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

Project Deliverable D3.1

Design of software architecture for radio control

Contractual date of delivery:	30-06-2015
Actual date of delivery:	30-06-2015
Beneficiaries:	CNIT, IMINDS, TCD, TUB, SNU, RUTGERS
Lead beneficiary:	CNIT
Authors:	Ilenia Tinnirello (CNIT), Pierluigi Gallo (CNIT), Domenico Garlisi (CNIT), Daniele Croce (CNIT), Bart Jooris (IMINDS), Peter Ruckebusch (IMINDS), Ingrid Moerman (IMINDS), Nicholas Kaminski (TCD), Anatolij Zubow (TUB), Sunghyun Choi (SNU), Ivan Seskar (RUTGERS)
Reviewers:	Nicholas Kaminski (TCD)
Work package:	WP3 – Radio Control
Estimated person months:	8
Nature:	R
Dissemination level:	PU
Version:	2.1

Abstract:

This deliverable describes the first specification of UPI_R interface defined for configuring the MAC/PHY stack of the WiSHFUL programmable radio nodes. The specification is based on the abstraction of three architectures, namely Iris, TAISC and WMP, into a general radio architecture with the following common building blocks: i) an **execution engine** able to run radio programs defined in an high-level programming language, ii) a **radio manager** responsible of loading and activating the radio programs in the engine, iii) a **hardware platform** providing some primitive commands and signals, used for defining the radio programs; iv) a **radio controller** responsible of configuring the radio programs and the hardware platform according to the rules specified in the control program. The UPI_R interface works on data structures which implement the main abstractions of the general radio and exposes three sets of functions working on the control, management and data plane. **Control functions** deal with the monitoring and configuration of the radio platforms during their operation; **management functions** permit to handle radio programs by loading the program in the execution engines and activating them; **data plane functions** make use of the higher layer flow classification and can adapt or switch the radio program based on the flow

type.
Keywords: Programmable radio architecture; radio capabilities; MAC/PHY adaptations.

Executive Summary

This public deliverable reports the WISHFUL general software architecture for radio control and its unified programming interface, named UPI_R, offered to experimenters in the first open call at the end of year 1.

First, we introduce the concept of programmable radio architecture, i.e. a hardware and software platform exposing an advanced programming interface which allows the experimenter to easily program low-level MAC/PHY mechanisms. We present three programmable radio architectures that will be integrated in the WiSHFUL testbed, Iris, Time Annotated Instruction Set Computer (TAISC) and Wireless MAC Processor (WMP), by describing the general architectures, the relevant programming languages, and the support of the general programmability requirements emerged in D2.1.

Second, we generalize these architectures in a common radio architecture, in which the hardware platform is abstracted in a set of radio capabilities. The desired radio flexibility is achieved by tuning the hardware configuration parameters and by loading specific radio programs.

Finally, we detail the specification of the UPI_R interface and the design of the platform-specific adaptation modules that will be implemented for exposing the same UPI_R interface on Iris, TAISC and WMP. The complete definition of UPI_R is reported using a C-style pseudo-code, with data structures derived from the general radio architecture and functions grouped into control, management and data-plane functions. Two examples of MAC/PHY adaptations programmed by using the UPI_R interface are also presented.

List of Acronyms and Abbreviations

WMP	Wireless MAC Processor
TAISC	Time Annotated Instruction Set Computer
TDMA	Time Division Multiple Access
CSMA	Carrier Sense Multiple Access
UPI	Unified Programming Interface
UPI_R	Unified Programming Interface radio
UPI_N	Unified Programming Interface network
VM	Virtual Machine
XFSM	Extended Finite State Machines
CISC	Complex Instruction Set Computer
ISC	Instruction Set Computer
RISC	Reduced Instruction Set Computer
TAISC	Time Annotated Instruction Set Computer
TSCH	Time Synchronized Channel Hopping
HII	Hardware Independent Interface

Table of contents

1 Introduction

This deliverable describes the software architectures provided by WiSHFUL for supporting radio control. Three different **programmable radio architectures**, designed by project partners in previous research projects for different hardware platforms (namely, **microcontroller devices with a radio chip**, **general purpose devices with a wireless network interface**, and **software defined radios** (**SDR)**) are integrated into the WiSHFUL testbeds and offered to experimenters of novel services and protocols with a common programming model and interface. With programmable radio architectures we mean a **hardware and software platform exposing an advanced programming interface** which allows easy programing of low-level MAC/PHY mechanisms, usually implemented in a closed standard-compliant form, without knowing the internal details of the platform.

The deliverable describes: i) how the different radio architectures support the WiSHFUL general requirements; ii) the abstraction of these programmable radio architectures into a common programming model; ii) the design of a platform-independent UPI_R interface and the mapping of this interface to the different platforms; iii) some UPI_R utilization examples.

Two different utilization paths are considered, as originally discussed in the proposal:

- *Path 1 (Black boxes approach)* offers limited flexibility, but maximal ease of use. WiSHFUL programmable radio architectures are offered with a pre-defined set of functionalities and **radio programs** (implementing different protocols and transceivers) that can be selected by experimenters who focus on service creation. Radio programs support a simple parametric interface, defined in terms of configurable parameters and output signals. The experimenter can also define a **control program** for dynamically reconfiguring the radio programs based on simple rules that work on the program output signals and platform-supported measurements.
- *Path 2 (White boxes approach)* offers full flexibility, and hence requires more expert knowledge. Experimenters can also define **novel radio programs**, by using the platform-specific programming languages and interfaces, for testing not only novel services, but also novel protocols and networking models. To this purpose, WiSHFUL will offer a common tool for editing and debugging the radio programs and for compiling the programs into platform-specific executable components.

This document focusses on the black-box approach, while the white-box approach will be considered during the second year activities of the project.

2 Programmable Radio Architectures available in WISHFUL

With programmable radio architectures, we mean a **hardware and software platform exposing an advanced programming interface** that allows to easy program low-level MAC/PHY mechanisms, usually implemented in a closed standard-compliant form, without knowing the internal details of the platform. This is different from classical SDR platforms, which offer the possibility of programming every MAC/PHY function from scratch, but do not consider unified abstractions that span several platforms (even non-SDR platforms).

Examples of programmable platforms, which focus on the design of hardware and software systems able to support MAC/PHY programmability, without exposing these simplifying abstractions, include GNUradio [1], WARP [2], USRP [3], SORA [4], AirBlue [5]. Conversely, the programmable radio platforms offered by WiSHFUL offer the possibility to load several MAC/PHY programs already available for experimenters or to define novel wireless protocols and radio behaviors by means of platform-agnostic software-defined programs specified in a high-level programming language.

The following subsections discuss the three different programmable radio architectures available in WiSHFUL and provided to experimenters.

i. The ***Iris*** architecture, developed on top of PCs connected to Universal Software Radio Periperals (USRPs).
ii. The ***Time Annotated Instruction Set Computer (TAISC)*** architecture conceived for sensor nodes and developed on top of RM090 sensor nodes.
iii. The Wireless MAC Processor (WMP) architecture, conceived for local-network wireless cards, and developed on top of two different hardware platforms (a commercial Broadcom card and the WARP board).

2.1 Iris

Iris is a software platform for realizing radios consisting of core functionality set and a set of libraries written in C++. This platform provides the means for accomplishing most of the tasks of a wireless system in software, including generating a signal for transmission or receiving and extracting data from a transmitted signal. In the typical case this software runs on general-purpose processors such as those found in PCs or laptops, connected to some flexible radio frontend such as the USRP. This configuration provides the flexibility to realize nearly any system that sends or receives data wirelessly. Furthermore, software provides the means for changing the change the properties of the radio more easily (for example, switching from a garage door opener to a model radar system or vice versa). Such flexibility and reconfigurability form the core goals of the Iris system.

2.1.1 Iris architecture

The software of Iris fundamentally rests on a plugin architecture. The core functionality of Iris focuses entirely on managing the interactions of these plugins. Each plugin wraps the operation of a library that does a specific job (e.g. data scrambling, OFDM modulation, etc.) to interface with the core Iris functionality. Libraries wrapped for use by the Iris core are referred to as ***components***. The architecture of Iris is organized into five entities that facilitate the connection and operation of these components.

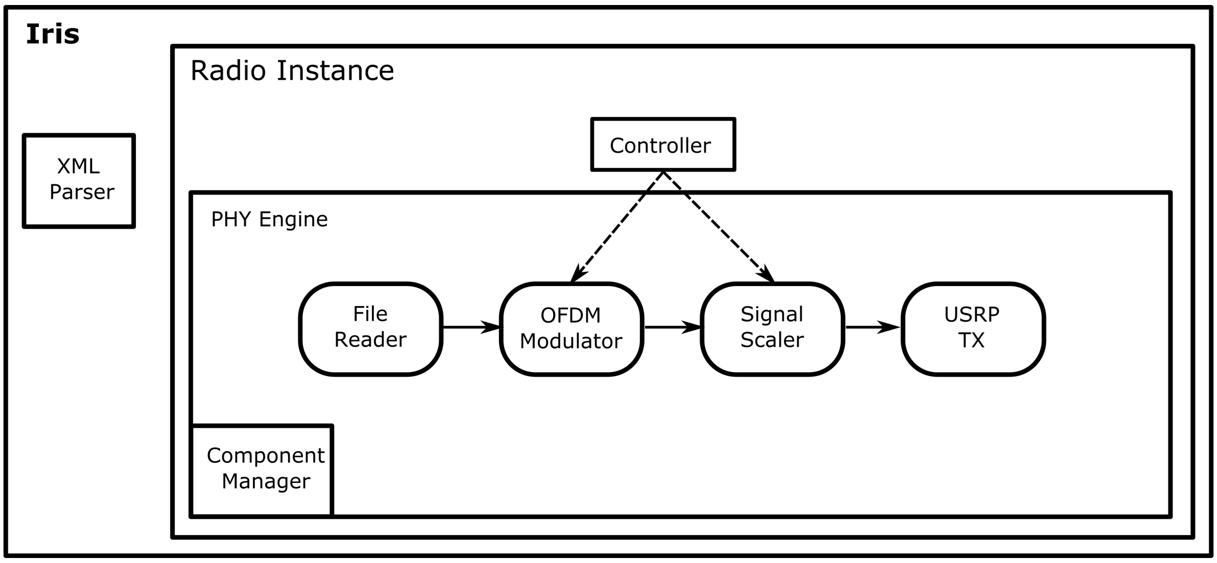

Figure 1– Iris Architecture

Components:

Component themselves are the central entity of the Iris system. Typically, components process streams of data to provide some specific aspect of a radio system (e.g. OFDM modulation, or signal scaling). They have input and output ports and work by reading data from one or more of their inputs and writing data to one or more outputs. This stream handling occurs in a loop, repeatedly processing sets of data from the time that the component is loaded until it is destroyed and unloaded. Component designers can expose ***parameters*** to control the operation of the component and reconfigure a radio while it is running. The specific functionality encapsulated within a component is at the discretion of component designer.

Iris currently has two main types of components – the ***Phy*** engine and the ***Stack*** engine. Phy components operate on a stream of signal data that flows in one direction from input to output. Examples of Phy components include modulators and demodulators, channel coders and decoders, data scrambles, etc. Stack components are a more experimental component type aimed at supporting bidirectional data, coming from both above and below. This will allow Stack components to include complete MAC layers, network routing layers and data encryption layers.

Engines

In Iris, components run within an ***engine***. The engine oversees the loading and initialization of components, schedules the input of data, calls the component work functions, collects output data, and finally destroys and unloads the component when the radio shuts down. Each component type runs within a matching engine type.

Component Managers

The component manager handles the loading and unloading of components within Iris engines. Additionally, this entity manages the collection of components available on a particular machine that may be loaded into Iris. The component designer must register components with the component manager prior to their use in a radio.

Controllers

Controllers provide a specialized category of components in that they are plugin libraries wrapped for use in Iris, but the operation of controllers is entirely different to the components discussed above. Controllers do not run in an engine and typically do not operate on streams of data. Instead, a controller has a global view of a running radio and employs user-defined logic to reconfigure components through their exposed parameters at any time. This functionality is typically used for radio management.

2.1.2 Iris Programming Language

Iris radio systems are described in an XML configuration file as a set of interconnected Iris entities. An XML parser interprets the radio design specified in such a configuration file, which enables the instantiation and operation of the necessary component managers and engines. Furthermore, users may specify initial parameter settings for each component in the XML configuration file.

