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

Project Deliverable D3.2

First operational radio control software platform

Contractual date of delivery:	31-12-2015
Actual date of delivery:	23-12-2015
Beneficiaries:	IMINDS, TCD, CNIT, TUB
Lead beneficiary:	CNIT
Authors:	Ilenia Tinnirello (CNIT), Pierluigi Gallo (CNIT), Domenico Garlisi (CNIT), Fabrizio Giuliano(CNIT), Peter Ruckebusch (IMINDS), Ingrid Moerman (IMINDS), Anatolij Zubow (TUB), Mikolaj Chwalisz (TUB), Nicholas Kaminski (TCD), Luiz DaSilva (TCD)
Reviewers:	Anatolij Zubow (TUB), Mikolaj Chwalisz (TUB)
Work package:	WP3 – Radio Control
Estimated person months:	25
Nature:	R
Dissemination level:	PU
Version:	1.6

Abstract:

This deliverable gives a detailed description of the capabilities and performance of the first radio control software platform that follows the Path 1 innovation strategy (black box). It also provides a list of improvements and extensions that are planned to be implemented in Year 2.

Keywords:

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

Executive Summary

This deliverable reports the first operational radio control software platform and provides details about the implementation of the unified programming interface, named UPI_R, that is offered to experimenters from the first open call at the end of Year 1.

We briefly present a general overview of the WiSHFUL radio control software platform, for providing a high-level representation of the software modules available to experimenters and better specifying the role of the platform-agnostic UPI_R interface. The interface represents a unified abstraction for dealing with different experimentation platforms for wireless nodes and ***configuring a specific solution under test at the radio level*****,** able to perform dynamic adaptations**.** As described in D3.1, the experimenters can access the same UPI_R functions for working on the IRIS architecture for SDR platforms, the Time Annotated Instruction Set Computer (TAISC) architecture sensor nodes, and the Wireless MAC Processor (WMP) architecture for wireless network interfaces. Additionally, we also provide the same UPI_R functions for traditional driver architecture of a wireless network interface based on the Atheros chipset, that we call Atheros platform.

We detail the implementation of the UPI_R functions for the different experimentation platforms, by specifying how heterogeneous ***platform-dependent tools*** and ***operating system tools*** have been harmonized by exposing a single unified interface to experimenters. This harmonization allows experimenters to easily prototype a novel wireless solution, by also porting their solutions from one platform to another or from one operating system to another. As a preliminary operation, the UPI_R functions need to ***acquire information*** about the platform capabilities, because different platforms can support different programmability models and configuration parameters. Then, according to the general design described in D3.1, the UPI_R functions are organized into three groups dealing with three main goals: ***configuring*** the experimentation platform, at both the hardware and radio program levels, ***monitoring*** the node and network conditions by accessing all the signals and internal state information of the experimentation platforms, ***adapting*** on-the-fly the node behavior by loading and activating context-specific radio programs.

Finally, we present some control program ***examples***, mainly extracted by the showcases described in D2.3, which have been used for testing the interface and demonstrating its main functionalities. In the Appendix A, the exemplary code snippets are complemented with the list of all the UPI_R tuneable parameters and available measurements.

List of Acronyms and Abbreviations

CSMA	Carrier Sense Multiple Access
GCP	Global Control Program
GITAR	Generic extension for Internet-of-Things ARchitectures
LCS	Local Control Service
MCE	Monitor and Configuration Engine
STA	Wireless Station
TAISC	Time Annotated Instruction Set Computer
TDMA	Time Division Multiple Access
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
VM	Virtual Machine
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 network nodes according to time-varying estimates of the network operating conditions. This document is focused on the presentation of the UPI_R interface, which is one important component of the WiSHFUL unified interfaces (UPI_N, UPI_R, UPI_G, UPI_HC, UPI_M) devised to configure the node behaviour at the lower MAC and PHYHY layers. However, such an interface can be used by experimenters by means of the functionalities provided by the control framework, whose implementation is documented in the companion deliverable D4.2.

The document is organized as follows. In section 2, we provide a high-level description of the WiSHFUL control framework functionalities that can be exploited by the experimenter for **configuring the radio** and the **dynamic radio adaptations** of the solution under test. We remark that for radio configuration we mean both the configuration of the PHY layer and lower MAC layer, i.e. the configuration of the transceiver (transmission formats, spectrum, antennas, etc.) and the configuration of the time-critical access rules for utilizing the wireless resources allocated to the solution under test. The framework allows working on a global control program that can act on single nodes or groups of nodes. The global control program can execute the UPI_R local calls by means of the UPI_G interface and can define some control functions that can migrate on the local nodes for supporting time-critical reconfigurations. Indeed, the global control framework is based on a proactive approach, i.e. it reconfigures the nodes on the basis of the global view of the network by means of timer-driven actions. Since these global views and reconfigurations introduce latency, time-critical adaptations can be supported by exploiting the local monitoring and configuration engines. The rest of the document deals with the UPI_R implementation and is more technical. Section 3 provides the documentation of the implemented UPI functions, while section 4 details the implementation of the UPI functions on the different experimentation platforms available in WiSHFUL. Some code snippets and exemplary function usages are discussed in section 6.

2 General description of WiSHFUL radio control software platform

In this section, we provide a high-level description of the WiSHFUL control framework and its implications for radio control. The framework allows orchestrating the utilization of the UPI_R and UPI_N interface 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. Nodes can be monitored and controlled singularly or in clusters, using the global and local controllers of the WiSHFUL architecture, called ***Monitor and Configuration Engines (MCE)***. The control framework, as detailed in D4.2, provides some basic services for coordinating the UPI_R calls, which basically include **time synchronization** among the nodes (for relying on a common temporal signal), **blocking or non-blocking** interface calls, **time-scheduled** and **remote execution** of UPI_R functions, **loading** of local control programs on the nodes. These services can be exploited for the definition of the control programs, which work on both radio and network control.

2.1 Global, local and hybrid control

The WiSHFUL architecture supports a ***two-tier control hierarchy:*** one global Monitoring and Configuration Engine (MCE) orchestrates several remote MCEs residing on wireless nodes. 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. These two tiers work in a coordinated manner, being orchestrated at the global level. Indeed, global control programs can instantiate local control programs on wireless nodes, performing a sort of *control by delegation,* or can act directly on the wireless nodes in a coordinated manner. Control by delegation is needed when the reconfiguration decisions or the parameters to be monitored have strict time constraints, which cannot be guaranteed by the control network. In fact, the physical channel used for conveying control messages to/from the global controller can be unreliable and introduce some latencies. Since radio performance depends on highly variable network conditions (e.g. channel propagation, fading, interference, access timings, etc.), control by delegation is particularly important for radio control. The architecture also supports hybrid approaches, in which some control operations are managed at the global level, while some others are demanded to wireless nodes. The coordination between global and local control programs is obtained by using the **UPI_HC**, which is the main driver for this hierarchical control.

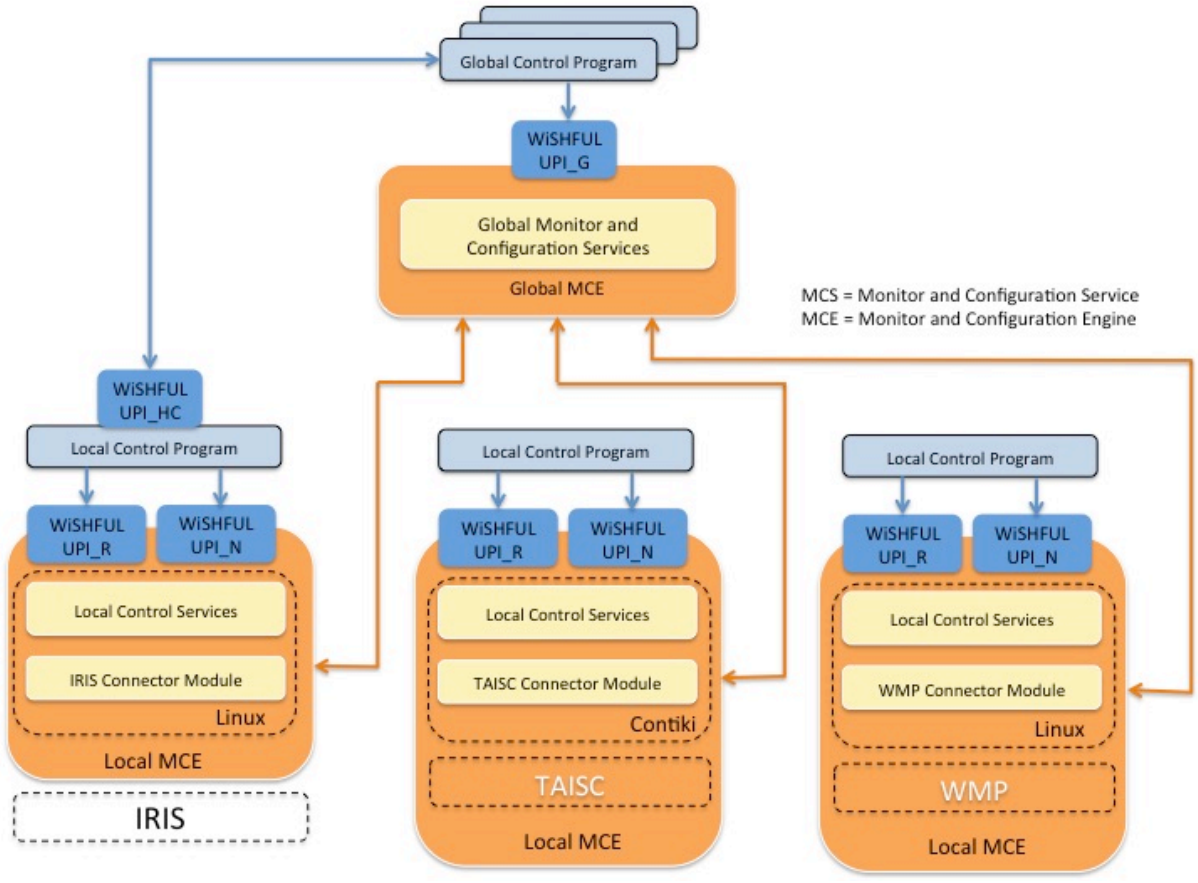


Figure 1 - WiSHFUL architecture, UPIs and supported platforms

Figure 1 illustrates how the WiSHFUL radio control works on three different radio platforms (namely, IRIS, TAISC, WMP). The global MCE runs remotely on a Linux machine and allows implementing node configurations that depend on network-level decisions and can be executed in a time-coordinated fashion among multiple nodes. Each of the WiSHFUL enabled nodes runs a local MCE that offers the same local services and the same UPI_R functions on different radio platforms

(IRIS, TAISC, WMP, ATHEROS) by means of a specific connector module. 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*). These tools, indicated in Figure 1 as Local Control Services, are heterogeneous upon platforms/operating systems and depend on the hardware and software configuration of the device under test. To provide unified interfaces over different technologies, platforms and programmable models, we implemented some **platform-specific connector modules.** These connectors expose the same UPI functions, by linking them to platform-specific implementation. Furthermore, they also disambiguate the platform-specific configuration tools.

2.2 WiSHFUL Connector Module

The Connector Modules, also called adaptation modules in D3.1, are responsible of exposing the same UPI_R functions on completely different hardware and software radio platforms. The module has been designed for achieving two main goals: i) **diverting** platform-independent UPI_R calls to platform-dependent implementation,s and ii) **providing a unified way to deal with a plethora of tools** provided by heterogeneous operating systems (e.g. *iw*, *iwconfig*, *iptable*) or platforms (e.g. *bytecode-manager* for the WMP).

Figure 2 represents a zoomed view of the MCE reported in Figure 1. It illustrates how the connector maps the UPI_R calls on the different radio platforms currently supported by WiSHFUL. The local MCE delegates each UPI_R call to the appropriate UPI_R connector that executes the call using platform-specific sub modules. In general all local MCEs and connector modules are currently implemented in Python, except for Contiki sensor nodes that, next to the Python implementation, also have a native implementation using GITAR. The native implementation is used when the sensor nodes are decoupled. In case they have a Linux host PC (e.g. in testbeds) the Python implementation can be used. This allows to easily prototype wireless solutions for sensors that can also work in real deployments, when the host PCs are not available.

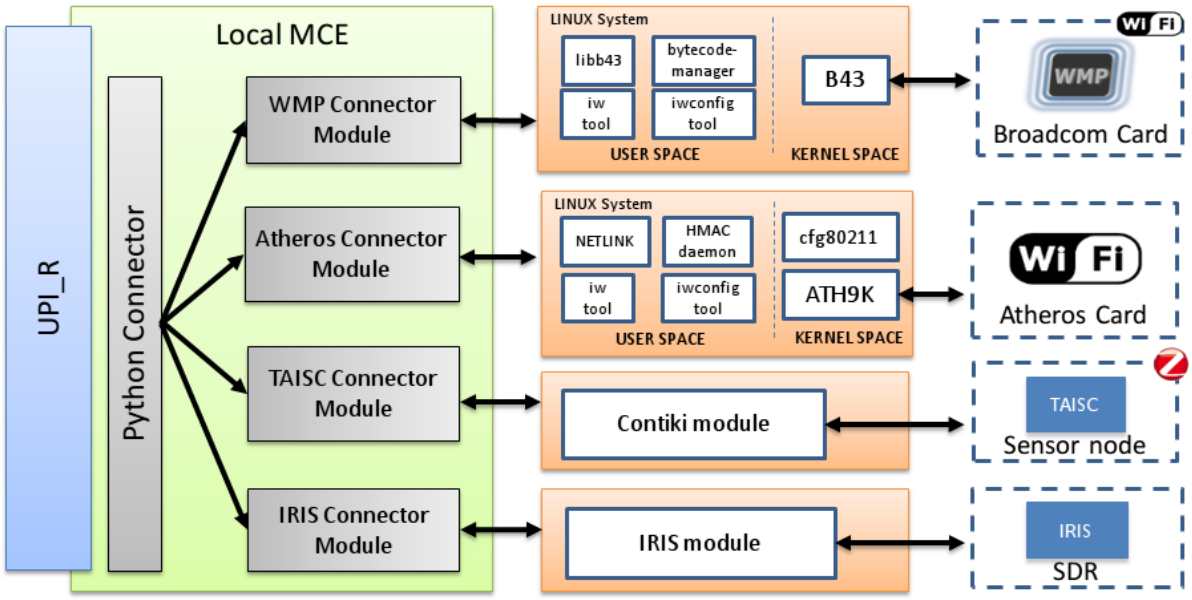


Figure 2 WiSHFUL adaptation modules

The following python code shows how the WiSHFUL connector is implemented in python. It automatically takes the right action depending on the radio platform of the device under test.

```
def set_up_connector(self, connector_module):
    if connector_module == CM_DOT80211_LINUX:
        self._cm = Dot80211_Linux_Impl()
    elif connector_module == CM_WMP_LINUX:
        self._cm = WMP_Linux_Impl()
    else:
        self.log.fatal('Unknown connector module ...')
        os._exit(0)
```

During the initialization phase, the interfaces of the devices under test provide information about their platform type; then, a specific connector is selected by passing to the *set_up_connector()* the right parameter (e.g. CM_DOT80211_LINUX, CM_WMP_LINUX). This function sets the value of self._cm in UPI_RN_Impl class in order to point the platform-specific functions of the UPI_R interface. When the specific module is *connected*, all the calls to UPI_R functions will trigger the execution of the platform-specific code, even if the UPI_R functions are completely generic as shown by the following code.

```
"""
Delegate all function calls to the UPI_R/N implementation in connector module.
"""
def getParameterLowerLayer(self, interface, myargs):
    return self._cm.getParameterLowerLayer(interface, myargs)

def setParameterLowerLayer(self, interface, myargs):
    return self._cm.setParameterLowerLayer(interface, myargs)
```

3 UPI_R implementation

In this section we present the UPI_R implementation, and we report the documentation of the UPI_R usage, automatically created using Sphinx [1] in the Appendix B.

The UPI_R interface is responsible of radio configuration for setting-up wireless links between the nodes and allocate per-flow resources on each link, i.e. for programming the wireless datapath. A programmable datapath gives the possibility to set per-flow specific transmission formats, transmission power, access priorities, etc. As introduced in D3.1, UPI_R range of action includes spectrum allocations, transceiver configurations, link set-up, statistic collections, medium access logic, and virtualization.

As a preliminary operation, the UPI_R functions need to ***acquire information*** about the platform capabilities, because different platforms can support different programmability models and configuration parameters. Then, according to the general design described in D3.1, the UPI_R functions are organized into three groups dealing with three main goals: ***configuring*** the experimentation platform, at both the hardware and radio program levels, ***monitoring*** the node and network conditions by accessing all the signals and internal state information of the experimentation platforms, ***adapting*** on-the-fly the node behavior by loading and activating context-specific radio programs. In the following, we present the UPI_R functions by grouping them according to these four goals:

1) Acquisition of nodes information;
2) Setting nodes capabilities;
3) Monitoring nodes state;
4) Changing the radio program on-the-fly.

