Wireless Software and Hardware platforms for Flexible and Unified radio and network controL

Project Deliverable D3.4

Second operational radio control software platform

Contractual date of delivery:	31-12-2016
Actual date of delivery:	23-12-2016
Beneficiaries:	IMEC, TCD, CNIT, TUB
Lead beneficiary:	CNIT
Authors:	Domenico Garlisi (CNIT), Daniele Croce (CNIT), Fabrizio Giuliano(CNIT), Ilenia Tinnirello (CNIT), Pierluigi Gallo (CNIT), Bartholomeu Liberato (TCD), Diarmuid Collins (TCD), Francisco Paisana (TCD), Peter Ruckebusch (IMINDS), Spilios Giannoulis (IMEC), Bart Jooris (IMEC), Jan Bauwens (IMEC), Anatolij Zubow (TUB), Piotr Gawłowicz (TUB)
Reviewers:	Ingrid Moerman (IMEC), Mikołaj Chwalisz (TUB)
Work package:	WP3 – Radio Control
Estimated person months:	24
Nature:	R
Dissemination level:	PU
Version:	5.1

Abstract:

This deliverable gives a detailed description of the capabilities and performance of the second operational radio control software platform that supports both the Path 1 (black box) and Path 2 (white box) innovation strategy.

Keywords:

Programmable radio architecture; radio capabilities; lower MAC/PHY adaptations, implementation.

Executive Summary

This deliverable reports the second operational radio control software platform and provides details about the implementation of the unified programming interface for radio control, named UPI_R, that is offered to experimenters at the end of Y2.

Radio control is meant for configuring the lower MAC and PHY layers of the wireless nodes, also called radio platform, i.e. the physical parameters characterizing the transceiver operations, such as the central frequency and the transmission format, as well as the logic for reacting to physical events and accessing the wireless channel. Usually, the configuration of these aspects requires a deep understanding of the hardware and software architecture of wireless nodes. Thanks to WiSHFUL, it is possible to **abstract the internals of the nodes with a unified configuration interface** able to work on completely different hardware and software architectures.

Indeed, the experimenters can use the same UPI_R functions for working on the IRIS architecture for SDR platforms, the Time Annotated Instruction Set Computer (TAISC) architecture for sensor nodes [1], the Wireless MAC Processor (WMP) architecture for WiFi interfaces [2], the Atheros driver and chipset architecture for off-the-shelf WiFi interfaces [3]. The technology-specific and platform-specific details can be completely hidden by the UPI_R abstractions, in case experimenters want to follow **a black-box approach** for testing a wireless solution.

During Y2 activities, the definition of the UPI_R interface has been extended in two main directions: i) **at the platform level**, by integrating new hardware and software architectures in the WiSHFUL framework, by means of adaptation modules supporting UPI_R abstractions, and by improving the capabilities of the platforms already available at the end of Y1; b) **at the functional level**, by introducing novel functionalities dealing with specific **technology-dependent configurations** of MAC/PHY layers or platform-dependent definitions of new radio programs. As far as concerns the platform extensions, we integrated a new implementation of the TAISC architecture and WMP architecture on SDR platforms, extended the UPI_R functions supported by the IRIS architecture, and added the GNU radio architecture and an architecture for programmable antennas as new platforms. Regarding the functional extensions, we added configuration functions which do not follow generic MAC/PHY models, but work on technology-specific protocols, such as WiFi or LTE standardized protocols. Moreover, we can now support more advanced experiments based on a **white-box approach**, according to which we give access to platform specific functionalities, such as compiling new custom radio programs. To this purpose, we developed two programming tools for facilitating the editing of radio programs on the WMP and TAISC architecture.

This approach obviously requires a more advanced understanding from experimenters and, in turn, offers full flexibility to the design and test of wireless solutions. Further, the refinements and extensions of the UPI_R interface implemented during Y2 have been driven by the requirements emerged during the realization of the showcases and by extensions supported by Open Call 1.

List of Acronyms and Abbreviations

AP	Access Point
BLER	Block Error Rate
BS	Base Station
CCA	Clear Channel Assessment
CQi	Channel Quality Indicator
CSMA	Carrier Sense Multiple Access
CTS	Clear to Send
DCF	Distributed Coordinator Function
DL	Download Link
E-UTRAN	Evolved Universal Terrestrial Access Network
EARFCN	Radio Frequency Channel Number
eNB	Evolved Node B
EPC	Evolved Packet Core
EPS	Evolved Packet System
GCP	Global Control Program
GITAR	Generic extension for Internet-of-Things Architectures
ISC	Instruction Set Computer
LAN	Local Area Network
LCS	Local Control Service
LTE	Long Term Evolution
MCE	Monitor and Configuration Engine
MCS	Modulation and Coding Scheme
MS	Mobile Station
P-GW	Packet Data Network Gateway
PBCH	Physical Broadcast Channel
PDN	Packet Data Network
PDSCH	Physical Downlink Shared Channel
PLMNID	Public Land Mobile Network ID
PSCH	Physical Shared Channel
PUCCH	Physical Uplink Control Channel
PUSCH	Physical Uplink Shared Channel
RACH	Random Access Channel
RTS	Request to Send
SDR	Software Defined Radio

SINR	Signal to Interference plus Noise Ratio
SSCH	Signal Synchronization Channel
STA	Wireless Station
TAC	Track Area Code
TAISC	Time Annotated Instruction Set Computer
TDMA	Time Division Multiple Access
UCS	Unified Clock System
UE	User Equipment
UL	Upload Link
UPI	Unified Programming Interface
UPI_G	Unified Programming Interface Global
UPI_HC	Unified Programming Interface Hierarchical Control
UPI_M	Unified Programming Interface Management
UPI_N	Unified Programming Interface Network
UPI_R	Unified Programming Interface Radio
USCI	Universal Serial Communications Interface
VM	Virtual Machine
WLAN	Wireless LAN
WMP	Wireless MAC Processor
XFSM	Extended Finite State Machines

Table of contents

1 Introduction

The WiSHFUL architecture is devised to provide i) **unified interfaces** to experimenters for easily prototyping novel and adaptable wireless solutions on different radio platforms, ii) a **control framework** for supporting dynamic on-the-fly reconfigurations of the wireless nodes according to time-varying estimates of the network operating conditions. For wireless node we mean a complete hardware and software architecture implementing: i) the MAC/PHY layers of the protocol stack, also called **radio platform**, based in turns on a hardware and software architecture supporting a **wireless technology** (such as standardized IEEE 802.11 and IEEE 802.15.4 technologies or non-standard technologies), ii) the upper layer of the protocol stack based on an operating system (such as a Linux operating system for wireless local area networks, or a Contiki operating system for sensor nodes); iii) additional hardware platforms, such as a system with programmable antennas, with the relevant drivers. Note that a given wireless technology can be supported by different radio platforms. i.e. by different hardware and drivers. For example, WiSHFUL includes IEEE 802.11 nodes based on commercial interfaces (namely, the Atheros cards), commercial interfaces with customized non-standard firmware (namely, the Wireless MAC Processor), and software defined radios. Programmable radio platforms are able to execute different MAC/PHY protocol stacks which are coded in **radio programs** that can be dynamically loaded into the platforms.

This document is focused on the presentation of the UPI_R interface, which is one important component of the WiSHFUL unified interfaces devised to configure the node behaviour at the lower MAC and PHY layers. The definition of the UPI_R interface has been carried out by considering two different utilization paths of wireless nodes, as originally discussed in the proposal:

- *Path 1 (black-box approach):* offers limited flexibility but maximal ease of use by completely hiding the platform-specific details of the wireless nodes used by experimenters. This implies that WiSHFUL nodes are offered with a pre-defined set of high-level configuration capabilities and radio programs (implementing different protocols and transceivers) that can be selected by experimenters. Moreover, technology-specific functions are offered with a unified interface working on heterogeneous platforms. Examples of technology-specific functionalities include the configuration of operation modes and parameters that depend on a given standard, such as the configuration of the RTS threshold for a node based on the IEEE 802.11 technology.
- *Path 2 (white-box approach):* offers full flexibility, and hence requires more expert knowledge because it allows to access **platform-specific** functionalities. Examples of platform-specific functionalities are compiling a new radio program for nodes based on the Wireless MAC Processor or Time Annotated Instruction Set Computer (TAISC) architecture, or configuring a new waveform for nodes based on software defined radio.

UPI_R interface has been designed by following an iterative approach: the black-box functionalities have been implemented during Y1 and refined during Y2 on the basis of the requirements emerged during the realization of the showcases; the white-box functionalities have been fully implemented during Y2. Rather than presenting a differential description of the new UPI_R functions developed during Y2, we decided to prepare this deliverable as an inclusive document, in which the current state of the UPI_R interface is described as a whole with the complete list of supported functions. Details about the implementation of UPI_R interface are provided in this document for new functions and radio platforms, and in D3.2 for functions implemented in Y1.

The rest of the document is organized as follows. First, we present the radio platforms currently included in the WiSHFUL framework, by detailing the device class and the hardware and software architecture of each one. We then present the abstraction of these programmable platforms in a common programming model. Second, we describe the UPI_R interface, by differentiating the general, technology-specific and white-box functionalities. Third, we provide some examples of UPI_R utilization. Finally, we draw our conclusions about the Y2 design phase.

2 General description of Y2 release of the WiSHFUL radio control architecture

In this section, we provide a high-level description of the WiSHFUL framework for controlling programmable wireless nodes by means of unified interfaces. The framework allows orchestrating the utilization of both the UPI_R and UPI_N interfaces at a global and local level, thus supporting dynamic adaptations of the wireless nodes according to the aggregation of radio parameters monitored by different nodes and estimates of the network state.

Figure 1 shows how the WiSHFUL architecture supports a ***two-tier control hierarchy*** and interacts with wireless nodes based on heterogeneous radio platforms (namely, IRIS, TAISC and WMP): one global Monitoring and Configuration Engine (MCE) orchestrates several remote MCEs residing on each wireless node of the testbed. The ***global MCE*** provides monitor and configuration services that can be used by the experimenter to write a *Global Control Program (GCP),* controlling the behaviour of the solution under test by means of the **UPI_G** interface. On the other hand, *local control programs* running on ***local MCEs*** control single devices by means of the **UPI_R** and **UPI_N** interfaces, respectively for radio and network control. The same UPI_R and UPI_N functions are exposed on the heterogeneous platforms by means of adaptation modules. This unified approach unloads the experiment from the burden to deal with a multiplicity of configuration and utility tools (e.g. *iw, iwconfig, iptables, iwlist, iperf, b43fwdump, etc*).

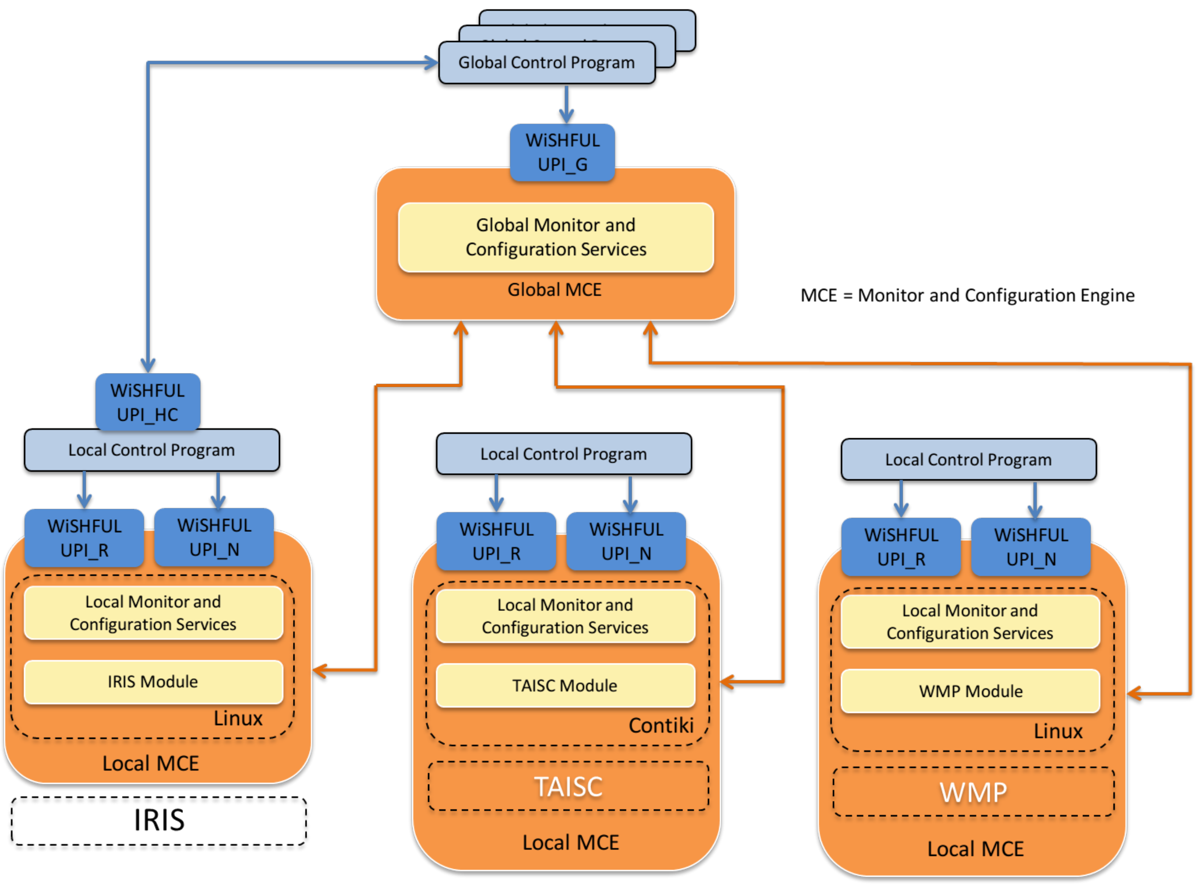


Figure 1 - WiSHFUL architecture, UPIs and supported platforms

While the detailed description of the WiSHFULframework is provided in the companion deliverable D4.4, in this document we describe the general concepts and implications for the control of the lower layers of the wireless nodes, which include the **hardware systems integrated into the nodes** (transceivers, antennas, sensors, etc.) and the relevant **software architectures devised to drive the hardware systems**.

We refer to a **hardware system and relevant software modules** exposing a configuration UPI interface and abstract programming model as **platform.** This definition generalizes the concept of radio platform to any hardware system, which does not necessarily include a radio transceiver (such as intelligent antennas or measurement sensors) and can be added to wireless nodes for providing new capabilities. **According to this vision, a wireless node can be equipped with multiple platforms, including at least one radio platform providing communication capabilities**; all the platforms are orchestrated as a whole by the WiSHFUL control programs running on the wireless node.

For clarifying this concept, in Figure 2 we consider an exemplary wireless node (e.g. a multi-technology gateway), which integrates heterogeneous hardware technologies, such as ZigBee, WiFi and a configurable antenna system, and the relevant software architectures. Thanks to the adaptation modules available for each platform, the node exposes to WiSHFUL the aggregated capabilities in a list of available UPI functions. The functions abstract the specific node architecture and just provide experimenter the possibility to communicate with ZigBee and WiFi nodes and to steer the antenna beam in a desired direction. The experimenter exploits the complete list of supported functions for writing the desired control program.

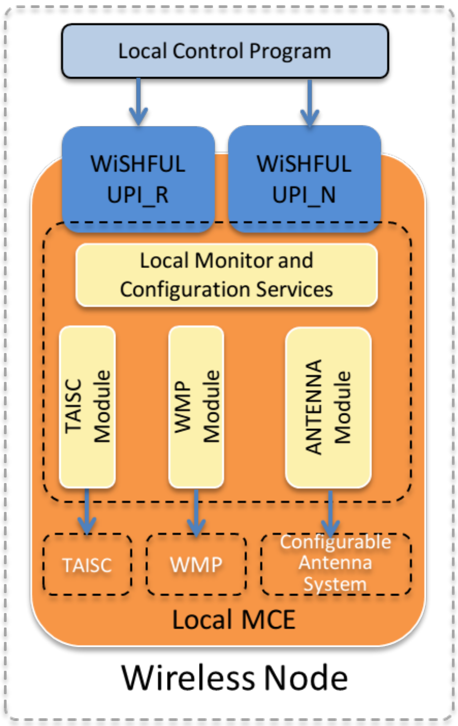


Figure 2 - Example of wireless node supporting three different platforms.

2.1 Supported Platforms

The WiSHFUL framework allows the control of heterogeneous classes of devices (micro-controller devices, general-purposes devices and software defined radio) and radio technologies by means of unified interfaces and control models available for some reference platforms. In particular, the initial set of reference platforms (supported at the end of Y1) are: i) the **Wireless MAC Processor (WMP) architecture**, that has been conceived for programmable wireless nodes in local area networks and has been implemented on top of a legacy IEEE 802.11 card by Broadcom; ii) the **Time Annotated Instruction Set Computer (TAISC)** architecture, that has been conceived for sensor nodes and developed on top of RM090 [4] sensor nodes; iii) the **IRIS architecture**, that has been conceived for wireless nodes exploiting SDR capabilities and has been developed on top of host PCs connected to Universal Software Radio Peripherals (USRPs); iv) the **Atheros platform**, that extends the WiSHFUL UPI to commercial off-the-shelf IEEE 802.11 Atheros cards.

During Y2, apart from the refinements of previous interfaces and adaptation modules, we worked for supporting the WiSHFUL UPI and control models on two additional platforms: i) the **GNU radio** architecture, for providing an alternative platform supporting software-defined radio capabilities; ii) the **Reconfigurable Antenna Systems (RAS)**, which is a platform of programmable antennas, able to configure the radiation pattern in the azimuth plan. This last platform has been classified as a non-radio platform, because it does not provide radio communication capabilities. Moreover, the TAISC architecture and the WMP architecture have been extended for working on SDR, while the IRIS adaptation module has been extended for supporting a widest set of UPI_R functions.