The following XML configuration file illustrates an example for a simple OFDM transmitter:

```
<softwareradio name="Radio1">
<engine name="phyengine1" class="phyengine">
<component name="filerawreader1" class="filerawreader">
   <parameter name="filename" value="testdata.txt"/>
   <parameter name="blocksize" value="140"/>
   <parameter name="datatype" value="uint8_t"/>
   <port name="output1" class="output"/>
</component>

<component name="ofdmmod1" class="ofdmmodulator">
```

```
      <port name="input1" class="input"/>
      <port name="output1" class="output"/>
   </component>

   <component name="signalscaler1" class="signalscaler">
      <parameter name="maximum" value="0.9"/>
      <port name="input1" class="input"/>
      <port name="output1" class="output"/>
   </component>

   <component name="usrptx1" class="usrptx">
      <parameter name="frequency" value="5010000000"/>
      <parameter name="rate" value="1000000"/>
      <parameter name="streaming" value="false"/>
      <port name="input1" class="input"/>
   </component>
   </engine>

   <link source="filerawreader1.output1" sink="ofdmmod1.input1" />
   <link source="ofdmmod1.output1" sink="signalscaler1.input1" />
   <link source="signalscaler1.output1" sink="usrptx1.input1" />
   </softwareradio>
```

This example XML configuration defines four Phy components, running within a single Phy engine. The data, read from the file “testdata.txt”, is modulated into an OFDM signal, then scaled in magnitude and finally transmitted. The resulting radio is also visualized in Figure 1.

2.2 TAISC

Like the Iris and WMP solutions, the Time Annotated Instruction Set Computer (TAISC) also aims to simplify the development of new protocols. The main advantage of TAISC is its time-awareness. In a TAISC program, the duration of each instruction is known. This not only includes the time required by the core to execute this instruction, but also the time required by external components to complete the action indicated by the instruction. By using timing information for all instructions TAISC MAC protocols can automatically be fine-tuned for optimal energy consumption and spectrum efficiency. Unlike Iris and WMP, TAISC is designed to run on very constrained hardware and introduces time-awareness for the execution of protocols. This means TAISC is able to guarantee the exact time of transmission of packets, while minimizing the radio-on time.

2.2.1 Time aware computer

TAISC aims to simplify the development of new lower MAC protocols by providing a hardware independent interface (HII) for radio primitives without losing their real time aspect. For this, TAISC needs to be aware of the duration of each radio primitive.

First, four time classifications are introduced and explained using example radio primitives.

1. Static execution: switching the radio off will always take a fixed amount of time execute
2. Dynamic execution: the time it takes to compare a field against a reference in the last received frame will be a function of the number of bytes in the field. As the number of bytes to check is variable, we the time needed to handle 1 byte will be stored.
3. Static transition: starting up the oscillator of the radio consists of a limited fixed amount of execution time and a static transition time. The latter will take into account the time the oscillator needs to settle.
4. Dynamic transition: the time needed to transmit a frame will depend on the PHY bit rate, preamble bytes, etc. At least three different time classes can be derived to transmit a frame that is already loaded: fixed-length execution time to switch the radio into TX mode, static transition time to transmit the preamble, and a variable transition time to transmit the bytes of the frame. As the number of bytes to transmit is unknown at design time of the MAC, only the time needed to handle one unit (here byte) will be stored.

The time needed to finalize all example primitives can be mapped on this sequence of time fields. Depending on the mapped instruction, some of the fields might be zero.

Static Execution	Dynamic Execution	Static Transition	Dynamic Transition

Figure 1 Time classification fields of radio primitives.

By classifying the duration of a radio primitive, it is possible to transform the primitive into a time-annotated instruction and add it the TAISC instruction set. The timing information of all instructions is stored per platform in the TAISC library used by the TAISC compiler to create TAISC binaries. The TAISC compiler will be discussed later.

The TAISC architecture is a slightly modified Von Neumann machine that annotates each instruction (stored in the program memory) with timing information (both execution time and offset). Other well-known implementations of the Von Neumann machine are the RISC (Reduced Instruction Set Computer)[6] and the CISC (Complex Instruction Set Computer)[6]. In comparison with TAISC, RISC and CISC only take into account the execution time of an instruction and then starts executing the next instruction immediately. By also taking into account the execution offset, TAISC allows for a more fine-grained scheduling of instruction. In general, an instruction on an ISC abstracts functionality like add, copy, etc. In contrast to a RISC and CISC, TAISC can take into account all of the earlier described time classifications per instruction. By also adopting a rich addressing scheme for every instruction parameter, a TAISC instruction with a single parameter can have eight different meanings, and can, as such, be seen as extension of CISC.

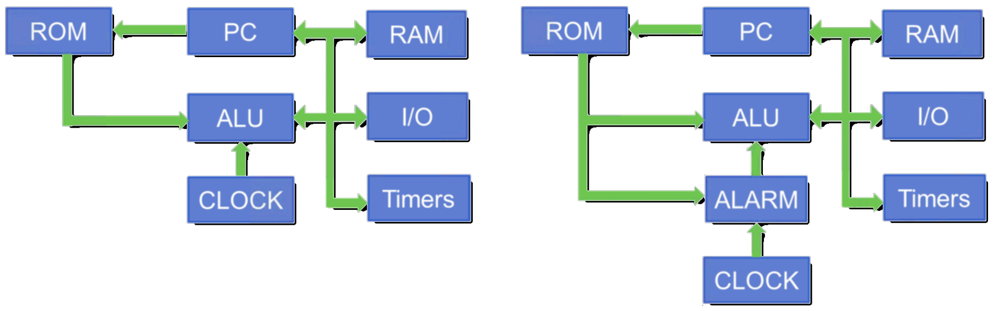

Figure 2 RISC/CISC (left) versus TAISC block diagrams (right). In the TAISC an ALARM block was added between the CLOCK and the ALU. The alarm gets configured with the annotated time information from the current instruction (where the program counter (PC) is pointing to). The ALU gets triggered when the ALARM and its timer have the same value.

To optimize the spectrum usage in a coordinated system we need two enablers:

- time synchronization,
- starting the execution of an instruction at a specific time.

Both are equally important to achieve high spectrum efficiency. Time synchronization is necessary to align TX and RX slots. Executing instructions on a predefined time will enable TAISC to allow the PHY to start shaping its first symbol immediately when a TX slot starts and implies further optimization (i.e. decreasing) of the slot durations.

To optimize power consumption we need:

- to pack all instructions before (example instructions: radio on and load frame) and after (example instructions: radio off) a reference instruction (example instruction: TX), without code nesting of higher layers.

TAISC packs every instruction in the time domain with respect to the time it needs to finalize (execution and transition). This is in a way similar to the 'Packed' attribute in the C language, which indicates that a data structure should be stored in memory with the smallest possible space.

We define a sequence of instructions with one optional reference instruction as a TAISC chain. By using the timing information in the TAISC instruction set, the TAISC compiler can compile a non time-aware chain, written in a C dialect, into a time-aware TAISC binary. For this purpose, the TAISC compiler will translate one or more chains into a byte code (binary) and add the time annotation to every instruction. Since the timing information depends on the radio hardware platform, this information is also stored in the TAISC library and used by the TAISC compiler when compiling for a specific target. It is hence possible to compile the same chain into different radio hardware platform specific binaries. After compilation, the TAISC binary is ready to be added and executed by the TAISC execution engine where the lower MAC protocol is executed.

In contrast to the classic software interfaces, the TAISC interface does not provide the instructions, but instead the TAISC binary needs to be uploaded into the TAISC execution engine via the management interface.

2.2.2 TAISC architecture

Similar to the level of detail found in the hardware representation of commercially available microcontrollers/radios, the basic building block presented in the TAISC block diagram (Figure 4 Mapping (pink) of the TAIC block diagram (see Figure 2) on msp430f5437. Extra (blue) blocks claimed for the cc2520 radio.Figure 2) will be discussed in more detail in this section.

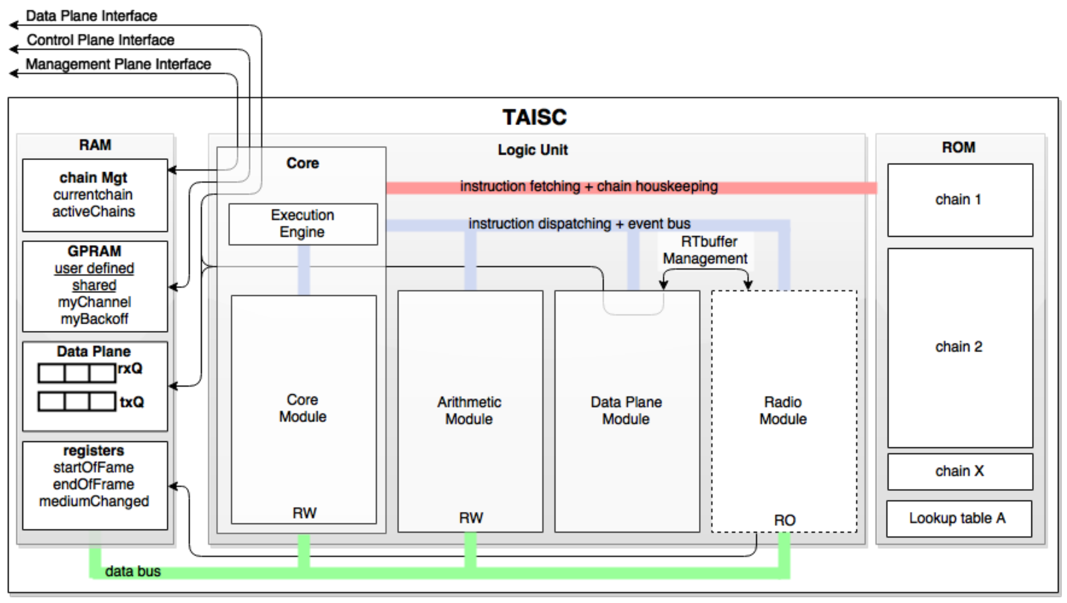

Figure 3 TAISC architecture

- The Logic Unit ((A)LU) of the TAISC is covered by its core and a flexible number of modules.
    - The execution engine (EE) in the TAISC core has the following tasks:
        - deciding the execution time of the next instruction and scheduling it using an Alarm;
        - managing the Program Counter (PC);
        - fetching the instruction and its parameters from the ROM;
        - dispatching the instruction to its responsible module;
        - interpreting the incoming events from the different modules;
    - Every module holds the implementation of the module, related instructions. These 3 fixed base modules are identical for all platforms (reuse of code):
        - The Data Plane module manages the access to and from the Data Plane RAM section. It avoids conflicts between the TAISC module that implements the Real Time Buffer Management (RTbuffer Management) and the upper layers that implement the Data Plane interface. Example instructions of this module are the rxTrigger and txTrigger, which can temporally transfer the ownership of a selected buffer to the upper layers via the Data Plane interface.
        - The Core module contains instructions that manipulate the Program Counter (PC), which includes jumps and chain management. A typical one-liner provided by this module abstracts “if event X triggers before timeout Y then…” (waitForTrigger instruction). This module minimizes the interactions between upper and lower layer MAC by concentrating the interactions for Data, Control and Management Plane interfaces at the end of every chain. This improves the performance of the chains and preserves their atomic behavior.

          The Arithmetic module implements classic arithmetic instructions like copy, add, subtract, etc. The datatypes they operate on are typically arrays. Hereby a complex instruction set is implied (TAISC extends CISC). Array operations for example ease the life of the 802.15.4e MAC developer to support operations on the five byte Absolute Slot Number (ASN)[9].

- Flexible platform specific modules (chip specific implementations)
  - One or more communication modules which abstract radio and bus chips functionality typically with Rx, Tx, Off,… instructions and interact to the Data plane module via the RTbuffer Management.
  - Other communication related hardware abstraction modules to cover RF switches, RF amplifiers, power supply chips and more.

The platform specific modules have restricted access on the data bus to the RAM. Read only in general, write access for the module specific registers and write access to the Data plane section via the RTbuffer Management.

- The TAISC ROM can store one or more compiled TAISC chains and other chain related constant data like fixed lookup table used in fixed TDMA schemes (a lookup table can translate a slot number identifier into slot functionality identifier (sleep, RX, TX,…).