UPI_R functions are called exactly in the same way over different platforms and examples about their usage are reported in Section 5. Figure 3 reports the UPI_R class diagram and its member functions, whose platform-specific implementation will be described in the rest of the section. From the figure, we can easily identify the classes used for representing the main WiSHFUL abstractions (as detailed in D3.1) for describing the wireless node behaviors at the radio level: the execution engine, the radio programs, and the radio NIC.

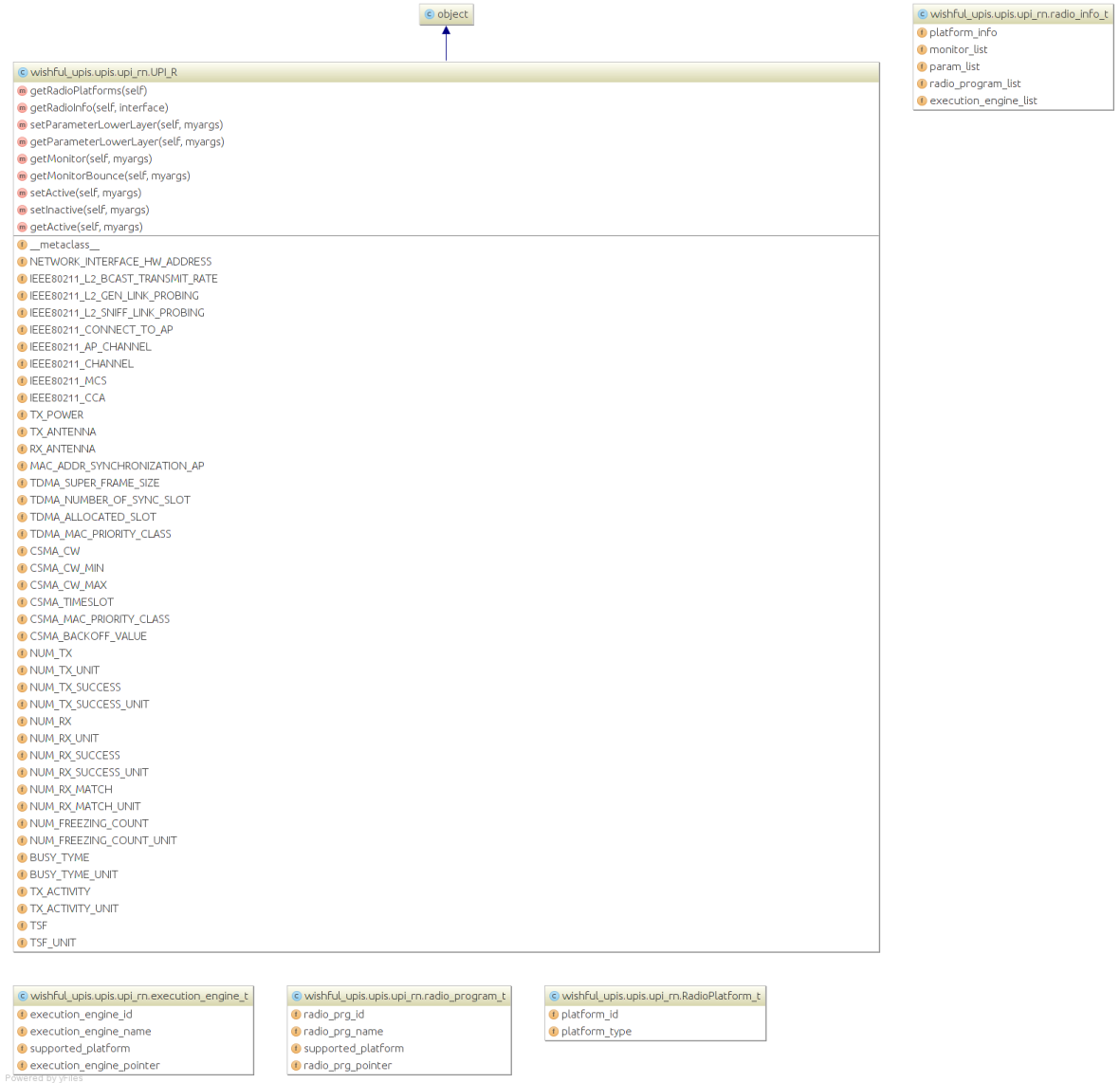


Figure 3 - UML diagram of the UPI_R class and its ancestors

During the implementation phase, few minor details diverged from indications given in deliverable D3.1 [2]. In fact, some changes were required in order to push further the abstractions about radio program concepts, and better clarify their usage or decouple them from the platform in use.

- Two functions names were changed in order to be more significant to experimenters. getRadioNICs() and getRadioNICinfo() have been changed in getRadioPlatforms() and getRadioInfo(). This better clarifies that such functions are related to the radio capabilities of the platform.
- Two functions were removed from UPI_R: inject() and getInject(). This simplifies the radio program activation phase, because they are executed anyway but the experimenter does not

need to explicitly call them. Inject() is now automatically called by setActive(), if needed, i.e. if the radio program was not previously injected on the radio platform.

- A new function was added, namely getMonitorBounce(). This avoids the experimenter to call getMonitor() several times in a row. getMonitorBounce() receives several parameters that specify how often measurements have to be sampled, reported and when the reporting process has to be stopped.
- The support for Atheros cards has been added. This permits to control Atheros tunable parameters and behavior (e.g. controlling when data packets of a particular flow are allowed to be transmitted or not).

Additional features described in deliverable D3.1 will be implemented at theYear 2 of the project, such as setMonitor(), defineEvent(), and related radio capabilities.

3.1 Common implementation

The UPI_R implementation relies both on **generic Linux system tools** (e.g. *iw*, *iwconfig*, etc.) and **platform-specific tools**. The description of platform-specific tools is postponed to the following subsections, which deal with WiSHFUL supported platforms. UPI_R functions adhere to code style for returning values: retval = 0 for SUCCESS, retval = 2 for FAILURE, retval = 1 for expressing function-dependant results.

Linux radio configuration tools

The first configuration tool is the "iw" command, a Linux system tool based on Netlink nl80211. It allows managing wireless interfaces. This permits to:

- establish a basic connection;
- get station statistics;
- modify transmit bitrates;
- set TX power.

Exemplary `iw` tool results are shown in Table 1, where interface statistics are collected.

Table 1 – Example of 'iw' tool result

```
#iw dev wlan0 station dump
station 00:14:a4:62:c8:24 (on wlan0)
        inactive time:  72 ms
        rx bytes:       3272040
        rx packets:     90890
        tx bytes:       0
        tx packets:     0
        tx retries:     0
        tx failed:      0
        signal:         -43 dBm
        signal avg:     -41 dBm
        tx bitrate:     1.0 MBit/s
        authorized:     yes
        authenticated:  yes
        preamble:       long
        WMM/WME:        no
        MFP:            no
        TDLS peer:      no
```

To have such level of detail, `iw` has to be followed by parameters indicating the interface (e.g. '`wlan0`') and 'station dump', to indicate reporting all available information on the station. A

complete list of `iw` parameters can be found in [3]. Another command, `iwconfig` is also specialized for setting and getting wireless-specific network parameters. Its popularity is due to its similarity with `ifconfig`, the most used Linux system tool for interface configuration. However, it is older than iw and its use is deprecated.

3.2 WMP

In this section we describe the implementation of the UPI_R for the WMP platform [4]. After a brief introduction on the WMP platform and its own tools, the UPI_R implementation for the WMP is presented, maintaining the structure proposed in the previous section, distinguishing functions by their goal: loading and activating, monitoring as well as configuring.

The WMP realization that is made available to experimenters using the WiSHFUL framework is implemented on a Broadcom AirForce54G wireless card (see Figure 4). The AirForce54G chipset is built around an 88 MHz processor with 64 registers supporting arithmetic, binary, logic and flow control operations. The other main blocks include a transmission and reception engine supporting 802.11b/g CCK and OFDM encodings and verifying the frame checksum; a set of internal registers for keeping hardware configuration settings; data memory for storing variables and composing arbitrary frames as well as a code memory.

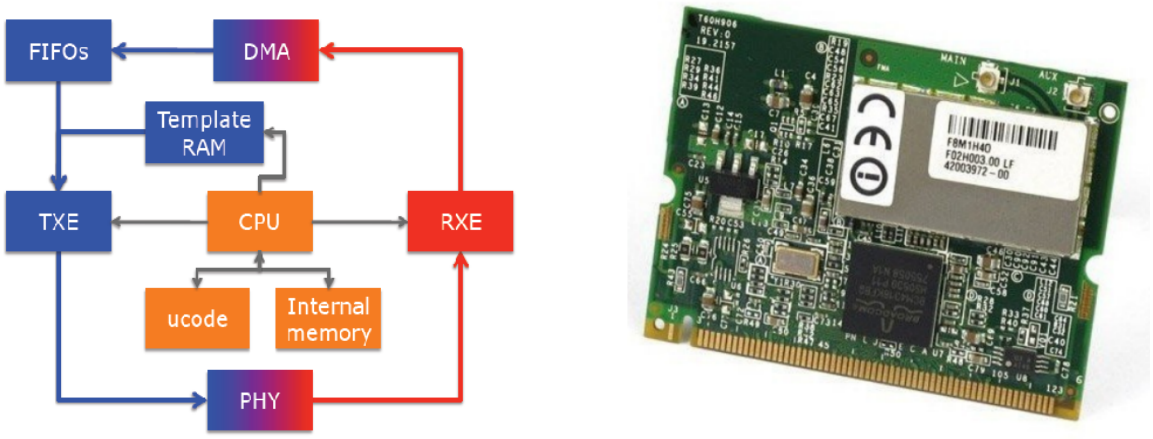


Figure 4 - Broadcom B43 architecture (a) and miniPCI card (b)

The WMP Engine implementation replaces the original card firmware with an assembly code implementing the execution environment able to run the radio program. For supporting the upper-MAC operations and interacting with the other protocol layers, we use the b43 soft-MAC driver, which adapts the Linux internal mac80211 interface to network card. The UPI_R implementation for the WMP includes two WMP-specific tools: the bytecode-manager and the libb43 library.

The bytcode-manager or MAClet-manager is a userspace C# program that implemented the interface with the WMP platform, the main functionality are related to the management of the radio program [5]. To enable the control of the WMP platform, the bytecode-managers uses the libb43 library [6].

The libb43 library is a user space library implemented in Python and C#. It provides a set of functions to read/write the registers and the memory areas of Broadcom b43 wireless cards. The library permits to manage two Linux kernel modules: one for debugfs and the other for B43 driver.

The debugfs is a simple memory-based filesystem, designed specifically to debug Linux kernel code. This helps user space developers to interface their program with the driver and the underlying hardware. Table 2 reports the main functions provides by the libb43 library.

Table 2 – Main functions provides by the libb43 functions

```
read16(reg):
""Do a 16bit MMIO read on physical registers and frame queue (template RAM)"""

read32(reg):
"""Do a 32bit MMIO read on physical registers and frame queue (template RAM)"""

write16(reg, value):
""Do a 16bit MMIO write on physical registers and frame queue (template RAM)"""

write32(reg, value):
"""Do a 32bit MMIO write on physical registers and frame queue (template RAM)"""

shmRead16(routing, offset):
"""Do a 16bit SHM read on general purpose registers and internal memory"""

shmWrite16(routing, offset, value):
"""Do a 16bit SHM write on general purpose registers and internal memory """

shmRead32(routing, offset):
"""Do a 32bit SHM read on general purpose registers and internal memory """

shmWrite32(routing, offset, value):
"""Do a 32bit SHM write on general purpose registers and internal memory """
```

3.2.1 Acquiring nodes information

In this UPI_R group, we can find the two functions getRadioNICs() and getRadioNICinfo(). These two functions allow acquiring the number of wireless interfaces available on the node and their platform information. Nodes may be required to provide the list of available measurements, the type of execution engine, the supported radio programs. This information resides within the node or the wireless network interface card. This can be acquired by the mean of the bytecode-manager. Therefore, getRadioNICs() and getRadioNICinfo() are translated, for the WMP platform, in WMP-specific calls to the bytecode-manager.

3.2.1.1 getRadioNICs()

This function gets available NICs on board, their names, their identifiers and the types of their platforms. This function wraps a call to the bytecode-manager, including its parameters. It checks if the installed card is a Broadcom b43 wireless card and if it supports the WMP platform. Finally it returns this result and the name of the interface.

The following code snippet shows the core implementation of the getRadioNICs(). This uses the python `subprocess` module to exec the bytecode-manager and its argument **"--get-WMP-interface".**

```
def getRadioNICs(self):
   import subprocess
    command = './runtime/connectors/wmp_linux/adaptation_module/src/bytecode-
manager -get-WMP-interface-name'
    nl_output = subprocess.check_output(command, shell=True,
stderr=subprocess.STDOUT)
    flow_info_lines = nl_output.rstrip().split('\n')
    radio_list = [radio_platform_t() for i in range(len(flow_info_lines))]

    for ii in range(len(flow_info_lines)):
        tmp = flow_info_lines[ii]
        items = tmp.split(",")
        radio_list[ii].platform_id = items[0]
        radio_list[ii].platform_type = items[1]

    radio_list_string = [radio_list[ii].platform_id, radio_list[ii].platform_type]
    return radio_list_string
```

3.2.1.2 getRadioNICInfo(interface, param_key)

This function gets the radio capabilities of a given network card NIC_t in terms of supported measurements, parameters and radio programs. This information is organized into data structures, which provide information on the platform type and radio capabilities.

This function, in case of the WMP is a wrapper for the bytecode-manager program, called with the argument **"–get-WMP-interface-capabilities interface".** The following code snippet shows the core of the function implementation.

```
   def getRadioNICInfo(self,interface,param_key):
        import subprocess
       nic_id = interface
       platform = param_key['platform']

       command = './runtime/connectors/wmp_linux/adaptation_module/src/bytecode-
       manager -get-WMP-interface-capabilities' + interface
       nl_output = subprocess.check_output(command, shell=True,
       stderr=subprocess.STDOUT)
       flow_info_lines = nl_output.rstrip().split('\n')
       radio_info = radio_info_t()
       radio_info.radio_info =radio_platform_t()

       radio_info.radio_info.radio_id = nic_id
       radio_info.radio_info.platform = platform

       #get available engines
       exec_engine_current_list_name = []
       exec_engine_current_list_pointer = []
       for row in flow_info_lines:
               exec_engine_current_list_name.append(row['execution_engine_name'])

       exec_engine_current_list_pointer.append(row['execution_engine_pointer'])
           radio_info.execution_engine_list = [execution_engine_t() for i in
       range(len(exec_engine_current_list_name))]
           for ii in range(len(exec_engine_current_list_name)):
               radio_info.execution_engine_list[ii].execution_engine_name =
       exec_engine_current_list_name[ii]
               radio_info.execution_engine_list[ii].execution_engine_pointer =
       exec_engine_current_list_pointer[ii]


       #get available radio program
       radio_prg_current_list_name = []
       radio_prg_current_list_pointer = []
```

```
        for row in flow_info_lines:
                radio_prg_current_list_name.append(row['radio_prg_name'])
                radio_prg_current_list_pointer.append(row['radio_prg_pointer'])
            radio_info.radio_program_list = [radio_program_t() for i in
        range(len(radio_prg_current_list_name))]
            for ii in range(len(radio_prg_current_list_name)):
                radio_info.radio_program_list[ii].radio_prg_name =
        radio_prg_current_list_name[ii]
                radio_info.radio_program_list[ii].radio_prg_pointer =
        radio_prg_current_list_pointer[ii]


        b43 = B43(b43_phy)
        radio_info.monitor_list = b43.monitor_list
        radio_info.param_list = b43.param_list

        ret_lst = []
        ret_lst = {'radio_info' : [radio_info.radio_info.radio_id,
        radio_info.radio_info.platform],
                   'event_list' : [''], 'monitor_list' : [b43.monitor_list],
        'param_list' : [b43.param_list],
                   'radio_prg_list_name' : [radio_prg_current_list_name], '
        radio_prg_list_pointer' : [radio_prg_current_list_pointer],
                   'exec_engine_list_name' : [exec_engine_current_list_name], '
        exec_engine_list_pointer' : [exec_engine_current_list_pointer],
                   }
        return ret_lst
```

3.2.2 Configuring nodes

In this UPI_R group, we can find the two functions setParametersLowerLayer() and getParameterLowerLayer(). These two functions get and set the parameters capabilities of the platform. The complete list of parameters capabilities can be found in the WiSHFUL framework documentation [7] or in the Appendix section of this document. The parameters supported by the WMP platform are identified through the execution of the function getRadioInfo(). The functions of this group use the libb43 library, because the parameters capabilities are totally locate on the platform register and memory area of the wireless card.