Note that a given wireless technology can be supported by different radio platforms. i.e. by different hardware and drivers. For example, the WiSHFUL UPI are available for IEEE 802.11 nodes based on commercial interfaces (namely, the Atheros cards), commercial interfaces with customized non-standard firmware (namely, the Wireless MAC Processor), and software defined radios. While some UPI_R functionalities are technology-agnostic, some others refer to specific technologies and therefore it is important to know which technologies are supported by a given platform for accessing these functionalities. Consequently, we categorized the radio platforms as WiFi, LTE, and Lowpan (IEEE 802.15.4) platforms, according to the wireless technologies that can be supported. Figure 3 shows a graphical representation of UPI_R functions grouped according to the technology they refer to: the common intersection represents the technology-agnostic functionalities.

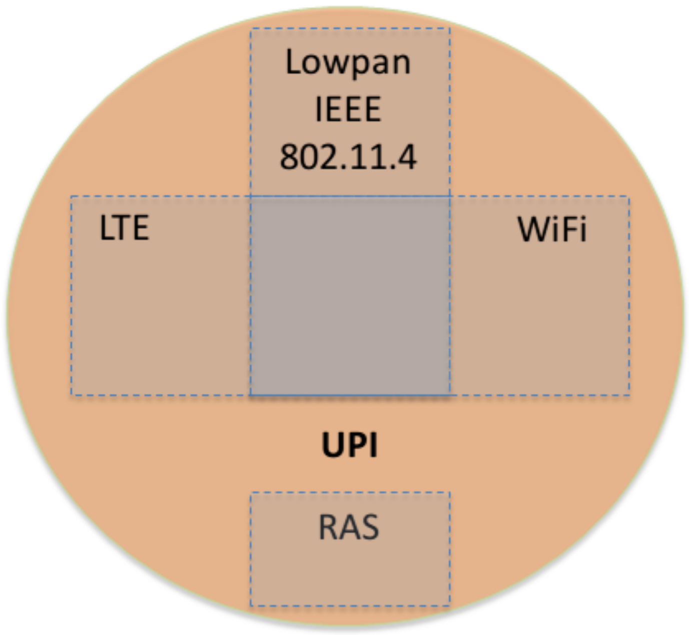


Figure 3 – Graphical representation of UPI_R functionalities supported by different platforms.

WiSHFUL also abstracts the radio platform programming model, in terms of generic execution engine and radio programs. According to this model, each radio platform offers the possibility to load several MAC/PHY programs, already available for experimenters in the WiSHFUL repository, or to define novel wireless protocols and radio behaviors by means of high-level programming languages.

Table 1provides a summary of the platforms that are currently supported in WiSHFUL at the end of Y2 activities. We will also present the complete list of available UPIs in Section 3, whose implementation details were provided in D3.2 for the platforms already available in Y1, and in this deliverable for the new platforms.

Module name	Description
WMP	WMP follows a programming model that decouples the Medium Access Control protocol logic (described in an abstract form via eXtended Finite State Machines – XFSM) from the wireless device design, implementing the radio primitives as well as an XFSM execution engine called "Wireless MAC processor" [2]. The core of the architecture is an execution Engine capable of running programs defined as eXtended Finite State Machines (XFSMs). The WMP is implemented on a **Broadcom AirForce54G wireless card** and (partially) on a SDR platform (namely, the WARP board). The original platform based on AirForce54G chipset (supporting IEEE 802.11b/g standard) is fully described in Deliverable 2.1, Section 2.3 and the UPI_R implementation for WMP is fully described in deliverable D2.2 section 3.2.
TAISC	TAISC (Time-Annotated Instuction Set Computer) consist of a cross-platform MAC protocol compiler and execution engine [1]. The cross-compilation approach allows developers to design MAC protocols once, and then compile them for reuse on different radio platforms. This approach has been successfully implemented for IEEE 802.15.4 MAC protocols on embedded wireless nodes (**RM090 and Zolertia RE-Mote**) and on a Xilinx Zynq-based SDR platform. This platform is fully described in deliverable D2.1 Section 2.2 and the UPI_R implementation for TAISC is fully described in deliverable D2.2 Section 3.3.
IRIS	IRIS is a software defined radio framework that allows users to design and construct radios from the composition of user defined signal processing blocks. The processing blocks of IRIS are written in C++ and run on the general purpose processor of a computer with a Linux based operating system. This computer is then interfaced to a universal software radio peripheral (**USRP**) frontend device, which handles the radio frequency aspects of the radio, which are limited to basic up or down conversion and minor filtering in the typical case. The Y1 implementation of this platform is fully described in deliverable D2.1 section 2.1 and the UPI_R implementation for IRIS is fully described in deliverable D2.2 section 3.4.
Atheros platform	Atheros-based IEEE 802.11 platform is a Commercial off-the-shelf IEE 802.11 compliant chip on a Linux platform. Following the Software-defined networking (SDN) paradigm we separate the control plane from the data plane and provide an API to allow local or global control programs to configure the channel access function. In particular we allow configuring the airtime sharing protocol access like define the number and size of time slots in which the transmission is enabled. Moreover, for each time slot a medium access policy can be assigned which allows restricting the medium access for particular stations (identified by their MAC address) and traffic identification (e.g. VoIP or video). The latter can

	be used to program flow-level medium access. The UPI_R implementation for Atheros platform is fully described in deliverable D2.2 section 3.5. More details can be found in [3].
GNU radio	GNU Radio is a free software development toolkit that provides signal processing blocks to implement software-defined radios and signal-processing systems. It can be used with external compatible RF hardware in order to deploy SDR transceiver, moreover GNU Radio allows to deploy innovative solutions in simulation-like environment. The UPI functions implementation for this platform is described in the section 3.3.1 of this deliverable.
RAS antenna	The Reconfigurable Antenna Systems (RAS) has been developed in the Open Call 1 extension of the WiSHFUL project. The antenna is capable of steering the radiation pattern dynamically on demand from typical omnidirectional to directional shape in the azimuth plane. RAS antenna is fully supported from WiSHFUL that provides UPI function to set the antenna direction.

Table 1 - WiSHFUL supported platforms

2.2 Adaptation Modules

In the initial WiSHFUL architecture, we started from the assumption that each wireless node would have been built on top of a single radio platform. For this reason, adaptation modules (also called connector modules) were designed for mapping the generalized UPI interface into platform-specific function calls, thus hiding the implementation details of each platform to experimenters.

This view has been generalized during Y2 activities, by considering that wireless nodes can integrate heterogeneous platforms: i) **exposing different hardware capabilities and software functions,** and ii) **supporting standard and/or non-standard radio technologies**. To cope with this generalized view, we revised the WiSHFUL architecture by allowing the definition of **multiple adaptation modules** in the same wireless node, thus decoupling the wireless node capabilities from a specific radio platform. The UPI interface exposed by a wireless node is given by the collection of functions supported by the adaptation modules, which have been installed for driving the available platforms. Moreover, the concept of adaptation modules have been further generalized for addressing the purely software architectures **implementing the higher layers of the protocol stack,** such as the operating systems or the traffic source generators. In other words, adaptation modules provide a set of UPI functions available in a given wireless node because of the installed platforms, operating system and software tools. The complete list of loaded adaptation modules and capabilities for each node are reported to the control program by the monitoring and configuration engine (MCE). Indeed, only the UPI functions presented in the loaded modules can be called by the control program. All the local MCEs and adaptation modules are implemented in Python except for Contiki sensor nodes where, in addition to the Python implementation residing on a host linux PC, also a native C software module exists that is used as an interface to the GITAR (Generic extension for Internet-of-Things Architectures) reconfiguration services on the node [5]. GITAR middleware offers a generic solution to integrate a vertical control plane within the protocol stack of constrained sensor devices and it is detailed in deliverable D4.4.

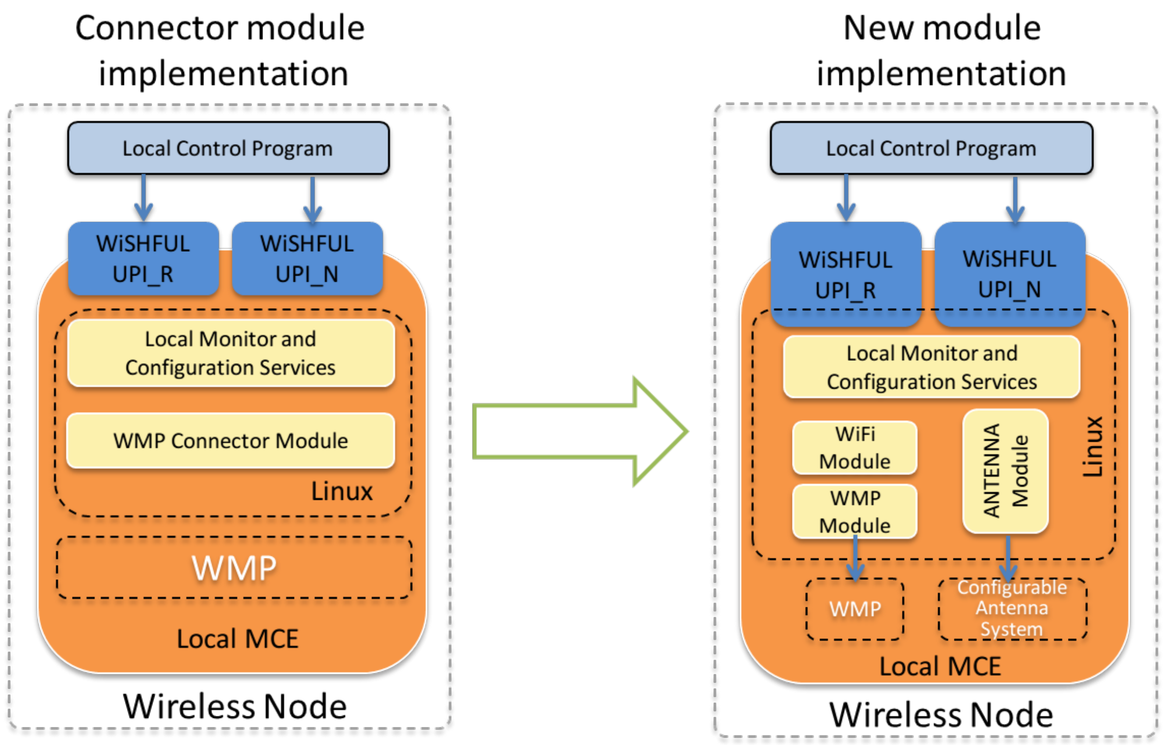


Figure 4 – Connector modules versus new WiSHFUL adaptation modules

Figure 4 shows an example of wireless node, with multiple loaded adaptation modules (on the right) and compare it with the Y1 architecture where a single connector module was responsible of hiding platform-specific details. In the new architecture, modules refer both to hardware radio platforms (*devicemodules*, such as the WMP Module) and to protocols (*protocol modules*, such as WiFi).

The configuration of the wireless node is given by the specification of the device and protocol modules to be loaded, which are addressed in yaml format. As an example, we provide in Table 2 a node configuration file with two device modules, *wishful_module_wifi_wmp* and *wishful_module_ras_antenna*, and one protocol module *wishful_module_iperf*. The MCE framework implementation is named node agent and the relative configuration file is named node agent config. Consequently, the MCE running on the node exposes only the UPI functions provided by these three modules.

```
## WiSHFUL node agent config file

agent_info:
  name: 'ras_antenna_test'
  info: 'node for testing RAS antenna system'
  iface: 'eth0'

modules:
  wlan:
      module : wishful_module_wifi_wmp
      class_name : WmpModule

  ras:
      module : wishful_module_ras_antenna
      class_name : RasAntennaModule
  iperf:
      module : wishful_module_iperf
      class_name : IperfModule
```

Table 2 – Module configurations in a WiSHFUL wireless node

The following Table 3 reports the complete list of platform and protocol modules provided by WiSHFUL, with a brief description of the supported functionalities. The whole documentation of the adaptation modules is included in the WiSHFUL code repository.

Module name	Description
module_wmp	This is the implementation of the WMP adaptation module, providing the Unified Programming Interfaces (UPIs) for WMP radio platform control. https://github.com/wishful-project/module_wifi_wmp
module_taisc	This is the implementation of the TAISC adaptation module, providing the Unified Programming Interfaces (UPIs) for the TAISC radio platform control. https://github.com/wishful-project/module_contiki
module_iris	This is the implementation of the IRIS adaptation module, providing the Unified Programming Interfaces (UPIs) for the IRIS radio platform control. https://github.com/wishful-project/module_iris
module_ath	This is the implementation of the Atheros adaptation module, providing the Unified Programming Interfaces (UPIs) for the Atheros radio platform control. https://github.com/wishful-project/module_wifi_ath
module_gnuradio	This is the implementation of the GNU radio adaptation module, providing the Unified Programming Interfaces (UPIs) for the GNU radio platform control. https://github.com/wishful-project/module_gnuradio
module_ras_antenna	This is the implementation of the RAS antenna adaptation module, providing the Unified Programming Interfaces (UPIs) for the RAS antenna platform control. https://github.com/wishful-project/module_ras_antenna
module_wifi	This is the implementation of the WiFi adaptation module providing the Unified Programming Interfaces (UPIs) for controlling WiFi technology. . https://github.com/wishful-project/module_wifi
module_lte	This is the implementation of the LTE adaptation module providing the Unified Programming Interfaces (UPIs) for the controlling LTE technology.. https://github.com/wishful-project/module_lte

Table 3 – List of adaptation modules

3 General UPI_R functions

The UPI_R interface is responsible of configuring and monitoring the platforms available in the wireless nodes according to an abstraction model of the platform capabilities and behaviors. Radio platforms are abstracted into a list of **parameters** that can be tuned, **measurements** that can be collected from the hardware, and **radio programs** that can define the logic for driving time critical operations. Examples of UPI_R functions for radio platforms include spectrum allocations, transceiver configurations, link set-up, statistic collections, definition of medium access logic, and virtualization. The main functions provided by this interface are organized into three groups dealing with three main goals: **configuring** the platform, at both the hardware and radio program levels, **monitoring** the node and network conditions by accessing all the signals and internal state of the platforms, **adapting** on-the-fly the node behavior by loading and activating context-specific radio programs. By means of the UPI_R interface, the WiSHFUL framework implements network-wide aggregation of radio parameters for estimating the overall network context and can enforce intelligent adaptation decisions in the network.

UPI_R functions are **unified across heterogeneous radio platforms** because the network controllers can access these functions in the same way and with the same parameters **regardless of the specific device class of the wireless nodes**. The unified functions can be further divided into **general functions**, which do not depend on the specific wireless technology supported by the nodes, and **technology-specific functions**, which deal with specific MAC/PHY protocols but are agnostic of their implementation. In this section, we describe the abstractions used for the definition of UPI_R and the complete list of general (technology-independent) functions developed during Y1 and Y2.

3.1 WiSHFUL Radio capability

The UPI_R interface is able to monitor and configure the radio behavior thanks to the abstraction of the hardware platform and radio programs in terms of **Radio Capabilities**. In D3.1, we defined two different types of radio capabilities: **Parameters** (in reading/writing mode), and low-level **Measurements** (in reading mode only). The parameters specify the parametric configuration interface exposed by the hardware and the global variables of the radio program loaded into the platform. Examples of global variables of radio programs are the contention windows for a CSMA radio program or the frame size and slot allocation for a TDMA radio program. The low-level measurements report the internal state of the hardware or the variable state of the radio program loaded into the platform. Examples of measurements are the signal strength of the last received packet or the number of experienced retransmissions. Summarizing, the parameters are configurable variables that influence how the platform or the radio program work, while the measurements are variables that trace how the platforms or the radio program evolve.

The list of radio capabilities is intrinsically extensible because they depend on software and hardware releases, which are continuously updated. However, we define a core set of basic capabilities, which are represented by a pre-defined list of identifiers.

Table 4 provides the list of basic parameters corresponding to the configuration registers of the hardware platforms provided in WiSHFUL and to the variables used in some reference radio programs (namely, TDMA and CSMA).

NAME	DESCRIPTION
TX_POWER	Transmission power in dBm
TX_Antenna	Antenna number selected for transmission
RX_Antenna	Antenna number selected for reception
NETWORK_INTERFACE_HW_ADDRESS	MAC address of wireless network interface card
TDMA_SuperFrameSize	Duration of periodic frames used for slot allocations
TDMA_NumberOfSyncSlots	Number of slots included in a frame
TDMA_AllocatedSlot	Assigned slot
TDMA_MAC_PRIORITY_CLASS	QUEUE class service associated with TDMA radio program
CSMA_BackoffValue	CSMA backoff value
CSMA_CW	CSMA current value of the Contention Window
CSMA_CWmin	CSMA minimum value of the Contention Window
CSMA_CWmax	CSMA maximum value of the Contention Window
CSMA_timeslot	CSMA duration of the backoff slot
CSMA_eifs	CSMA duration of the EIFS time
CSMA_difs	CSMA duration of the DIFS time
CSMA_sifs	CSMA duration of the SIFS time
CSMA_MAC_PRIORITY_CLASS	QUEUE class service associated with CSMA radio program

Table 4 - List of UPI_R core parameters

Table 5 provides the list of basic **measurements** with the relevant identifier and description. These measurements provide information and statistics about the state of the physical links or the internal state of the node. Also the list of measurements is extensible.

NAME	DESCRIPTION
RSSI	Received Signal Strength Indication (RSSI); it refers to the last received frame in dBm.
SNR	Signal-to-noise ratio (SNR) of the last received frame in dB.
LQI	Link Quality Indicator (LQI)
FER	Frame Erasure Rate (FER)
BER	Bit Error Rate (BER)
NUM_GOOD_PREAMBLE	Number of preambles correctly synchronized by the receiver.
NUM_BAD_PREAMBLE	Number of receiver errors in synchronizing a valid preamble.
NUM_GOOD_PLCP	Number of valid PLCP synchronized by the receiver.
NUM_BAD_PLCP	Number of wrong PLCP errors triggered by the receiver.
NUM_GOOD_CRC	Number of success of CRC checks.