- The TAISC RAM is divided into 4 sections:
  - General Purpose RAM (GPRAM) stores the defined variables needed in the different chains. Some of these variables can store the parameters for a TAISC chain which implements (part of) a MAC. The Control Plane will enable the exchange of data to and from this section of RAM. In the MAC upper layer a translation can be made from a CSMA_basic.setBackoff(X) to a specific location in the RAM. The TAISC compiler generates related code snippets to be used in the upper layers.
  - The Data plane section stores the reception and the transmission queues. The number of buffers and the size of the buffer is instance specific. Both queues implement a FIFO flavour: “First in Filtered/First Out”. While retrieving data from the queue, a filter can be defined to match a number of bytes at a given offset, the oldest buffer matching the given filter will be selected.
  - The register section holds the module specific volatile registers. For example, every time the medium changes from idle to busy and vice versa, the radio stores the timestamp of this event in a dedicated register.
  - The chain management section is kept consistent by the TAISC core which arbiters for the EE and the upper layers which interacting via the Management Plane interface. The number of chains stored in this section is instance specific.

- The upper layer TAISC interfaces:
  - Data Plane interface: discussed in the Data Plane RAM section and Data Plane Module.
  - Control Plane interface: discussed in the GPRAM section.
  - Management Plane interface: provides functionality to upload new compiled chains into the ROM and activating chains while keeping all the chains consistent regarding ROM and RAM boundaries of every chain.

2.2.3 TAISC on RM090

A proof of concept of the TAISC, which is a hardware concept, is implemented as a Virtual Machine (VM) on top of a MSP430F5437 microcontroller and the CC2520 802.15.4 radio integrated on the RM090 platform[7] (a RMoni product, same chipset is used on LSR ProFLEX01[8]). The TAISC block diagram is mapped on the msp430 as shown in the next figure.

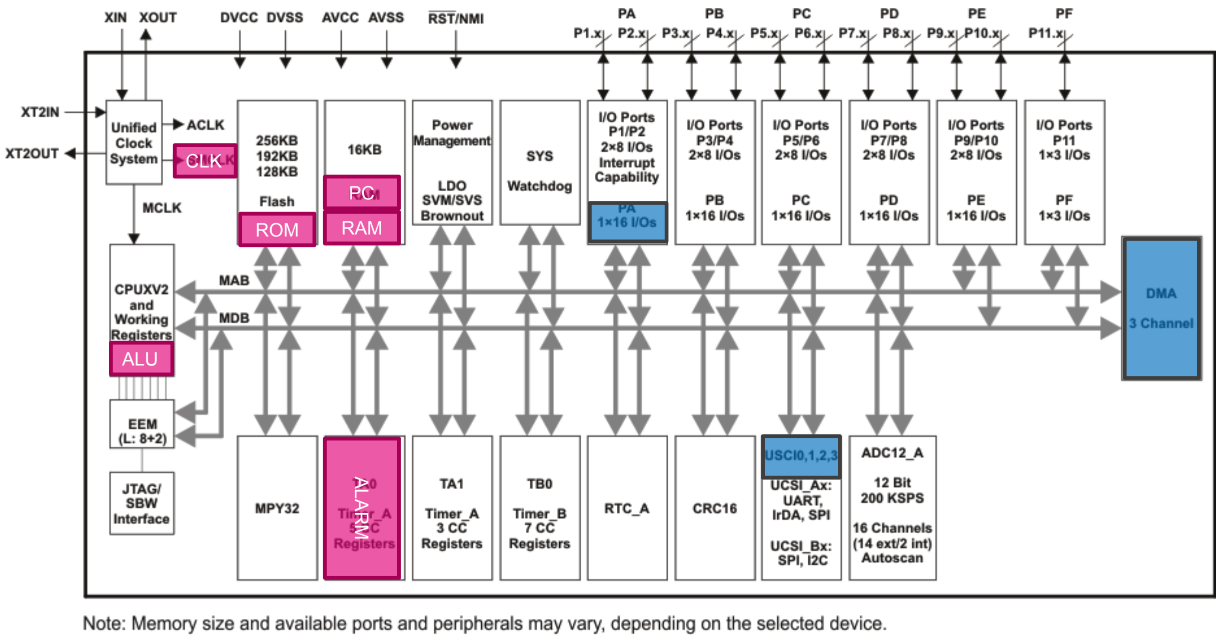

Figure 4 Mapping (pink) of the TAIC block diagram (see Figure 2) on msp430f5437. Extra (blue) blocks claimed for the cc2520 radio.

Targeting a lower MAC TSCH[9]**Error! Reference source not found.**, we aim at 15 KB of ROM (max 256KB) for the TAISC VM 2K ROM for TAISC chains (binary format of TSCH) , 1KB of RAM (max 10KB) for the TAISC core and buffers, and 10% of CPU (16bit @16MHz) resources. Further A VM brings some non-negligible extra overhead as shown in Figure 5.

pre VM	Addressing decoding	Static Execution	Dynamic Execution	Static Transition	Dynamic Transition	post VM

Figure 5 Extra overhead for TAISC instructions inside a VM

A timer compare block is used to schedule the execution of the next instruction. The time between timer compare interrupt and the execution of the instruction itself is labelled as preVM. The time needed to prepare the schedule of the next instruction is labelled as postVM. The addressing for every parameter also happens in software and implies some overhead and is labelled Addressing Decoding. We can adopt preVM and Addressing Decoding in the static execution part as a constant platform overhead. The postVM can be adopted in a minimum inter instruction time which was already foreseen in the TAISC compiler. Both adoptions imply that the primarily defined time classifications can also cover a VM implementation of the TAISC.

2.3 Wireless MAC Processor

The Wireless MAC Processor (WMP) [10] is a programmable architecture for wireless cards based on: i) an abstraction of the card hardware capabilities and ii) a generic behavioral model for running a control logic which drives the hardware based on the execution of state machines. It allows to completely repurpose the card hardware by simply loading a novel state machine program, from the execution of a TDMA-like or CSMA-like access protocol, to the collection of statistics for radio localization or interference characterization.

2.3.1 WMP architecture

In the WMP architecture, the card hardware capabilities are abstracted by the following sub-systems:

i. **The transceiver** deals with the reception and the transmission of the frames according to a pre-defined set of modulation and coding schemes.
ii. **The transmission queues** enqueues traffic flows or control and management frames into separate queues for achieving different medium access performance.
iii. **The reception queue** stores incoming packets before they forwarding them to the host.

Each sub-system exposes some configuration **parameters**, provides some signals at the occurrence of specific **events** (such as the arrival of a new frame) and is able to execute some primitive **actions** (such as the selection of a modulation and coding scheme for transmitting a frame).

Rather than been controlled by a given protocol, these sub-systems can be governed by a generic MAC Engine able to run programs defined in terms of Extended Finite State Machines (XFSMs). The state machines are composed by reusing the set of signals provided by the hardware sub-systems by means of an *interruption block*, the set of elementary actions implemented into an *operation block*, the *set of registers* for saving the system state and configuration parameters.

The XFSM defining the MAC protocol is coded in a table, which specifies all the possible state transitions in terms of input state, triggering event, enabling condition, transition action and output state. Starting from the initial state and parameters, the MAC Engine fetches the table entry corresponding to the state, reads the list of all the possible events that can trigger a transition from that state, and loops until one of these events occurs. It then evaluates the associated conditions on the system parameters, and if this is the case, it triggers the associated action and parameters' updates (if any), executes the state transition, and fetches the new table entry for such destination state.

The MAC engine workflow allows easy extensions for supporting code switching. Indeed, the MAC engine does not need to know to which MAC program a new fetched state belongs, so that a code switching is basically achieved by moving to a state in a different transition table and by updating the system configuration registers and protocol variables (e.g. the operating channel, the transmission power, etc.) when needed. To this purpose, the program switching transitions can be coded into a dedicated transition table and added to the list of transitions to be verified from each state of the program under execution.

Error! Reference source not found. Figure 6 illustrates the Wireless MAC Processor architecture [10], by enlightening the hardware sub-systems and relevant event/action abstractions. Multiple state machines can be simultaneously loaded on the engine, while a high-level controller can configure the protocol variable or switching conditions. A client process for receiving the commands sent by the controller (not shown in the figure) has been implemented and called MAClet manager, being the platform-independent MAC program similar to a JAVA applet [11].

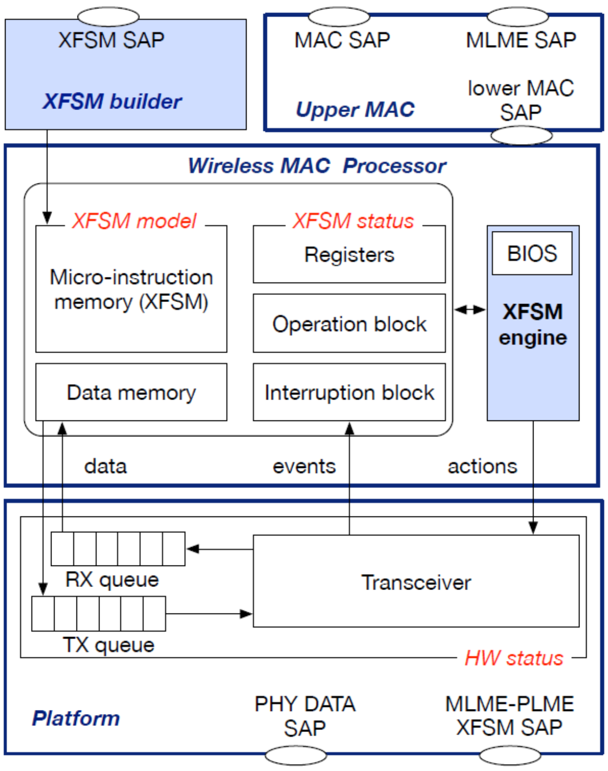

Figure 6 Wireless MAC Processor architecture.

2.3.2 WMP Primitives

Table 1 summarizes the list of WMP primitives that can be exploited for defining the MAC programs. The primitives include the elementary actions that can be performed on the card hardware, the signals triggered by the card and the data that can be configured on the hardware system and program variables.

Actions are commands acting on card hardware. Ordinary arithmetic and memory related (set/get) operations work on WMP data, which are given by protocol variables, physical registers and packet queues, while dedicated actions implement atomic MAC functions such as transmit a frame, set a timer, write a header field, switch to a different frequency channel, etc. Actions are not programmable. As the instruction set of an ordinary CPU, the hardware vendor provides them. The set of actions may be extended by the device vendor, and include advanced operations on the PHY, such as the configuration of the physical channel and the selection of the desired encoding scheme.

Events include hardware interrupts such as channel up signals, indication of reception of a valid preamble or end of a valid frame, expiration of timers, and signals conveyed from the higher layers such as the enqueuing of a new packet. As in the case of actions, the list of supported events is a-priori provided by the hardware design.

Conditions are Boolean expressions evaluated by comparing the card internal registers, the fields of the enqueued frames or the protocol variables with a given parameter. Registers, frame fields and program variables are either explicitly updated by actions, or implicitly updated by events. Registers store general hardware parameters whose settings represent the hardware configuration and operational state (such as the channel, antenna, transmit power, rate, etc.). Frames, enqueued in the transmission or reception queue, store the payload and MAC header fields (e.g., source or destination addresses, frame size, frame type, etc). The programmer can define the protocol variables (e.g. contention window, backoff counter, etc). Registers provide an interface to the PHY, while variables are used to achieve a more compact protocol description.

Actions, events, and registers on which conditions may be set, form the application programming interface exposed to third party programmers. This API is implemented (in principle) once-for-all, meaning that programs may use such building blocks to compose a desired operation, but have no mean to modify them. Since a MAC program is a list of labels specifying the events, actions and conditions associated to each state transition, by defining a common set of labels for the API (i.e. a machine language), the MAC program can be coded in a *bytecode* and transported over data frames from one node to another.

