to a device-specific data structure

Returns:

VSNSIGNALGENERATOR_OK on success

typedef int(* vsnSignalGenerator_setupFunc)(const void *priv, const struct vsnSignalGenerator_txConfig *txConfig)

Function to setup the transmission.

Parameters:

priv Pointer to a device-specific data structure

txConfig Pointer to the transmission description

Returns:

VSNSIGNALGENERATOR_OK on success

typedef int(* vsnSignalGenerator_startFunc)(const void *priv)

Function to start the transmission.

Parameters:

priv Pointer to a device-specific data structure

Returns:

VSNSIGNALGENERATOR_OK on success

typedef int(* vsnSignalGenerator_stopFunc)(const void *priv)

Function to stop the transmission.

Parameters:

priv Pointer to a device-specific data structure

Returns:

VSNSIGNALGENERATOR_OK on success

FUNCTION DOCUMENTATION

int vsnSignalGenerator_addDevice (const struct vsnSignalGenerator_device * device)

Register a new signal generation device to the system.

Parameters:

device Pointer to the device structure to add

Returns:

VSNSIGNALGENERATOR_OK on success

int vsnSignalGenerator_reset (void)

Reset all signal generation devices.

Returns:

VSNSIGNALGENERATOR_OK on success

int vsnSignalGenerator_start (const struct vsnSignalGenerator_device * device, const struct vsnSignalGenerator_txConfig * txConfig)

Start transmission on a device.

Parameters:

device Pointer to the device structure to use

txConfig Pointer to the transmission description

Returns:

VSNSIGNALGENERATOR_OK on success

int vsnSignalGenerator_stop (const struct vsnSignalGenerator_device * device)

Stop transmission on a device.

Parameters:

device Pointer to the device structure to use

Returns:

VSNSIGNALGENERATOR_OK on success