NUM_BAD_CRC	Number of failures of CRC checks.
NUM_TX	Total number of transmitted frames measured since the interface has been started
NUM_TX_DATA_FRAME	Total number of transmitted frames measured since the interface has been started
NUM_TX_SUCCESS	Total number of successfully transmitted frame measured since the interface has been started
NUM_RX	Total number of received frames since the interface has been started
NUM_RX_ACK_RAMATCH	Total number of received frames addressed to the node since the interface has been started. This measurement traces the number of received frame in which the receiver address field matches with the network interface card MAC address
NUM_RX_ACK	Total receive ack frame measured since the interface has been started
NUM_RX_SUCCESS	Total number of successfully transmitted frame measured since the interface has been started
CSMA_NUM_FREEZING_COUNT	Total number of freezing during the backoff phase
BUSY_TYME	Time interval in which the transceiver has been active (including reception, transmission and carrier sense).
TX_ACTIVITY	Time interval in which the transceiver has been involved in transmission.
LOW LEVEL TIME	Time provided by platform chipset

Table 5 - List of UPI_R core measurements

3.2 WiSHFUL UPI Radio Functions List

This section reports the complete list of UPI_R functions developed during the first and second year of project activities. In particular, on the basis of user feedback and requirements emerged during the showcase implementation, we performed a few minor updates on the list described in D3.2. Some changes were required in order to harmonize the UPI_R implementation between the different platforms and better clarify their usage. About this second aspect, as detailed in the following Table 6, we decided to add some functions, which explicitly refer to the parameters they are working on, rather than using the generic set/get functions.

Table 6 shows the UPI Radio complete list: for each function, we detail the current function name and parameters, the list of platforms that support them, and the old version of the function (if any).

Function	Supported platforms	Old version
set_parameters	All radio platforms	No change
get_parameters	All radio platforms	Not change
get_measurements	All radio platforms	get_monitor
get_measurements_periodic	All radio platforms	get_monitor_bounce
subscribe_events	All radio platforms	define_event
activate_radio_program	All radio platforms	set_active
deactivate_radio_program	All radio platforms	set_inactive
get_running_radio_program	All radio platforms	get_active
get_radio_platforms	All radio platforms	No change
get_radio_info	All radio platforms	No change
set_tx_power	All radio platforms	NEW
get_tx_power	All radio platforms	NEW
get_noise	All radio platforms	NEW
configure_radio_sensitivity	All radio platforms	NEW
configure_cca_threshold	All radio platforms	NEW
set_rx_channel	All radio platforms	NEW
get_rx_channel	All radio platforms	NEW
set_tx_channel	All radio platforms	NEW
get_tx_channel	All radio platforms	NEW
set_rx_bandwidth	All radio platforms	NEW
set_tx_bandwidth	All radio platforms	NEW
set_rx_antenna	All radio platforms	NEW
set_tx_antenna	All radio platforms	NEW
get_hwaddr	All radio platforms	NEW
get_airtime_utilization	All radio platforms	NEW
perform_spectral_scanning	Atheros platform	NEW
get_csi	Atheros platform	NEW
set_sas_conf	RAS antenna	NEW

Table 6 - Complete list of UPI_R general functions

From the previous table we can observe that UPI_R functions support two different styles for reading and configuring relevant parameters of the MAC/PHY stack. On one side, as considered in Y1, UPI_R provides two common get_parameter/set_parameter functions and the common get_measurements, in which the capabilities parameters and measurements are given as a *(key, value)* for monitoring and enforcing configuration value. On the other side, in Y2, we added a list of monitor and configuration functions, whose name has been differentiated for directly addressing the radio parameter controlled by each function. This choice has been motivated by the need of keeping backward compatibility with Y1 implementation, and improving code readability with self-explaining function names.

As an example, assume that experimenter wants to read the Channel State Information (CSI) in a OFDM-based radio platform. The same result can be obtained by calling the get_measurements(csi_key) function or the get_csi() function. Figure 5 shows the results of these calls, by plotting per OFDM subcarrier and RX antenna SNR estimated from the Channel State Information (CSI) in a real link. We see that link is frequency-selective (up-to 10 dB for most rx antennas). Moreover, there is a large difference between the best antenna and the worst antenna.

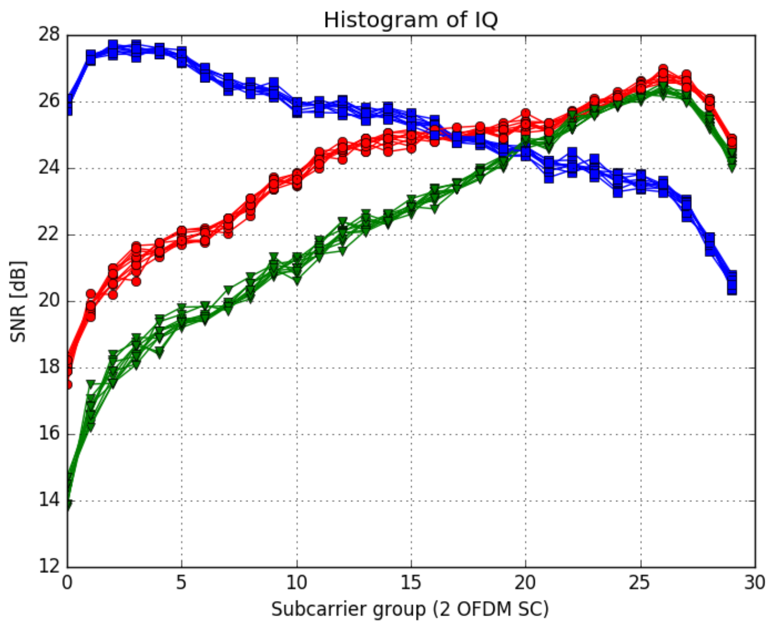


Figure 5. SNR per OFDM subcarrier and RX antenna (color) estimated from CSI of 10 consecutive packets.

3.3 Enhancements in existing platforms and new implementation

As anticipated in section 2, we refer to a **hardware system and relevant software modules** exposing a configuration UPI interface and abstract programming model as **platform.** This definition generalizes the concept of radio platform to any hardware system, which does not necessarily include a radio transceiver (such as intelligent antennas or measurement sensors) and can be added to wireless nodes for providing new capabilities.

During Y2, we extended the original platforms supported by WiSHFUL and integrated two more platforms into the project: **an additional software-defined radio platform**, which is a de-facto reference platform for the research community working on SDR, and **an configurable-antenna platform**, which has been integrated as an Open Call extension.

3.3.1 TAISC SDR framework and Zolertia RE-Mote (AVR CPU) based platforms

TAISC (Time-Annotated Instruction Set Computer) is an architecture for designing, implementing and runtime configuration of flexible MAC schemes. A MAC scheme can be designed by composing a TAISC chain. A TAISC chain is defined as a sequence of instructions with one optional time reference instruction. The TAISC architecture annotates each instruction of a TAISC chain (stored in the program memory) with timing information (both execution time and offset with respect to the time reference). To optimize time synchronisation, all instructions are packed before and after the reference instruction. TAISC packs every instruction in the time domain with respect to the time it needs to finalize. By using the timing information in the TAISC instruction set, the TAISC compiler can compile a non time-aware chain, written in a C dialect, into a time-aware TAISC binary. For this purpose, the TAISC compiler will translate one or more chains into a byte code (binary) and add the time annotation to every instruction. Since the timing information depends on the radio hardware platform, this information is also stored in the TAISC library and used by the TAISC compiler when compiling for a specific target. It is hence possible to compile the same chain into different radio hardware platform specific binaries. After compilation, the TAISC binary is ready to be added and executed by the TAISC execution engine where the lower MAC protocol is executed. To this end, the TAISC binary needs to be uploaded into the TAISC execution engine via the management interface.

Major refactoring was needed in order to port TAISC into an AVR based platform and keep the ability to integrate it within CONTIKI as a MAC layer protocol implementation. TAISC was closely coupled with the hardware modules of the MSP430 processor that is used in the RM090 mote, the first sensor node platform TAISC was supporting, so a redesign of TAISC including the definition of a generic southbound interface to the HAL (Hardware Abstraction Layer) of any specific platform was required.

a. North-bound interfaces

General interfaces of TAISC are the DATA, Management and Control interfaces. Data and Control is always exposed through the Upper Mac implementation of a specific MAC protocol, while Management is a TAISC core interface. All interfaces and how they are actually exposed to the upper layer are described in the next paragraphs.

The Management plane interface provides functionality to upload and/or activate new compiled MAC protocols. This interface is minimalistic for now and has only two main functions ClearAll and Append in order to avoid fragmentation of the TAISC ROM. With the ability to have multiple MACS enabled, there is also a SetActive function that enables the selected MAC to run. One of the consequences of this minimalistic approach is that all upperMACs will always need to store a local copy of the lowerMAC and provide it in case the TAISC ROM needs to be repopulated after a ClearAll.

The data plane interface interacts with incoming/outgoing frames from/to the upperMAC. The upperMAC must always expose the functionality of this interface to the upper layer in order to be able to send and receive data to/from it.

The control interface provides the upperMAC access to the lowerMAC specific variables which are stored in GPRAM section in the TAISC RAM. This interface is also exposed to the user by the upperMAC with GET/SET_parameter type of functions in order for the user of the MAC to have access to alter parameters of the lowerMAC implementation. It is up to the implementer of the upperMAC to decide which parameters of the lowerMAC will be exposed to the MAC layer user.

We distinguish two main discrete cases where the interface of a developer working on TAISC is different and it of course varies because due to a white box approach the developer may choose to work in different levels of TAISC. So the developer may choose to write a new MAC or he may choose to go deeper and try to enhance the TAISCparser for instance. Those 2 cases are presented below and of course the list is not exhaustive. We present those examples to make clear that in a white box approach, TAISC interfaces change based on the software level the developer wishes to work on. Following the white box approach we also present an example of a developer going into TAISC, to present what is considered for him a northbound interface of TAISC.

Case 1. Writing a MAC

In order to write a MAC protocol in TAISC, a number of instructions are made available to the protocol developer by the different modules in TAISC. The developer should chain those instructions together in order to create a functioning MAC protocol. Example instructions per module are:

- Core module:
    - stop(TAISC_relBigTimestampT duration): stops the execution of the current chain, and reschedules the chain in “duration” μs.
    - loadChain(TAISC_ChainIDT id, TAISC_relBigTimestampT timeOffset): loads the chain with given id in “timeOffset” μs.
    - report(TAISC_sizeT size, TAISC_ReportIDT id, TAISC_dataT data, TAISC_boolT byVal): reports a given variable to the upper layers.
- Arithmetic module:
    - add(TAISC_sizeT size, TAISC_dataT a, TAISC_dataT b, TAISC_dataT c): adds b and c and places the result in a.
    - copy(TAISC_sizeT size, TAISC_dataT dst, TAISC_dataT src, TAISC_dataT mask): copies src to dst, given a certain mask
    - random(TAISC_absBigTimestampRWT rand_value, TAISC_absBigTimestampT min_value, TAISC_absBigTimestampT max_value): generates a random value between a min and max
- Data plane module:
    - rxTrigger(): triggers the upper layer that a packet has been received
    - txTrigger(): triggers the upper layer that a packet has been transmitted.
- Radio module:
    - on(): puts the radio in “on” modus
    - setChannel(TAISC_channelT channel): sets the current channel of the radio
    - tx(): transmits the current packet

The protocol developer can create one (or more) chains for a specific MAC protocol. A good rule of thumb is to put all time critical operations in a chain (lower MAC), and the less time critical operations in the upperMAC.

Case 2. Using a MAC

To use a MAC protocol in an operating system, the correct interfaces should be initialised. In order to achieve this, TAISC needs to be presented as a virtual MAC protocol to Contiki with the same interfaces as other Contiki MAC protocols (send packet, receive packet, input packet, etc). This interfacing happens in the so called TAISC upperMAC which is responsible for all non-time critical operations, as well as getting/setting of parameters in the TAISC chain. Northwards it interfaces with the Contiki operating system, southwards it interfaces with TAISC via one if its control interfaces (dataplane, control plane and management).

Example functionality from upperMAC to TAISC contains:

- taiscAPI__Init__init: initialise TAISC and start the execution of the first chain.

- taiscAPI__taiscControlplane__assign: will update a variable in TAISC with a given value
- taiscAPI__taiscControlplane__startChain: starts the execution of a given chain. In most cases this will be an configuration chain which is responsible to start the specific TAISC MAC protocol.

Example functionality from TAISC to upperMAC contains:

- taiscAPI__taiscDataplane__handledTxBufferDone: transmission has been completed (initiated from the the txTrigger instruction in a TAISC chain)
- taiscAPI__taiscDataplane__completedDataBuffer: packet has been received (initiated from the the rxTrigger instruction in a TAISC chain)
- taiscAPI__taiscControlplane__report: report a specific value to the upper layer (initiated from the the report instruction in a TAISC chain)
-

The general architecture of TAISC is presented so that any developer wishing to enhance a specific aspect of TAISC can do so based on this architecture presentation. TAISC is a hardware concept but it has been implemented as a VM (virtual machine) so far. Like any other **ISC** (Instruction Set Computer) TAISC has a logic unit, which fetches the instruction out of the ROM and executes the instruction, which could manipulate the RAM. The compiled instructions which are fed to the TAISC core are **TA** (Time Annotated) and will be executed accordingly (which is on time).

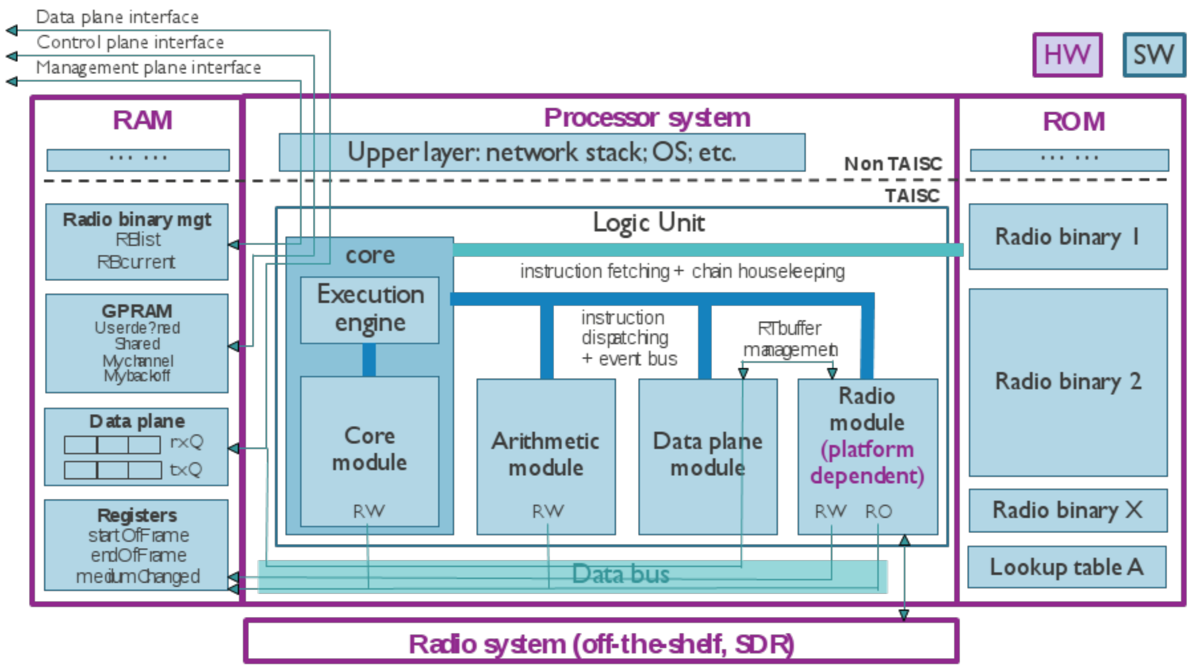


Figure 6 - TAISC architecture

The explicit North-bound interfaces of the VM are shown in Figure 6 which include data, control and management control interfaces.

If a developer decides to work on the TAISCParser level and wants to either enhance the parser or create a new one, then for him the northbound interface of TAISC will be specific implementation details like:

- the binary format used to store the compiled instructions in the ROM
  - TAISCParser compiles any platform-independent MAC chains (which are MAC protocols coded using platform-independent instructions) into platform-specific binaries according to the defined binary format.
  - the compiler uses a library with platform-specific execution times of the instructions regarding the underlying platform the structure of the TAISC RAM (see Figure 6) and the data types used in the TAISC MACs and defined by TAISC chains
  - the endianness and the data size of the data types used in the MACs must be compatible on all TAISC VM instances and independent of the underlying platform. For instance, different sensor platforms in WiSHFUL have the same endianness, but a different data size:
    - MSP430 (on RM090): little endian 16 bit CPU
    - ARM Cortex-M3 (on Zolertia RE-Mote): little endian 32 bit CPU
- LED and Logic Analyzer interfaces (used for debugging purposes) which are compatible on all TAISC VM instances
  - 16 GPIOs are selected on the target platform to observe the TAISC activities via a Logic Analyzer like: instruction and chain envelope, radio off, sleep, on, reception, transmission, etc.

b. South-bound interfaces

The next section describes how to maximize code reusability while porting the VM to another underlying platform. The architecture redesign of TAISC can be seen in Figure 7 (rm090 specific object monolithic architecture) and Figure 8 (redesigned generic object modular architecture) and is explained in detail hereafter.

In Figure 7 the contents of the TAISC software library for the RM090 platform in Contiki OS is presented. It contains hardware specific implementations targeting timer, SPI, DMA, and CC2520 wireless transceiver drivers. Further, the utilities used for TAISC debugging through Logic Analyser and LED drivers are integrated in this software library.