3.2.2.1 def setParameterLowerLayer(self, myargs)

Parameters correspond to the configuration registers of the hardware platform and to the variables used in the radio programs. This function (re)set the value(s) of the specified Parameters Radio Capabilities specified in the dictionary argument.
The following snippet code reports the core of the function implementation, where it is shown how the b43.shmWrite16() function is used to set registers and memory area of the Broadcom wireless card.

```
def setParameterLowerLayer(self, myargs):

        b43 = B43(b43_phy)
        write_share = False
        write_gpr = False

        …

        if myargs.has_key(UPI_R.CSMA_CW):
            offset_parameter_share= b43.SHM_EDCFQCUR + b43.SHM_EDCFQ_CWCUR
            offset_parameter_gpr= b43.GPR_CUR_CONTENTION_WIN
            write_share = True
            write_gpr = True
        elif myargs.has_key(UPI_R.CSMA_CW_MIN):
```

```
            offset_parameter_share= b43.SHM_EDCFQCUR + b43.SHM_EDCFQ_CWMIN
            offset_parameter_gpr= b43.GPR_MIN_CONTENTION_WIN
            write_share = True
            write_gpr = True
        elif myargs.has_key(UPI_R.CSMA_CW_MAX):
            offset_parameter_share= b43.SHM_EDCFQCUR + b43.SHM_EDCFQ_CWMAX
            offset_parameter_gpr= b43.GPR_MAX_CONTENTION_WIN
            write_share = True
            write_gpr = True

        if write_share :
            b43.shmWrite16(b43.B43_SHM_SHARED, offset_parameter_share, value)
        if write_gpr :
            b43.shmWrite16(b43.B43_SHM_REGS, offset_parameter_gpr, value)

        ...
```

3.2.2.2 def getParameterLowerLayer(self, myargs)

Parameters correspond to the configuration registers of the hardware platform and to the variables used in the radio programs. This function get the value(s) of the specified Parameters Radio Capabilities specified in the dictionary argument. A list of parameters reports the UPI_R attributes. The following snippet of code shows the core of the function implementation, we use the b43.shmRead16() to read registers and memory are on Broadcom wireless card.

```
def getParameterLowerLayer(self, myargs):

    offset_parameter = myargs['parameters']
          ret_lst = []

    …

    if offset_parameter == UPI_R.CSMA_CW:
        val = b43.shmRead16(b43.B43_SHM_REGS, b43.GPR_CUR_CONTENTION_WIN)
    elif offset_parameter == UPI_R.CSMA_CW_MIN:
        val = b43.shmRead16(b43.B43_SHM_REGS, b43.GPR_MIN_CONTENTION_WIN)
    elif offset_parameter == UPI_R.CSMA_CW_MAX:
        val = b43.shmRead16(b43.B43_SHM_REGS, b43.GPR_MAX_CONTENTION_WIN)
    elif offset_parameter == UPI_R.TDMA_SUPER_FRAME_SIZE :
        if active_slot == 1 :
            val = b43.shmRead16(b43.B43_SHM_SHARED,
    b43.SHM_SLOT_1_TDMA_SUPER_FRAME_SIZE)
            val_2 = b43.shmRead16(b43.B43_SHM_SHARED,
    b43.SHM_SLOT_1_TDMA_NUMBER_OF_SYNC_SLOT)
        else :
            val = b43.shmRead16(b43.B43_SHM_SHARED,
    b43.SHM_SLOT_2_TDMA_SUPER_FRAME_SIZE)
            val_2 = b43.shmRead16(b43.B43_SHM_SHARED,
    b43.SHM_SLOT_2_TDMA_NUMBER_OF_SYNC_SLOT)

        self.log.error('readRadioProgramParameters(): val %s : val_2 %s' %
    (str(val), str(val_2)))
        val = val * val_2
    elif offset_parameter == UPI_R.TDMA_NUMBER_OF_SYNC_SLOT :
        if active_slot == 1 :
            val = b43.shmRead16(b43.B43_SHM_SHARED,
    b43.SHM_SLOT_1_TDMA_NUMBER_OF_SYNC_SLOT)
        else :
            val = b43.shmRead16(b43.B43_SHM_SHARED,
    b43.SHM_SLOT_2_TDMA_NUMBER_OF_SYNC_SLOT)
    elif offset_parameter == UPI_R.TDMA_ALLOCATED_SLOT :
        if active_slot == 1 :
```

```
                val = b43.shmRead16(b43.B43_SHM_SHARED,
        b43.SHM_SLOT_1_TDMA_ALLOCATED_SLOT)
            else :
                val = b43.shmRead16(b43.B43_SHM_SHARED,
        b43.SHM_SLOT_2_TDMA_ALLOCATED_SLOT)
        else:
            self.log.error('readRadioProgramParameters(): unknown parameter')
```

3.2.3 Monitoring nodes

In this UPI_R group, we can find the two functions getMonitor() and getMonitorBounce(). These two functions allow collecting the measurements capabilities from the platform. The complete list of measurements capabilities can be found in the WiSHFUL framework documentation [7] and also in appendix, for the sake of completeness. The measurements supported by the WMP platform are identified through the execution of the function getRadioInfo(). Functions of this group use the libb43 library and the iw tool. In the first case we make a direct reading from the memory areas and registers of the Broadcom wireless card. In the second case we first run the iw tool and then we parse the text result and reformat it according to the needs of the WiSHFUL framework.

3.2.3.1 getMonitor(myargs)

This UPI_R function is able to get the radio measurements thanks to the abstraction of the hardware platform and radio programs in terms of Radio Capabilities. This function get the current value(s) of the Measurements Radio Capabilities specified in the dictionary argument.
The following snippet of code shows the core of the function implementation, we implement two cases, one in which we run the iw tool, and other, where we use the libb43 library.

```
...
...
if  iw_command_monitor:
    cmd_str = 'iw dev ' + interface + ' station dump'
    cmd_output = subprocess.check_output(cmd_str, shell=True,
stderr=subprocess.STDOUT)
    # parse serialized data and create data structures
    flow_info_lines = cmd_output.rstrip().split('\n')
    for ii in range(len(flow_info_lines)):
        tmp = flow_info_lines[ii]
        items = tmp.split("\t")
        if ii == 3:
            rx_packet = items[2]
        elif ii == 5:
            tx_packet = items[2]
        elif ii == 6:
            tx_retries = items[2]
        elif ii == 7:
            tx_failed = items[2]
        else:
            continue
    tx_packet_success = int(tx_packet)
    tx_packet = int(tx_packet) + int(tx_retries) + int(tx_failed)
    if tx_packet_success < 0 :
        tx_packet_success = 0

for ii in range(0,len(key)):
    if key[ii] == UPI_RN.TSF:
        while True :
                v3 = b43.read16(b43.B43_MMIO_TSF_3)
                v2 = b43.read16(b43.B43_MMIO_TSF_2)
                v1 = b43.read16(b43.B43_MMIO_TSF_1)
```

```
                v0 = b43.read16(b43.B43_MMIO_TSF_0)
                test3 = b43.read16(b43.B43_MMIO_TSF_3)
                test2 = b43.read16(b43.B43_MMIO_TSF_2)
                test1 = b43.read16(b43.B43_MMIO_TSF_1)
                if v3 == test3 and v2 == test2 and v1 == test1 :
                    break
        ret_lst.append( (v3 << 48) + (v2 << 32) + (v1 << 16) + v0 )
    if key[ii] == UPI_RN.BUSY_TYME:
        ret_lst.append( b43.shmRead32(b43.B43_SHM_SHARED, b43.BUSY_TIME_CHANNEL)
)
    if key[ii] == UPI_RN.NUM_FREEZING_COUNT:
        ret_lst.append( b43.shmRead16(b43.B43_SHM_SHARED, b43.NUM_FREEZING_COUNT)
)
    if key[ii] == UPI_RN.TX_ACTIVITY:
        ret_lst.append( b43.shmRead32(b43.B43_SHM_SHARED, b43.TX_ACTIVITY) )
    if key[ii] == UPI_RN.NUM_RX:
        total_receive = b43.shmRead16(b43.B43_SHM_SHARED, b43.BAD_PLCP_COUNTER)
#trace failure
        total_receive += b43.shmRead16(b43.B43_SHM_SHARED,
b43.INVALID_MACHEADER_COUNTER)    #trace failure
        total_receive += b43.shmRead16(b43.B43_SHM_SHARED, b43.BAD_FCS_COUNTER)
#trace failure
        total_receive += b43.shmRead16(b43.B43_SHM_SHARED,
b43.RX_TOO_LONG_COUNTER)          #trace failure
        total_receive += b43.shmRead16(b43.B43_SHM_SHARED,
b43.RX_TOO_SHORT_COUNTER)         #trace failure
        total_receive += b43.shmRead16(b43.B43_SHM_SHARED,
b43.RX_CRS_GLITCH_COUNTER)        #trace failure
        total_receive += b43.shmRead16(b43.B43_SHM_SHARED, b43.GOOD_FCS_COUNTER)
#trace success
        ret_lst.append(total_receive)
    if key[ii] == UPI_RN.NUM_RX_SUCCESS:
        ret_lst.append(b43.shmRead16(b43.B43_SHM_SHARED, b43.GOOD_FCS_COUNTER))
    if key[ii] == UPI_RN.NUM_RX_MATCH:
        ret_lst.append(rx_packet)
    if key[ii] == UPI_R.NUM_TX:
        ret_lst.append(tx_packet)
    if key[ii] == UPI_R.NUM_TX_SUCCESS:
        ret_lst.append(tx_packet_success)
...
...
```

3.2.3.2 getMonitorBounce(myargs)

The UPI_R interface is able to get the radio measurements thanks to the abstraction of the hardware platform and radio programs in terms of radio capabilities. The `getMonitorBounce()` function works like `getMonitor()`, but it reports measurements periodically. More specifically, the function accepts three extra arguments and implements two more functionality in comparison to getMonitor(). The additional arguments are SLOT_PERIOD, FRAME_PERIOD, and ITERATION.
Several calls are scheduled every FRAME_PERIOD, which corresponds to the reporting period of measurements. The slot period indicates the periodicity of measurement reading in the interface, while the number of ITERATIONs specifies when this function terminates its periodic measurement reporting. The following code snippet shows the usage of this function and its parameters.

```
def getMonitorBounce(self, myargs):

    import subprocess

    iw_command_monitor = False
    microcode_monitor = False
```

```
    key = myargs ['measurementes']
    slot_period = myargs ['slot_period']
    frame_period = myargs ['frame_period']
    interface = myargs ['interface']

    cumulative_reading = []
    reading = []

    ...

    num_sampling_total = frame_period / slot_period
    num_sampling = 0
    while True :
    ...
       # getMonitor() call
    ...

        cumulative_reading.append(reading)
        reading = []
        num_sampling += 1
        if num_sampling == num_sampling_total :
            #self.log.debug('sampling num %d' % num_sampling)
            #self.log.debug('call result: %s' % str(cumulative_reading))
            return cumulative_reading
        time.sleep(slot_period/1000000.0)

    return cumulative_reading
```

3.2.4 Changing the radio program on-the-fly

In this section we describe the implementation of the UPI_R functions devoted to manage the radio program on WMP. The WMP platform that is currently available in WiSHFUL offers two radio programs, one for CSMA and another for TDMA. We assume that both radio programs are available on a central repository, a database or a file.
Before being called, the radio program needs to be loaded on the microinstruction memory of the platform: being there, it can be executed by the execution engine. An UPI_R function is used to copy the radio program in the microinstruction memory (where it can be executed by the execution environment) and immediately set it as active, i.e. it starts to be executed. `setActive()` is the UPI_R function devoted to inject and run radio programs.

The Figure 5 shows the component element present in the WMP platform. The system consists in the platform architecture and a software program, called Bytecode-Manager to inject, and control the radio program, the software running at the application level, and interacts with the WMP. The Bytecode-Manager is responsible of enabling and loading the radio program on the WMP, and switching the activation status of the injected radio programs.
The WMP platform receives commands from the Bytecode-Manager and performs the requested operation. As shown in Figure 5**Error! Reference source not found.**, two independent radio programs can be stored on the WMP, this design choice enables immediate reconfiguration of the MAC algorithm on wireless cards because the switch between two radio programs in the microinstruction memory requires the time of few instructions. This permits to automatically switch between these two behaviours accordingly to user requests or even periodically.

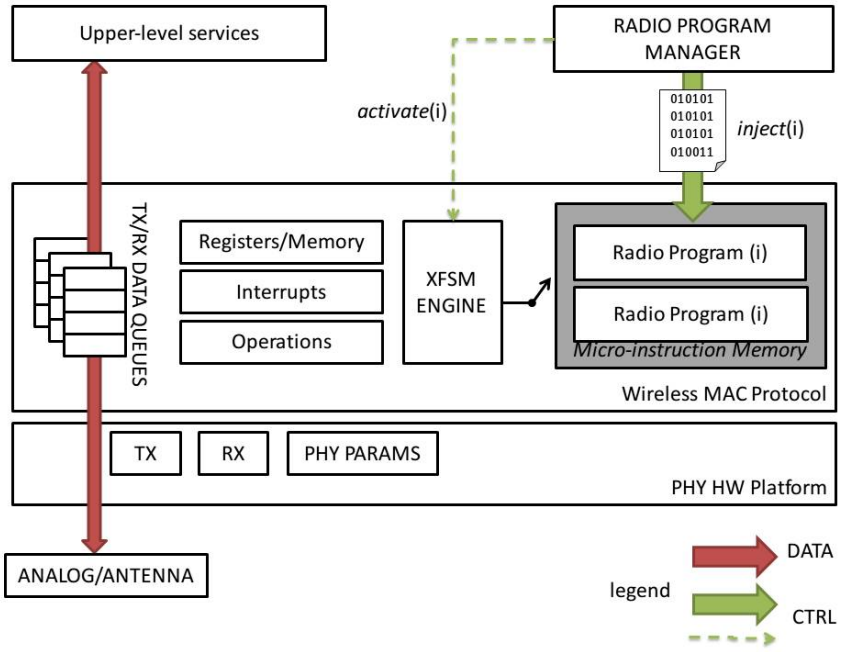


Figure 5 - WMP architecture

3.2.4.1 *setActive(myargs)*

This function activates the passed radio program, one the platform. This operation required that the radio program file description is present on the micro instruction memory of the platform, or in other words, that it is injected. Each radio program injected on platform is associated with an index. This represents the memory slot in which the radio program is copied.

For the Broadcom version of the WMP, only two radio program can be contemporary injected in the microinstruction memory and only one can be active. The implementation of this function provides, two separated phases, in the first phase we check is the passed radio program is already present on microinstruction memory, two cases there may be, the radio program is present, in this case we get the index of the memory slot in which the radio program is stored (see Figure 5). In the second case, the radio program is not present on the microinstruction memory, we get the most old used index, or a new index (if available), and use it to copy the radio program description in the correspondently memory slot. The second phase of function implementation uses the slot memory index, in which the radio program is injected, to active it.

The function uses the python module subprocess to exec the user level bytecode-manager program.

When we inject the radio program, we use the argument **"–l <index>"** that specifies the index of the memory slot in which the radio program is copied, and the argument **"–m <rp_path>"** specifies the file system path where the radio program txt file can be found.

When we active the radio program, we use the argument **"–a <index>"** that specifies the index of the memory slot in which the radio program is stored. The complete function implementation is shown below.

```
def setActive(self,  myargs):
    import subprocess
    radio_program_name = myargs ['radio_program_name']
    position = None
    position = myargs ['position']

   #get the current radio program injected
   injected_radio_program = getInjectedRadioProgram()

   #get the position of the radio program to active, if it is not present, we
```

```
    inject it
    if not(injected_radio_program.has_key(radioProgramName)) :
              radio_program_path = myargs['path']
              if position == None :
            position = getOldPosition()
              command =
    './runtime/connectors/wmp_linux/adaptation_module/src/bytecode-manager -l ' +
    position + ' -m ' + radio_program_path
        nl_output = subprocess.check_output(command, shell=True,
stderr=subprocess.STDOUT)
        flow_info_lines = nl_output.rstrip().split('\n')
        if not(flow_info_lines[5] == 'end load file') :
           return FAILURE
     else :
        position = injected_radio_program[radioProgramName].position


     command = './runtime/connectors/wmp_linux/adaptation_module/src/bytecode-
manager -a ' + position
     nl_output = subprocess.check_output(command, shell=True,
stderr=subprocess.STDOUT)

     flow_info_lines = nl_output.rstrip().split('\n')
     if position == '1' and flow_info_lines[0] == 'Active byte-code 1' :
         return SUCCESS
     elif position == '2' and flow_info_lines[0] == 'Active byte-code 2' :
         return SUCCESS
     else :
         return FAILURE
```

3.2.4.2 *setInactive(myargs)*

When setInactive is called, one of the two radio programs stored in the microinstruction memory is inactivated. The index of the radio program represents the memory slot in which the radio program is stored (see Figure 5**Error! Reference source not found.**). The function uses the python module subprocess to exec the user level bytecode-manager program. The bytecode-manager is called with the argument **"–d <index>"** that specifies the index of the memory slot in which the radio program is stored. The complete function implementation is shown below.

```
def setInactive(self, myargs):
     import subprocess
     position = myargs ['position']

     command = './runtime/connectors/wmp_linux/adaptation_module/src/bytecode-
manager -d ' + position
     nl_output = subprocess.check_output(command, shell=True,
stderr=subprocess.STDOUT)

     flow_info_lines = nl_output.rstrip().split('\n')
     if position == '1' and flow_info_lines[0] == 'Inactive byte-code 1' :
         return SUCCESS
     elif position == '2' and flow_info_lines[0] == 'Inactive  byte-code 2' :
         return SUCCESS
     else :
         return FAILURE
```

3.2.4.3 *getActive(myargs)*

When getActive is called, the index of the current radio program active is reported. The index of the radio program represents the memory slot in which the radio program is stored (see Figure 5 **Error!**

Reference source not found.). This function uses the python module subprocess to exec the bytecode-manager, passing the argument "-v". The complete function implementation is showed below.

```
def getActive(self, myargs):
    import subprocess
    command = './runtime/connectors/wmp_linux/adaptation_module/src/bytecode-
manager -v'
    nl_output = subprocess.check_output(command, shell=True,
stderr=subprocess.STDOUT)

    flow_info_lines = nl_output.rstrip().split('\n')
    items = flow_info_lines[1].split(" ")

    active_radio_program = items[4]
    return active_radio_program
```

3.3 TAISC

This section describes the UPI_R implementation in TAISC, to simplify the structure, after a short introduction on TAISC and the available hardware platforms; this section is organized in four subsections, one for each group of UPI_R functions, as defined in the previous section.