Table 1 WMP Application programming interface: supported events, actions and configuration data.

events	*actions*	*data*
CH_UP	rx_header()	channel
CW_DOWN	rx_msdu()	antenna
RX_PLCP_END	start_timer(reg,prm)	power
RX_MAC_HEAD_END	extract_bk(reg, prm)	txrx_on
RX_END	tx_start(prm)	backoff_slot
RX_ERROR	update_cw(reg, prm)	rx_checksum
QUEUE_OUT_UP	repor_to_host(prm)	busy_time
IFS_EXPIRED	start_ifs(prm)	backoff_value
TX_END	set(reg/var, var/prm)	bandwidth
TIMER_EXPIRED	get(reg/var, var/prm)	slot_time
	write(queue, field, var/prm)	+ *protocol*
	read(queue, field, var/prm)	*variables*
	incr(var)	+ *payload*
	hw_reset()	*fields*

2.3.3 WMP Programming Language

MAC programs are defined in terms of eXtensible Finite State Machines (XFSM). XFSMs are a generalization of the finite state machine model and permit to conveniently control, with a more compact description, the actions performed by the MAC protocol as a consequence of the occurrence of events. In particular, they allow reducing the state space by decoupling a program state (tracked by the state machine executor) from a set of system variables updated by actions and events.

In order to show the API potential for programming different behaviors on the same platform, Figure 7 shows two simple state machines implementing, respectively, a TDMA and a CSMA protocol. Self-explaining labels indicate MAC program states, capitals represent events and square brackets enclose the conditions (when associated to a transition). The examples are limited to the management of frame transmissions (i.e. acknowledgments and frame receptions are not included in the figures) for better enlightening the possibility to program the medium access timings according to completely different logics.

In the leftmost part of Figure 7, for example, a simple program for accessing the channel access at regular time intervals is modeled with two states: a waiting state and a transmission state. From the waiting state, a first transmission event is scheduled in case of reception of a synchronization signal, which in the figure is given by the reception of a beacon frame. The event is scheduled after a time interval (called *slot*) from the reception of the beacon header. When the timer expires, in case the transmission queue is not empty, the transceiver is activated by calling the action *start_tx(queue)*. At the end of the frame transmission, a timer is set for the next transmission event by considering the difference between the protocol variable representing the inter-transmission *period* and the duration of the transmitter activity *busy_time*. When no frame is available for transmission, the same timer is set during the entire inter-transmission time.

The rightmost part of Figure 7 shows a simple program with three protocol states, for accessing the channel at random intervals. From the idle state, in case of enqueuing of a novel packet, a sensing interval equal to a inter-frame time or to the inter-frame time plus a random backoff is set by calling the *start_ifs(prm)* action according to the state of the medium (i.e. to the activity state of the transceriver). At the expiration of the sensing interval, the transmitter is activated by calling the *start_tx(queue)* action; at the end of the transmission, a transition to a novel sensing interval or to the idle state is triggered and performed according to the state of the transmission queue.

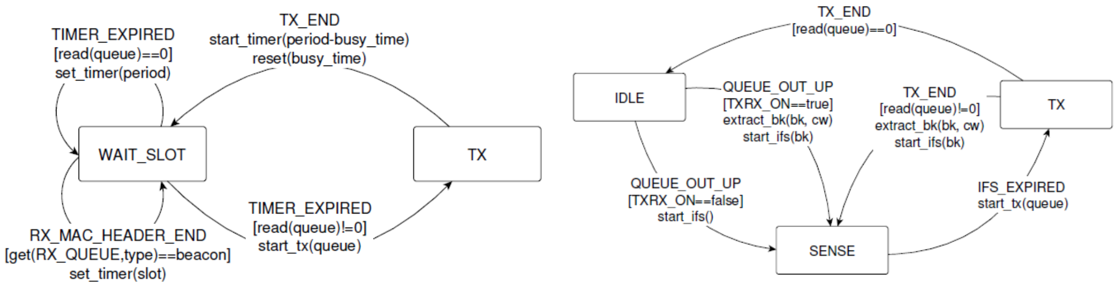

Figure 7 – Examples of MAC programs: simplified TDMA and CSMA channel access operations.

2.3.4 WMP Implementation over a commercial WiFi card

The first WMP prototype was implemented on a Broadcom card based on the AirForce54G chipset. The 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 storing hardware configuration settings; data memory for storing variables and composing arbitrary frames and code memory.

Our implementation replaces the original card firmware with an assembly code implementing the state machine execution engine, and mapping the previously described WMP programming interface into actual signals, operations, and registers of the card. All the register configuration actions and non-blocking actions summarized in Table 1 were natively supported by the chipset; the remaining ones were developed and pre-installed as micro-code firmware procedures.

Moreover, the internal card registers automatically track the state of the hardware sub-systems (e.g. the starting of a frame demodulation after a valid preamble, the arrival of a new frame, etc.). All the WMP events can be revealed by monitoring the change of status of the relevant internal registers. The only exceptions are given by the CH_UP and CH_DOWN events, which required monitoring two registers.

2.3.5 WMP Implementation over a WARP v3 board

The WARP v3 provides a Xilinx Virtex-6 FPGA, two MAX2829 transceivers, and a complete 802.11 Reference Design that we used as a starting point for our implementation. The legacy 802.11 architecture includes some custom FPGA cores (dedicated to the implementation of the 802.11g OFDM transceiver, the required timers, the carrier sense mechanism and the interface between the transmission and reception blocks) and two MicroBlaze CPUs running the DCF MAC protocol (written in C) according to the usual upper-MAC and lower-MAC decomposition. The architecture includes a single transmission queue with 15 available slots for storing data frames that can be explicitly addressed. A last slot of the queue is dedicated to the storage of an acknowledgment template.

For our implementation, we replaced the programs executed by the two CPUs with two different programs: the high-level one, adding the WMP control interface to the upper MAC functionalities, and the low-level one implementing the MAC Engine and part of the WMP actions. Some other actions have been added to the custom cores dealing with the interface towards the physical layer. For example, the non-blocking *start_ifs* action, which manages the backoff countdown, has been added to the transmission core.

For detecting the events, we kept the register-based solution already adopted in the 802.11 reference design, which works similarly to the Airforce54G chipset. Indeed, each WMP event corresponds to a specific register whose status change signals the event occurrence to the low-level CPU. Finally, we added some other blocks for supporting a dedicated BRAM to store FSMs, the relevant controller, a mutex for regulating the BRAM accesses performed by the high-level and low-level CPUs, and some software registers.

Note that, for this platform, thanks to the availability of programmable hardware sub-systems, the core WMP API described in [10]**Error! Reference source not found.** can be extended for including more advanced operations, such as tuning the transmission bandwidth or configuring the queuing disciplines.

3 Mapping UPI_R Requirements into WiSHFUL programmable Radio Architectures

The UPI_R interface is responsible of radio configuration. In D2.1 we discussed how this configuration is related to the set-up of wireless links between the nodes, whose capacity is affected by propagation and interference conditions, bandwidth allocations, and also medium access schemes. More detailled, the radio configuration requires an explicit consideration of the following main aspects:

i. **The spectrum allocation** lists the operating channel, bandwidth, multiple bands or sub-bands, etc., on each node.
ii. **The transceiver configurations** and **link set-up**, in terms of selection of transmission power, antenna, transmission format and error protection schemes, as well as physical links enabled for transmissions.
iii. **The statistic collections** include the spectrum occupancy, the channel busy intervals, the receiver errors, the received power and interference levels, the energy consumption.
iv. **The medium access logic** defines the rules for timely controlling the access to the shared medium, the acknowledgement policies, and the unicast/multicast delivery modes.
v. **The radio virtualization** creates multiple logical interfaces with different configurations on top of the same hardware.

Before specifying the UPI_R interface, we analysed how the programmable radio architectures offered by WiSHFUL support these general requirements. These architectures have been developed on top of very heterogeneous hardware, in terms of computational resources and reconfiguration capabilities. Each architecture has some intrinsic limits that have to be explicitly considered by experimenters. These aspects are reported in tabular form (rows represent different use cases) for Iris, TAISC and WMP. Different tables are used for spectrum allocation, wireless link set-up, radio monitoring and virtualization functionality.

Table 2 illustrates an example of such a table. Each row maps the high-level UPI_R requirements on a programmable radio architecture. The first three columns were copied from the requirements defined in D2.1. The last column describes the level of support given by a specific radio architecture (Iris, TAISC or WMP) for that requirement.

Table 2 Example table for mapping the UPI_R requirements into the current support given by the WiSHFUL programmable radio architectures (Iris, TAISC or WMP).

UPI_R aspect (e.g. Spectrum allocation)			
UC#	**Actors**	**UC name and short description**	**Architecture x support level and short description**
3.1	Local Controller Physical layer	*Use Case Name*	*Level of support (No, Low, Medium, High)*
		Use Case short description.	*Description of current support level.*

3.1 Iris

Table 3 Support for spectrum allocation use cases in Iris.

Spectrum allocation			
UC#	**Actors**	**UC and short description**	**Iris**
3.1	Local Controller Physical layer	Get spectral capabilities of the RF hardware.	Low
		The local controller must be able to obtain the spectral capabilities of the RF hardware such as: supported bandwidth, supported frequency range(s), etc.	The frequency and bandwidth are set within the XML description of the radio system and available to the core system. The fundamental hardware limitation is defined by the USRP daughterboard used.
3.2	Local Controller Lower MAC	Get supported channel configurations.	Low
		The local controller must be able to obtain the channel configurations supported by the system, including channel bonding, subcarrier selection, etc.	Same as 3.1
3.3	Global Controller Lower MAC	Restrict spectrum usage	No
		The global controller must be able to configure the spectrum that can be used by a node.	The core goal of flexibility and software nature of Iris make the limitation of functionality difficult. This may be possible but is currently not supported.
3.4	Local Controller Lower MAC / Physical layer	Change RX and TX spectrum configurations.	High
		The local controller must be able to control the spectrum used for sending and receiving data choosing from the configurations supported according to 3.1 and 3.2 and within the restrictions imposed by 3.3.	Reconfiguration of radio systems is a core goal for Iris.

Table 4 Support for wireless link set-up use cases in Iris.

Wireless link set-up			
UC#	**Actors**	**UC and short description**	**Iris**
3.5	Local Controller Transceiver	Get supported link configurations	High
		The local controller must be able to obtain the supported link parameters of the transceiver such as TX power, RX gain, CS thresholds, etc.	All configurable parameters in Iris components are registered with the component manager.
3.6	Local Controller Antenna	Get supported antenna configurations	Low
		The local controller must be able to obtain the supported antenna configurations.	Antenna configuration depends purely on the underlying hardware configuration, which is unavailable to Iris. However, in purely static settings this may be known.
3.7	Local Controller Baseband	Get supported baseband functionality	High
		The local controller must be able to obtain the supported configurations of the baseband (or equivalent) such as modulation scheme, coding scheme, etc.	The Iris components that define this functionality must be registered with the component control.
3.8	Local Controller Physical layer	Get MIMO capabilities	Low
		The local controller must be able to determine the MIMO capabilities of the whole RF system (diversity, beamforming, etc.)	See 3.6
3.9	Local Controller Physical layer	Configure link parameters	High
		The local controller must be able to control and modify the parameters related to the different RF subsystems as specified in 3.5 – 3.8.	The supported parameters can be configured through Iris controllers.
3.10	Global Controller Local Controller	Query and restrict link parameter boundaries	No
		The global controller must be able to obtain the relevant link information from the local controller and impose restrictions on the used link configurations	See 3.3

Table 5 Support for radio monitoring use cases in Iris.

Radio Monitoring			
UC#	**Actors**	**UC and short description**	**Iris**
3.11	Local Controller Physical layer	Get raw RF measurements	Medium
		The local controller must be able to obtain raw RF measurements as reported by the underlying hardware. Depending on the hardware these can be I/Q samples, RSSI values, LQI/LSI estimates, channel occupation, noise level, etc.	Iris components typically operate on I/Q samples and these may be made available.
3.12	Local Controller Physical layer	Trigger spectrum scanning	Medium
		The local controller must be able to trigger spectrum scanning and to get results (e.g. received PHY frames on each scanned channel).	Iris may be easily reconfigured to scan the spectrum or spectrum scanning functionality can be deployed in parallel to other functions.
3.13	Local Controller Lower MAC	Sniff and inject raw PHY frames	Low
		The local controller must be able to get a copy of each received physical layer frame. Moreover, it needs a way to inject externally created PHY frames into the wireless network device.	Implementing a radio instance with this functionality should be possible.
3.14	Local Controller Lower MAC	Get channel measurements and tune measurement parameters	Medium
		The local controller must be able to obtain MAC measurements such as error statistics, channel occupancy, etc., as well as control the parameters of these measurements such as the energy detection threshold.	Iris is well suited to implement monitoring and tuning with Iris controllers.