We achieved the migration of the TAISC VM (virtual machine) from a RM090 platform (MSP430 based) written in Tinyos, to a modular architecture by applying a generic redesigning approach in order to be able to port it in the future to any target platform. Of course there are always still some platform specific issues especially regarding the radio driver which most of the times won't provide immediately all the advanced functionality that TAISC requires from it. In Figure 8 we present the case for the ARM based platform that we targeted, the CC2538 wireless microcontroller system-on-chip module, but the approach is similar for any other platform (and OS):

- A number of ***South bound interfaces*** are defined to address all hardware specific implementations as external:
  - the TAISC core scheduler South bound interface should contain the following functions, the last one is a call-back which has an implementation in the TAISC core.

```
error_t taiscAPI__taiscAlarmInit__init(void);

void taiscAPI__taiscAlarm__start(uint32_t scheduleAt);

uint32_t taiscAPI__taiscAlarm__getNow();

uint32_t taiscAPI__taiscAlarm__getAlarm();

void taiscAPI__taiscAlarm__fired();
```

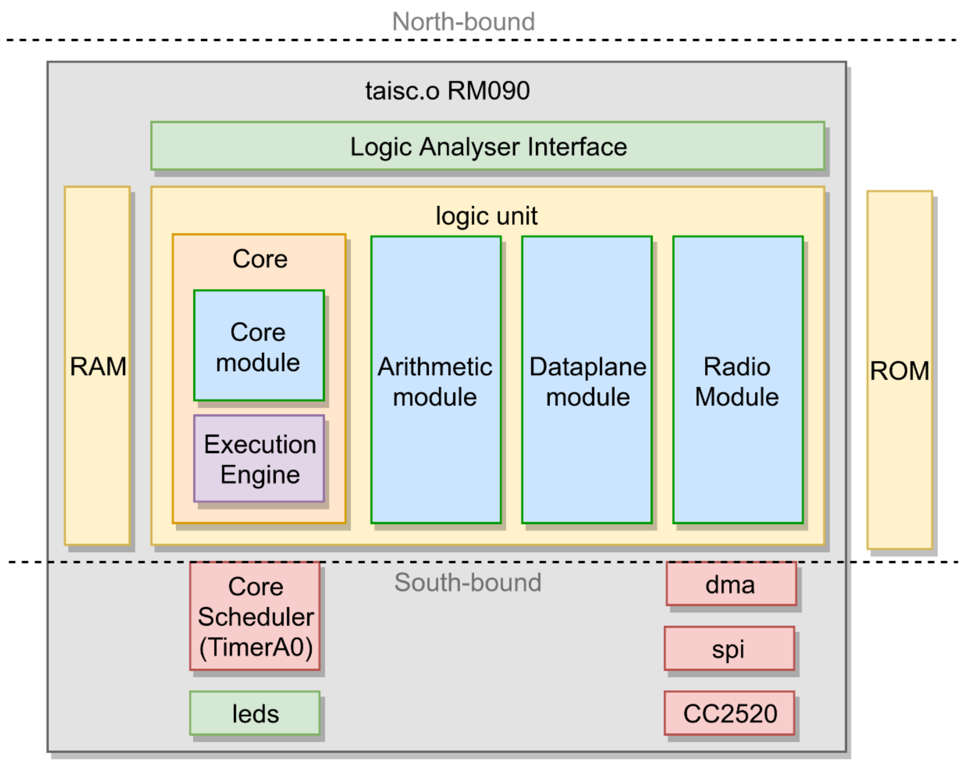


Figure 7 - Contents of TAISC library object when targeting MSP430 CPU

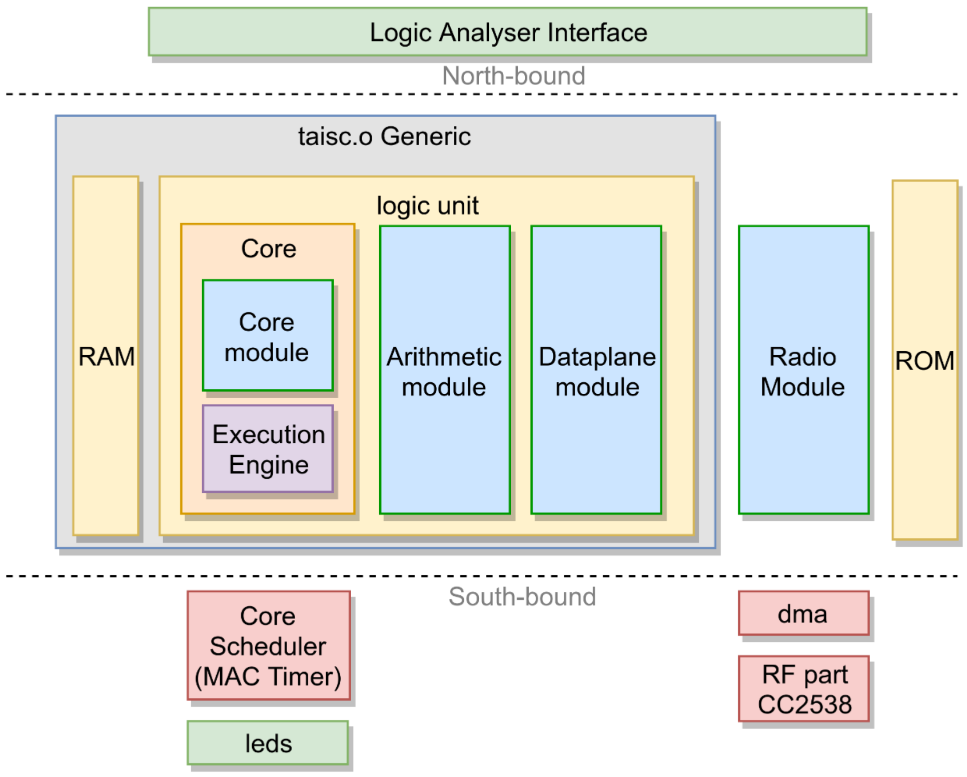


Figure 8 - Contents of the new generic TAISC library object

- in order to address the radio as an external module, all radio module specific interfaces are defined as external:
  - realTimeBufferManagement

    ```
    command void getRxBuffer(uint8_t ** msg, TAISC_DPFrmSizeT * size);

    command void getTxBuffer(uint8_t ** msg, TAISC_DPFrmSizeT * size);

    command void rxBufferReady(uint8_t * msg, TAISC_DPFrmSizeT  size,
    TAISC_conditionT autoTrigger);

    command   void   txBufferLoaded(uint8_t   *   msg,   TAISC_conditionT
    autoTrigger);
    ```

  - Instruction and Event Bus

    ```
    command TAISC_bitMAPTlocal filter(TAISC_bitMAPT f);

    command void clearFilter();

    event   void   trigger(TAISC_bitMAPT   t,   TAISC_relBigReferenceT
    timeInfo);
    ```

  - Data bus (no specific interface needed as RAM is shared resource).
- All hardware specific implementations reside on the target platform (in Contiki OS) like
  - the radio module(s), which will need DMA and hardware specific implementations targeting the RF part of the CC2538.
  - the TAISC core scheduler driver implementation on the CC2538 is written on top of its MAC timer. The CC2538 system clock is 32MHz and the TAISC resolution is 1 µs. The MAC Timer has a 16 bit lower part which is configured to divide the system clock to 1MHz and the 24 bit overflow part of the MAC timer will be used to store the 3 LSBs (lowest significant bytes) of the 32bit TAISC timer. This implies that the MSB (most significant byte) is tracked in software. We would have expected a performance gain on the ARM Cortex M3 compared to the msp430 with respect to the databus (32bit versus 16bit) and system clock (32MHz versus 16MHz). But the hardware coupling from the ARM CPU towards the RF part is done by an 8-bit data bus. The expected performance gain is in reality lost due to the poor hardware coupling.
  - LEDs and the logic analyser drivers which are used for debugging purposes.

As the TAISC core becomes Tinyos-independent there is no need to create a new Tinyos platform per target platform. The TAISC core, arithmetic and data plane modules are compiled, due to cascading nesC and the target platform compiler, to a C software library compatible to the target platform. The target platform linker bonds this TAISC core software library (taisc.o) with the rest of the Contiki code. This approach can be applied to any IoT platform and implies code reusability of the TAISC core modules.

c. *Design refactor actions on the TAISC core during Y2*

Besides many little things to make the TAISC more performant and stable, the first three refactor actions had a very big impact on the stability and the code size of the TAISC core:

c.1 *Moving from auto grab towards instruction based frame grab implementation*

The term grab is used in the context of getting a received frame out of the radio part and transfer it to the CPU where the TAISC core is running. The implementation used during Y1 of WiSHFUL scheduled the transaction automatically based on a specific interrupt triggered by the radio part.

But at the same time the TAISC core could be executing an instruction and also imply a transaction towards the radio part, for example changing the transmission power. An arbiter was implemented to handle multiple transactions coming from multiple concurrent processes and sequence them in the order as they were invoked. The arbiter has been optimized a few times, and seems to have a big impact on the execution time, and even worse its execution was not deterministic. Further it could conflict with a running MAC. For example: a frame comes in and the MAC decides that the radio should go into a sleep mode. Until now it was the responsibility of the MAC designer to always keep a window open to handle the auto grab functionality. By moving this trigger from an interrupt-based trigger to an instruction-based execution it had a big positive impact on the stability and code size of the TAISC core:

- radio interrupts are no longer state-full. MAC designers don't need to keep track on the number of transactions that are needed to get all received frames out of the radio. Before this refactoring, missing an interrupt could result in a TAISC core that was no longer synchronized with the radio state.
- There is no longer need for the arbiter that keeps tracks on the radio transactions as there is only one process, the TAISC core, which initiates the transactions with the radio. It is now possible to have a flat implementation towards the radio without storage (queuing) and related functions.
- The MAC designer has now complete control over when and how the radio transactions are taken place with the “grabframe” instruction. The drawback is of course is that the MACs need to be alerted for incoming frames. One advantage is, if the MAC is not interested in receiving a frame it can flush the frame in the radio itself instead of first transferring it to the data plane and then drop it.

c.2 Only enable interrupts when needed

This refactoring is related to the previous refactoring, where we designed interrupts to be no longer state-full, meaning that interrupts can be turned off if no longer needed. By doing so, the radio instructions are further optimized and have lower execution time. We activate the interrupts when requested by the “waitForTrigger” instruction. For example if the MAC designer wants to wait until the medium becomes idle an according interrupt is enabled. This refactoring has an impact not only on the real nodes but also in the Cooja environment (see section 5.3), and more specifically relaxes the CPU overhead while debugging.

c.3 Handle radio exceptions during the stop instructions

Radio exceptions inform us about unexpected radio activity, for example if the radio is demodulating an incoming frame and the MAC decides to disable the RF part of the radio by invoking an “idle” instruction. In this specific case the radio will throw an aborted frame exception which needs to be handled and cleaned up. Before this refactoring, we scheduled this exception as soon as possible, which resulted in non-deterministic execution of the following instructions. Now we handle the radio exception during the “stop” instruction and this has only an impact on the latter.

c.4 Introduce a fake event: onEventStoreTimeStamp

The TAISC core always stored the timestamp for every event. As these timestamps are only used occasionally we added a fake event “onEventStoreTimeStamp” in order to inform the TAISC core to also store the timestamp when the mentioned event triggers. This results in minor optimization of the execution, but it implies that we only need one register to store the timestamp (“storedTimestamp”). Before this refactoring we had TAISC registers for “startOfFrameTimestamp”, “endOfFrameTimestamp” and “mediumChangedTimestamp” as well.

c.5 Resize TAISC event bitmap

As the number of events is only 14, we scaled the data type used to store an event bitmap ("TAISC_bitMAPT") from 32 to 16 bits. Which implies that all instructions, which include a parameter of this data type shrink by two bytes in the TAISC ROM and the fetching of these instructions is also faster now.

3.3.2 IRIS SDR Framework

In this section, we present the implementation details and capabilities provided through the integration between the WiSHFUL UPI_R and the IRIS SDR framework. This work has been developed as part of the DVB-TX-IRIS Open Call 1 extension, led by the Department of Engineering of the University of Perugia (UPG). The DVB-TX-IRIS Open Call 1 aims at designing an extension of the WiSHFUL framework that leverages the reconfiguration capabilities offered by modern SDR technology. This integration work can be summarized into the following phases:

1. Extension of the IRIS SDR platform to support the reception of multiple asynchronous concurrent control events.
2. Development of the appropriate UPI_R for the configuration of SDR parameters. This capability was tested with Ettus USRP N210 boards.
3. Design of new modules and testing examples that enable the interconnection and configuration on-the-fly of the IRIS SDR framework through the respective UPI_Rs.

We highlight that in this deliverable we are considering the definition of the UPI_R. Another important point is that the WiSHFUL projects focus is on the integration with the IRIS SDR implementation supported by the commercial radio front-end hardware USRP N210 provided by Ettus Company.

a. Extension of the IRIS SDR Framework to receive multi-event

The IRIS SDR platform enables the development of PHY and MAC radio components in software, which support reconfiguration of radio components and parameters on-the-fly. The IRIS SDR platform it is very flexible. As components are developed in software, any part of the radio can be reconfigured. However, it was necessary to redevelop the IRIS SDR to support receiving several reconfiguration events at the same time.

This extension was designed to give the DVB-TX-IRIS Open Call 1 team more options when performing experiments. The previous version of the IRIS SDR framework did not support receiving multiple events. Figure 9 illustrates the original scenario. In this example the operator sends four events. However, only one event is processed. The other three events were dropped.

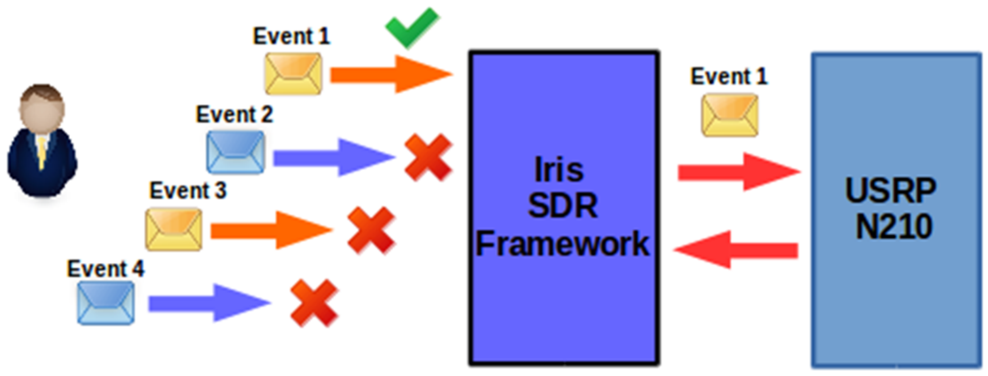


Figure 9 - IRIS SDR Framework receive just one event

In the extension, the IRIS SDR Framework allows multiple reconfiguration events to be received at the same time. In Figure 10 we illustrate this scenario. In this example, the operator sends four events. These events are received and put into a FIFO queue before being processed by the IRIS SDR Framework.

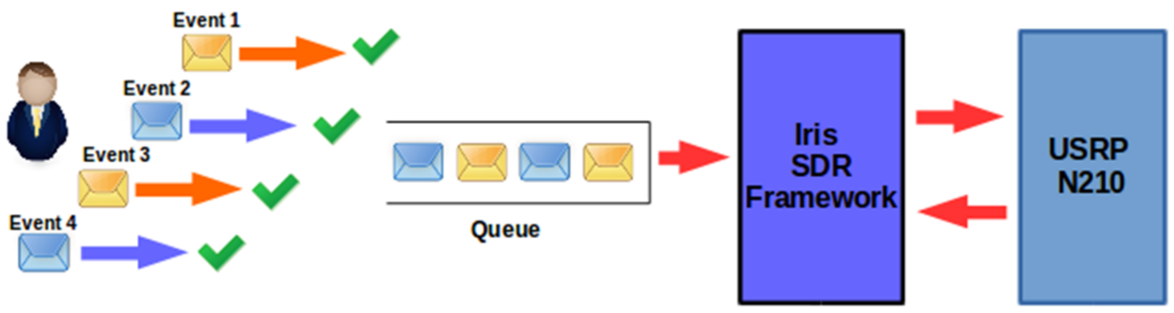
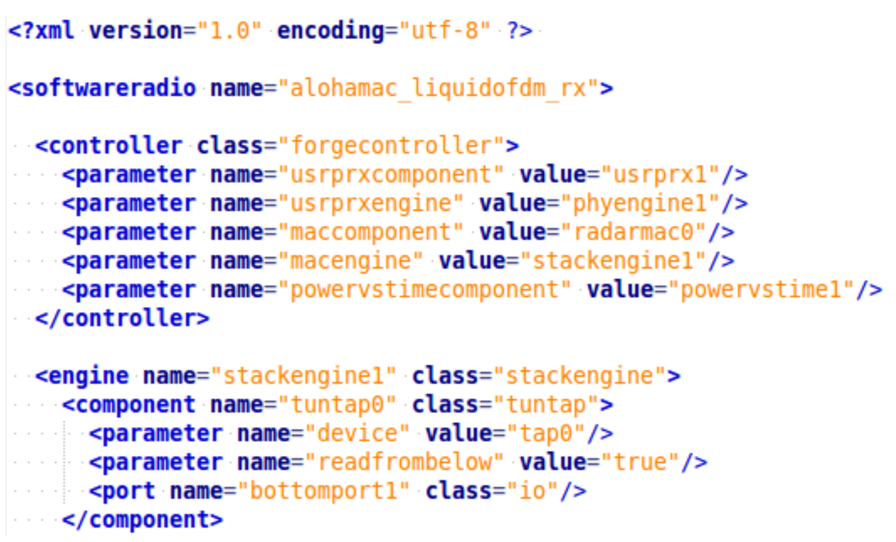