TAISC is implemented on a RM090 [8] sensor node. The RM090 has a 16MHz msp430f5437 CPU, 128 kB ROM and 16kB RAM. It is equipped with the CC2520 IEEE-802.15.4-compliant transceiver. Both the hardware platform and radio are supported in Contiki. TAISC, however, replaces the lower levels of the Contiki network stack (e.g. radio driver, radio dutycycling (RDC) protocol and mac protocol (MAC)). For this purpose, a TAISC specific MAC was created that enables to link TAISC with the upper layers in Contiki.

The UPI_R implementation for TAISC platform can be used in two settings, depending on where the local control program and MCE is executed, as also discussed in D4.2.

- Testbed setting: The local control program and MCE are executed on the Linux host-pc to which the sensor node is connected. For this purpose a connector module was provided, enabling all UPI interactions over ZeroMQ [9] / ZeroRPC [10] as used for the other platforms. The connector module uses a custom python module libContiki to enable UPI usage over serial.
- Decoupled setting: The local control program and MCE are executed on the sensor node in a fully decoupled setting. Now, the WiSHFUL global MCE uses CoAP to remotely execute the UPI functions.

The TAISC architecture, as defined in D3.1, is illustrated in Figure 6. Important for the WiSHFUL UPIs are the control and management interfaces provided by TAISC. They allow reconfiguring parameters, obtaining monitoring information or receiving monitoring events from the radio programs (radio binaries on the figure) by reading/writing to the general purpose RAM (left side of figure). Moreover, new radio programs can be injected by writing to the TAISC ROM (right side of figure) and using the radio binary mgmt functionality.

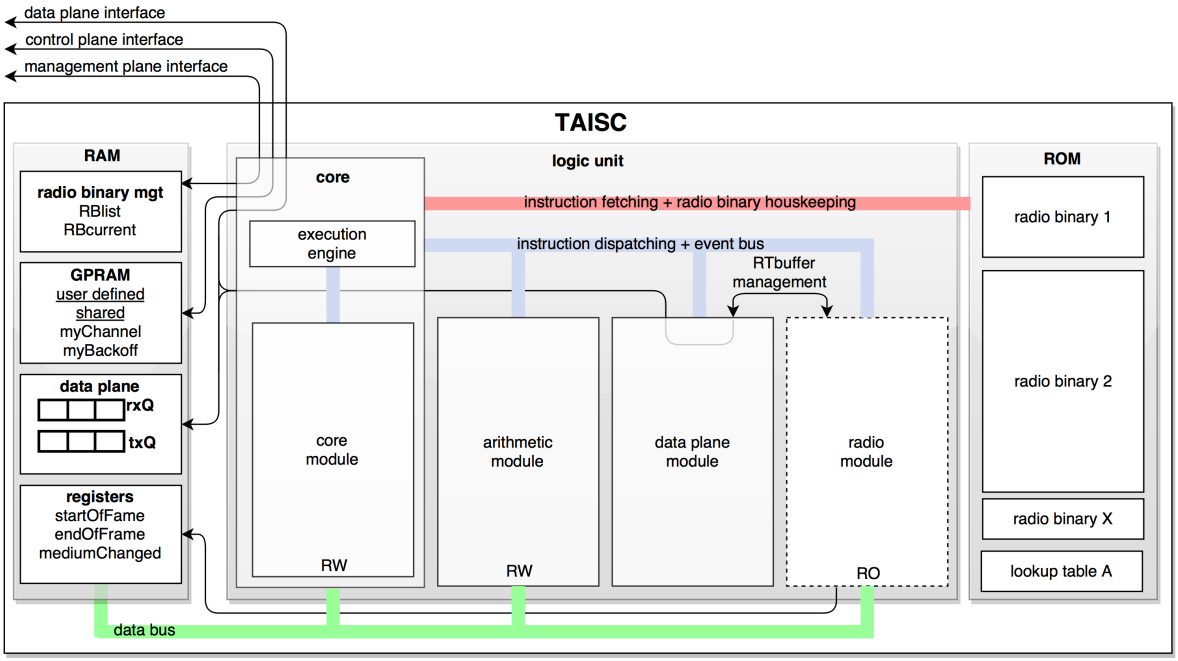


Figure 6 Overview of the TAISC architecture. Left: RAM memory blocks. Middle: logical processing unit. Right: ROM memory.

3.3.1 Acquiring nodes information

Two functions are defined in this UPI group: getRadioNICs() and getRadioNICinfo(). They allow acquiring the following information:

- Number of wireless interfaces of the node and supported platforms
- List the capabilities of the node
  - Supported configuration parameters
  - Measurements node capabilities
  - Execution environment used on the NIC (TAISC)
  - Available radio programs (CSMA, TDMA)

This information resides in a (user-friendly) string format on the Linux local MCE, in the testbed setting, and on the global MCE proxy (see D4.2), in the decoupled setting. To reduce the memory overhead on sensor nodes, the unique names (string format) are replaced by unique IDs (integer format). The IDs are obtained by calculating the 16-bit CRC of the unique name.

3.3.1.1 getRadioPlaftorms()

This function returns information about the available interfaces on the node. The returned NIC_t object contains the name of the interface (i.e. 'wpan 0' for TAISC) and the supported platform type (i.e. TAISC). This function is implemented based on the motelist command as illustrated in the next code snippet.

```
def getRadioNICs(self, myargs):
        import subprocess
        command = 'motelist -c'
        motelist = subprocess.check_output(command, shell=True,
stderr=subprocess.STDOUT).rstrip()
        NICList = []
        for mote in motelist:
                if mote.split(',')[0] == '/dev/rm090':
                        NICList.append(NIC_t('wpan0','TAISC'))
        return NICList
```

3.3.1.2 getRadioPlatformInformation(interface, param_key):

This function returns the radio capabilities of the given interface (NIC_t object) in terms of supported configuration parameters, monitoring measurements and supported radio programs. The information elements used by the UPI_R interface, to manage parameters, measurements and radio program, are organized into data structures, which provide information on the platform type and radio capabilities. When executed, this function return information about available radio capabilities (measurements and parameters, radio_info_t) of each interface (NIC_t) on the available radio programs (radio_prg_t) available for transmissions over the radio interface. The following code snippet illustrates how this is done for TAISC, it uses the custom python module libTAISC to convert unique IDs into unique names.

```
def getRadioNICInfo(self, myargs):
        interface = myargs['interface']
        radio_info = radio_info_t()
        radio_info.NIC_info = interface
        radio_info.param_list = libTAISC.getSupportedParameters(interface)
        radio_info.event_list = libTAISC.getSupportedEvents(interface)
        radio_info.monitor_list = libTAISC.getSupportedMeasurements(interface)
                return radio_info
```

3.3.2 Configuring nodes

There are two UPI_R functions in this group: setParametersLowerLayer() and getParameterLowerLayer(). They allow getting/setting the configuration parameters of the radio platform. The complete list of configuration parameters is listed in the WiSHFUL documentation (http://wirelesstestbedsacademy.github.io/wishful_upis/) and in the Appendix of this document. The parameters supported by TAISC can also be retrieved at runtime using the function getRadioNICinfo(). The functions of this group use the custom libTAISC python module for retrieving TAISC specific (NIC) info and on the custom libContiki Python module for using the UPIs over serial.

3.3.2.1 setParameterLowerLayer(interface,param_key_values)

The parameters that can be configured correspond to the configuration registers of the radio hardware platform and to the variables used in the radio programs executed in TAISC. 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 parameter names are converted to unique IDs for use on the sensor node. The following code snippet illustrates how this is done on the Linux MCE.

```
def setParameterLowerLayer(self, myargs):
        interface = myargs['interface']
        param_keynames_values = myargs['param_key_values']
        param_keyids_values = {}
        for param_key in dct_param_key_values.keys():
                param_key_id = libTAISC.mapUniqueNameOnId(param_key)
                param_keyids_values[param_key_id] = param_key_values[param_key]
       return libContiki.setParameters(interface, dct_param_keyids_values)
```

Internally, another conversion is required to map the unique IDs of configuration parameters on offsets in the TAISC RAM memory. For this purpose an offset table is maintained in the mac wrapper linking TAISC to Contiki, as illustrated in the next code snippet.

```
error_t setParameter(void* value, param_t* p){
```

```
        int i;
        //first find the correct index in the offset table
        for(i=0;i<NUM_PARAMETERS;i++){
                if(offsets[i].uid == p->uid) break;
        }
        //now try to change the parameter in TAISC RAM
        if(i<NUM_PARAMETERS){
                return taisc_set_parameter(value,offsets[i].offset,p->len);
        }
        return FAIL;
}
```

3.3.2.2 getParameterLowerLayer(interface, param_keys)

The parameters correspond to the configuration registers of the radio hardware platform and configuration settings of the radio programs. This function gets the value(s) of the specified parameter keys added to the param_keys list. The possible parameter keys are defined as attributes of the UPI_R class. The parameter names are converted to unique IDs for use on the sensor node. The following code snippet illustrates how this is done on the Linux MCE.

```
def getParameterLowerLayer(self, myargs):
        interface = myargs['interface']
        lst_param_keynames = myargs['param_keys']
        lst_param_keyids = []
        for param_key in lst_param_keynames:
                lst_param_keyids.append(libTAISC.mapUniqueNameOnId(param_key))
        return libContiki.getParameters(interface,lst_param_keyids)
```

Internally, another conversion is required to map the unique IDs of configuration parameters on offsets in the TAISC RAM memory. For this purpose an offset table is maintained in the mac wrapper linking TAISC to Contiki, as illustrated in the next code snippet.

```
void* getParameter(param_t* p){
        int i;
        for(i=0;i<NUM_PARAMETERS;i++){
                if(offsets[i].uid == p->uid){
                        return taisc_wrapper_get_parameter(offsets[i].offset);
                }
        }
        return NULL;
}
```

3.3.3 Monitoring nodes

There are two UPI functions in this group: getMonitor() and getMonitorBounce(). They allow obtaining (*getMonitor*) and collecting (*getMonitorBounce*) the measurements values from the radio platform. The complete list of possible measurement values is listed in the WiSHFUL documentation (http://wirelesstestbedsacademy.github.io/wishful_upis/) and in the Appendix of this document. The measurements supported by TAISC can be retrieved at runtime using the function getRadioNICinfo(). The functions of this group use the custom libTAISC python module for retrieving TAISC specific (NIC) info and on the custom libContiki Python module for using the UPIs over serial.

3.3.3.1 getMonitor(interface, measurement_keys)

This UPI_R function gets the current value(s) of the measurement values specified in the measurement key list. The following snippet of code shows the core of the function implementation in the Linux MCE for TAISC in Contiki.

```
def getMonitor(self, myargs):
        interface = myargs['interface']
        lst_measurement_keys = myargs['measurement_keys']
        lst_measurement_keysids = []
        for key in measurement_keys:
                lst_measurement_keysids.append(libTAISC.mapUniqueNameOnId(key)
        return libContiki.getMonitor(interface,lst_param_keyids)
```

Internally, the getMonitor function is implemented similarly to the getParameter function.

3.3.3.2 getMonitorBounce(interface, measurement_keys, collect_period, report_period ,num_iterations, result_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 following snippet of code shows the core of the function implementation in the Linux MCE for TAISC in Contiki.

```
def getMonitorBounce(self, myargs):
        interface = myargs['interface']
        lst_measurement_keys = myargs['measurement_keys']
        callback = myargs['result_callback']
        collect_period = myargs['collect_period']
        report_period = myargs['report_period']
        num_iterations = myargs['num_iterations']
        lst_measurement_keysids = []
        for key in measurement_keys:
                lst_measurement_keysids.append(libTAISC.mapUniqueNameOnId(key))
        libContiki.getMonitorBounce(interface,lst_measurement_keysids,collect_perio
d,report_period,num_iterations,callback)
        return True
```

To implement this in Contiki for TAISC, some additional steps are required. First off all, a Process must be created that gets the measurements every *collect_period* and generate a report every *report_period*. Secondly, the total memory size required for storing the measurements needs to be considered. Currently 128 bytes are foreseen. Then a timer is scheduled for every *collect_period* after which measurements are added to a report. The reports are sent every *report_period* by counting how many *collect_period* fit in a *report_period*. This repeats for *num_iterations*. The following code snippet illustrates this process:

```
PROCESS_THREAD(monitor_bounce_process, ev, data)
{
  PROCESS_BEGIN();
  getMonitorBounceArgs_t* args= ((getMonitorBounceArgs_t*) data);
  uint32_t collect_period = args->collect_period;
  uint32_t report_period = args->report_period;
  uint8_t num_iterations = args->num_iterations;
  uint8_t num_values = args->num_measurements;
  int i,j,n;
  uint8_t num_collects_in_report = report_period / collect_period;
  measurement_info_t* msrmnt_info_list = args->measurement_info;
  etimer_set(&collect_timer, collect_period);
```

```
  //check if buffer can contain measurements
  if(check_collect_buffer_size(num_values, msrmnt_info_list,
num_collects_in_report)){
       for(i=0;i<num_iterations,i++){
              for(j=0;j<num_collects_in_report;j++){
                     PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&collect_timer));
                     for(j=0;j<num_measurements;j++){
                            //first get a pointer to the parameter
                            measurement_t* m = paramDB_get_parameter(
msrmnt_info_list[i]);
                            //call get function to obtain the value
                            appendMeasurement(msrmnt_info_list[i],m->get(m));
                     }
                     etimer_set(&collect_timer, collect_period);
              }
              //generate the measurement report
              generateMeasurementReport();
       }
  }
  PROCESS_END();
}
```

3.3.4 Changing the radio program on-the-fly

This section describes how the radio program, executed in TAISC, can be changed on-the-fly via the UPI_R interface. TAISC currently provides a CSMA and a TDMA radio program. Both radio programs are pre-installed in TAISC and can be (de-)activated at runtime by using the setActive/setInActive function. The run-time injection of radio programs (e.g. TAISC bytecodes) is planned for Year 2 as a high priority extension and already supported by design.

To allow injection of radio program bytecode, TAISC pre-allocates ROM and RAM memory from the sensor node memory by defining an extra section for this purpose. An interface is provided that allows writing bytecode to the TAISC ROM and loading bytecode from the TAISC ROM. As illustrated in the next code section:

```
/**
* The startChain command makes it possbile to schedule a chain from upper MAC.
* Will return FAIL if the chain is alreay scheduled, not found or
* if the diff value is not in the range of 0..0x7FFFFF
*
* @param  'TAISC_ChainIDT cid'          the chainId reference
* @param  'TAISC_relBigReferenceT diff' relative timestamp to start the chain.
should be in range of 0..0x7FFFFF
* @return 'TAISC_ChainIDT'              the chain_id of the installedChain
*/
TAISC_ChainIDT taiscAPI__taiscControlplane__installChain(void* bytecode, uint16_t
bytecode_len);

/**
* The startChain command makes it possbile to schedule a chain from upper MAC.
* Will return FAIL if the chain is alreay scheduled, not found or
* if the diff value is not in the range of 0..0x7FFFFF
*
* @param  'TAISC_ChainIDT cid'          the chainId reference
* @param  'TAISC_relBigReferenceT diff' relative timestamp to start the chain.
should be in range of 0..0x7FFFFF
*/

error_t         taiscAPI__taiscControlplane__startChain(TAISC_ChainIDT         cid,
TAISC_relBigReferenceT diff);
```

This interface is used by the local monitoring and configuration engine to implement the UPI functions described in the next subsections.