Table 6 Support for medium access logic use cases in Iris.

Medium Access Logic			
UC#	**Actors**	**UC and short description**	**Iris**
3.15	Local Controller Upper MAC	Control framing	Medium
		The local controller must be able to control the segmentation and aggregation of packets, as well as the definition of customized header fields.	Supported.
3.16	Local Controller Lower MAC	Handshake mechanism	Medium
		The local controller must be able to define the sequence of control packets to be sent in each channel access (e.g. 2-way or 4-way handshake, 2-way handshake with reverse link, etc.) and the control information to be included in RTS/CTS frames.	Handshake logic is programmable through the Iris controller.
3.17	Local Controller Lower MAC	Carrier sense mechanism	Medium
		The local controller must be able to modify the collision avoidance policy used by the MAC logic such as the mechanism used (energy-based, preamble-based, virtual, etc.) and the classification method (binary, multi-level).	Adaptive control and MAC logic is supported by Iris.
3.18	Local Controller Lower MAC	Definition of access times	No
		The local controller must be able to modify the timing parameters of the MAC logic. This includes but is not limited to: contention window selection, back-off freezing, inter-frame space, multi-node synchronization, etc.	The processing platform of Iris (GPP) does not lend itself to tight timing control. This may be possible through using the FPGA available on the USRP, although this method is not well developed.
3.19	Local Controller Lower MAC	Definition of ACK policy	High
		The local controller must be able to control the ACK policy used by the MAC logic. This includes artificial dropping, accurate time-stamping, handshaking, frame configuration, etc.	ACK policies are defined in the Iris controller and may be adapted.

Table 7 Support for virtualization functionality use cases in Iris.

Virtualization functionality			
UC#	**Actors**	**UC and short description**	**Iris**
3.20	Global Controller Local Controller	Definition of partitioning rules	High
		The controller has to allocate the hardware resources for each instantiated virtual interface.	Iris supports the definition of multiple radio chains and their association to a hardware platform.
3.21	Global Controller Virtual Radio Manager	Definition of access priorities	High
		The controllers must be able to define the hardware access priority of the different virtualized interfaces.	See 3.20.
3.22	Local Controller Virtual Radio Manager	Conflict resolution	Low
		The virtual radio manager should be able to handle conflicts and notify the local controller when they arise.	Iris supports some conflict identification functionality, but additional resolution abilities are necessary.

3.2 TAISC

Table 8 Support for spectrum allocation use cases in TAISC.

Spectrum allocation			
UC#	**Actors**	**UC and short description**	**TAISC CC2520**
3.1	Local Controller Physical layer	Get spectral capabilities of the RF hardware	Low
		The local controller must be able to obtain the spectral capabilities of the RF hardware such as: supported bandwidth, supported frequency range(s), etc.	The management interface contains a function to list all the implemented modules by the TAISC on the current platform. Based on the result (ex. CC2520 module) the local controller looks up the requested info.
3.2	Local Controller Lower MAC	Get supported channel configurations	Low
		The local controller must be able to obtain the channel configurations supported by the system, including channel bonding, subcarrier selection, etc.	Same as 3.1
3.3	Global Controller Lower MAC	Restrict spectrum usage	Low
		The global controller must be able to configure the spectrum that can be used by a node.	The restriction policy/decision is Upper MAC functionality. But the configuration happens over UPI_R and will be supported.

3.4	Local Controller Lower MAC / Physical layer	Change RX and TX spectrum configurations	Possible
		The local controller must be able to control the spectrum used for sending and receiving data choosing from the configurations supported according to 3.1 and 3.2 and within the restrictions imposed by 3.3.	Supported but not always applicable (TSCH has a strict scheme for RX/TX slots).

Table 9 Support for wireless link set-up use cases in TAISC.

Wireless link set-up			
UC#	**Actors**	**UC and short description**	**TAISC CC2520**
3.5	Local Controller Transceiver	Get supported link configurations	Medium
		The local controller must be able to obtain the supported link parameters of the transceiver such as TX power, RX gain, CS thresholds, etc.	We will be able to list the supported link parameters.
3.6	Local Controller Antenna	Get supported antenna configurations	Low
		The local controller must be able to obtain the supported antenna configurations.	TAISC can implement an antenna module. See 3.1: based on the result we can retrieve this info on antenna availability.
3.7	Local Controller Baseband	Get supported baseband functionality	Low
		The local controller must be able to obtain the supported configurations of the baseband (or equivalent) such as modulation scheme, coding scheme, etc.	See 3.1 (for CC2520 there is only one MCS, for other radio chips they may more basedband functionality).
3.8	Local Controller Physical layer	Get MIMO capabilities	Low
		The local controller must be able to determine the MIMO capabilities of the whole RF system (diversity, beamforming, etc.)	See 3.1 (not available for constrained sensor platforms, could be interesting for SDR platforms)
3.9	Local Controller Physical layer	Configure link parameters	High
		The local controller must be able to control and modify the parameters related to the different RF subsystems as specified in 3.5 – 3.8.	The supported parameters can be configured. Number of paramters are limited for constrained sensor platforms (mainly TX power, CCA, RX sensitivity).
3.10	Global Controller Local	Query and restrict link parameter boundaries	Low
		The global controller must be able to	The restriction policy/decision is

	Controller	obtain the relevant link information from the local controller and impose restrictions on the used link configurations.	Upper MAC functionality. But the configuration happens over UPI_R and will be supported.

Table 10 Support for radio monitoring use cases in TAISC.

Radio Monitoring			
UC#	**Actors**	**UC and short description**	**TAISC CC2520**
3.11	Local Controller Physical layer	Get raw RF measurements	High
		The local controller must be able to obtain raw RF measurements as reported by the underlying hardware. Depending on the hardware these can be I/Q samples, RSSI values, LQI/LSI estimates, channel occupation, or noise level, etc.	Currently supported through the same interface via meta-data attached to each regular data packet (via data plane). Chains can store RSSI measurements and pass them on. For single measurements, the data can also be passed via getParameter in the control interface (via control plane).
3.12	Local Controller Physical layer	Trigger spectrum scanning	High
		The local controller must be able to trigger spectrum scanning and to get results (e.g. received PHY frames on each scanned channel).	A scanning chain can be activated (see above).
3.13	Local Controller Lower MAC	Sniff and inject raw PHY frames	High
		The local controller must be able to get a copy of each received physical layer frame. Moreover it needs a way to inject externally created PHY frames into the wireless network device.	Sniffing is supported, injection possible through chain support.
3.14	Local Controller Lower MAC	Get channel measurements and tune measurement parameters	Medium
		The local controller must be able to obtain MAC measurements such as error statistics, channel occupancy, etc., as well as control the parameters of these measurements such as the energy detection threshold.	Highly related to the MAC, which is implemented by the active chains. Supported by TAISC.

Table 11 Support for medium access use cases in TAISC.

Medium Access Logic			
UC#	**Actors**	**UC and short description**	**TAISC CC2520**
3.15	Local Controller Upper MAC	Control framing	Medium
		The local controller must be able to control the segmentation and aggregation of packets, as well as the definition of customized header fields.	Supported.
3.16	Local Controller Lower MAC	Handshake mechanism	High
		The local controller must be able to define the sequence of control packets to be sent in each channel access (e.g. 2-way or 4-way handshake, 2-way handshake with reverse link, etc.) and the control information to be included in RTS/CTS frames.	Handshake logic is programmed in a chain combined with its wrapper.
3.17	Local Controller Lower MAC	Carrier sense mechanism	High
		The local controller must be able to modify the collision avoidance policy used by the MAC logic such as the mechanism used (energy-based, preamble-based, virtual, etc.) and the classification method (binary, multi-level).	TAISC has a strong focus on this kind of functionality. Fully supported and highly flexible by programming the right sequence of instructions in one ore more chains.
3.18	Local Controller Lower MAC	Definition of access times	High
		The local controller must be able to modify the timing parameters of the MAC logic. This includes but is not limited to: contention window selection, back-off freezing, inter-frame space, multi-node synchronization, etc.	Fully supported and highly flexible by programming the right sequence of instructions in one ore more chains.
3.19	Local Controller Lower MAC	Definition of ACK policy	High
		The local controller must be able to control the ACK policy used by the MAC logic. This includes artificial dropping, accurate time-stamping, handshaking, frame configuration, etc.	Fully supported and highly flexible by programming the right sequence of instructions in one ore more chains.

Table 12 Support for virtualization functionality use cases in TAISC.

Virtualization functionality			
UC#	**Actors**	**UC and short description**	**TAISC CC2520**
3.20	Global Controller Local Controller	Definition of partitioning rules	No
		The controller has to allocate the hardware resources for each instantiated virtual interface.	Not yet supported. Only one chain can be active at the time. Time based switching between protocols is supported.
3.21	Global Controller Virtual Radio Manager	Definition of access priorities	No
		The controllers must be able to define the hardware access priority of the different virtualized interfaces.	Not yet supported, see 3.20.
3.22	Local Controller Virtual Radio Manager	Conflict resolution	No
		The virtual radio manager should be able to handle conflicts and notify the local controller when they arise.	Not yet supported

3.3 Wireless MAC Processor

For WMP an extra column is added to the tables because two implementations exist, one for the Broadcom NIC and one for the WARP board.

Table 13 Support for spectrum allocation use cases in WMP.

Spectrum allocation				
UC#	**Actors**	**UC and short description**	**WMP Broadcom**	**WMP WARP**
3.1	Local Controller Physical layer	Get spectral capabilities of the RF hardware.	High	High
		The local controller must be able to obtain the spectral capabilities of the RF hardware such as: supported bandwidth, supported frequency range(s), etc.	Supported through OS command (iw/iwlist)	Supported through custom-made control interface.
3.2	Local Controller Lower MAC	Get supported channel configurations	High	High
		The local controller must be able to obtain the channel configurations supported by the system, including	Supported through OS command (iw/iwlist).	Supported through custom-made control interface.

		channel bonding, subcarrier selection, etc.		
3.3	Global Controller Lower MAC	Restrict spectrum usage	Possible	Possible
		The global controller must be able to configure the spectrum that can be used by a node	Through driver-level configurations of regulatory domains.	Through custom-made control interface.
3.4	Local Controller Lower MAC / Physical layer	Change RX and TX spectrum configurations	Medium	Medium
		The local controller must be able to control the spectrum used for sending and receiving data choosing from the configurations supported according to 3.1 and 3.2 and within the restrictions imposed by 3.3.	Central frequency can be controlled through OS command or Engine action; bandwidth is fixed to 20 MHz.	Central frequency can be controlled through custom-made control interface or Engine action; bandwidth scale supported by reprogramming the FPGA (static tuning).

Table 14 Support for wireless link set-up use cases in WMP.

Wireless link set-up				
UC#	**Actors**	**UC and short description**	**WMP Broadcom**	**WMP WARP**
3.5	Local Controller Transceiver	Get supported link configurations	Low	Medium
		The local controller must be able to obtain the supported link parameters of the transceiver such as TX power, RX gain, CS thresholds, etc.	TX power can be read through OS command; AGC can be read in HAL registers; CS thresholds cannot be read.	TX power can be read through custom-made control interface; other readings possible but not implemented yet.
3.6	Local Controller Antenna	Get supported antenna configurations	Low	Medium
		The local controller must be able to obtain the supported antenna configurations.	Current selected antenna can be read through OS command (only two antennas in diversity mode supported).	Current selected antenna can be read through custom-made control interface; more complex antenna systems can be added, but the control interface has to be extended.

3.7	Local Controller Baseband	Get supported baseband functionality	High	High
		The local controller must be able to obtain the supported configurations of the baseband (or equivalent) such as modulation scheme, coding scheme, etc.	Supported through OS command (iw/iwlist).	Supported through custom-made control interface.
3.8	Local Controller Physical layer	Get MIMO capabilities	No	No
		The local controller must be able to determine the MIMO capabilities of the whole RF system (diversity, beamforming, etc.)	MIMO is not supported in the Broadcom card.	MIMO is not currently supported in the WARP prototype (possible in future extensions).
3.9	Local Controller Physical layer	Configure link parameters	Medium	Medium
		The local controller must be able to control and modify the parameters related to the different RF subsystems as specified in 3.5 – 3.8.	Transmission antenna and modulation scheme can be specified in OS commands and Engine actions; TX power only in OS command.	Transmission antenna and modulation scheme can be specified in custom-made control interface commands and Engine actions; TX power only in control interface command.
3.10	Global Controller Local Controller	Query and restrict link parameter boundaries	Possible	Possible
		The global controller must be able to obtain the relevant link information from the local controller and impose restrictions on the used link configurations.	Not yet supported.	