Figure 10 - IRIS SDR Framework receiving multi-events

b. *Implementation details*

In order to run the IRIS SDR Framework, an XML configuration file is necessary. This file tells the core IRIS program what engines will be used to create the radio SDR, and what components will run within the engine. It also includes the initial parameter settings for each component. An example XML configuration file for a simple OFDM transmitter is shown in Table 7.



```xml
<?xml version="1.0" encoding="utf-8" ?>

<softwareradio name="alohamac_liquidofdm_rx">

  <controller class="forgecontroller">
    <parameter name="usrprxcomponent" value="usrprx1"/>
    <parameter name="usrprxengine" value="phyengine1"/>
    <parameter name="maccomponent" value="radarmac0"/>
    <parameter name="macengine" value="stackengine1"/>
    <parameter name="powervstimecomponent" value="powervstime1"/>
  </controller>

  <engine name="stackengine1" class="stackengine">
    <component name="tuntap0" class="tuntap">
      <parameter name="device" value="tap0"/>
      <parameter name="readfrombelow" value="true"/>
      <port name="bottomport1" class="io"/>
    </component>
```

Table 7 - XML file configuration example

The configuration example shows the specification of the controller class. This class supports reception of commands to dynamically change radio components and parameters.

c. *Adaptation Module*

According with the WiSHFUL IRIS SDR adaptation module implementation (Table 8), the parameters that can be changed by the WiSHFUL framework correspond to what has been exposed by the radio designer. This function sets the values of the specified parameters in the argument passed.

```
@wishful_module.bind_function(upis.radio.set_frequency)
    def set_frequency(self, frequency):
        self.log.debug("IrisModule set_frequency: %s on interface: %s"
                       % (frequency, self.interface))
        return self.generic_interact('set', 'frequency', frequency)

@wishful_module.bind_function(upis.radio.set_rate)
    def set_rate(self, rate):
        self.log.debug("IrisModule set_frequency: %s on interface: %s"
                       % (rate, self.interface))
        return self.generic_interact('set', 'rate', rate)

@wishful_module.bind_function(upis.radio.set_gain)
    def set_gain(self, gain):
        self.log.debug("IrisModule set_frequency: %s on interface: %s"
                       % (gain, self.interface))
        return self.generic_interact('set', 'gain', gain)

@wishful_module.bind_function(upis.radio.set_bandwidth)
    def set_bandwidth(self, bandwidth):
        self.log.debug("IrisModule set_frequency: %s on interface: %s"
                       % (bandwidth, self.interface))
        return self.generic_interact('set', 'bandwidth', bandwidth)
```

Table 8 – WiSHFUL IRIS SDR adaptation module implementation

The main IRIS configuration parameters are presented in the Table 8. In this module, a server socket is created, which receives commands from the WiSHFUL UPIs that are passed to the IRIS SDR framework. The same socket is used to send a response from IRIS to the WiSHFUL UPIs. Table 9 shows the WiSHFUL UPI parameters and their descriptions.

Parameter	Description
set_frequency	Configures the value of the frequency in SDR.
set_gain	Configures the value of the gain in SDR.
set_ bandwidth	Configures the value of the bandwidth in SDR
set_rate	Configures the value of the rate in SDR.

Table 9 - IRIS SDR Framework UPI_R specific technology functions

d. Examples

In this section we present the WiSHFUL files using the IRIS SDR Framework. In this example, we use the WiSHFUL framework to configure the USRP N210 frequency parameter on the fly.

Table 10 illustrates the IRIS agent configuration. In this agent (Agent_1) it is necessary to run the following python module to start the IRIS WiSHFUL agent: wishful_module_iris.

```
## WiSHFUL Agent config file
agent_info:
  name: 'agent_1'
  info: 'agent_info'
  iface: 'lo'
```

```
modules:
  discovery:
      module : wishful_module_discovery_pyre
      class_name : PyreDiscoveryAgentModule
      kwargs: {"iface":"lo", "groupName":"Iris_1234"}
  iris:
      module : wishful_module_iris
      class_name : IrisModule
      interfaces : ['iris']
```

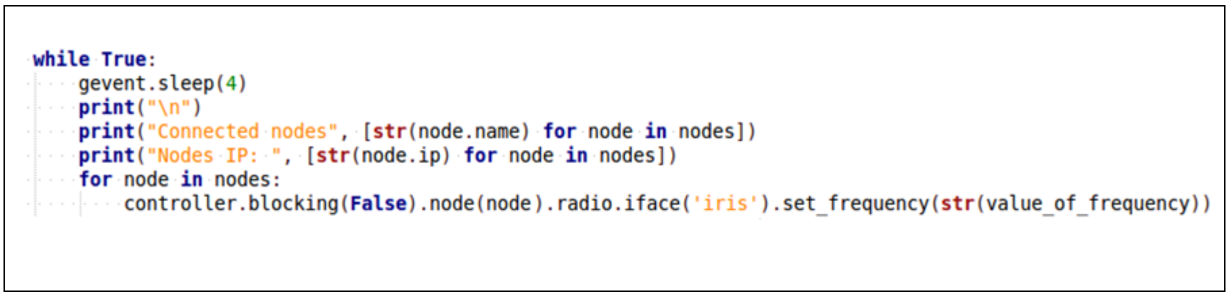
Table 10 - Agent_config.yaml file with the configuration example

Table 11 illustrates the WiSHFUL Controller configuration file. The controller will use this set up for on-the-fly configuration changes in the IRIS SDR modules.

```
## WiSHFUL Controller's config file
controller:
    name: "Controller"
    info: "WiSHFUL Controller"
    dl: "tcp://127.0.0.1:8990"
    ul: "tcp://127.0.0.1:8989"
modules:
    discovery:
        module : wishful_module_discovery_pyre
        class_name : PyreDiscoveryControllerModule
        kwargs: {"iface":"lo", "groupName":"Iris_1234",
"downlink":"tcp://127.0.0.1:8990", "uplink":"tcp://127.0.0.1:8989"}
```

Table 11 - Controller_config.yaml file with the configuration example

Table 12 illustrates the WiSHFUL Controller code. The controller waits to receive commands from the experimenter. When an IRIS node connects to the WiSHFUL Controller, it can receive configuration commands to change parameters via the IRIS radio.iface('iris') interface.



```
while True:
    gevent.sleep(4)
    print("\n")
    print("Connected nodes", [str(node.name) for node in nodes])
    print("Nodes IP: ", [str(node.ip) for node in nodes])
    for node in nodes:
        controller.blocking(False).node(node).radio.iface('iris').set_frequency(str(value_of_frequency))
```

Table 12 - The WiSHFUL Controller file with the configuration example (change the frequency value).

The complete WiSHFUL example controller and agent that integrate with the IRIS SDR is available at the WiSHFUL GitHub repository project:

https://github.com/wishful-project/examples/tree/master/iris

3.3.3 WMP WARP SDR framework

In this section, we present the implementation details and capabilities provided through the integration between the WiSHFUL UPI_R and the WMP implementation on the WARP SDR platform.

The WARP v3 research platform is a FPGA-based software-defined-radio platform that provides a Xilinx Virtex-6 FPGA, two MAX2829 transceivers, and a complete IEEE 802.11 Reference Design that we used as a starting point for our implementation. As shown in the Figure 11, the global architecture of this research board can be divided in two parts: an RF hardware interface, given by the transceiver (MAX2829), ADC/DAC chip (AD9963) and hardware for clocking (AD9512), and an FPGA programmable part (Virtex-6 Xilinx).

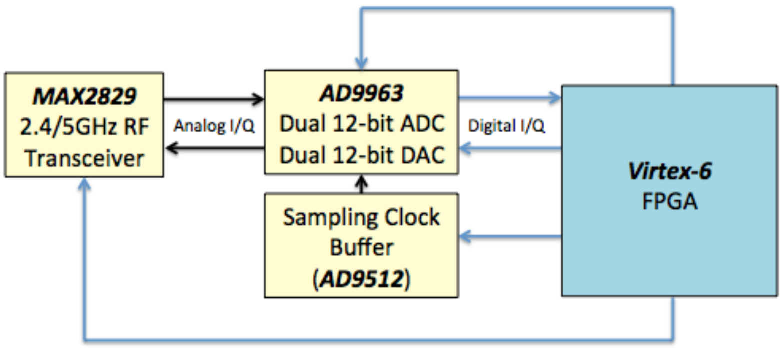


Figure 11 – Architecture of the WARP research board

For this board, some implementations of MAC/PHY stacks are already available. In particular, an IEEE 802.11g MAC/PHY stack has been developed under the name of IEEE 802.11 reference design. The architecture is based on two MicroBlaze processors and some dedicated IP cores developed within the FPGA board.

As detailed in the WARP Project [6], the architecture includes: two MicroBlaze CPUs, called CPU High and CPU Low, for executing respectively upper-MAC and lower-MAC operations; a MAC DCF core; two PHY cores, one for the transmission Tx and the other for the receiving Rx; Hardware Support cores. The two MicroBlaze CPUs run the MAC protocol (written in C) according to the usual upper-MAC and lower-MAC decomposition. Specifically, CPU High is responsible of network management operations (probe request/response, association request/response, etc.), which are non-time-critical functionalities. It is also responsible of bridging operations to a wired network, implementing encapsulation and de-encapsulation of Ethernet frames according to the wired-wireless integration described in the IEEE 802.11-2012 standard. On the other hand, CPU Low is responsible of PHY tuning and low-level MAC operations. These include transmission of ACKs, scheduling of backoffs, maintaining the contention window and initiating re-transmissions. There is also a MAC core block, which acts as the interface between the MAC software and the Tx/Rx PHY cores. This core implements the timers required for the DCF (timeout, backoff, DIFS, SIFS, etc.) and the employed carrier-sensing mechanisms. The MAC core monitors the Tx and Rx PHY cores and the relevant events trigged by these cores. PHY Tx/Rx cores implement the OFDM physical layer transceiver specified in the IEEE 802.11-2012 standard. The legacy IEEE 802.11 reference design architecture includes some custom FPGA cores (dedicated to the implementation of the IEEE 802.11g OFDM transceiver, the required timers, the carrier sense mechanism and the interface between the transmission and reception blocks) and two MicroBlaze CPUs running the DCF MAC protocol (written in C) according to the usual upper-MAC and lower-MAC decomposition.

a. Extension WMP SDR Framework

For integrating a WMP architecture based on WARP in WiSHFUL, **we started from an implementation developed by the CNIT research group within other projects (namely, an Open Call extension of the FP7 project CREW)**. However, this architecture was not integrated into the WiSHFUL framework and was not able to exploit more advanced flexibility at the PHY layer, offered by the possibility to work on the receiver implementation.

In our implementation, we replaced the programs executed by the two CPUs with two different programs: the high-level one, adding the WMP control interface to the upper MAC functionalities, and the low-level one implementing the MAC Engine and part of the WMP [2]. In the WMP control interface, a server socket is created, which receives commands from the WiSHFUL UPIs that are passed to the WMP SDR platform. The same socket is used to send a response from WMP to the WiSHFUL framework.

Table 13 shows the WiSHFUL UPI functions implemented and their descriptions. We also added some other blocks for supporting a dedicated BRAM to store radio program, the relevant controller, a mutex for regulating the BRAM accesses performed by the high-level and low-level CPUs, and some software registers.

Function	Description
set_parameters	Setting radio parameter
get_parameters	Getting radio parameter
activate_radio_program	Active radio program
deactivate_radio_program	Deactive radio program
get_running_radio_program	Get the running radio program
get_radio_platforms	Get the radio platform available
get_radio_info	Get the radio platform capabilities
set_tx_power	Configures the value of the TX power in WMP SDR
configure_cca_threshold	Configure the Clear Channel Assessment
set_rxchannel	Configures the value of the RX channel in WMP SDR
set_txchannel	Configures the value of the TX channel in WMP SDR
set_rx_bandwidth	Configures the value of the RX bandwidth in WMP SDR
set_tx_bandwidth	Configures the value of the TX bandwidth in WMP SDR
set_gain	Configures the value of the **gain** in WMP SDR.
set_rate	Configures the value of the **rate** in WMP SDR.

Table 13 - SDR WARP UPI functions

b. *Implementation details*

In order to support more advanced flexibility at the physical layer, which include also non-standard configurations, we envisioned the possibility to tune dynamically the central frequency and the transmission bandwidth of the nodes, and to compose different transmitter and receiver chains from pre-defined available blocks. These requirements imply the design of a novel engine and a novel OFDM-based transceiver architecture. While the novel engine is simply programmed by a firmware code to be executed by the low CPU, the desired transceiver re-configurability has some implications on the hardware architecture. On one side, it is possible to exploit the configuration capabilities of some blocks by acting on the hardware registers, which correspond to the tuning of the operation parameters enabled from WiSHFUL.

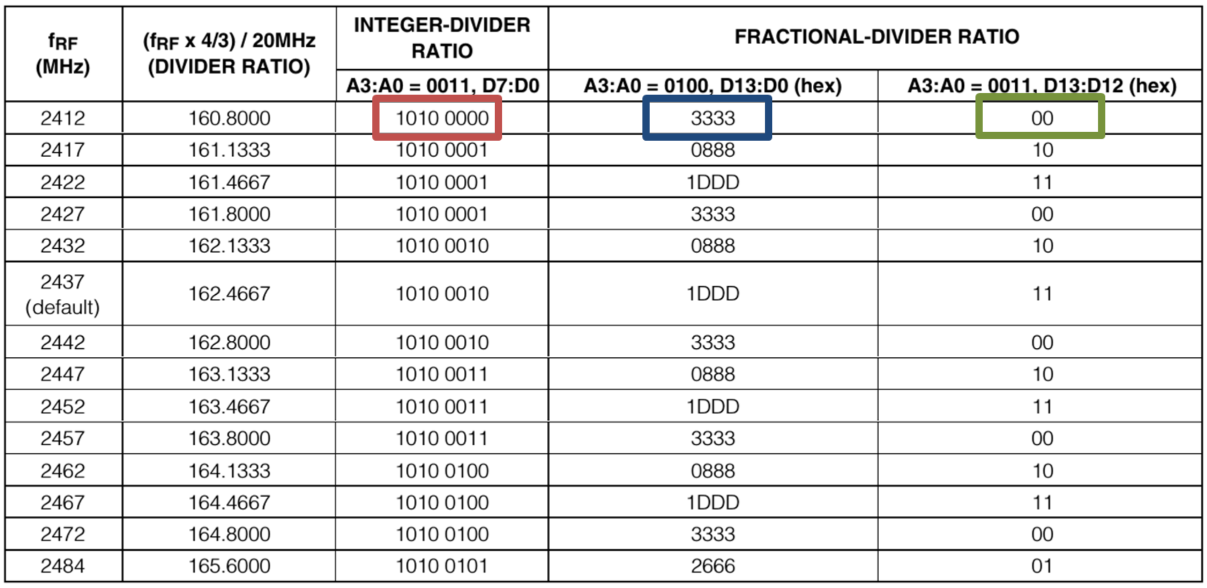
More into details, for changing the carrier frequency it is possible to act on two registers in the MAX2829. These values are the integer part and the fractional part of a parameter, called DIVIDER RATIO, given by the following expression:

$$DIVIDER\ RATIO = \frac{F_{req} \times 4}{3 \times 20}$$

where F_{req} is the desired central frequency. Both the integer and fractional part can be assigned by setting the registers:

$$REG3 = DIVIDER\_RATIO\_LSB |\ INT(DIVIDER\_RATIO)$$
$$REG4 = FRACT\_DIVIDER\_RATIO\_MSB$$

For example, for the legacy IEEE 802.11b/g channels, the central frequencies can be obtained by tuning the REG3 and REG4 registers to the values specified in the third/fifth column and fourth column of the following Table 14.



f_{RF} (MHz)	(f_{RF} x 4/3) / 20MHz (DIVIDER RATIO)	INTEGER-DIVIDER RATIO	FRACTIONAL-DIVIDER RATIO	
		A3:A0 = 0011, D7:D0	A3:A0 = 0100, D13:D0 (hex)	A3:A0 = 0011, D13:D12 (hex)
2412	160.8000	1010 0000	3333	00
2417	161.1333	1010 0001	0888	10
2422	161.4667	1010 0001	1DDD	11
2427	161.8000	1010 0001	3333	00
2432	162.1333	1010 0010	0888	10
2437 (default)	162.4667	1010 0010	1DDD	11
2442	162.8000	1010 0010	3333	00
2447	163.1333	1010 0011	0888	10
2452	163.4667	1010 0011	1DDD	11
2457	163.8000	1010 0011	3333	00
2462	164.1333	1010 0100	0888	10
2467	164.4667	1010 0100	1DDD	11
2472	164.8000	1010 0100	3333	00
2484	165.6000	1010 0101	2666	01

Table 14 – Tuning of the transmission carrier.

For changing the transmission bandwidth, we exploited a multi-clock architecture for both the digital-to-analog converters and the modulator, thus enabling the scaling of the OFDM sub-carrier channels. The digital to analog (DAC) converter is fed by the AD9512 chip (see Figure 11), which gives the clock reference (clk_ref). Therefore, for changing the converting rate, it is possible to work on the reference clock. The clock is nominally set to 80MHz, but it can be reduced to a lower rate thanks to a divider integrated in this chip with a configurable scaling factor (whose values range from 1 to 32). The outgoing clock can be again managed inside the AD9963 chip, following the formula:

$$Channel\ bandwidth = \frac{(clk\_ref \times DLL\_M)}{(DLL\_N \times D)}$$

where DLL_M is a multiplier factor, DLL_N is a divider factor and D is the interpolation factor. It is important that clk_ref x DLL_M must be greater than 100MHz.