3.3.4.1 *setActive(interface, radio_program_name, radio_program_index)*

This function activates the radio program with name `radio_program_name` installed on index `radio_program_index` in the TAISC ROM. The index of the radio program represents the chain ID in TAISC. The following code snippet illustrates the implementation in the Linux MCE.

```
def setActive(self, myargs):
        interface = myargs['interface']
        radio_program_name = myargs['radio_program_name']
        radio_program_index = myargs['radio_program_index']
        return libContiki.activateRadioProgram(interface,
libTAISC.mapUniqueNameOnId(radio_program_name), radio_program_index)
```

Radio programs are represented in TAISC as chains of instructions. Currently only one radio program / chain can be active. To activate a chain, the load chain function must be executed. If another radio program chain was active, it must be stopped first as illustrated in the next code snippet.

```
error_t activateRadioProgram(uint16_t radio_program_id, uint8_t
radio_program_index){
        radio_program_t rp* = get_radio_program(radio_program_id,
radio_program_index);
        if(rp != NULL){
                if( rp->id != activeRadioProgram->id){
                        taisc_wrapper_stop_chain(activeRadioProgram->index);
                        taisc_wrapper_load_chain(rp->index);
                        activeRadioProgram = rp;
                        return SUCCESS;
                }
        }
        return FAIL;
}
```

3.3.4.2 *setInactive(interface, radio_program_name)*

This function de-activates the radio program with name `radio_program_name`. The following code snippet illustrates the implementation in the Linux MCE.

```
def setInactive(self, myargs):
        interface = myargs['interface']
        radio_program_name = myargs['radio_program_name']
        return libContiki.deActivateRadioProgram(interface,
libTAISC.mapUniqueNameOnId(radio_program_name))
```

To activate de-activate a radio program, the stop chain function must be executed.

```
error_t deActivateRadioProgram(uint16_t radio_program_id){
        if(activeRadioProgram != NULL && activeRadioProgram->id ==
radio_program_id){
                taisc_wrapper_stop_chain(activeRadioProgram->index);
                activeRadioProgram = NULL;
                return SUCCESS;
        }
        return FAIL;
}
```

3.3.4.3 getActive(interface):

This function returns the name of the active radio program. The following code snippet illustrates the implementation in the Linux MCE.

```
def getActive(self, myargs):
        interface = myargs['interface']
        libContiki.getActiveRadioProgram(interface)
        return None
```

In Contiki this translates to the following code.

```
error_t geActiveRadioProgram(){
        if(activeRadioProgram != NULL){
                return activeRadioProgram->id;
        }
        return 0;
}
```

3.4 Iris

This section describes the UPI_R implementation in Iris, to simplify the structure, after a short introduction on Iris and the available hardware platforms; this section is organized in four subsections, one for each group of UPI_R functions, as defined in the previous section.

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 central operation of the Iris framework is then the management of user defined signal processing blocks within so-called engines. These engines may be organized to support operation at either the PHY or MAC layers of radio operation. Finally, user radio programs, composed by their signal processing blocks, are described with a XML document to the managing engines. This XML document specifies which blocks should be loaded, any parameters that should be based to these blocks, and how blocks should be connected. Such XML descriptions are primarily used to configure radios at the beginning of their operation.

The UPI_R implementation for Iris focuses on providing a standard way for users to expose radio control and monitoring functionality. This is achieved by developing an Iris engine to manage processing blocks which supports the operation of the UPI_R. Thus, the implementation of UPI functionality sits on top of an enabling Iris engine as depicted below. To take advantage of the capabilities of WiSHFUL, users must simply specify which parameters should be exposed in the enabling Iris Engine. A slim layer of WiSHFUL UPI functionality is then able to translate UPI calls into an internal socket based Iris call for accomplishing radio control in a standard manner. This approach maintains the flexibility of Iris to realize the needs of users in terms of custom signal processing blocks and chains, while allowing these elements to be exposed to and controlled by the unified WiSHFUL framework.

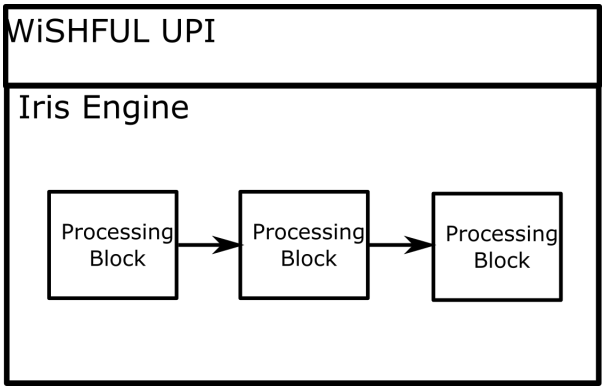


Figure 7 - Organization of WiSHFUL UPI functionality in relation to Iris

3.4.1 Acquiring nodes information

Two functions are defined in this UPI group: getRadioPlatforms() and getRadioiNFO(). They allow to acquire the following information:

- Number of wireless interfaces of the node and supported platforms
- List the capabilities of the node
  - Supported configuration parameters
  - Measurements node capabilities
  - Available radio chains (e.g. CSMA, TDMA)

Recall that the radio elements within Iris are defined in software, meaning that the NICs for Iris are the user defined radio chains. These radio chains, including specification of processing blocks, are defined within XML files that the Iris engine uses to load radio programs. Each such radio chain maintains the unique string formatted name specified by the user.

3.4.1.1 getRadioPlatforms ()

This function returns information about the available interfaces on the node. The returned NIC_t object contains the name of the interface (i.e. the user defined name of the radio program for Iris) and the supported platform type (i.e.Iris). This function makes use of an Iris socket call through the custom libIris python module as illustrated in the next code snippet.

```
def getRadioNICs(self, myargs):
        return libIris.call('getRadioNIC:all')
```

3.4.1.2 getRadioiNFO(interface, param_key):

This function returns the radio capabilities of the given interface (NIC_t object) in terms of supported configuration parameters, monitoring measurements and supported radio programs. The information elements used by the UPI_R interface, to manage parameters, measurements and radio program, are organized into data structures, which provide information on the platform type and radio capabilities. When executed, this function return information about available radio capabilities (measurements and parameters, radio_info_t) of each interface (NIC_t) on the available radio programs (radio_prg_t) available for transmissions over the radio interface. The following code snippet illustrates how this is done for Iris, it uses Iris socket calls through the custom libIris python module to the engine which collects the parameters and events specified for availability to WiSHFUL.

```
def getRadioNICInfo(self, myargs):
        interface = myargs['interface']
```

```
        radio_info = radio_info_t()
        radio_info.NIC_info = interface
        radio_info.param_list = libIris.getSupportedParameters(interface)
        radio_info.event_list = libIris.getSupportedEvents(interface)
        radio_info.monitor_list = libIris.getSupportedMeasurements(interface)
        return radio_info
```

3.4.2 Configuring nodes

There are two UPI_R functions in this group: setParametersLowerLayer() and getParameterLowerLayer(). They allow getting/setting the configuration parameters of the radio platform. The complete list of configuration parameters is listed in the WiSHFUL documentation (http://wirelesstestbedsacademy.github.io/wishful_upis/) and in the Appendix of this document. Note that the parameters supported are ultimately the choice of the user defining the underlying radio chains within Iris. For ease of discussion we will assume a chain that exposes all supported parameters here. Should an unsupported parameter be requested, an error is returned. The parameters supported by Iris radio chains can be retrieved at runtime using the function getRadioPlatforms(). The functions of this group use the custom libIris python module for retrieving Iris specific (NIC) info.

3.4.2.1 setParameterLowerLayer(interface,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 following code snippet illustrates how this is done in Iris.

```
def setParameterLowerLayer(self, myargs):
        interface = myargs['interface']
        param_key_values = myargs['param_key_values']
       return libIris.setParameters(interface, param_key_values)
```

3.4.2.2 getParameterLowerLayer(interface, 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 possible parameter keys are defined as attributes of the UPI_R class. The following code snippet illustrates how this is done in Iris.

```
def getParameterLowerLayer(self, myargs):
        interface = myargs['interface']
        lst_param_keynames = myargs['param_keys']
        return libIris.getParameters(interface,lst_param_keynames)
```

3.4.3 Monitoring nodes

There are two UPI functions in this group: getMonitor() and getMonitorBounce(). They allow obtaining (*getMonitor*) and collecting (*getMonitorBounce*) the measurements values from the radio platform. The complete list of possible measurement values is listed in the WiSHFUL documentation (http://wirelesstestbedsacademy.github.io/wishful_upis/) and in the Appendix of this document. Note that the parameters supported are ultimately the choice of the user defining the underlying radio chains within Iris. For ease of discussion we will assume a chain that exposes all supported parameters here. Should an unsupported parameter be requested, an error is returned. The parameters supported by Iris radio chains can be retrieved at runtime using the function getRadioNICinfo(). The functions of this group use the custom libIris python module for retrieving Iris specific (NIC) info.

3.4.3.1 *getMonitor(interface, measurement_keys)*

This UPI_R function gets the current value(s) of the measurement values specified in the measurement key list. The following snippet of code shows the core of the function implementation in Iris.

```
def getMonitor(self, myargs):
        interface = myargs['interface']
        lst_param_keynames = myargs['measurement_keys']
        return libIris.getMonitor(interface,lst_param_keynames)
```

3.4.3.2 *getMonitorBounce(interface, measurement_keys, collect_period, report_period ,num_iterations, result_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 following snippet of code shows the core of the function implementation in Iris.

```
def getMonitorBounce(self, myargs):
        interface = myargs['interface']
        lst_measurement_keys = myargs['measurement_keys']
        callback = myargs['result_callback']
        collect_period = myargs['collect_period']
        report_period = myargs['report_period']
        num_iterations = myargs['num_iterations']
        lst_measurement_keysids = []
        libIris.getMonitorBounce(interface,lst_measurement_keys,collect_period,repo
rt_period,num_iterations,callback)
        return True
```

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

3.4.4 Changing the radio program on-the-fly

This section describes how the radio program, executed in Iris, can be changed on-the-fly via the UPI_R interface. For the purposes of this discussion we will consider a CSMA and a TDMA radio program. Both radio programs are added to the Iris engine via XML description and can be (de-)activated at run-time by using the setActive/setInActive function.

3.4.4.1 *setActive(interface, radio_program_name, radio_program_index)*

This function activates the radio program with name `radio_program_name` loaded into the Iris engine. If the radio program is not present in the Iris engine, it will be loaded before to active it. Each radio program should be described by an XML file, conforming to standard Iris format, that specifies the elements of the radio program. Furthermore the processing blocks employed by the radio program to be inject must be available to the Iris engine in a precompiled form. The libIris

custom python module is then used to load and active the radio program.The index of the radio program is not necessary in Iris. The following code snippet illustrates the implementation in Iris.

```
def setActive(self, myargs):
        radio_program_name = myargs['radio_program_name']
        xml_file_location = myargs[‘xml_path’]
        return libIris.activateRadioProgram(radio_program_name, xml_file_location)
```

3.4.4.2 setInactive(interface, radio_program_name)

This function de-activates the radio program with name `radio_program_name`. The following code snippet illustrates the implementation in Iris.

```
def setInactive(self, myargs):
        radio_program_name = myargs['radio_program_name']
        return libIris.deActivateRadioProgram(radio_program_name)
```

3.4.4.3 getActive(interface):

This function returns the name of the active radio program. The following code snippet illustrates the implementation in Iris.

```
def getActive(self, myargs):
        return libIris.getActiveRadioProgram()
```

3.5 Atheros-based IEEE 802.11 Subsystem

In this section we describe the implementation of the UPI_R for the Atheros-based IEEE 802.11 platform. After a brief introduction on the platform and its own tools, the UPI_R implementation is presented, maintaining the structure proposed in the previous section, distinguishing functions by their goal: loading and activating, monitoring as well as configuring.

WiSHFUL allows building TDMA on top of today’s off-the-shelf WiFi hardware by providing a flexible and extensible software solution. Currently, we are focusing on the programmability of the downlink whereas in the future also the uplink will be considered. As a platform we use the Atheros-based IEEE 802.11 platform which is a Commercial off-the-shelf (COTS) 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 TDMA downlink channel access like define the number and size of time slots in the TDMA superframe. 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 data plane itself resides in each AP and is controlled by the WiSHFUL runtime system.

The control plane in our design is managed by the either a global or local WiSHFUL controller which takes as input the channel access scheme specified by applications. Any application is self-responsible to decide on how to map the per-flow QoS requirements on the channel access. An example would be to measure which wireless links are suffering from hidden node problem and to assign exclusive time slots for flows requiring high QoS.

The provided centralized coordination for channel access requires a tight time synchronization among APs. In WiSHFUL time synchronization is performed using the wired backhaul network and

hence is not harming the performance of the wireless network. We use the Precise Time Protocol (PTP) giving us an accuracy in microsecond level.

The WiSHFUL agent running on each AP locally is responsible for coordination of channel access as configured by the local or global controller.

WiSHFUL provides hybrid TDMA MAC on commodity devices. So far a connector is provided for Linux boxes using Atheros WiFi chips supporting the Ath9k driver. Specifically, we provide a patch to the Linux Compat Wireless. The implementation was tested with Intel x86 nodes.

The following figure illustrates the components involved. The TDMA mac processor, called HMAC, for the platform is realized as a userspace daemon written in C. It receives control commands like the slot duration and the number of slots in a superframe from either the local WISHFUL execution engine or the remote agent. Therefore, it starts the HMAC daemon. The agent controls (reconfiguration) the HMAC daemon using a message passing system (ZMQ). The task of the daemon is to pass slots configuration information to the wireless network driver using the NETLINK protocol. Moreover, it is responsible to inform the wireless driver about the beginning of each time slots. The patched wireless driver uses the slot configuration information to control which network queues are active and which are freezed. Only packets from active queues are allowed to be sent.

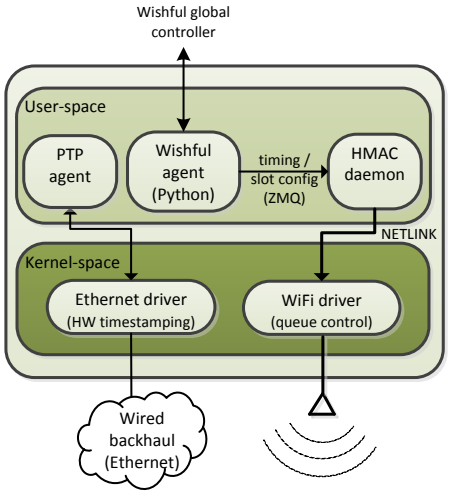
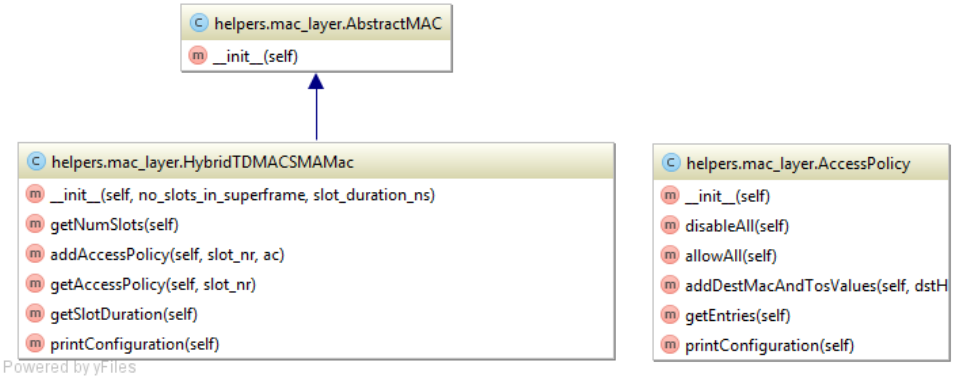


Figure 8. Components of the Atheros-based IEEE 802.11 subsystem.

The UPI functions allow the installation, at runtime reconfiguration and uninstallation of a hybrid TDMA MAC. The mac_profile is an object-oriented representation of the hybrid MAC configuration.



The following example show how to set-up a new hybrid MAC instance using the provided helper classes which are hiding the UPI calls:

```
# create new MAC for each node
mac = HybridTDMACSMAMac(no_slots_in_superframe=7,slot_duration_ns=20e3)
# assign access policy to slot 0
acBE = AccessPolicy()
# MAC address of the link destination
dstHWAddr = '12:12:12:12:12:12'
# best effort
tosVal = 0
acBE.addDestMacAndTosValues(dstHWAddr, tosVal)
slot_nr = 0
mac.addAccessPolicy(slot_nr, acBE)
# assign time guard slot 1
acGuard = AccessPolicy()
acGuard.disableAll() # guard slot
slot_nr = 1
mac.addAccessPolicy(slot_nr, acGuard)
# UPI call for activating the MAC processor
radioHelper.installMacProcessor(node, iface, mac)
```

3.5.1 Acquiring nodes information

Two functions are defined in this UPI group: getRadioPlatforms() and getRadioiNFO(). They allow to acquire the following information:

- Number of wireless interfaces of the node and supported platforms
- List the capabilities of the node
  - Supported configuration parameters, i.e. slot duration, total number of slots, …
  - Measurements node capabilities
  - Available radio chains (e.g. CSMA, hybrid TDMA)

3.5.1.1 getRadioPlatforms ()

This function returns information about the available interfaces on the node. The returned NIC_t object contains the name of the interface (i.e. the user defined name of the radio program for Atheros platform) and the supported platform type (i.e. Atheros).