Table 15 Support for radio monitoring use cases in WMP.

Radio Monitoring				
UC#	**Actors**	**UC and short description**	**WMP Broadcom**	**WMP WARP**
3.11	Local Controller Physical layer	Get raw RF measurements	Medium	High
		The local controller must be able to obtain	Through driver-level commands, it is	Through custom-made control interface, it is

		raw RF measurements as reported by the underlying hardware. Depending on the hardware these can be I/Q samples, RSSI values, LQI/LSI estimates, channel occupation, noise level, etc.	possible to read RSSI values; low-level channel traces (idle/busy timings) can be programmed inside the Engine and read through the custom-made control interface.	possible to read RSSI, I/Q samples; low-level channel traces (idle/busy timings) can be programmed inside the Engine and read through the custom-made control interface; more memory available for traces.
3.12	Local Controller Physical layer	Trigger spectrum scanning	Low	Low
		The local controller must be able to trigger spectrum scanning and to get results (e.g. received PHY frames on each scanned channel).	Supported through OS command (iw/iwlist), but limited statistics.	Supported through custom-made control interface, but limited statistics.
3.13	Local Controller Lower MAC	Sniff and inject raw PHY frames	Medium	Medium
		The local controller must be able to get a copy of each received physical layer frame. Moreover, it needs a way to inject externally created PHY frames into the wireless network device.	Sniffing can be supported through OS commands (iwconfig) and/or MAC Engine actions. Frame forging can be supported by means of MAC Engine actions.	Sniffing can be supported through custom-made control interface and/or MAC Engine actions. Frame forging can be supported by means of MAC Engine actions.
3.14	Local Controller Lower MAC	Get channel measurements and tune measurement parameters	Medium	Low
		The local controller must be able to obtain MAC measurements such as error statistics, channel occupancy, etc., as well as control the parameters of these measurements such as the energy detection threshold.	Receiver errors can be logged by programming the MAC Engine and can be read by the custom-made control interface.	Limited statistics on receiver errors currently available.

Table 16 Support for medium access logic use cases in WMP.

Medium Access Logic				
UC#	**Actors**	**UC and short description**	**WMP Broadcom**	**WMP WARP**
3.15	Local Controller Upper MAC	Control framing	Medium	Medium
		The local controller must be able to control the segmentation and aggregation of packets, as well as the definition of customized header fields.	Customized headers supported by MAC Engine API (write/read actions). Aggregation possible at the driver level (not currently supported).	Customized headers supported by MAC Engine API (write/read actions).
3.16	Local Controller Lower MAC	Handshake mechanism	High	High
		The local controller must be able to define the sequence of control packets to be sent in each channel access (e.g. 2-way or 4-way handshake, 2-way handshake with reverse link, etc.) and the control information to be included in RTS/CTS frames.	Handshake mechanisms are programmed in the XFSM logic by combining the elementary MAC Engine API.	Handshake mechanisms are programmed in the XFSM logic by combining the elementary MAC Engine API.
3.17	Local Controller Lower MAC	Carrier sense mechanism	Low	Medium
		The local controller must be able to modify the collision avoidance policy used by the MAC logic such as the mechanism used (energy-based, preamble-based, virtual…) and the classification method (binary, multi-level).	Feasible only at the MAC level (e.g. virtual carrier sense) by programming a specific XFSM.	Currently, only at the MAC level (e.g. virtual carrier sense) by programming a specific XFSM; more classification methods of channel state possible, but not yet implemented.
3.18	Local Controller	Definition of access times	High	High

	Lower MAC	The local controller must be able to modify the timing parameters of the MAC logic. This includes but is not limited to: contention window selection, back-off freezing, inter-frame space, multi-node synchronization, etc.	Fully supported by programming the MAC logic and parameters used by the XFSM.	Fully supported by programming the MAC logic and parameters used by the XFSM.
3.19	Local Controller Lower MAC	Definition of ACK policy	Medium	Medium
		The local controller must be able to control the ACK policy used by the MAC logic. This includes artificial dropping, accurate time-stamping, handshaking, frame configuration, etc.	Fully supported by programming the MAC logic and parameters used by the XFSM, if the ack preparation time is compatible with the protocol timings.	Fully supported by programming the MAC logic and parameters used by the XFSM, if the ack preparation time is compatible with the protocol timings.

Table 17 Support for Virtualization functionality use cases in WMP.

Virtualization functionality				
UC#	**Actors**	**UC and short description**	**WMP Broadcom**	**WMP WARP**
3.20	Global Controller Local Controller	Definition of partitioning rules	Low	Medium
		The controller has to allocate the hardware resources for each instantiated virtual interface.	Only time-based switching between two different protocols are supported; rules are defined in a custom-made controller.	Currently only time-based with a local controller, but up to 16 protocols can be loaded simultaneously; in principle, more complex rules are possible by instantiating multiple engines.
3.21	Global Controller Virtual Radio Manager	Definition of access priorities	No	Possible
		The controllers must be able to define the hardware access priority of the different virtualized	Only one protocol is active at a given time.	In case of multiple engines, priorities can be defined in the custom-made control interface.

		interfaces.		
3.22	Local Controller Virtual Radio Manager	Conflict resolution	No	Possible
		The virtual radio manager should be able to handle conflicts and notify the local controller when they arise.		In case of multiple engines, virtualization conflicts arise an event named “virtualization_conflict”.

4 Abstracting Radio Architectures

From the analysis of Iris, TAISC and WMP, it is interesting to observe that there is a valuable common set of functionalities (implemented in different ways) and approaches that can be abstracted for the definition of UPI_R. More detailled, all discussed architectures rely on **primitive components**, called data processing blocks, i.e. commands or actions, which depend on the hardware capabilities and cannot be programmed. They abstract the hardware systems in a set of **configuration parameters,** which may change the hardware operating conditions (e.g. the transmission channel). The architectures **signal events** indicating the occurrence of specific hardware operations (e.g. the reception of a new packet). Moreover, the architectures define an **execution environment** (called PHY engine, Stack engine, MAC engine, or TAISC engine) able to run **radio programs** defined in a high-level programming language. In case of Iris, such a language describes the links between primitive components in an acyclic graph with the sequential processing operations required by the PHY, while in case of TAISC and WMP the program has a more complex logic, which requires the definition of a control flow specified in a chain or in a state machine. A set of radio programs implementing very similar operations (e.g. CSMA and TDMA access protocols) have been defined for all the architectures by using the architecture-specific programming languages. Both for Iris and WMP, some **local controllers** can reconfigure the hardware parameters or the radio program according to the logic defined in the **control program**.

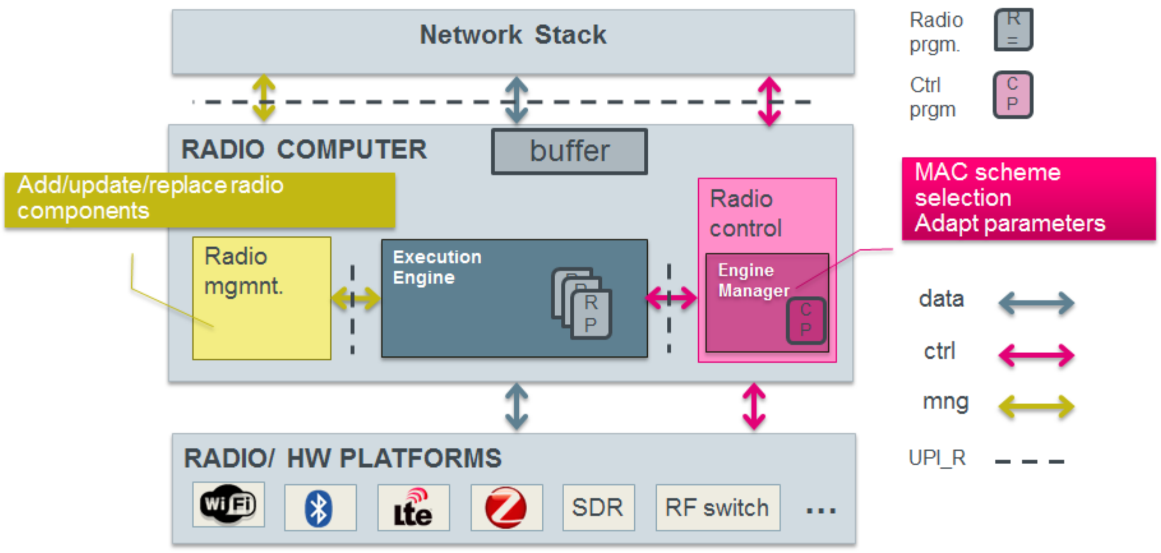

Figure 8 Generic architecture for programmable radio platforms: radio programs are loaded into the execution engine, while the radio controller manages program configuration and switching.

Figure 8 shows the generalization of the WiSHFUL radio programmable architectures (built on top of different hardware platforms) in terms of Radio Programs, Execution Engine, Radio Manager, Radio Controller and Control Program. The **Radio Programs** specify the logic for driving the hardware platforms and implementing lower-MAC protocols, modulation/demodulation schemes or other processing operations on the hardware platform (e.g. spectrum scanning schemes, interference estimation schemes, localization schemes). The **Execution Engine** provides the environment for running the Radio Programs, while the **Radio Manager** is responsible of injecting, compiling and activating different radio programs in the execution environment. Finally, the **Radio Controller** configures or reconfigures the Radio Programs and the hardware platform during the initialization of the radio or during the radio activity, according to the rules specified in the **Control Program**.

The radio programs loaded on the platform and the hardware systems expose a set of configuration parameters, which represent the **platform capabilities**. Moreover, the hardware platform is able to expose some low-level signals and internal states to the Radio Controller to be used as **monitor measurements** for estimating the network conditions and trigger radio adaptations. The Control Program configures the platform capabilities (e.g. transmission power, channel, radio program to be activated, etc.) and the program-dependent capabilities (e.g. slot size, contention window, etc.) in a list of parameters and relevant values, which may change as a function of the monitored measurements.

5 Unified Radio Control Interface

According to the general architecture for programmable radio platforms, presented in the previous section, the UPI_R interface is responsible of three main actions:

i. **Loading and activating** the selected radio programs on the execution environments.
ii. **Monitoring** the node and network conditions by aggregating the platform signals and internal states into measurements and logical contexts.
iii. **Configuring** the hardware platform and the radio programs according to the available capabilities and to the logic defined in the control program.

To support these actions, the UPI_R interface exposes some commands, that enable the configuration of the radio platforms, and catches some signals raised by radio platforms, that enable the estimation of the network operating contexts and the definition of the adaptation mechanisms implemented by the controllers.

5.1 Radio Capabilities

The UPI_R interface is able to monitor and configure the radio behavior thanks to the abstraction of the hardware platform and radio programs in terms of **Radio Capabilities**. We define three different types of radio capabilities: configurable **Parameters**, low-level **Measurements** and hardware **Events**. The configurable parameters specify the configuration of the hardware platform and the initialization of the global variables of the loaded radio program. The low-level measurements are provided by the platform in some internal registers that track the received signal strength, the receiver errors, etc. The hardware events are asynchronous signals, which can be directly exposed to the controller or aggregated in a sequence of events whose occurrence can be signaled to the controller.

The list of radio capabilities is intrinsically extensible because they depend on software and hardware releases, which are continuously updated. However, we define a core set of basic capabilities, which are represented by a pre-defined list of identifiers. Each platform can obviously support the whole list of capabilities or a subset of such a list, depending on the hardware flexibility and on the loaded radio programs.

Parameters correspond to the configuration registers of the hardware platform and to the variables used in the radio programs. For each parameter, a range of valid values can also be specified. Table

18 summarizes the list of core parameters, with an identifier, the architectural element dealing with such a parameter (the hardware platform or the radio program) and a short description.