In order to support the selection of receive bandwidths and channels, we implemented a receiver, able to filter the desired signals according to the bandwidth configured by the UPI_R function. An extension of this solution for automatically filtering the received signals by using in-band or out-band signalling mechanisms is currently considered within another research project.

c. Future work

As soon as more advanced capabilities will be supported by the WMP implementation on the WARP SDR board, we plan to integrate these capabilities in the UPI_R adaptation module. For example, we plan to integrate the possibility of configuring an agile receiver, able to autonomously identifying the bandwidth and the central frequency of the signals, and/or the possibility to configure more advanced antennas.

3.3.4 GNU Radio

This section describes the UPI_R implementation in GNU Radio. After a short introduction on GNU Radio, this section is organized in four subsections related to the UPI_R functions that perform the radio program management.

GNU Radio is an open-source software architecture that provides signal processing blocks for implementing a radio transceiver in software, by means of the so-called flow graphs. It can be used with external RF hardware (e.g. USRP) to create software-defined radios, or without hardware in a simulation-like environment. It is widely used for wireless communications research and real-world radio systems.

a. Generic Support

We integrated GNU radio in WiSHFUL, by allowing an experimenter to write his own radio programs using GNU Radio and to control it by using the WiSHFUL framework. To this purpose, we developed a generic GNU Radio adaptation module shown in Figure 12, which provides an implementation of four generic UPI_R functions: i) *activate_radio_program()*, ii) *deactivate_radio_program()*, iii) *set_parameters()* and iv) *get_parameters()*. Hence, the experimenter is able to activate and deactivate a GNU Radio program, represented as a GRC XML file, on-the-fly. Moreover, with the help of the latter two generic functions, the experiment can introspect the state of the program represented by variables, such as the central frequency of the transceiver.

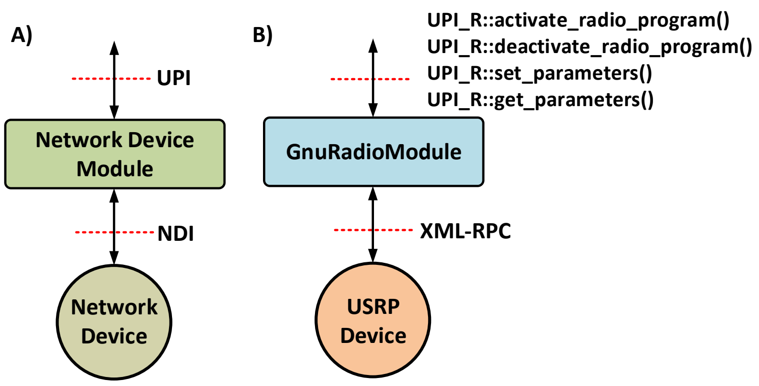


Figure 12 - WiSHFUL device modules: A) concept, B) module to support GNU Radio

b. *Acquiring nodes information*

Two functions are defined in this UPI_R group: get_radio_platforms() and get_radio_info(). They allow acquiring the following information: i) present platforms on wireless node; ii) list the capabilities of the node. This last function, in turns includes:

- Supported configuration parameters
- Measurements node capabilities
- Available radio programs (e.g. CSMA, TDMA)
-

c. *Configuring nodes*

There are two UPI_R functions in this group: set_parameters () and get_parameters (). They allow getting/setting the configuration parameters of the radio platform. The complete list of configuration parameters is given in Section 3.1. The parameters supported by GNU Radio platform can be retrieved at runtime using the function get_radio_platforms(). The functions of this group use the custom ctrl_socket python module for retrieving GNU Radio specific (NIC) info.

set_parameter(param_key_values):

The parameters that can be configured correspond to those exposed by the radio designer to the WiSHFUL framework. This function (re)set the value(s) of the specified parameters in the param_key_values dictionary argument. The keys of this dictionary are the parameter names, the values are the configuration values. The code snippet presented in Table 15 illustrates how this is done in this platform.

```
def set_parameter(self, **kwargs):
    if self.gr_state == RadioProgramState.RUNNING or self.gr_state ==
                                                   RadioProgramState.PAUSED:
        self.init_proxy()
        for k, v in kwargs.items():
            try:
                getattr(self.ctrl_socket, "set_%s" % k)(v)
            except Exception as e:
                self.log.error("Unknown variable '%s -> %s'" % (k, e))
    else:
        self.log.warn("no running or paused radio program; ignore command")
```

Table 15 - GNU radio set_parameter UPI implementation

get_parameters(param_keys):

The parameters correspond to those exposed by the radio designer to the WiSHFUL framework. This function gets the value(s) of the specified parameter keys added to the param_keys list. The code snippet presented in Table 16 illustrates how this is done in GNU Radio.

```
def get_parameter(self, **kwargs):
    if self.gr_state == RadioProgramState.RUNNING or self.gr_state ==
                                                       RadioProgramState.PAUSED:
        rv = {}
        self.init_proxy()
        for k, v in kwargs.items():
            try:
                res = getattr(self.ctrl_socket, "get_%s" % k)()
                rv[k] = res
            except Exception as e:
                self.log.error("Unknown variable '%s -> %s'" % (k, e))
        return rv
    else:
        self.log.warn("no running or paused radio program; ignore command")
        return None
```

Table 16 - GNU radio get_parameter UPI implementation

d. *Monitoring nodes*

There are two UPI functions in this group: get_measurements() and get_measurements_periodic(). They allow obtaining (get_measurements) and collecting (get_measurements_periodic) the measurements values from the radio platform. The complete list of configuration parameters is listed in Section 3.1. The parameters supported by GNU Radio platform be retrieved at runtime using the function get_radio_info(). The functions of this group use the custom `ctrl_socket` python module for retrieving GNU Radio specific (NIC) info.

get_measurements (measurement_keys)

This UPI_R function gets the current value(s) of the measurement values specified in the measurement key list. The snippet of code presented in Table 17 shows the core of the function implementation in GNU Radio.

```
def get_measurements (self, **kwargs):
    if self.gr_state == RadioProgramState.RUNNING or self.gr_state ==
                                                       RadioProgramState.PAUSED:
        rv = {}
        self.init_proxy()
        for k, v in kwargs.items():
            try:
                res = getattr(self.ctrl_socket, "get_measurement_%s" % k)()
                rv[k] = res
            except Exception as e:
                self.log.error("Unknown measurement '%s -> %s'" % (k, e))
        return rv
    else:
        self.log.warn("no running or paused radio program; ignore command")
        return None
```

Table 17 - GNU radio get_measurements UPI implementation

get_measurements_periodic(measurement_key_list, collect_period, report_period, num_iterations, report_callback):

This UPI_R function enables to schedule the collection of the measurement values specified in the measurement key list. The measurements are collected every *collect_period* and reported every *report_period*. This is repeated a *num_iterations* number of times. For every report, the *result_callback* is called. The snippet of code presented in Table 18 shows the core of the function implementation in GNU Radio.

```
def get_measurements_periodic(self, measurement_key_list, collect_period,
report_period, num_iterations, report_callback):
   if self.gr_state == RadioProgramState.RUNNING or self.gr_state ==
                                                       RadioProgramState.PAUSED:
        self.init_proxy()
        try:
              thread.start_new_thread(self.ctrl_socket,
    self.get_measurements_periodic_worker, (node, measurement_key_list,
    collect_period, report_period, num_iterations, report_callback,))

        except Exception as e:
              self.log.error("get_measurements_periodic %s" % e)
        return
    else:
        self.log.warn("no running or paused radio program; ignore command")
        return None
```

Table 18 - GNU radio get_measurements_periodic UPI implementation

To implement this functionality by using the ctrl_socket python module, a monitor periodic thread is instantiated containing two timer threads. The first of these timer threads collects the requested measurements every *collect_period*, by calling the get_measurements function discussed above and adds measurements to an internal queue. The second timer thread reads the contents of this queue every *report_period* for delivering measurements to the defined callback.

e. Changing the radio program

This section describes how the radio program, executed in GNU Radio platform, can be changed on-the-fly via the UPI_R interface.

activate_radio_program (radio_program_name)

This function activates the radio program with name grc_radio_program_name loaded into the GNU Radio engine. Each radio program is described by an GRC XML file. Furthermore the processing blocks employed by the radio program to be injected must be available to the GNU Radio engine in a pre-compiled form. The ctrl_socket custom python module is then used to load and activate the radio program. The code snippet presented in Table 19 illustrates the implementation in GNU Radio.

```
def active_radio_program(self, **kwargs):

    # params
    grc_radio_program_name = kwargs['grc_radio_program_name'] # name of the radio
program
```

```
    if self.gr_state == RadioProgramState.INACTIVE:
        self.log.info("Start new radio program")
        self.ctrl_socket = None

        """Launches Gnuradio in background"""
        if self.gr_radio_programs is None or grc_radio_program_name not in
self.gr_radio_programs:
            # serialize radio program to local repository
           self.add_program(**kwargs)
        if self.gr_process_io is None:
            self.gr_process_io = {'stdout': open('/tmp/gnuradio.log', 'w+'),
'stderr': open('/tmp/gnuradio-err.log', 'w+')}
        if grc_radio_program_name not in self.gr_radio_programs:
            self.log.error("Available layers: %s" % ",
".join(self.gr_radio_programs.keys()))
            raise AttributeError("Unknown radio program %s" %
grc_radio_program_name)
        if self.gr_process is not None:
            # An instance is already running
            self.gr_process.kill()
            self.gr_process = None
        try:
            # start GNURadio process
            self.gr_radio_program_name = grc_radio_program_name
            self.gr_process = subprocess.Popen(["env", "python2",
self.gr_radio_programs[grc_radio_program_name]],

stdout=self.gr_process_io['stdout'], stderr=self.gr_process_io['stderr'])
            self.gr_state = RadioProgramState.RUNNING
        except OSError:
            return False
        return True

    elif self.gr_state == RadioProgramState.PAUSED and self.gr_radio_program_name
== grc_radio_program_name:
        # wakeup
        self.log.info('Wakeup radio program')
        self.init_proxy()
        try:
            self.ctrl_socket.start()
            self.gr_state = RadioProgramState.RUNNING
        except xmlrpc.Fault as e:
            self.log.error("ERROR: %s" % e.faultString)
    else:
        self.log.warn('Please deactive old radio program before activating a new
one.')
         return None
```

Table 19 - GNU radio active_radio_program UPI implementation

deactivate_radio_program ()

This function de-activates the radio program. The code snippet presented in Table 20 illustrates the implementation in GNU Radio.

```
def deactive_radio_program(self, **kwargs):

    pause_rp =  bool(kwargs['do_pause'])

    if self.gr_state == RadioProgramState.RUNNING or self.gr_state ==
RadioProgramState.PAUSED:

        if pause_rp:
            self.log.info("pausing radio program")
            self.init_proxy()
            self.ctrl_socket.stop()
            self.ctrl_socket.wait()
            self.gr_state = RadioProgramState.PAUSED

        else:
            self.log.info("stopping radio program")

            if self.gr_process is not None and hasattr(self.gr_process, "kill"):
                self.gr_process.kill()
            if self.gr_process_io is not None and self.gr_process_io is dict:
                for k in self.gr_process_io.keys():
                    #if self.gr_process_io[k] is file and not
self.gr_process_io[k].closed:
                    if not self.gr_process_io[k].closed:
                        self.gr_process_io[k].close()
                        self.gr_process_io[k] = None
            self.gr_state = RadioProgramState.INACTIVE
    else:
        self.log.warn("no running or paused radio program; ignore command")

                return none
```

Table 20 - GNU radio deactive_radio_program UPI implementation

get_running_radio_program ():

This function returns the name of the active radio program. The code snippet presented in Table 21 illustrates the implementation in GNU Radio.

```
def get_running_radio_program(self):
   if self.gr_state == RadioProgramState.RUNNING:
        self.init_proxy()
        return self.gr_radio_programs[self.gr_radio_active]
    else:
        self.log.warn("no running;")
        return None
```

Table 21 - GNU radio get_running_radio_program UPI implementation

f. IEEE 802.11 WiFi Support

In addition to the generic GNU Radio adaptation module, we provide technology-specific GNU Radio adaptation modules. The WiFi (IEEE 802.11) GNU Radio adaptation module (Figure 13) is based on the IEEE 802.11 implementation in GNURadio by Bloessl et al. [7]. We extend the WiFi transceiver to provide the possibility to monitor and change variables (TX power, center frequency, etc.) via an XML-RPC interface. Figure 13 shows a list of UPI_R functions implemented by WiFi GNU Radio adaptation module, i.e. the UPI functions are mapped to the corresponding XML-RPC calls.

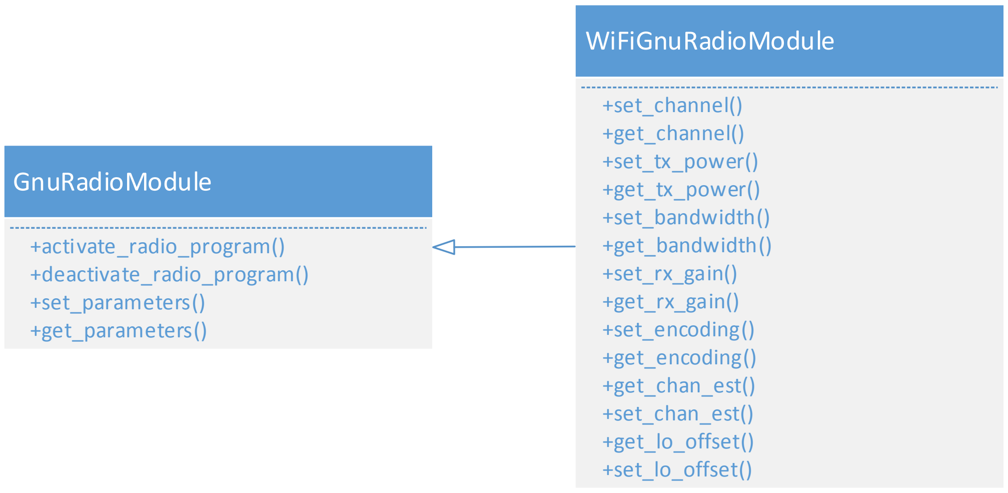


Figure 13. Functionality provided by generic GNU Radio adaptation module (left) and WiFi (IEEE 802.11) adaptation GNU Radio module.

The WiSHFUL framework supports both the ***black-box*** and ***white-box*** approach with respect to GNU Radio. In the black-box approach, the WiFi experimenter will use our IEEE 802.11 GNU Radio implementation and control its behaviour using the well-defined set of UPI_R functions (Figure 13, right). In contrast, the white-box approach allows the experimenter to modify the IEEE 802.11 GNU Radio program (flow graph) to its needs and use the generic UPI_R functions to activate/deactivate the Radio program at runtime.

3.3.5 RAS antenna

This platform represents an example of hardware and software system, which does not provide a radio transceiver, but only focuses on the radiation antenna system. It has been developed during the Open Call 1 as an extension.

The Reconfigurable Antenna Systems (RAS) uses radiating elements that can be programmed for dynamically changing the radiation characteristics of the antenna. Each of the radiating elements has embedded switches that can be toggled to change the current distribution over the antenna, and therefore radiate the energy in different directions. The RAS system provides one omni-directional mode and 8 different directional modes. At 2.4 GHz and 5 GHz, as illustrated in the intermediate and right-most radiation diagram in Figure 14.

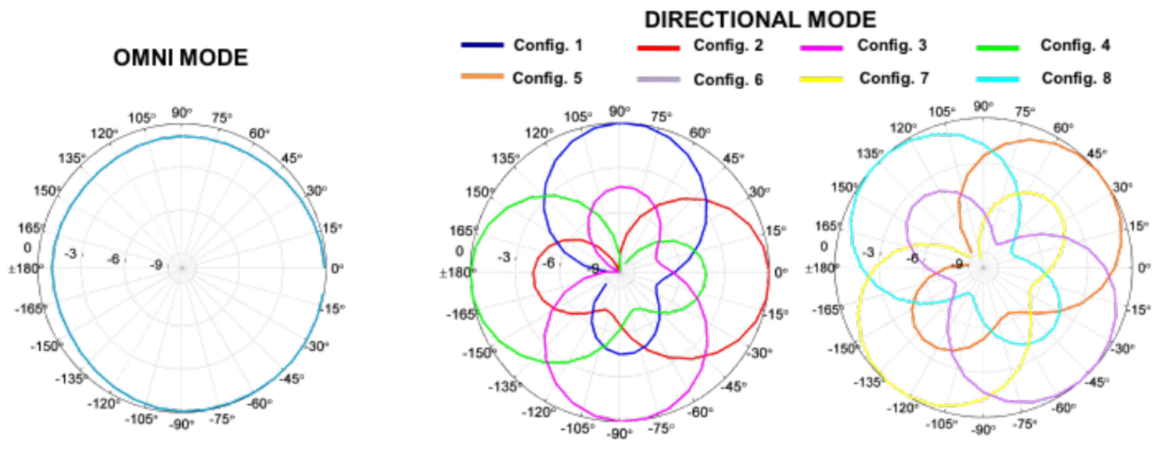


Figure 14 – Radiation patterns that can be excited with RAS antenna

The system has been integrated in WiSHFUL by providing UPI_R functions for configuring the antenna mode. Moreover, in order to connect the antenna to the wireless node, the Open Call 1 extension provides a controller able to control up to 4 different RAS antennas, working both at 2.4GHz and 5GHz bandwidth. In particular, a new function named **set_sas_conf** allows to set the configurations of the smart antennas using the WiSHFUL control framework, specifying the desired band and the configuration of each available smart antenna.

set_sas_conf(band, conf_ant1, conf_ant2, conf_ant3, conf_ant4)

• **band:**	set antenna band to 2.4 or 5 GHz;
• **conf_ant1:**	set configuration for smart antenna 1;
• **conf_ant2:**	set configuration for smart antenna 2 (if present);
• **conf_ant3:**	set configuration for smart antenna 3 (if present);
• **conf_ant4:**	set configuration for smart antenna 4 (if present).