3.5.1.2 getRadioiNFO(interface, param_key):

This function returns the radio capabilities of the given interface (NIC_t object) in terms of supported configuration parameters, monitoring measurements and supported radio programs. The information elements used by the UPI_R interface, to manage parameters, measurements and radio program, are organized into data structures, which provide information on the platform type and radio capabilities. When executed, this function return information about available radio capabilities (measurements and parameters, radio_info_t) of each interface (NIC_t) on the available radio programs (radio_prg_t) available for transmissions over the radio interface.

3.5.2 Configuring nodes

There are two basic UPI_R functions in this group: setParametersLowerLayer() and getParameterLowerLayer(). They allow getting/setting the configuration parameters of the radio platform. The complete list of configuration parameters is listed in the WiSHFUL documentation (http://wirelesstestbedsacademy.github.io/wishful_upis/) and in the Appendix of this document.

However, we provide helper classes, which allow a more user-friendly way to program the HMAC processor. See functions provided by the RadioHelper class.

3.5.2.1 setParameterLowerLayer(interface,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. This function is used to configure an MAC processor, i.e. setting the number of slots and slot durations in TDMA MAC.

3.5.2.2 getParameterLowerLayer(interface, 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 possible parameter keys are defined as attributes of the UPI_R class.

3.5.2.3 Wireless mode control

In this section we provide a description of implemented UPI that allows to configure the EDCA parameters and per-flow transmission power.

a. EDCA parameters settings

The Enhanced Distributed Channel Access (EDCA) is an extension of the basic DCF defined in IEEE-802.11e standard. It was introduced to support Quality-of-Service. The EDCA mechanism defines four access categories (AC): *AC_BK (background), AC_BE (best effort), AC_VI (video) and AC_VO (voice).* Each AC is characterized by specific values for the access parameters. Different settings allow to statistically prioritizing channel access for one AC over another. As there are four packet queues associated with each AC, it is possible to assure prioritized access for some flows (packets).

There are four parameters that are possible to set for each AC:

- Minimal Contention Window (cwmin) value,
- Maximal Contention Window (cwmax) value,
- Arbitration Inter-frame Space (aifs) value,
- Transmission Opportunity (txop) value

We have implemented UPI functions that allow changing each parameter listed above. In Table 3, example configuration of configuration of EDCA parameters is presented. First, an experimenter has to create *EdcaQueueParameters* object that contains all mentioned parameters. Then with usage of ***setEdcaParameters()*** function she is able to configure EDCA for specific WiFi interface in SUT node.

We have implemented a patch for ath9k driver that allows changing EDCA parameters. The *DebugFs* was used to set/retrieve EDCA parameters to/from proper WiFi interface. We used Atheros AR928X card in our implementation.

Table 3 Example of configuration of EDCA parameters

```
#Define EDCA parameters for each queue
queueParam0 = EdcaQueueParameters(aifs=1,  cwmin=1,  cwmax=3,   txop=9)
queueParam1 = EdcaQueueParameters(aifs=50, cwmin=15, cwmax=63,  txop=4)
queueParam2 = EdcaQueueParameters(aifs=55, cwmin=63, cwmax=127, txop=2)
queueParam3 = EdcaQueueParameters(aifs=99, cwmin=127,cwmax=511, txop=0)

#Configure hardware queues of wireless NIC in node
radioHelper.setEdcaParameters(node=node1, ifname='wlan0', queueId=0,
             qParam=queueParam0)
radioHelper.setEdcaParameters(node=node1, ifname='wlan0', queueId=1,
             qParam=queueParam1)
radioHelper.setEdcaParameters(node=node1, ifname='wlan0', queueId=2,
             qParam=queueParam2)
radioHelper.setEdcaParameters(node=node1, ifname='wlan0', queueId=3,
             qParam=queueParam3)
```

In order to get to know what are currently set EDCA parameters, one needs to use ***getEdcaParameters()*** function. Then retrieved values can be displayed using helper function ***printEdcaParameters()*** (see Table 4).

Table 4 Getting and printing EDCA parameters

```
#Get and print EDCA parameters from node
qParams = radioHelper.getEdcaParameters(node=node1, ifname='wlan0')
radioHelper.printEdcaParameters(node=node1, ifname='wlan0', qParam=qParams)
```

We also provided function ***setFlowTransmissionQueue()***, which can be used to instruct the SUT node to send flow defined by 5-tuple, with speficied hardware queue. An example of such function is presented in Table 5. We exploit the fact that MAC layer is sending packets to proper queue based on Type-of-Service value, thus ***setFlowTransmissionQueue()*** is configuring proper iptable rules to set proper TOS value in packets of defined flow.

Table 5 Sending flows with specified hardware queue

```
#Define flow
flowDesc = FlowDesc(src="192.168.1.4", dst="192.168.1.5")

#Instruct node to send flow with speficied hardware queue
radioHelper.setFlowTransmissionQueue(node=node1, ifname="wlan0", queueId=0,
             flowDesc=flowDesc)
```

b. Per-flow transmission power

Besides the UPI function to set transmission power for a specific interface, a function to set per-flow transmission power is provided.

Example of configuration of per-flow transmission power is presented in Table 6. First, an experimenter has to create ***FlowDesc*** objects that describe the flow with usage of 5-tuple. Second, using UPI function, ***setPerFlowTxPower()***, one can instruct the node to send all packets of a defined flow using a specified transmission power.

We have implemented a patch for ath9k driver that allows setting per-flow transmission power. The *DebugFs* was used for communicating with driver. We used Atheros AR928X card in our

implementation. The Per-Flow TX Power list is an associative table with flow mark being a key and TX power being a value. The function ***setPerFlowTxPower()*** adds new entry in driver's Per-Flow TX Power list and configures iptable rule to set mark in packets of defined flow.

Table 6 Example of configuration of Per-Flow TX Power

```
#Define flows
flow1 = FlowDesc(src="192.168.1.4", dst="192.168.1.5", prot='tcp',
              dstPort="5001")
flow2 = FlowDesc(src="192.168.1.4", dst="192.168.1.5", prot='tcp',
              dstPort="5123")

#Configure TX power for flow
radioHelper.setPerFlowTxPower(node=node0, ifname='wlan0', flow=flow1,
              txPower=10)
radioHelper.setPerFlowTxPower(node=node0, ifname='wlan0', flow=flow2,
              txPower=20)
```

In order to get current per-flow transmission power list from node for specified interface, one needs to use the ***getPerFlowTxPowerList()*** function (see Table 4). The retrieved list can be displayed using helper function ***printPerFlowTxPowerList().***

Table 7 Getting and printing Per-Flow TX Power list

```
#Get and print Per-Flow TX Power list from wireless interface in node
perFlowTxPowerList = radioHelper.getPerFlowTxPowerList(node=node0,
              ifname='wlan0')
radioHelper.printPerFlowTxPowerList(node=node0, ifname='wlan0',
              data=perFlowTxPowerList)
```

Finally, in Table 8, we present UPI function ***cleanPerFlowTxPowerList(),*** that can be used to clear Per-Flow TX Power list of specified wireless interface in SUT node.

Table 8 Cleaning Per-Flow TX Power entry/list

```
#Clean Per-Flow TX Power list of wireless interface in node
radioHelper.cleanPerFlowTxPowerList(node=node0, ifname='wlan0')
```

3.5.3 Monitoring nodes

Specific UPI functions are implemented to get measurements values from the radio platform. The complete list of possible measurement values is listed in the WiSHFUL documentation (http://wirelesstestbedsacademy.github.io/wishful_upis/) and in the Appendix of this document.

3.5.3.1 getMonitor(interface, measurement_keys)

This UPI_R function gets the current value(s) of the measurement values specified in the measurement key list. So far no functionality was implemented.

3.5.4 Changing the radio program on-the-fly

This section describes how the radio program, executed in Iris, can be changed on-the-fly via the UPI_R interface. The application developer can switch between CSMA and a TDMA radio program. Moreover, he can change the behaviour of a running radio program by calling either setParameterLowerLayer() directly or using the helper class.

3.5.4.1 setActive(interface, radio_program_name, radio_program_index)

This function activates the radio program. So far a CSMA and TDMA is implemented and can be activated.

3.5.4.2 setInactive(interface, radio_program_name)

This function de-activates the radio program with name `radio_program_name`. The allowed options are CSMA and TDMA.

4 UPI_M implementation

All management related functions are grouped in the UPI_M interface because they are required for managing protocol software modules at any layer. Moreover, software management requires functionality on both the local and global level. UPI_M deploying, installing and activating software packages. We consider how software package the execution environment to deploy in all the platform. Moreover, we use the UPI_M interface to deploy the radio program on platform. The UPI_M interface will be implemented in the Year 2 of the project, however, we need to implement some part of this features and only for the WMP platform in this year. In this year we do not consider the feature to deploy the software package and the radio program, but we consider the feature to install the execution environment. This section introduces UPI_M, its functions and how the implementation is done. In order to execute the radio program on WMP platform we need install the execution environment on platform, this operation is addressed by the UPI_M interface. Another feature addressed by UPI_M and implemented in this year is the operation to setup the wireless link.

4.1 UPI_M implementation

The UPI_M functions implemented on the WMP platform are two, installExecutionEngine() and initTest(). The first allows to configure the Broadcom wireless card to use the WMP engine (ables to run the radio program), the second allows to establish the links of the wireless network.

installExecutionEngine(param_key):
The architectures of the nodes present in the testbed define an execution environment or execution engine able to run radio programs defined in a high-level programming language. This management function install an execution environment on a node. In the case of WMP platform the execution engine is implemented on the microcode wireless card, the function action is to copy the WMP supported microcode on the precise linux file system directory. Consequently, the reload of microcode on Broadcom wireless card provide the correct execution environment able to run the two radio program provided by the WiSHFUL framework.

initTest(myargs):
This function is a management function used to setup the wireless link. The function performs three different action:

1) restart the broadcom driver module

2) creates infrastructure BSS, the node in which is executed is used such Access Point
3) associate node to infrastructure BSS.

The different action are addressing by the key argument 'OPERATION' in the dictionary data type of arguments. The supported value for the key 'OPERATION' are :

- 'module', used to restart the module;
- 'create-network', used to create the IBSS;
- 'association' used to associate the node to the IBSS.

This function uses the Linux system command "rmmod" and "modprobe" to restart the card module driver, it uses the "hostapd" tool to create the IBSS network on Access Point node, and the "iwconfig" command to creating the wireless link on station node.
The following snippet code shows the core of the function implementation, according with the type of operation, we perform the correct command.

```
            def initTest(self, interface, param_key):
                import subprocess
                key = param_key['OPERATION']

                 if key[ii] == "module":
      ...
      #restart module
      ...

                 if key[ii] == "association":
                     value_1 = param_key['SSID']
                     value_2 = param_key['IP_ADDRESS']
      ...
      #association the station
      ...

                 if key[ii] == "create-network":
                     value_1 = param_key['SSID']
                     value_2 = param_key['IP_ADDRESS']
      ...
      #restart module
      ...
```

5 Examples of control programs using UPIs

The section provides some examples of control programs that can be used by experimenters as a starting point to develop their own controllers. These examples are taken from the control programs used in the showcases discussed in D2.3.

5.1.1 Defining a local controller

The local controller is a code that runs locally, i.e. on the System Under Test (SUT). The implementation of the local controller can be done directly on the node itself (using the LocalManager), or, under the coordination of the global controller. In this last case, it can be defined on the global controller (the definition of the run_local_controller function), then it is remotely sent to SUTs (when run_local_controller is called).

It contains the definition of the following nested functions, which have to be customized by the experimenter:

- def customLocalCtrlFunction(myargs)
- def resultCollector(json_message, funcId)
- def ctrlMsgCollector(json_message)
- def stop_local_controller(mytestbed)

These functions define, respectively, the actions to be run by the local controller, passing parameters that have to be defined by the experimenter, those to manage messages, to … and those to be executed when the local controller has to be stopped.

```
def run_local_controller(mytestbed, disable=0):

    """
    Custom function used to implement local WiSHFUL controller
    """
    def customLocalCtrlFunction(myargs):
```

u

```
    """
    Custom callback function used to receive result values from scheduled calls,
i.e. if you schedule the execution of a
    particular UPI_R/N function in the future this callback allows you to be
informed about any function return values.
    """
    numCBs = {}
    numCBs['res'] = 0
    # use in while to lern if the local logic stopped e.g.
    # while numCBs['res'] < 2:

    def resultCollector(json_message, funcId):
```

e

```
    """
    Custom callback function used to receive control feedback results from local
controllers.
    """
    def ctrlMsgCollector(json_message):
```

t

```
    """
    Stop function used to send stop function to local controllers.
    """
    def stop_local_controller(mytestbed):
```

t

```
    # START MAIN PART

    if disable:
        stop_local_controller(mytestbed)
        return

    # register callback function for collecting results
    mytestbed.global_mgr.setCtrlCollector(ctrlMsgCollector)
    # deploy a custom control program on each node
    CtrlFuncImpl = customLocalCtrlFunction
    CtrlFuncargs =  {'INTERFACE' : ['wlan0']}
    # get current time
    now = get_now_full_second()
    # exec immediately
    exec_time = now + timedelta(seconds=3)
    log.info('Sending local WiSHFUL controller on all nodes - start at : %s',
```

```
str(exec_time))

    #nodes = upi_hc.getNodes()
    for node in mytestbed.wifinodes:
        node.measurement_types.append('FREEZING_NUMBER')
        node.measurement_types.append('CW')
    nodes = mytestbed.nodes

    try:
        # this is a non-blocking call
        callback = partial(resultCollector, funcId=99)
        #isOntheflyReconfig = True
        mytestbed.global_mgr.runAt(nodes, CtrlFuncImpl, CtrlFuncargs,
unix_time_as_tuple(exec_time), callback )
    except Exception as e:
        log.fatal("An error occurred in local controller WiSHFUL sending and
running : %s" % e)

    log.info("Local logic STARTED")
    return
```

5.1.2 Waiting for events

In several cases it is useful to wait for an event. In the following snippet a traffic flow is waited for 100 seconds.

```
"""
Wait for traffic stopped
"""
seconds = 0
traffic_number = get_traffic()
while seconds < 100 and traffic_number != 0:
    traffic_number = get_traffic()
    log.debug('waiting for traffic end , traffic_number = %d (%d)' %
(traffic_number, seconds) )
    time.sleep(1)
    seconds += 1
```

5.1.3 Setting a radio program

A radio program can be set on a specific node by indicating the node index, using the active_TDMA_radio_program function, passing TDMA configuration parameters in the form KEY : value.

```
"""
Set TDMA radio program on nodes
"""
node_index = 0
#superframesize in ms
superframe_size_len = 700 * len(mytestbed.wifinodes) #at modulation rate 24Mbps
for node in mytestbed.nodes:
    active_TDMA_radio_program(node, log,mytestbed.global_mgr,
nodes_NIC_info[node_index])
    tdma_params={'TDMA_SUPER_FRAME_SIZE' : superframe_size_len,
'TDMA_NUMBER_OF_SYNC_SLOT' : len(mytestbed.wifinodes), 'TDMA_ALLOCATED_SLOT':
node_index}
    set_TDMA_parameters(node, log,mytestbed.global_mgr, tdma_params)
    node_index += 1
```

The full set of tuneable/readable parameters can be found in upi_rn.py, groped by PHY, TDMA, and CSMA parameters, where also it is reported the full list of available radio measurements.

5.1.4 Collecting measurements

Measurements to be collected have to be listed in a vector, then the request for these measurements is broadcasted to all nodes indicated in the first parameter. The request for measurements can be done 'one shot', i.e. only once. In most cases, however, it is needed a periodical reading. The UPI permits to indicate the reporting period (how often measurements are reported to the controller), the number of iterations (how many readings), the sampling time (how often measurements are read on the SUT). Meaningful values are those for which my_reporting_period is greater than sampling_time. In fact, obviously, measurements samples have to be collected on the SUT before being reported. The number of iterations implicitly defines the ending time of the measurement process.

```
"""
Start measurement collector for measure FREEZING_NUMBER on all nodes
"""
measurements = (UPI_RN.TSF, UPI_RN.NUM_FREEZING_COUNT)
my_reporting_period=2000000
my_iterations=60
meas_collector.collect_values_from_nodes(nodes=mytestbed.wifinodes,
     node_list=mytestbed.nodes, measurement_types=measurements,
     ucallback=programmable_callback, sampling_time=1000000,
     reporting_period = my_reporting_period, iterations = my_iterations)
```

5.1.5 Configuring Logs

```
"""
Init WiSHFUL framework
"""
FORMAT = '> %(asctime)-15s %(message)s'
logging.basicConfig(format=FORMAT)
log = logging.getLogger()
log.setLevel(logging.DEBUG)
```

This snippet defines the logging format to be used in all logging messages by the controller. The available logging levels are inherited from python, as follows.

```
LEVELS = {'debug': logging.DEBUG,
          'info': logging.INFO,
          'warning': logging.WARNING,
          'error': logging.ERROR,
          'critical': logging.CRITICAL}
```

5.1.6 Defining a testbed

The testbed definition is implemented using OMF. As an extra alternative, a testbed is an object of the class TestbedTopology, to which we need to pass the name of the experiment, the logging object and the number of nodes.

```
"""
*********************
Start framework and get information from Experiment Controller
*********************
"""
mytestbed = TestbedTopology("SC3", log, 4)  # uses 4 nodes and "SC3" name for
```

```
nodes group name
mytestbed.initializeTestbedTopology()        # discovery and allocate nodes
```

When the global controller requires measurement data coming from the nodes, we need to launch the measurement collector. In our case the controller requests information about the available NICs on wireless nodes and their related information. These can be requested to a global manager, which handles nodes in a testbed.