Table 18 – List of UPI_R core parameters

CATEGORY	ID	NAME	ARCHITECTURE COMPONENT	DESCRIPTION
Parameter	1	IEEE802.11_channel	Platform	IEEE 802.11 PHY channel
Parameter	2	IEEE802.11_MCS	Platform	IEEE 802.11 Modulation and Coding Scheme (MCS) index value
Parameter	3	IEEE802.11_CCA	Platform	IEEE 802.11 Clear channel assessment (CCA) threshold
Parameter	4	IEEE802.15.4_channel	Platform	IEEE 802.15.4 PHY channel
Parameter	5	IEEE802.15.4_CCA	Platform	IEEE 802.15.4 Clear channel assessment (CCA) threshold
Parameter	6	TxPower	Platform	Transmission power in dBm
Parameter	7	TxAntenna	Platform	Antenna number selected for transmission
Parameter	8	RxAntenna	Platform	Antenna number selected for reception
Parameter	9	TDMA_SuperFrameSize	TDMA RP	Duration of periodic frames used for slot allocations
Parameter	10	TDMA_NumberOfSyncSlots	TDMA RP	Number of slots included in a frame
Parameter	11	TDMA_AllocatedSlot	TDMA RP	Assigned slot
Parameter	12	TDMA_MAC_PRIORITY_CLASS	TDMA RP	QUEUE class service associated with TDMA radio program
Parameter	13	CSMA_BackoffValue	CSMA RP	CSMA backoff value
Parameter	14	CSMA_CW	CSMA RP	CSMA current value of the Contention Window
Parameter	15	CSMA_CWmin	CSMA RP	CSMA minimum value of the Contention Window
Parameter	16	CSMA_CWmax	CSMA RP	CSMA maximum value of the Contention Window
Parameter	17	CSMA_timeslot	CSMA RP	CSMA duration of the backoff slot
Parameter	18	CSMA_MAC_PRIORITY_CLASS	CSMA RP	QUEUE class service associated with CSMA radio program

The low-level **measurements** are continuously monitored by the hardware platform and by the radio programs. The measurement capabilities can be used to get information and statistics about the state of the physical links or the internal state of the node. In addition, the list of measurements is extensible, although a core list of measurements is provided in Table 19 with the relevant identifier and description.

Table 19 – List of UPI_R core measurements

CATEGORY	ID	NAME	DESCRIPTION
Measurements	1	IEEE802.11_RSSI	Received Signal Strength Indication (RSSI); it refers to the last received frame in dBm.
Measurements	2	IEEE802.11_SNR	Signal-to-noise ratio (SNR) of the last received frame in dB.
Measurements	3	IEEE802.11_busytime	Time interval in which the transceiver has been active (including reception, transmission and carrier sense).
Measurements	4	IEEE802.11_TxActivity	Time interval in which the transceiver has been involved in transmission.
Measurements	5	IEEE802.15.4_LQI	Link Quality Indicator (LQI)
Measurements	6	FER	Frame Erasure Rate (FER)
Measurements	7	BER	Bit Error Rate (BER)
Measurements	8	goodPreamble	Number of preambles correctly synchronized by the receiver.
Measurements	9	badPreamble	Number of receiver errors in synchronizing a valid preamble.
Measurements	10	IEEE802.11_goodPLCP	Number of valid 802.11 PLCP synchronized by the receiver.
Measurements	11	IEEE802.11_badPLCP	Number of wrong 802.11 PLCP errors triggered by the receiver.
Measurements	12	IEEE802.11_goodCRC	Number of success of 802.11 CRC checks.
Measurements	13	IEEE802.11_badCRC	Number of failures of 802.11 CRC checks.
Measurements	14	IEEE802.11_tooLongFrame	Number of receiver errors due to the detection of 802.11 frames exceeding the maximum possible size.
Measurements	15	IEEE802.11_tooShortFrame	Number of receiver errors due to the detection of 802.11 frames shorter than the minimum size.
Measurements	16	Active	Identifier of the active radio program.

The **events** signal the occurrence of a state change that may involve the hardware platform or the logical state of the active radio program. Table 20 shows the core list of envisioned events, with the relevant identifier and description.

Table 20 – UPI_R core list of events.

CATEGORY	ID	NAME	DESCRIPTION
Event	1	ChannelUp	Triggered when the wireless channel switches from idle to busy
Event	2	ChannelDown	Triggered when the wireless channel switches from busy to idle
Event	3	QueueOutUp	Triggered when the frame is injected into the

			physical queue of the platform from the upper MAC
Event	4	RxEnd	Triggered when that receiver operation is finished
Event	5	IEEE802.11_RxPLCPEnd	Triggered at the end of PLCP reception
Event	6	RxPreambleEnd	Triggered at the end of preamble reception
Event	7	RxMACHeaderEnd	Triggered at the end of MAC header reception
Event	8	RxErrorBadCRC	Triggered at the occurrence of a receiver error due a CRC failure
Event	9	RxErrorBadPLCP	Triggered at the occurrence of a receiver error due a PLCP check failure
Event	10	RxErrorQueueOverflow	Triggered at the occurrence of a receiver error due to the overflow of the reception queue
Event	11	TxErrorQueueUnderflow	Triggered at the occurrence of a transmission error due to the underflow of the transmission queue
Event	12	EndTimer	Triggered at the occurrence of a timer expiration
Event	13	IFSExpired	Triggered at the end of inter frame spaces in which the medium has to be sensed as idle
Event	14	CSMA_BackoffExpired	Triggered at the expiration of the backoff countdown
Event	15	TDMA_SlotStart	Triggered at the beginning of a TDMA slot
Event	16	TDMA_SlotEnd	Triggered at the end of a TDMA slot
Event	17	TDMA_FrameStart	Triggered at the beginning of a TDMA frame
Event	18	TDMA_FrameEnd	Triggered at the end of a TDMA frame

5.2 UPI_R Data and Functions

In this section, we detail the first specification of the UPI_R interface by using a C-style pseudo-code. The definition is given by generic abstract functions, whose platform-specific implementation is provided by the adaptation modules defined for the Iris, TAISC and WMP platforms.

5.2.1 UPI_R Data Structures

The information elements used by the UPI_R interface are organized into data structures, which provide information on the platform type and radio capabilities (**event_t, monitor_t, param_t**) of each interface (**NIC_t**) on the available radio programs (**radio_prg_t**), and on the traffic queues (**queue_t**) available for transmissions over the radio interface.

```
/* structure representing each radio interface
it contains an identifier and the platform type (Iris, TAISC, WMP)
*/
struct NIC_t{NIC_id; platform;}

/* structures representing events, monitored measurements and
parameters in terms of couples identifiers/values */
```

```
struct event_t{ event_id; event_value; }
struct monitor_t{ monitor_id; monitor_value; }
struct param_t{ param_id; param_value; }

/* structure coding a radio program in with an identifier,
the platform type for which it has been compiled and the pointer
to the compiled program */
struct radio_prg_t{
   radio_prg_id;
   platform;
   <radio_program_pointer>;
}

/* structure representing a traffic queue with an identifier,
the traffic type and the radio program managing the frame
transmissions for the queue */
struct queue_t{ queue_id;  queue_name;  radio_prg_t; }

/*structures for listing elementary information types:
event_t, monitor_t, param_t, radio_prg_t, NIC, queue */
event_list{ list event_t; }
monitor_list{ list monitor_t; }
param_list{ list param_t; }
radio_prg_list{ list radio_prg_t; }
NIC_list{ list NIC_t; }
queue_list{ list queue_t; }

/* structure representing the radio capabilities of a given
network card NIC_t in terms of event, measurement and
parameter lists */
struct radio_info_t{
   NIC_t;
   event_list;
   monitor_list;
   param_list;
}
```

5.2.2 UPI_R Functions

For defining abstract functions to be mapped into platform-specific implementations, we use the C-like formal definition based on function pointers. Input data and output data of UPI_R functions are based on the data structures defined in the previous section or on other primitive data.

The complete list of functions identified in the first iteration of UPI_R specification is given by the following list, which will be described by grouping the functions into control, management and data plane functions.

```
struct upi_r{
   NIC_list (*getNICs)();
   radio_info_t (*getNICInfo)(NIC_t);
   error_t (*setMonitor)( NIC_t, monitor_list);
   monitor_list (*getMonitor)(NIC_t);
   error_t (*defineEvent)( NIC_t, “name”, <handler>, optional);
   error_t (*setParameter)( NIC_t, param_list);
   param_list (*getParameter)( NIC_t, param_list);
   error_t (*inject)( NIC_t, radio_prg_list, boolean_list);
   radio_prg_list (*getInjected)(NIC_t);
   errot_t (*setActive)( NIC_t, radio_prg_t);
   radio_prg (*getActive)(NIC_t);
   queue_list (*getQueue)(NIC_t);
   error_t (*connect)(NIC_t, queue_t, radio_prg_t);
   error_t (*disconnect)(NIC_t, queue_t);
} UPI_R;
```

a. UPI_R CONTROL FUNCTIONS

Control functions deal with the monitoring and configuration of the radio interfaces during their operation.

The function `getRadioInfo()` for capability discovery is mostly used at bootstrap, but also after the loading of a novel radio program. It returns the list of events, monitor measurements and configuration parameters supported by the node according to the employed radio architecture (Iris, TAISC, WMP) and radio programs. The returned data are structured in three lists of <type,value> couples, which enumerate the supported events, measurements, parameters and their current values.

```
    /* get supported parameters, events and measurements for the platform
       under use. The platforms in Wishful have heterogeneous
       capabilities. */

   radio_info_t (*getRadioInfo)( NIC_t);
```

A second important set of control functions allow to gather low-level radio information and statistics to be used for estimating the network conditions (interference, propagation conditions, contention levels, etc.). For efficiency reasons, the measurements that the controller is willing to monitor can be configured as a sub-set of the whole measurement list by means of the **setMonitor(NIC_t, monitor_list)** function. The entire list of values of the monitored measurements can be read by calling the **getMonitor(NIC_t)** function, while single measurements can be tracked by calling the function **getMonitor(NIC_t, monitor_id)**. While the measurements are notified by explicitly calling the **getMonitor** functions, asynchronous events can also be tracked by opportunistically defining the conditions for triggering an interrupt signal, and the program handler for reacting to such an interrupt. The triggering conditions may correspond to the occurrence of an event supported by the platform, to a sequence of multiple elementary events or to the overcoming of a threshold for the counter of elementary events. The function responsible of event definitions is the **defineEvent** function**.**

```
    /* set a subset of monitor capabilities. If the value field is
    different from zero, the monitor is (re)set to the specified value.
    If the value is null, turns off the monitor. */
    error_t (*setMonitor)( NIC_t, monitor_list);

    // get the list of monitors currently running
    monitor_list (*getMonitor)( NIC_t);

    // get a single measurement of the monitored parameter monitor_id
    monitor_t (*getMonitor)( NIC_t, monitor_id);


    /* associate a function, pointed by <handler>, that must be executed
     * when the aggregated event “name” is triggered.
     * The triggering  condition is reached when the counter of
     * the event identified by event_id is equal to the threshold value
     */
    error_t  (*defineEvent)(  NIC_t,  “name”,  <handler>,  event_id,
threshold);

    /* The triggering  condition is reached when a sequence of events
    coded in the event_list is sequentially verified. */

    error_t (*defineEvent)( NIC_t, “name”, <handler>, event_list);
```

Finally, the last set of control functions permit to configure the platform in use by setting the values of a list of configurable parameters through the function **getParameter(NIC_t, param_list)** and by reading their current values through the function **setParameter(NIC_t, param_list)**.

```
      // (re)set the value(s) of the specified parameter list
      error_t (*setParameter)( NIC_t, param_list);

      // get the value(s) of the specified parameter list
      param_list (*getParameter)( NIC_t, param_list);
```

b. *UPI_R MANAGEMENT FUNCTIONS*

Management functions permit to handle radio programs. A radio program has to be previously loaded on the wireless node to be available. For sending the radio program code on the desired testbed nodes, it is necessary to perform some network-level operations, addressed by using a specific set of UPI_M functions. Section 7 in D4.1 describes the UPI_M interface. Once a radio program has been successfully loaded on the wireless node, it can be in the following states: **loaded**, **injected and active**.