4 Technology-specific UPI_R functions

As described in the introduction, UPI_R functions include **general functions**, which are agnostic of the radio platform technology and architecture, and **technology-specific functions**, which depend on a given MAC/PHY stack but are agnostic of the specific implementation. In this section, we describe the technology-specific UPI_R functions that have been developed during the second year of activities. In particular, we worked on abstractions for the **configuration of WiFi and LTE protocols**, which constitute two representative wireless technologies widely used by the research community. In particular, we developed a WiFi adaptation module, that can be loaded on heterogeneous radio platforms (the Atheros platform, the WMP platform) by exposing the same set of WiFi-related functions and configuration capabilities, and a LTE adaptation module, within one of the Open Call 1 extensions, that can be loaded in a specific radio platform. We plan to integrate other technology-specific adaptation modules, such as a module for low power personal area networks, during Y3.

4.1 WiFi

WiFi is the commercial name of a wide-spread technology for wireless local area networks, standardized as IEEE 802.11. WiFi devices can access a public network, such as the Internet, by means of a wireless Access Point (infrastructure mode) or can be organized into ad-hoc networks, which represent independent islands of nodes. WiFi works on 2.4GHz and 5GHz ISM radio bands and is based on physical layer with several available modulation formats (based on OFDM modulations, in the most recent versions) and on a contention-based MAC protocol called DCF. Moreover, at the MAC layer several management functionalities have been defined for creating networks, managing associations, security, or for managing more advanced networking modes including mesh networks.

In order to support the WiFi technology in the WiSHFUL project, we implemented a software module called *module_wifi*. This section describes the module implementation in terms of UPI_R functions. Examples of WiFi-specific functions are the configuration of IEEE 802.11e EDCA parameters and the configuration of physical layer parameters for different service queues and traffic flows. Figure 15 shows the *WifiModule* class implemented in *module_wifi*, its methods and its relationship using a UML class diagram. The module supports all the linux-compatible WiFi chipsets. Specific class extensions have been designed for other WiSHFUL platforms supporting WiFi technology. The Atheros platform is integrated by means of the *AthModule* class, which is further differentiated as a function of chipset version (IEEE 802.11n and IEEE 802.11b/g/1). The former version is supporting the configuration of a hybrid CSMA/TDMA MAC protocol. The WMP platform is integrated by means of the *WmpModule* class.

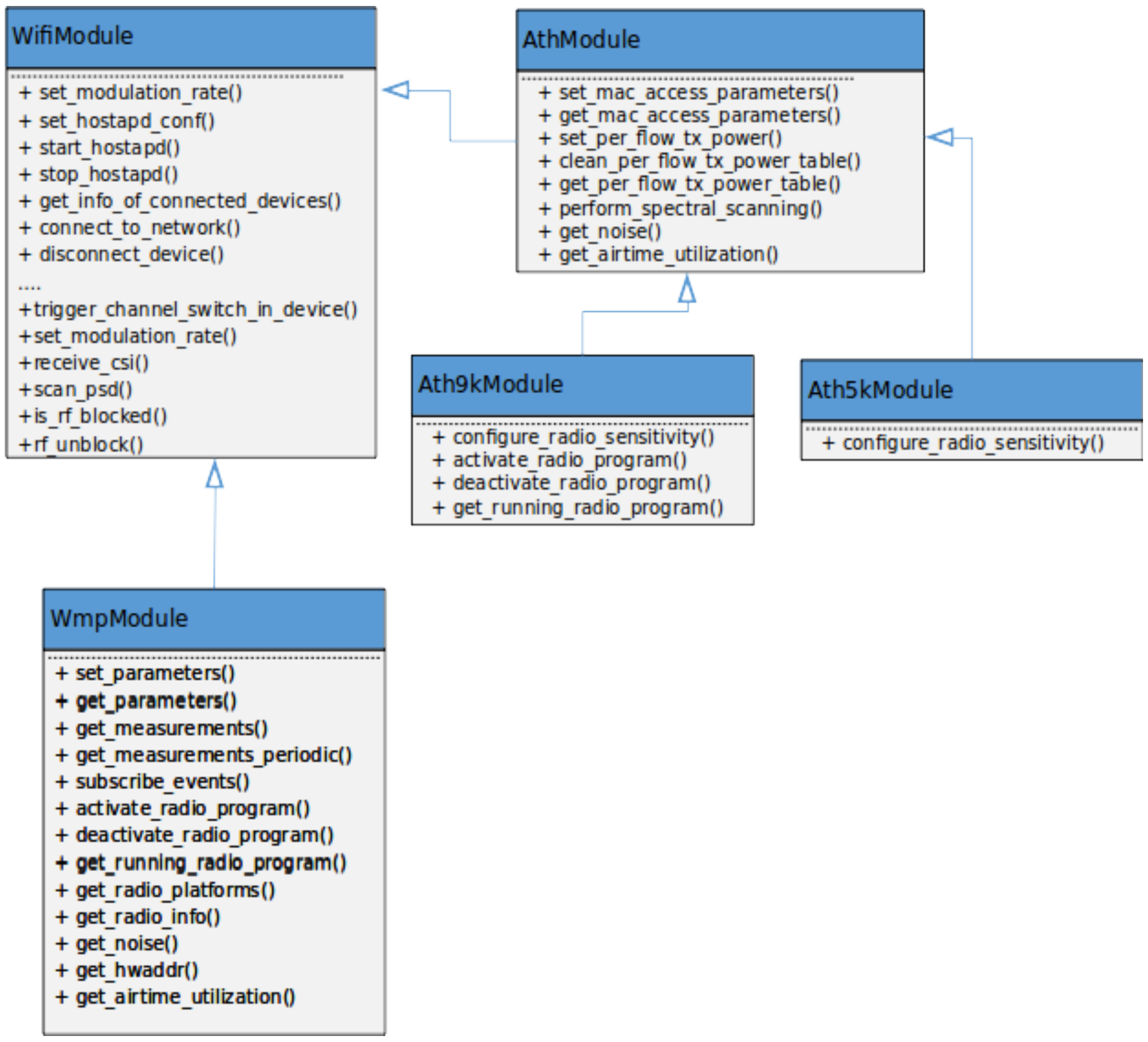

WifiModule
+ set_modulation_rate()
+ set_hostapd_conf()
+ start_hostapd()
+ stop_hostapd()
+ get_info_of_connected_devices()
+ connect_to_network()
+ disconnect_device()
....
+trigger_channel_switch_in_device()
+set_modulation_rate()
+receive_csi()
+scan_psd()
+is_rf_blocked()
+rf_unblock()
AthModule
+ set_mac_access_parameters()
+ get_mac_access_parameters()
+ set_per_flow_tx_power()
+ clean_per_flow_tx_power_table()
+ get_per_flow_tx_power_table()
+ perform_spectral_scanning()
+ get_noise()
+ get_airtime_utilization()
Ath9kModule
+ configure_radio_sensitivity()
+ activate_radio_program()
+ deactivate_radio_program()
+ get_running_radio_program()
Ath5kModule
+ configure_radio_sensitivity()
WmpModule
+ set_parameters()
+ get_parameters()
+ get_measurements()
+ get_measurements_periodic()
+ subscribe_events()
+ activate_radio_program()
+ deactivate_radio_program()
+ get_running_radio_program()
+ get_radio_platforms()
+ get_radio_info()
+ get_noise()
+ get_hwaddr()
+ get_airtime_utilization()

Figure 15. UML class diagram of the WiSHFUL WiFi adaptation module.

The overall list of UPI_R functions in the WiFi module includes: i) parametric functions; ii) measurement functions; iii) generic functions; and iv) specific functions. The parametric functions **specialize the generic black-box set/get functions** to configurations which are specific of the WiFi protocols, such as the configuration of the short and long retry limits. The measurements functions **specialize the generic black-box measurement functions** to WiFi performance metrics, such as the number of data frames transmitted or received packets from a given device. The generic functions include the **technology-independent UPI_R functions supported by WiFi**, such as the setting of the transmission power. Finally, the **WiFi specific functions allow the modification of operation modes defined in the WiFi technology**. The complete list of UPI functions provided by the WiSHFUL *module_wifi* represents for the experimenter a flexible and powerful software tool for testing wireless solutions exploiting the WiFi technology; the software module *module_wifi* can be found in the WiSHFUL project repository at link:

https://github.com/wishful-project/module_wifi

The remaining part of this section lists all the UPI_R functions supported by the WiFi module.

Table 22 shows the UPI_R functions to get/set the WiFi parameters with the relative descriptions.

Function	Description
set_mac_access_parameters	Set the MAC access parameters in IEEE 802.11e (configuration of the access categories).
get_mac_access_parameters	Get the MAC access parameters in IEEE 802.11e (configuration of the access categories).
set_power_management	Set the IEEE 802.11 power management configuration
get_power_management	Get the IEEE 802.11 power management configuration
set_retry_short	Set the retry short (number of transmission attempts before change the modulation rate)
get_retry_short	Get the retry short (number of transmission attempts before change the modulation rate)
set_retry_long	Set the retry long (number of transmission attempts before notify a frame lost to the upper level)
get_retry_long	Get the retry long (number of transmission attempts before notify a frame lost to the upper level)
set_rts_threshold	Set RTS threshold (threshold frame length to activate the RTS algorithm)
get_rts_threshold	Get RTS threshold (threshold frame length to activate the RTS algorithm)
set_fragmentation_threshold	Sets fragmentation threshold (threshold frame length to activate the fragmentation)
get_fragmentation_threshold	Get fragmentation threshold (threshold frame length to activate the fragmentation)

Table 22 - UPI to get/set WiFi parameters

Table 23 reports the UPI_R functions to get the WiFi measurements with the relative descriptions.

Function	Description
get_supported_modes	Get supported WiFi modes
get_supported_swmodes	Get supported WiFi software modes
get_rf_band_info	Get info about supported RF bands
get_ciphers	Get info about supported ciphers
get_supported_wifi_standards	Get info about supported WiFi standards, i.e. IEEE 802.11a/n/g/ac/b
get_wifi_mode	Get the mode of the interface: managed, monitor, master, ad-hoc.
get_wifi_card_info	Get info about the WiFi card: vendor, driver, etc.
get_info_of_connected_devices	Returns information about associated devices.

get_inactivity_time_of_connected_devices	Returns the inactivity time of the associated devices.
get_avg_sigpower_of_connected_devices	Returns the link average signal power of the associated devices.
get_sigpower_of_connected_devices	Returns the link signal power information of the associated devices.
get_tx_retries_of_connected_devices	Returns the transmission retries of the associated device.
get_tx_packets_of_connected_devices	Returns the transmission number of the associated device.
get_tx_failed_of_connected_devices	Returns the transmission failed of the associated device.
get_tx_bytes_of_connected_devices	Returns the transmission byte number of the associated device.
get_tx_bitrate_of_connected_devices	Returns the transmission bitrate of the associated device.
get_rx_bytes_of_connected_devices	Returns the receive byte number of the associated device.
get_rx_packets_of_connected_devices	Returns the receive frames number of the associated device.
get_tdls_peer_connected_device	

Table 23 - UPI to get WiFi measurements

Table 24 reports the UPI_R functions for setting the general (WiFi independent) parameters and the relative descriptions.

Function	Description
set_tx_power	Set the device transmission power
get_tx_power	Get the device transmission power
set_tx_channel	Set the device transmission channel
get_tx_channel	Get the device transmission channel
set_tx_bandwidth	Set the device transmission bandwidth

Table 24 - UPI to configure generic WiFi functions

Table 25 reports the UPI_R functions for setting the WiFi-specific functions and operation modes with the relative descriptions.

Function	Description
set_ap_conf	Set hostapd configuration, the function enables functionality to setting Access Point mode on WiFi device and relative configuration.

start_ap	Start hostapd, the function runs the Access Point.
stop_ap	Stop hostapd, the function stops Access Point.
network_dump	Return the connection information a given interface to some network.
add_device_to_blacklist	Add the given device to the AP's blacklist, i.e. AP will not reply with Probe Reply packets or Association Replies.
remove_device_from_blacklist	Remove the given device from the AP's blacklist.
disconnect_device	Disconnects a given interface (from Access Point side); Send a disassociation request frame to a client STA associated with this AP. See [8-9] for more details.
register_new_device	Register a new station to the Access Point (Register a new STA within the AP, i.e. afterwards the STA is able to exchange data frames). See [8-9] for more details.
disconnect_from_network	Disconnects a given interface (from station side).
connect_to_network	Connects a given interface (in managed WiFi mode) to some network e.g. WiFi network identified by SSID
trigger_channel_switch_in_device	Trigger a channel switch in the device connected to this AP node, see [8-9] for more details.
set_modulation_rate	Sets a fix PHY modulation rate (MCS).
get_csi	Receives Channel State Information (CSI) samples from the WiFi driver (currently supported by Atheros chipsets using ATH9K driver).
scan_psd	Receives Power Spectral Density (PSD) samples from WiFi driver (currently supported by Atheros chipsets using ATH9K driver).
is_rf_blocked	Returns information about rf blocks (Soft Block, Hard Block).
rf_unblock	Turn off the soft block.
sync_by_ap	Allows to synchronize the clock of a node (including an Access Point node) as a function of the timestamp provided by a given reference AP, which acts as a synchronization node.

Table 25 - UPI to management specific WiFi technology

4.2 LTE

LTE (Long Term Evolution) or the E-UTRAN (Evolved Universal Terrestrial Access Network) is the access part of the Evolved Packet System (EPS). The main features of the LTE access network are high spectral efficiency, high peak data rates, short round trip time as well as flexibility in frequency and bandwidth. The LTE access network is a network of base stations, evolved NodeB (eNB), generating a flat architecture. LTE has penetrated in the global market as the key 4G solution, able to deliver speeds ranging from 100Mbps to over 1Gbps per cell.

This section presents how WiSHFUL support LTE technology. The LTE support has been developed within the FLEXFUL Open Call 1 extension and can be summarized in the following phases:

1. Development or extension of the appropriate UPI_R for the radio configuration and control of the LTE BS parameters for any commercial equipment.
2. Development or extension of the appropriate UPI_N for the network configuration of the LTE EPC and the LTE access network. UPI_N components enable the interconnection of the EPC and the BSs in a modular way.
3. Development of the appropriate UPI for LTE User Equipment (UE). UPI_R and UPI_N have been developed in order to control and interconnect the nodes bearing the UE. Moreover, the UPI_R functions are able to monitor the LTE signal parameters (e.g. RSSI, RSRP, RSRQ, SNR, etc.)

However, in this deliverable, we consider only the definition of the UPI_R, being UPI_N functions reported in deliverable D4.4.

4.2.1 Implementation details

The WiSHFUL platform for supporting LTE technology is based on the commercial base station LTE245F provided by ip.access [10], and on a wide variety of different LTE dongles as user equipments. The LTE245F is dual band capable and is available in 3GPP Bands 1/13, 4/13, 2/5 or 7/13. Supporting 2x2 MIMO with an output power of +10dBm per port, the LTE245F provides comprehensive LTE operation in a compact form factor. The dongles are provided by Huawei or ZTE.

FLEXFUL is an extension project that deals with the integration of the existing LTE equipment, residing in several FIRE islands, to the WiSHFUL project. Apart from the NITOS testbed, which is offered as an extension, the same framework is able to handle identical resources that are installed at other WiSHFUL testbeds, e.g. the w-iLab.t testbed. Subsequently, the UPI framework has been appropriately tailored in order to appropriately expose the configuration parameters of the LTE equipment (Base Stations, EPCs, User Equipment), and enable their combination with the rest of the supported resources in a unified fashion. To this aim, FLEXFUL has developed UPI_Ns and UPI_Rs functions for the integration of the LTE equipment in WiSHFUL. The developments has addressed the following phases:

- Separation of the parameters that are altered as either Radio related or Network related, in order to respectively develop the UPI_Rs and UPI_Ns interfaces.
- Development of software blocks enabling the handling of the UPI_Rs interfaces in a similar way like the existing parameters.
- Support of UE configuration has been developed. UPI_Rs and UPI_Ns have been designed and implemented for the different types of UEs that exist in WiSHFUL.

Similar to WiFi technology support, we produced a list of UPI functions for LTE organized in three groups: i) parametric functions; ii) generic functions; and iii) specific functions. The parametric and measurement functions **specialize the generic black-box set/get functions** to configurations, which are specific of the LTE protocols, such as the RACH preambles, or to statistics which are specific of LTE. The generic functions are a set of **UPI_R black-box functions working on technology-independent parameters**, such as the transmission power, which are supported by LTE. Finally, the specific functions are a set of functions able to modify functionalities specific of the LTE technology. The complete list of UPI functions provided by the FLEXFUL project represents for experimenters a software tool useful for speeding-up the definition of experiments on LTE technology. The remaining part of this section reports the description of the UPI functions that support LTE technology.

Table 26 reports the UPI_R functions to get/set the LTE parameters with the relative descriptions.