5.1.7 Collecting measurements

```
"""
Start WiSHFUL controller measurement collector
"""
meas_collector = MeasurementCollector(mytestbed, log)
nodes_NIC_info = []
for node in mytestbed.nodes:
    nodes_NIC_info.append(getPlatformInformation(node, log, mytestbed.global_mgr))
```

5.1.8 Monitoring traffic flows

The global controller monitors the number of active traffic flows and waits until one flow is detected. The information about the number of traffic flows is obtained by reading a csv file (get_traffic() reads this file). This configuration file is written by the experiment controller, it is the current interface between the experiment controller and this controller.

```
"""
Wait for some traffic start in testbed
"""
seconds = 0
traffic_number = get_traffic()
while seconds < 100 and traffic_number == 0:
    traffic_number = get_traffic()
    log.debug('waiting some traffic start, traffic_number = %d (%d)' %
(traffic_number, seconds) )
    time.sleep(1)
    seconds += 1
```

6 Improvements and extensions

For the next year of the project, we envision some improvements and extensions on the WiSHFUL software architecture for radio control, which involve the unified UPI_R interface, the monitoring and configuration engines, and the general control framework.

As far as concerns the unified UPI_R interface, we plan to ***complete the implementation*** of the functions defined in D3.1 (basically, by adding the *defineEvent* function), and to add more advanced functions for ***supporting the white-box experimentation approach***. Indeed, in the current software release, experimenters can use non-standard MAC/PHY solutions by choosing among pre-defined radio programs available in the WiSHFUL repository. Although in some cases these programs implement similar protocols (e.g. CSMA or TDMA protocols) which expose the same set of configurable parameters, their implementation is platform-specific and is hidden to experimenters. We plan to extend the UPI_R functions for enabling some forms of more advanced MAC/PHY programming, in which experimenters can define their own protocols.

As far as concerns the WiSHFUL monitoring and configuration engines, during the implementation phase it clearly emerged that MCEs currently work on node-level abstractions, i.e. by enforcing desired configurations on single nodes or groups of nodes. However, network-level decisions could benefits of some ***network abstractions*** to be automatically translated into node-level decisions/measurements by the configuration engines. For example, being the concept of radio link dependent on time-varying channel conditions and node mobility, the estimation of the network topology (i.e. nodes in radio visibility or in interference ranges) can be considered a network abstraction relevant to any control program. Such an abstraction can be built by aggregating different types of measurements and actions on single nodes, and offered as a network-wide service integrated into the WiSHFUL framework (rather than leaving its implementation to experimenters).

Finally, as far as concerns the control framework, we noticed that the control of the solution under test and the definition of a specific experiment could be somehow harmonized for avoiding redundancies of similar functionalities and for efficiency reasons. More into details, WiSHFUL experimenters usually deal with two independent control problems: i) the WiSHFUL control, i.e. defining a network/radio control logic that is not experiment-specific, but implements a novel adaptive wireless solutions able to work in different network conditions and even in real networks; ii) the experiment control, i.e. defining a specific experiment in which it is possible to select different network nodes and to define the dynamics of the traffic flows, the control program and the relevant statistics. Although the two aspects are logically independent, i) the availability of UPI interface for the Experiment controller can facilitate the configuration of the nodes and network topology (e.g. node associations), ii) the availability of Experiment control information in the WiSHFUL controllers can avoid the duplication of some functionalities, such as node associations. Therefore, we plan to investigate on the possibility to add an ***interface between the WiSHFUL control framework and the experiment control framework***.

7 Conclusions

In this deliverable, we describe the first release of the WiSHFUL software architecture for radio control, which comprises of two main components: i) the ***WiSHFUL control framework***, for providing a global view of the solution under test to the experimenter and defining the radio control logics, and ii) the ***unified UPI_R interface*** for monitoring and configuring the radio behaviours of the nodes. Radio control is devised to easily prototype novel wireless solutions, which can include dynamic adaptations of the MAC/PHY of the devices. The solutions can be platform-agnostic, thanks to the abstractions provided by the UPI_R interface, and can work on heterogeneous hardware platforms, including sensors, wireless cards and software-defined-radio.

The implementation has been carried out according to the general design specified in D3.1 (apart from the minor modifications discussed in the first paragraph of section 4), and validated in the showcases presented in D2.3. While in the first design we only considered three different experimentation platforms (IRIS,TAISC and WMP), during the implementation phase the UPI_R functions have been provided for an additional platform, i.e. a commercial wireless card with an Atheros chipset, widely used by experimenters.

The UPI_R implementation is based on the development of ***connector modules***, able to map platform-independent function calls into platform-specific tools and functionalities (which may vary as a function of the platform capabilities). The implementation of the control framework includes the implementation of global and local Monitoring and Configuration Engines, and the implementation of control services to be used for both radio and network control. Because of its generality, the framework is able to support ***different control forms***, including global, local and hybrid decisions; however, for radio control, we expect that local control can be the dominant solution, because of the strict time constraints of MAC/PHY protocols.

Appendix A. Available and tuneable elements for radio programs

A.1 Parameters

```
"""
PHY PARAMETERS
"""
NETWORK_INTERFACE_HW_ADDRESS = "NETWORK_INTERFACE_HW_ADDRESS"
""" MAC address of wireless network interface card"""

IEEE80211_L2_BCAST_TRANSMIT_RATE = "IEEE80211_L2_BCAST_TRANSMIT_RATE"
""" To Measure the transmit rate of generated 802.11 broadcast traffic at full
rate """

IEEE80211_L2_GEN_LINK_PROBING = "IEEE80211_L2_GEN_LINK_PROBING"
""" To send out 802.11 broadcast link probes """

IEEE80211_L2_SNIFF_LINK_PROBING = "IEEE80211_L2_SNIFF_LINK_PROBING"
""" To Receive 802.11 broadcast link probes """

IEEE80211_CONNECT_TO_AP = "IEEE80211_CONNECT_TO_AP"
""" Connect to ap """

IEEE80211_AP_CHANNEL = "IEEE80211_AP_CHANNEL"
""" IEEE 802.11 PHY channel """

IEEE80211_CHANNEL = "IEEE80211_CHANNEL"
""" IEEE 802.11 PHY channel """

IEEE80211_MCS = "IEEE80211_MCS"
""" IEEE 802.11 Modulation and Coding Scheme (MCS) index value """

IEEE80211_CCA = "IEEE80211_CCA"
""" IEEE 802.11 Clear channel assessment (CCA) threshold """

TX_POWER = "TX_POWER"
""" Transmission power in dBm """

TX_ANTENNA = "TX_ANTENNA"
""" Antenna number selected for transmission """

RX_ANTENNA = "RX_ANTENNA"
""" Antenna number selected for reception """

MAC_ADDR_SYNCHRONIZATION_AP = "MAC_ADDR_SYNCHRONIZATION_AP"
""" To set the Access Point MAC address used for synchronizing the TSF timer """
"""
END PHY PARAMETERS
"""

"""
TDMA RADIO PROGRAM PARAMETERS
"""
TDMA_SUPER_FRAME_SIZE = "TDMA_SUPER_FRAME_SIZE"
""" Duration of a periodic TDMA frame """

TDMA_NUMBER_OF_SYNC_SLOT = "TDMA_NUMBER_OF_SYNC_SLOT"
""" Number of temporal slots included in a TDMA frame """

TDMA_ALLOCATED_SLOT = "TDMA_ALLOCATED_SLOT"
""" Slot number allocated to the NIC"""

TDMA_MAC_PRIORITY_CLASS = "TDMA_MAC_PRIORITY_CLASS"
""" Service class QUEUE associated to the TDMA radio program """
"""
END TDMA RADIO PROGRAM PARAMETERS
"""
```

```
"""
CSMA RADIO PROGRAM PARAMETERS
"""
CSMA_CW = "CSMA_CW"
""" Current value of the Contention Window used by the CSMA radio program"""

CSMA_CW_MIN = "CSMA_CW_MIN"
""" Minimum value of the Contention Window used by the CSMA radio program"""

CSMA_CW_MAX = "CSMA_CW_MAX"
""" Maximum value of the Contention Window used by the CSMA radio program"""

CSMA_TIMESLOT = "CSMA_TIMESLOT"
""" Duration of the backoff slot used by the CSMA radio program"""

CSMA_MAC_PRIORITY_CLASS = "CSMA_MAC_PRIORITY_CLASS"
""" Service class QUEUE c associated to the CSMA radio program """

CSMA_BACKOFF_VALUE = "CSMA_BACKOFF_VALUE"
""" Current backoff value used by the CSMA radio program"""
"""
END CSMA RADIO PROGRAM PARAMETERS
"""
```

A.2 Measurements

```
"""
RADIO MEASURAMENT
"""
NUM_TX = "NUM_TX"
""" Total number of transmitted frames measured since the interface has been
started"""

NUM_TX_UNIT = "samples"
""" Unit of measurement of NUM_TX """

NUM_TX_SUCCESS = "NUM_TX_SUCCESS"
""" Total number of successfully transmitted frame measured since the interface
has been started """

NUM_TX_SUCCESS_UNIT = "samples"
""" Unit of measurement of NUM_TX_SUCCESS """

NUM_RX = "NUM_RX"
""" Total number of received frames since the interface has been started """

NUM_RX_UNIT = "samples"
""" Unit of measurement of NUM_RX """

NUM_RX_SUCCESS = "NUM_RX_SUCCESS"
""" Total number of successfully received frames since the interface has been
started """

NUM_RX_SUCCESS_UNIT = "samples"
""" Unit of measurement of NUM_RX_SUCCESS """

NUM_RX_MATCH = "NUM_RX_MATCH"
""" 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_MATCH_UNIT = "samples"
""" Unit of measurement of NUM_RX_MATCH """

NUM_FREEZING_COUNT = "NUM_FREEZING_COUNT"
```

```
""" Total number of backoff freezes since the interface has been started """

NUM_FREEZING_COUNT_UNIT = "samples"
""" Unit of measurement of NUM_FREEZING_COUNT """

BUSY_TYME = "BUSY_TIME"
""" Time interval in which the transceiver has been active (including reception,
transmission and carrier sense).
The unit of measurement is microseconds since the interface has been started, the
register size is 32bit
(cycle on 4294 sec)"""

BUSY_TYME_UNIT = "us"
""" Unit of measurement of BUSY_TIME """

TX_ACTIVITY = "TX_ACTIVITY"
""" Time interval in which the transceiver has been involved in transmission.
The unit of measurement is microseconds since the interface has been started, the
register size is 32bit
(cycle on 4294 sec)"""

TX_ACTIVITY_UNIT = "us"
""" Unit of measurement of TX_ACTIVITY """

TSF = "TSF"
""" The Time Synchronization Function (TSF), i.e. the timer that all stations in
the same Basic Service Set (BSS) use
for the synchronization """

TSF_UNIT = "us"
""" Unit of measurement of TSF """
```

Appendix B. Implementation of WiSHFUL architecture for radio platforms

The next pages give the documentation of the UPI_R usage, automatically created using Sphinx [1], and also available on a public git repository that will be publicly accessible to experimenters [3].

wishful_upis package

wishful_upis.upis.upi_m module

class wishful_upis.upis.upi_m.**UPI_M**
Bases: object

initTest(*myargs*)
Thisfunctionis a managementfunctionusedto setupthenodesbeforethe experiment, the functioncan performs different action.Theyare i) restart the driver module, ii) createsinfrastructure BSS, the node is usedsuch AccessPoint, iii)associate nodetoinfrastructureBSS. Thedifferentactionare addressing the key 'OPERATION' in the dictionary data typeof arguments.

Parameters **myargs** – listof parameters, in termof a dictionarydata type (listof key: value) in which: Thekey"interface" specify thenetwork interface touse. The key 'operaiton' is used tospecify the action, thesupported valuefor the key'operation' are'module', usedto restart the module, 'create-network', used to createthe IBSS, and 'association'used to associate the nodeto the IBSS. The key'ssid'is usedtodefinethe essidofthe IBSS. The key 'ip_address' is used to define the ip address of thewireless interface.

Return result return0 iftheparametersettingcallwassuccessfullyperformed, 1 partialsuccess, 2error.

example: >>args={'interface': 'wlan0', 'operation': ['create-network'], 'ssid': ['MyNetwork'], 'ip_address' : ['192.168.1.1'] }]

>> result = UPI_M.initTest(args)

>> print result

0

installExecutionEngine(*myargs*)
Thearchitecturesofthenodespresentin thetestbeddefineanexecution environment or executionengine able to run radio programs definedin a high-levelprogramming language. This managementfunction install an execution environ ment on a node. If theexecutionengineisimplementedon themicrocode wireless card, the function the copy and ablethe correct microcode.

:param myargs:listof parameters, interm ofa dictionarydata type(list ofkey: value) in which: thekey is 'execution_engine'and thevalue is the pathof theexecutionengine file(s). :return result: return 0 ifthe parameter setting call was successfullyperformed, 1 partial success,2 error.

example: >>args = {'execution_engine' : ['runtime/connectors/wmp_linux/execution_engine/wmp']}

>> result = UPI_M.installExecutionEngine(args)

>> print result

0

wishful_upis.upis.upi_rn module

class wishful_upis.upis.upi_rn.**RadioPlatform_t**

The informationelements used bythe UPI_R interfaceare organized intodata structures, which provide information on theplatform type of each interface,over theradio interface. Thisclass representing the data structure information of a radio interface, it contains anidentifierandthe platformtype.

platform_id ='']

interface identifier or interface name

platform_type ='']

platform interface

class wishful_upis.upis.upi_rn.**UPI_R**

Bases: object

BUSY_TYME ='BUSY_TIME']

Timeinterval inwhichthe transceiver hasbeen active(includingreception, transmissionand carrier sense). Theunitof measurementis microseconds since theinterfacehas been started, theregister sizeis32bit (cycle on 4294 sec)

BUSY_TYME_UNIT ='us']

Unit of measurement of BUSY_TIME

CSMA_BACKOFF_VALUE ='CSMA_BACKOFF_VALUE']

Current backoff value used by the CSMA radioprogram

CSMA_CW = 'CSMA_CW']

Current value of the Contention Windowused by theCSMA radio program

CSMA_CW_MAX ='CSMA_CW_MAX']

Maximum value of the Contention Window usedby the CSMA radioprogram

CSMA_CW_MIN ='CSMA_CW_MIN']

Minimum value of the Contention Windowusedby theCSMA radio program

CSMA_MAC_PRIORITY_CLASS ='CSMA_MAC_PRIORITY_CLASS']

Service class QUEUE c associatedto theCSMA radio program

CSMA_TIMESLOT ='CSMA_TIMESLOT']

Duration of the backoff slot used by the CSMAradioprogram

IEEE80211_AP_CHANNEL ='IEEE80211_AP_CHANNEL']

IEEE 802.11 PHY channel

IEEE80211_CCA ='IEEE80211_CCA']

IEEE 802.11 Clear channel assessment (CCA)threshold

IEEE80211_CHANNEL = 'IEEE80211_CHANNEL']

IEEE 802.11 PHY channel

IEEE80211_CONNECT_TO_AP ='IEEE80211_CONNECT_TO_AP']

Connect to ap

IEEE80211_L2_BCAST_TRANSMIT_RATE ='IEEE80211_L2_BCAST_TRANSMIT_RATE'
To Measure the transmit rate of generated 802.11 broadcasttraffic at fullrate

IEEE80211_L2_GEN_LINK_PROBING ='IEEE80211_L2_GEN_LINK_PROBING'
To send out 802.11 broadcast link probes

IEEE80211_L2_SNIFF_LINK_PROBING = 'IEEE80211_L2_SNIFF_LINK_PROBING'
To Receive 802.11 broadcast link probes

IEEE80211_MCS ='IEEE80211_MCS'
IEEE 802.11 Modulation and Coding Scheme (MCS) indexvalue