Parameters	Description
get_tracking_area_code **set_tracking_area_code**	set/get the TAC of the LTE network
get_PLMNID **set_PLMNID**	set/get the PLMNID that is used
get_ENB_name **set_ENB_name**	set/get the eNB name
get_Eci **set_Eci**	set/get the CellID parameter
get_ENB_type **set_ENB_type**	**set/get the eNB type (0 is for home, 1 for macro)**
get_max_ERAB **set_max_ERAB**	set/get the maximum Radio Access Bearers per UE
get_PUSCH_power_control **set_PUSCH_power_control**	set/get the Power Control on the PUSCH channel
get_PDCCH_power_control **set_PDCCH_power_control**	set/get the Power Control on the PDSCH channel
get_SINR_PUCCH_power_contrl **set_ SINR_PUCCH_power_control**	set/get the SINR Power Control on the PUCCH channel
get_HARQ_PUCCH_power_control **set_HARQ_PUCCH_power_control**	set/get the HARQ Power Control on the PUSCH channel
get_Freq_PUSCH_power_control **set_Freq_PUSCH_power_control**	set/get the Power Control on the PUSCH channel for frequency selection
get_PUCCH_SINR_target **set_PUCCH_SINR_target**	set/get the target SINR for the PUCCH channel
get_PUCCH_BLER_target **set_PUCCH_BLER_target**	set/get the target BLER for the PUCCH channel
get_EARNFCN_dl **set_EARNFCN_dl**	set/get the EARFCN for the DL channel (center frequency)
get_EARNFCN_ul **set_EARNFCN_ul**	set/get the EARFCN for the UL channel (center frequency)
get_phy_cell_id **set_phy_cell_id**	set/get the Physical Cell ID
get_PBCH_power_offset **set_PBCH_power_offset**	set/get the power offset for the PBCH channel
get_PSCH_power_offset **set_PSCH_power_offset**	set/get the power offset for the PSCH channel
get_SSCH_power_offset **set_SSCH_power_offset**	set/get the power offset for the SSCH channel
get_num_RACH_preambles **set_num_RACH_preambles**	set/get the number of RACH preambles
get_tx_mode **set_tx_mode**	set/get the transmission mode
get_MCSDl	set/get the MCS used for the DL channel

set_MCSDl	
get_MCSUl **set_MCSUl**	set/get the MCS used for the UL channel
get_CQI_report **set_CQI_report**	set/get the CQi reporting
get_UE_report **set_UE_report**	set/get the UE reporting
get_UE_inactivity_timer **set_UE_inactivity_timer**	set/get the UE inactivity timer
get_cipher_algo **set_cipher_algo**	set/get the ciphering algorithm used

Table 26 – UPI_R to managing the LTE parameters

Table 27 reports the UPI_R functions for tuning the general (LTE independent) parameters and the relative descriptions.

Functions	Description
get_tx_channel	getting the transmission channel
set_tx_channel	setting the transmission channel (ip.access femtocell support bands 7 and 13)
get_tx_bandwidth	getting the bandwidth used for the DL channel
set_tx_bandwidth	setting the bandwidth used for the DL channel (5/10 MHz)
get_rx_bandwidth	getting the bandwidth used for the UL channel
set_rx_bandwidth	setting the bandwidth used for the UL channel (5/10 MHz)
get_tx_power	getting the transmission power of the cell
set_tx_power	setting the transmission power of the cell
register_new_device	attaching a UE to the LTE network
disconnect_device	detaching a UE from the LTE network

Table 27 - LTE UPI_R generic functions

Table 28 reports the UPI_R functions for configuring LTE-specific functionalities and operation modes with the relative descriptions.

Functions	Description
get_admin_state	checking whether the cell is radiating
set_admin_state	turning on/off the cell
restart	restarting the cell
UE_activate	activating the UE connection within a PDN
UE_deactivate	deactivating the UE connection within the PDN

Table 28 - LTE UPI_R specific technology functions

5 Additional software tools enabling the white-box approach

In this section we present a group of software tools available to experimenters for more advanced configurations of the wireless nodes. In particular, they permit to work on WiSHFUL facilities according to the **white-box** approach and offer platform-specific tools for defining new radio programs. Radio programs implement the logic of the so called lower-MAC operations, which are time critical operations for driving he hardware and reacting to hardware signals. Although the WiSHFUL repository provides some exemplary radio programs, based on CSMA and TDMA access schemes, two WiSHFUL platforms (namely, the WMP and TASIC platforms) allow the execution of more general schemes, which can be defined by composing elementary primitives in a high-level programming language, such as a state machine or a chain of operations with time contraints. For both the platforms, we made available a programming environment, devised to edit already available radio programs or define new radio programs from scratch.

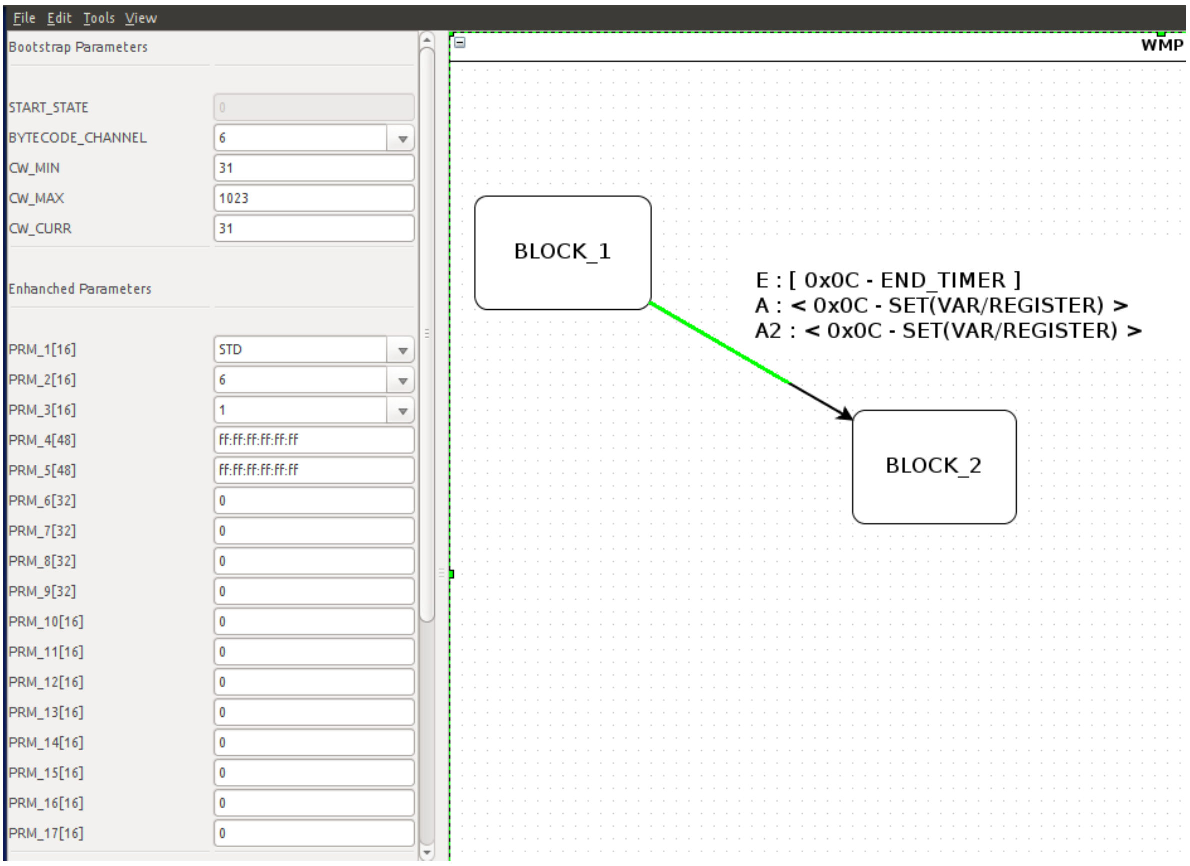
5.1 Editor for Radio Programs of the WMP platform

The Wireless MAC Processor Graphic Editor (WMPE or WMP-Editor) is a tool for the graphical editing and for compiling WMP-specific radio programs. The editor was developed as an extension of a previous tool that was designed for internal use and for the initial WMP release within the FP7 project FLAVIA. The extensions have been dedicated to the enrichment of the programming API and to the improvement of the user interface.

By using this graphical tool, the experimenter can design and implement a new radio program in terms of finite state machines. By the mean of graphical operations it is possible to assemble the finite state machine by adding states and transitions. Transitions can be customized by composing conditions, actions and events, which are the fundamental building blocks, as described in deliverable D3.1. The WMPE contains also a compiler that translates the finite state machine representation in a radio program. In this section we describe the WMPE introducing the different design styles supported by the tool, which is available on the gitHUB [11].

5.1.1 Editor elements and description

The radio program graphical representation provides two main types of editor elements: blocks and transitions. Blocks are graphical boxes representing states of the radio program, while transitions are graphical arrows representing state changes. Each state has a number of outgoing transitions triggered by the occurrence of events and enabled by the verification of one or more optional conditions.


File Edit Tools View
WMP
BLOCK_1
E : [0x0C - END_TIMER]
A : < 0x0C - SET(VAR/REGISTER) >
A2 : < 0x0C - SET(VAR/REGISTER) >
BLOCK_2

Bootstrap Parameters

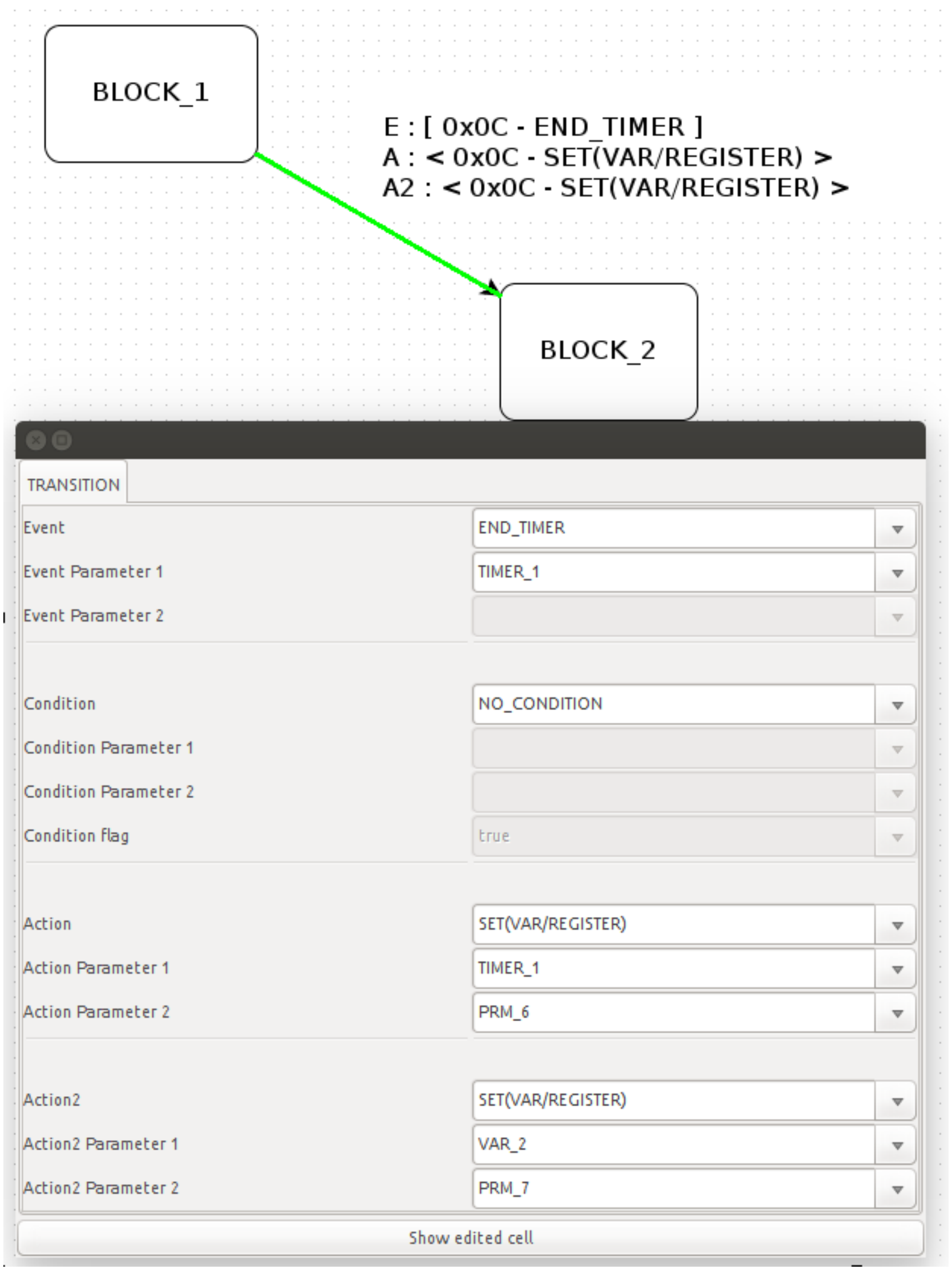
START_STATE	0
BYTECODE_CHANNEL	6
CW_MIN	31
CW_MAX	1023
CW_CURR	31

Enhanched Parameters

PRM_1[16]	STD
PRM_2[16]	6
PRM_3[16]	1
PRM_4[48]	ff:ff:ff:ff:ff:ff
PRM_5[48]	ff:ff:ff:ff:ff:ff
PRM_6[32]	0
PRM_7[32]	0
PRM_8[32]	0
PRM_9[32]	0
PRM_10[16]	0
PRM_11[16]	0
PRM_12[16]	0
PRM_13[16]	0
PRM_14[16]	0
PRM_15[16]	0
PRM_16[16]	0
PRM_17[16]	0

Figure 16 - WMP-Editor Layout

The basic layout of WMP-Editor is shown in Figure 16. It is an all-in-one window organized in two main frames. On the left side there are listed the tuneable parameters for the state machine and the window zoom. On the right frame there is the graphical representation of the state machine as it defined by the programmer. The WMP-Editor uses a simple right-click pop-up to add and edit state blocks. A transition links two states and is created by clicking on the starting state and dragging and releasing on the ending one. By double-clicking the selected transition, a new window opens and transition properties can be edited, using events, conditions and actions (see Figure 17). Two actions can be associated to one transition, by selecting them from the drop-down menu. Radio programs have only one starting state, which can be assigned with a right click on the block state and is displayed with a different shape. Events, conditions and actions are optional for transitions. Events trigger transitions, which result in the state machine evolution. Transitions without events are run immediately because no triggering event has to be waited for. By adding conditions to transitions it is possible to define more complex behavioural logics.


BLOCK_1
E : [0x0C - END_TIMER]
A : < 0x0C - SET(VAR/REGISTER) >
A2 : < 0x0C - SET(VAR/REGISTER) >
BLOCK_2
TRANSITION
Show edited cell

Event	END_TIMER
Event Parameter 1	TIMER_1
Event Parameter 2	
Condition	NO_CONDITION
Condition Parameter 1	
Condition Parameter 2	
Condition flag	true
Action	SET(VAR/REGISTER)
Action Parameter 1	TIMER_1
Action Parameter 2	PRM_6
Action2	SET(VAR/REGISTER)
Action2 Parameter 1	VAR_2
Action2 Parameter 2	PRM_7

Figure 17 – Window editor with transition properties (events, conditions and actions)

In Figure 18 an exemplary finite state machine is shown containing a condition associated to one transition. Consider being on the RX state; a transition occurs only if the RX_END event occurs.

If the RX_QUEUE(FRAME) is true, then the next state is TX_AC, otherwise the new status will be IDLE.

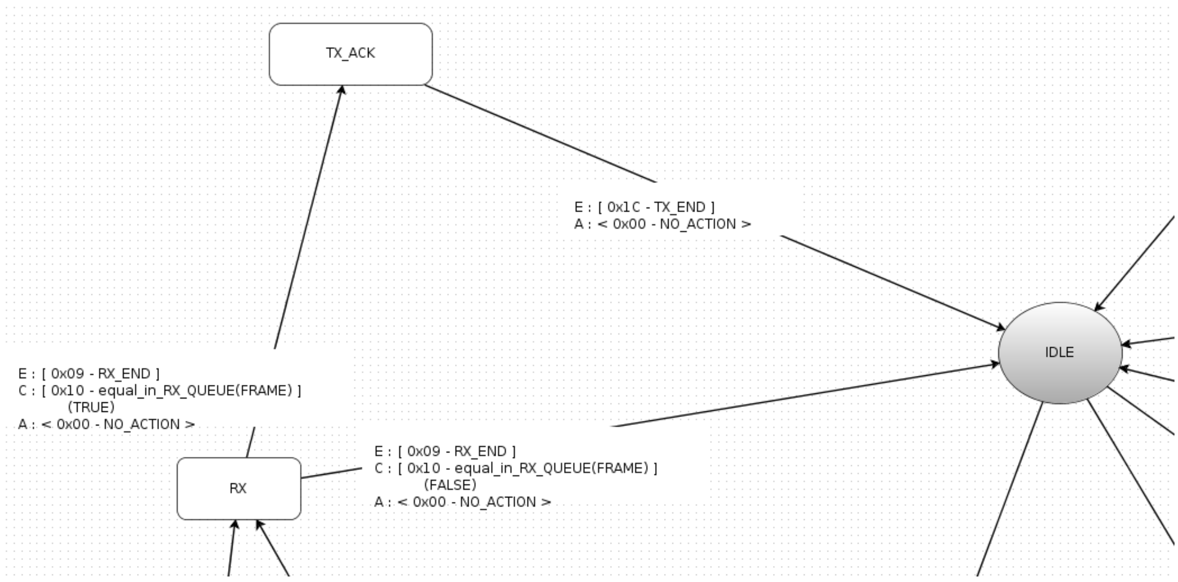


Figure 18 - Example of transitions with event and condition

5.1.2 New Radio Programs

This section describes an example of state machine graphical implementation, namely the Distributed Coordination Function (DCF). This and other examples of radio programs can be found at [12]. Files containing graphical radio programs have the scxml extension and can be edited by the WMP-Editor.

This DCF state machine implements the standard IEEE 802.11 DCF and is presented in Figure 19. Its graphical representation can be logically split into two parts: on the left it is described the ingress chain for dealing with the incoming frames, on the right they are managed the outgoing ones. The initial state is ‘IDLE’ and going to the left or to the right depends on the occurred event: if QUEUE_OUT_UP, then the machine starts the transmit operation mode, if RX_PLCP_END, it starts the receive operation mode. These two operation modes are described below.

During the **reception operation mode**, when in the RX_HEADER state, the RX_MAC_HEADER_END is waited for, indicating the end of the header of the incoming frame. After that, the status of the state machine shifts to CHECK_IF_SCHEDULE_ACK. This state has two outgoing transitions towards the RX state, without being triggered by any event. Depending the condition on the RX_QUEUE(FRAME) an acknowledgement is expected or not.