MAC_ADDR_SYNCHRONIZATION_AP ='MAC_ADDR_SYNCHRONIZATION_AP'
To set the Access Point MAC address used for synchronizing theTSF timer

NETWORK_INTERFACE_HW_ADDRESS ='NETWORK_INTERFACE_HW_ADDRESS'
MAC address of wireless network interfacecard

NUM_FREEZING_COUNT ='NUM_FREEZING_COUNT'
Total number of backoff freezes since theinterface has been started

NUM_FREEZING_COUNT_UNIT ='samples'
Unit of measurement of NUM_FREEZING_COUNT

NUM_RX = 'NUM_RX'
Total number of received frames since the interface hasbeen started

NUM_RX_MATCH = 'NUM_RX_MATCH'
Total number ofreceived frames addressed to thenode since the interfacehas been started. This measurement tracesthe number of received framein which thereceiver addressfield matches withthe network interface card MAC address

NUM_RX_MATCH_UNIT = 'samples'
Unit of measurement of NUM_RX_MATCH

NUM_RX_SUCCESS = 'NUM_RX_SUCCESS'
Total number of successfully received frames since the interface hasbeen started

NUM_RX_SUCCESS_UNIT = 'samples'
Unit of measurement of NUM_RX_SUCCESS

NUM_RX_UNIT ='samples'
Unit of measurement of NUM_RX

NUM_TX = 'NUM_TX'
Total number of transmitted frames measuredsince theinterface has beenstarted

NUM_TX_SUCCESS = 'NUM_TX_SUCCESS'
Total number of successfully transmitted frame measuredsincethe interface has beenstarted

NUM_TX_SUCCESS_UNIT = 'samples'
Unit of measurement of NUM_TX_SUCCESS

NUM_TX_UNIT ='samples'
Unit of measurement of NUM_TX

RX_ANTENNA = 'RX_ANTENNA'
Antenna number selected for reception

TDMA_ALLOCATED_SLOT = 'TDMA_ALLOCATED_SLOT'
Slot number allocated to the NIC

TDMA_MAC_PRIORITY_CLASS ='TDMA_MAC_PRIORITY_CLASS'
Service class QUEUE associated to the TDMA radio program

TDMA_NUMBER_OF_SYNC_SLOT = 'TDMA_NUMBER_OF_SYNC_SLOT'
Number of temporal slots included in a TDMA frame

TDMA_SUPER_FRAME_SIZE ='TDMA_SUPER_FRAME_SIZE'
Duration of a periodic TDMA frame

TSF = 'TSF'
The Time Synchronization Function (TSF), i.e. thetimerthatallstationsin thesame BasicServiceSet (BSS) use for the synchronization

TSF_UNIT = 'us'
Unit of measurement of TSF

TX_ACTIVITY ='TX_ACTIVITY'
Time intervalin whichthe transceiver has been involved in transmission. The unit of measurement is microseconds since the interface has been started, theregistersize is32bit (cycle on 4294 sec)

TX_ACTIVITY_UNIT ='us'
Unit of measurement of TX_ACTIVITY

TX_ANTENNA = 'TX_ANTENNA'
Antenna number selected for transmission

TX_POWER = 'TX_POWER'
Transmission power in dBm

getActive(*myargs*)
Eachradioprogramisassociatedwitha nameandanindex. Whenexecuted, thisfunctionreturntheindex of the radioprogram active.

Parameters myargs – a dictionary data type (key: value)where thekeys are: The key"interface" specify the network interface to use.

Return result the index of theactiveradioprogram.

Example >> args ={'interface': 'wlan0'}

>> result =UPI_RN.getActive(args)

>> print result

2

getMonitor(*myargs*)
The UPI_Rinterfaceis able to get the radiomeasurements thanksto theabstraction of the hardware platform and radio programs in terms of Radio Capabilities. Asubsetofradio capabilitiesarethe low-level measurements. The low-level measurements arecontinuously monitoredby the hardwareplatform and by the radioprograms. Themeasurementcapabilitiescanbe usedtogetinformation andstatistics about the state of the physical links or the internal state of the node. Thisfunction get the value(s) of the Measurements RadioCapabilities specifiedin the dictionaryargument. The list ofavailable measurements are defined as attribute of the UPI_R class, you canusetheUPI_RN.getRadioInfofunctionto findthe platform supported measurements.

Parameters myargs – listof parameters, in termof a dictionarydata type(list ofkey: value) inwhich: the keyis'measurements' andthe valueisa listof UPI_Rattributesfor measurements, the key in'interface' specify network interfaceto be uses. Anexampleof argument dictionaryis {"measurements": [UPI_RN.NUM_FREEZING_COUNT, UPI_RN.TX_ACTIVITY]}.

Return result list ofparameters andvalues, interm of a dictionary datatype (list ofkey: value) in whichthe key arethe UPI_R attributesfor measurements, andvalue is thereading of the

measurement. Anexample of argumentdictionary datatype is {UPI_RN.NUM_FREEZING_COUNT :150, UPI_RN.TX_ACTIVITY : 45670}.

Example >> args = {'interface': 'wlan0', 'measurements': [UPI_RN.NUM_FREEZING_COUNT]}

>> result =UPI_RN.getMonitor(args)

>> print result

{UPI_RN.NUM_FREEZING_COUNT :150}

getMonitorBounce(*myargs*)

The UPI_R interface is able to get the radio measurements thanksto the abstractionof the hardware platform and radioprograms in terms of Radio Capabilities. Asubset ofradiocapabilities arethe low-level measurements.The low-level measurements arecontinuously monitored by the hardware platformand by the radio programs. Themeasurement capabilitiescan beusedto getinformation andstatisticsabout the state of the physical links or theinternal state of the node. ThisfunctionworkssimilarlytogetMonitor, it gets thevalue(s)of theMeasurements RadioCapabilities specifiedin thedictionary argument, butin cycling mode. The function gets themeasurements every SLOT_PERIOD and stores them on node memory. EveryFRAME_PERIODallmeasurementsare reported to thecontroller, andthis operation is performed a number of times specified by ITERATION. A callback function is usedto receive the measurements results. The listofavailablemeasurementsaredefinedasattributeof theUPI_Rclass, youcan usethe UPI_RN.getRadioInfo function to find the platform supported measurements.

Parameters **myargs** – listof parameters, in termof a dictionarydata type (listof key: value) in which: Thekey'interface' specify thenetworkinterface tobeused. Thekey'measurements' is usedto give the listof UPI_R attributesfor measurements. Thekey'slot_period'is used todefinethe timebetweentwoconsecutivemeasurementreadings, inmicrosecond. The key 'frame_period' is used todefine the time between two consecutive reports to the controller, inmicroseconds. The key'iterator'is usedtodefine howmany timesthemeasurements have to be performed.

Return result list ofparameters andvalues, interm of a dictionary data type (list ofkey: value) in which the key are the UPI_R attributesfor measurements, and value isthe measurement reading. Anexampleofargumentdictionarydatatype is{UPI_RN.TX_ACTIVITY :45670}.

Example >> args= {'interface': 'wlan0', 'measurements': [UPI_RN.BUSY_TIME, UPI_RN.TX_ACTIVITY], 'slot_period': 500000,

'frame_period': 2000000, 'iterator': 60}

>> result =UPI_RN.getMonitorBounce(args)

>> print result

{UPI_RN.BUSY_TIME : 1505, UPI_RN.TX_ACTIVITY: 45670}

getParameterLowerLayer(*myargs*)

The UPI_R interface isable to configure the radio behavior thanks to the abstraction of the hardware platformand radioprograms in terms of Radio Capabilities. A subset of radio capabilities arethe parameters. Parameters correspondto the configurationregisters of the hardware platformand to the variables used in the radio programs. This function getthe value(s) of the Parameters Radio Capabilities specified in the dictionaryargument. TheavailableparametersaredefinedasattributesoftheUPI_Rclas, youcanusethe UPI_RN.getRadioInfo function to find the platform supported parameters.

Parameters **myargs** – listof parameters, in termof a dictionarydata type(list ofkey: value) in which:the key is 'parameters' andthe valueis alist ofUPI_R attributes for parameters, the key in'interface' specify network interface to be uses. An argumentdictionary example

is{'interface': 'wlan0', 'PARAMETERS': [UPI_RN.CSMA_CW,UPI_RN.CSMA_CW_-MIN, UPI_RN.CSMA_CW_MAX]}.

Return result list ofparameters andvalues, interm of a dictionary data type (list ofkey: value) in which the key is the UPI_R class attribute, and value is the current setting of the attribute. Anexampleofargumentdictionarydatatypeis{UPI_RN.CSMA_CW: 15, UPI_-RN.CSMA_CW_MIN : 15, UPI_RN.CSMA_CW_MAX :15}.

Example >>args={'interface': 'wlan0', 'parameters' : [UPI_RN.CSMA_CW]}

>> result =UPI_RN.getParameterLowerLayer(args)

>> print result

{UPI_RN.CSMA_CW :15}

getRadioInfo(*interface*)

Gets the radio capabilities of a given networkcard RadioPlatform_tin termsofsupported measurement and supported parameter and list of supp orted radio program. The information elements used by the UPI_-R interface, tomanage parameters, measurementsand radioprogram, are organized intodata structures, which provide information onthe platformtype andradio capabilities. When executed, this function return information about available radio capabilities (measurements and parameters) of each interface (RadioPlatform_t)on theavailable radio programs(radio_prg_t)availablefortransmissions over the radio interface.

Parameters **interface** – networkinterfaces to use

Return result returna listin termofa dictionarydatatype(listof key: value). in whichare presentthe keyshowed below: 'radio_info'–> a listof pairvalue, thefirst valueisthe interfaceidentifierandthe secondisthe supportedplatforms. 'monitor_list'–>a listofsupported measurementsbetween the attribute of the class UPI_R 'param_list' –> a list of supported Parameters between the attributeofthe class UPI_R'exec_engine_list_name' –> a listof supported execution environmentname'exec_engine_list_pointer'–> a list ofsupported execution environment path 'radio_prg_list_name'–> a list ofsupportedradio program name 'radio_prg_list_pointer' –> a list of supported radio program path

example: >> interface = 'wlan0'

>> current_platform_info =radio_info_t()

>>param_key = {'platform': 'wmp'}

>> current_platform_info_str = UPI_RN.getRadioInfo(interface,param_key)

>> current_platform_info.platform_info.platform_id = current_platform_info_str['radio_info'][0]

>> current_platform_info.platform_info.platform = current_platform_info_str['radio_info'][1]

>> current_platform_info.monitor_list = current_platform_info_str['monitor_list']

>> current_platform_info.param_list =current_platform_info_str['param_list']

>> current_platform_info.execution_engine_list_name =current_platform_info_str['exec_engine_-list_name']

>>current_platform_info.execution_engine_list_pointer = current_platform_info_str['exec_engine_-list_pointer']

>> current_platform_info.radio_program_list_name = current_platform_info_str['radio_prg_list_-name']

>> current_platform_info.radio_program_list_path = current_platform_info_str['radio_prg_list_-pointer']

getRadioPlatforms()

Gets available NIC on board andtype of supported platforms. The information elements used by the UPI_-R interface, tomanage parameters, measurementsand radioprogram, are organized intodata structures, whichprovideinformationon theplatformtypeandradiocapabilities. When executed, this functionreturn information aboutavailable interfaces onnode, the name orthe identifier ofthe interfaceand the supported platform type.

Return current_NIC_list a listof pair value, thefirst value is theinterface identifier and the second is the supported platforms.

example: >> current_NIC_list = RadioPlatform_t() >> current_NIC_list_string = UPI_-RN.getRadioPlatforms() >> current_NIC_list.platform_info = current_NIC_list_string[0] >> current_NIC_list.platform =current_NIC_list_string[1]

setActive(*myargs*)

Thisfunctionactivatesthepassedradioprogram, onetheplatform. Whenexecuted, thisfunctionstopsthe currentradio program and enables theexecution of theradioprogram specified inthe parameter radioPro-gramName. Twoadditionalparameterscanbeused, oneoftheseisrequired.Thepathoftheradioprogram description is required. Theoptionally parameterspecify theindex, inorder toassociate anindexto the radio program.

Parameters **myargs** –a dictionary data type (key: value) wherethe keysare: The key'inter-face'specify thenetwork interfaceto use. Thekey'radio_program_name'specifythe name of radio program. Thekey'path' in which the value specifythepath ofradio programde-scription, and 'position' in whichthe valuespecify the radio program indexassociated.

Return result return0 iftheparametersettingcallwassuccessfullyperformed, 1 partialsuccess, 2error.

Example >>args= {'interface': 'wlan0', 'radio_program_name': 'CSMA','PATH':'./radio_-program/csma.txt'}

>> result =UPI_RN.setActive(args)

>> print result0

setInactive(*myargs*)

When executed, this function stops the radio program specified in the parameter radio_program_name.

Parameters **myargs** – a dictionary data type (key: value)where thekeys are: The key"inter-face" specify thenetwork interface to useand thekey 'radio_program_name' inwhich the value specify the name of radio program,

Return result return0 iftheparametersettingcallwassuccessfullyperformed, 1 partialsuccess, 2error.

Example >>args={'interface': 'wlan0', 'radio_program_name' : 'CSMA'}

>> result =UPI_RN.setInactive(args)

>> print result

0

setParameterLowerLayer(*myargs*)

The UPI_R interface isable to configure the radio behavior thanks to the abstraction of the hardware plat-form andradio programs in terms of Radio Capabilities. A subsetof radio capabilitiesare the parameter. Parameters correspondto the configurationregisters of the hardware platformand to the variables used in theradioprograms. This function(re)setthevalue(s)of theParametersRadioCapabilitiesspecified inthe dictionary argument. ThelistofavailableparametersisdefinedasattributesoftheUPI_Rclass, youcan use the UPI_RN.getRadioInfo function to find the platform supported parameters.

Parameters **myargs** – listof parameters and values toset, in term ofa dictionarydata type (list of key: value)inwhichkeysis theinterface('interface')tospecify networkinterface tobeuses and the desired UPI_Rattribute, andvalueisthe valuetoset. An exampleof argument dictionary data typeis {UPI_RN.CSMA_CW : 15, UPI_RN.CSMA_CW_MIN: 15, UPI_RN.CSMA_CW_MAX : 15}.

Return result return0 iftheparametersettingcallwassuccessfullyperformed, 1 partialsuccess, 2error.

Example >> args= {'interface': 'wlan0', UPI_RN.CSMA_CW: 15, UPI_RN.CSMA_CW_-MIN :15, UPI_RN.CSMA_CW_MAX : 15}

>> result =UPI_RN.setParameterLowerLayer(args)

>> print result

[0, 0, 0]

class wishful_upis.upis.upi_rn.**execution_engine_t**

The informationelements used bythe UPI_R interfaceare organized intodata structures, which provide informationona specificexecutionenvironment. Thisclassrepresentingthe datastructurethatcontainsthe execution environment information.

execution_engine_id = ''
Identifier of the execution environment

execution_engine_name =''
Name of the execution environment

execution_engine_pointer = ''
Path of the execution environment

supported_platform =''
Platform of the execution environment

class wishful_upis.upis.upi_rn.**radio_info_t**

The information elementsused by the UPI_R interface are organized into data structures, whichprovide information on radio capabilities(monitor_t, param_t) of eachinterface (RadioPlatform_t)on the available radio programs (radio_prg_t), over theradio interface. This class representingtheradio capabilitiesof a given networkcardRadioPlatform_t in termsof measurementlist, parameters lists, execution environmentlistand radio program list.

execution_engine_list =None
The list of supported execution environment

monitor_list = []
The list of supported measurements

param_list = []
The list of supported parameters

platform_info =None
Interface information structured such as RadioPlatform_t object

radio_program_list =None
The list of supported radio program

class wishful_upis.upis.upi_rn.**radio_program_t**

The informationelements used bythe UPI_R interfaceare organized intodata structures, which provide information on a specific radio program. This class representingthe data structure thatcontains the radio program information.

radio_prg_id = ‘’
Identifier of the radio program

radio_prg_name = ‘’
Name of the radio program

radio_prg_pointer = ‘’
Path of the radio program

supported_platform =‘’
Platform of the radio program

References

[1] Sphinx – Python document generator - http://sphinx-doc.org

[2] WiSHFUL Project Deliverable D3.1 - Design of software architecture for radio control

[3] https://wireless.wiki.kernel.org/en/users/documentation/iw

[4] Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, "Wireless MAC Processors: Programming MAC Protocols on Commodity Hardware" IEEE INFOCOM, March 2012.

[5] G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, I. Tinnirello, "MAClets: Active MAC Protocols over Hard-Coded Devices" ACM CoNEXT'12, pp. 229-240, 2012.

[6] libb43 library - https://github.com/mbuesch/b43-tools

[7] WiSHFUL documentation - http://wirelesstestbedsacademy.github.io/wishful_upis/

[8] "rm090", http://www.rmoni.com/en/products/hardware/rm090

[9] ZeroMQ distributed messaging http://zeromq.org

[10] Zerorpc - An easy to use, intuitive, and cross-language RPC