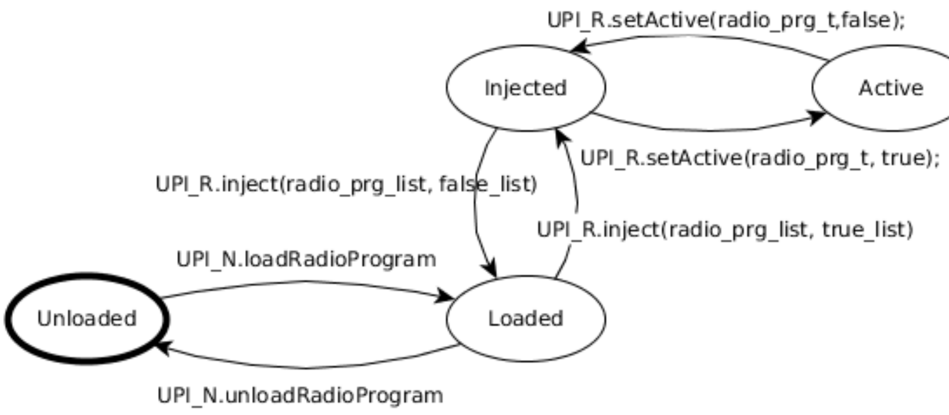
UPI_R.setActive(radio_prg_t,false);
Injected
Active
UPI_R.setActive(radio_prg_t, true);
UPI_R.inject(radio_prg_list, false_list)
UPI_R.inject(radio_prg_list, true_list)
UPI_N.loadRadioProgram
Unloaded
Loaded
UPI_N.unloadRadioProgram

Figure 9 - Management lifecycle for radio programs

Figure 9 illustrates the management lifecycle of WiSHFUL radio program. The lifecycle of a radio program over a WiSHFUL node includes several management states in the set {Unloaded, Loaded, Injected, Active}. These have to be considered in relation to the wireless node, therefore the same radio program can have different management state on different nodes (e.g. the same TDMA radio program can be in the Injected state on a node and in the Active state on another node).

When the experimenter defines his own radio program or wants to use a radio program available in a repository, the program has to be transferred to the wireless node. When the radio program is moved from the repository to the wireless node, it changes to the **Loaded** state. A loaded radio program is stored locally but it resides in a general-purpose memory area (depending on the architecture it may be the hard disk, a CF or SD card, etc). By calling the inject UPI_R function, the radio program is copied in the internal microinstruction memory of the programmable radio platform (where it can be executed by the Engine). Several radio programs can be **Injected** into the radio platform simultaneously, but their execution is started only after the call of the UPI_R **setActive** function, which switches the program state to **Active**.
The generalization of the WiSHFUL programmable radio architectures is currently based on a single Engine without multi-threading capabilities. The local controller can implement code switching by calling the **setActive** function with the desired radio program in different time intervals or after the occurrence of a given event. Note that the Iris platform can support multiple engines able to run multiple programs in parallel. The WMP and TAISC platforms support the execution of multiple programs when the Engine is programmed.

The following commands are defined for managing the radio programs:

```
// injects the radio programs as specified in the     // boolean_list
error_t (*inject)( NIC_t, radio_prg_list, boolean_list);

// Returns the radio programs which are injected on the interface
radio_prg_list (*getInjected)( NIC_t);
```

```
// Activates a radio program. If another program is already running,
// switches between the two radio programs.
errot_t (*setActive)( NIC_t, radio_prg_t);

// returns the radio program that is currently running
radio_prg (*getActive)(NIC_t);
```

c. UPI_R DATA-PLANE FUNCTIONS

In each programmable radio platform supported in WiSHFUL, packets can be marked using packet attributes, meta-data or packet header fields such as Type of Service (TOS). The architecture does not impose any constraint in terms of which layer of the protocol stack can mark a packet. In Contiki, the packet attributes can be read and updated by any protocol in the stack and are (normally) added by the application. In TinyOS using IDRA and OpenWSN, the meta-data can also be updated by any protocol while in Linux systems the marking is typically done by netfilter or Linux Traffic Control (tc).

For configuring the radio data plane, we envision the possibility to specify different MAC protocols for different traffic flows, in order to allocate different channel resources to different traffic types and prioritize some applications. To this purpose, it is required to differentiate the channel access operations performed for transmitting frames belonging to different traffic flows.

There are two options for implementing this feature:

- **Using multiple queues**. Each queue holds packets of a specific traffic flow (e.g. type of service). A different MAC program is simultaneously executed for each traffic queue and a priority is assigned to each queue in case of internal conflicts among multiple MAC programs. The queuing policies and the assignment of different MAC programs to each queue is decided by the local/global control program. This approach implies some level of memory overhead. To realize this functionality, the following functions need to be supported by UPI_R:

```
// get list of available queues
queue_list (*getQueue)(NIC_t);

// associate queue with the corresponding radio program
error_t (*connect)(NIC_t, queue_t, radio_prg_t);

// remove association to queue
error_t (*disconnect)(NIC_t, queue_t);
```

- **Using filtered de-queuing**. A single physical queue is implemented, but the de-queuing of frames is not necessarily performed according to a FIFO discipline. Multiple MAC programs are defined for different traffic flows. Each packet in the queue has some meta-data that indicates the type of service, which in turns addresses the MAC program dealing with the transmission of the frame. When a MAC program de-queues a packet, it asks for a packet with a specific type of service. The local/global control program decides which type of service is assigned to which MAC. This approach implies some additional complexity and hence processing overhead in the queuing system. Again, priorities have to be assigned to different programs for managing conflicts, as shown in the following example:

```
      //define param to set class priority
      param_t param;
      parma.id = TDMA_MAC_PRIORITY_CLASS;
      param.value = “CLASS 1”;

      //assign priority class
      param_list parameters;
      parameters[1]= param;
      UPI_R.setParameter(nic, param);
```

In short, it is thus a trade-off between processing and memory overhead.

Data-plane configuration functions for multiple queues are enabled for radio programmable architectures able to run simultaneously two or more MAC programs. This allows an advanced form of card virtualization because from the perspective of the higher layers (e.g. NET) it looks like there are two or more execution engines that can be independently configured. In this sense, we can make an analogy between a hypervisor that virtualizes computing resources and our “MAC hypervisor” approach that virtualizes network resources.

5.3 Adaptation modules

The UPI_R interface provides a set of functions that can be exploited by the local or global controllers to retrieve information from the programmable radio architectures and controlling their behavior, without knowing the internal details of the platform. This enables the definition of control programs that can drive heterogeneous WiSHFUL platforms, if they support the same required capabilities.

The adaptation modules, whose initial design is documented in this section, perform the mapping of the UPI_R generic functions into platform-dependent implementations.

5.3.1 UPI_R adaptation for Iris

Iris, as a software platform, supports a high degree of flexibility, which eases the addition of WiSHFUL functionality. As shown in Figure 10, the majority of the functionality required to support the WiSHFUL UPIs fits naturally into the Iris controller as this entity already supports the radio monitoring and reconfiguration of Iris.

The software natural of Iris imposes several challenges as well. Since the platform is inherently built to be extended as needed by users, the functionality provided by Iris is not consistent. The radio itself depends entirely on the components employed by a particular user and are typically highly customized to the user’s purposes. Some core functionality is consistently maintained, for example, the architecture of Iris and entry points for WiSHFUL do not change, but the radio operation regularly changes drastically. Therefore, the integration of WiSHFUL functionalities to Iris will center more on the definition of mechanisms to allow users to expose parameters through the WiSHFUL APIs, which occurs during the configuration of the radio platform.

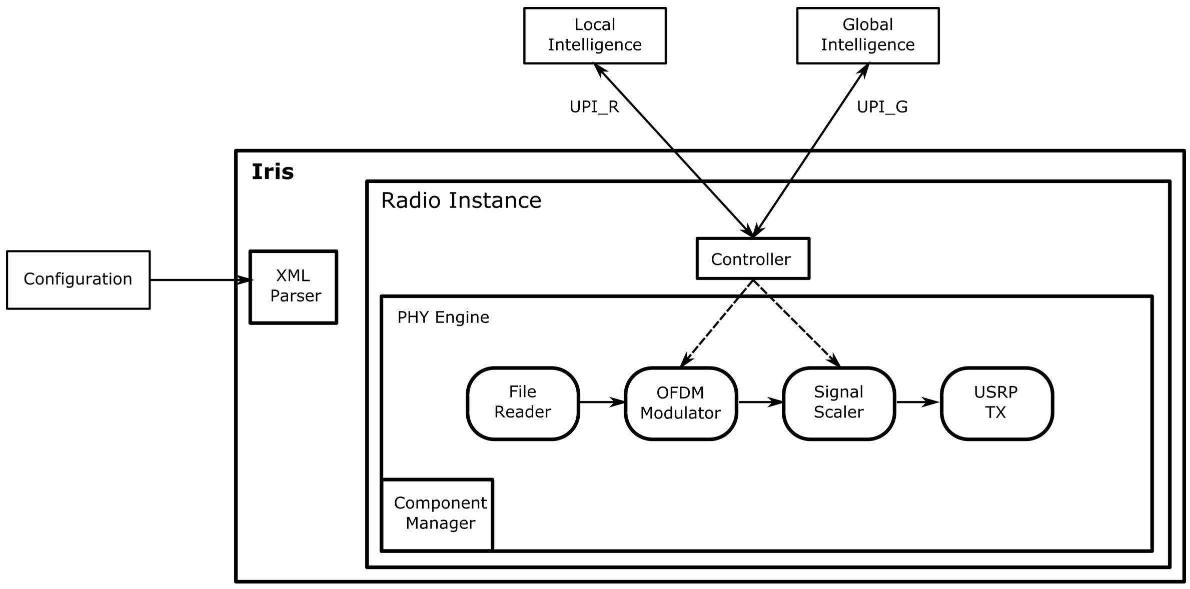

Figure 10 UPI_R adaptation for Iris

5.3.2 UPI_R adaptation for TAISC

Figure 11 illustrates the mapping of the WiSHFUL control and management plane extensions, proposed in Figure 8, on the TAISC architecture. The adaptation layer required for configuring, monitoring and managing TAISC chains implements two repositories:

- A parameter repository used for control purposes.
    - Setting and/or getting configuration parameters
    - Subscribing to monitoring events and/or receiving event updates.
- A chain repository used for management purposes.
    - Adding, activating, switching and/or removing chains.
    - Keeping track of the chain states (e.g. idle, active, running, etc.).

Using repositories introduces minimal the memory overhead, which is a crucial requirement for the constrained devices that are currently targeted by TAISC. It is possible to map all functions currently defined in UPI_R and to add all parameters and events defined in Table 18, Table 19 and Table 20 respectively. Each entry in the repositories (chain, configuration parameter and monitoring event) requires an UID and contains the ROM/RAM addresses that correspond to the chains and chain variables used in TAISC.

The local intelligence or control program uses the UPI_R interface to configure and monitor the TAISC. The adaption layer translates (if necessary) and forwards, based on information stored in the parameter repository, these requests to the TAISC Control Plane (narrow waist) interface described in the TAISC architecture (2.2.2). Similarly, the global intelligence or control program uses the UPI_G interface to configure, monitor and maintain chains on a group of nodes.

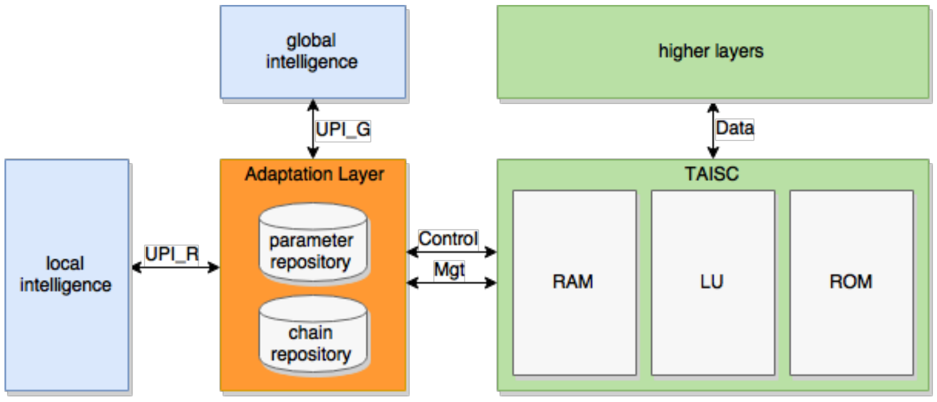

Figure 11 UPI_R adaptation for TAISC

5.3.3 UPI_R adaptation for WMP

For the initial design of this adaptation module, we refer to the WMP prototype available for the Broadcom commercial card (being the current WARP prototype equipped with a simplified network stack). For the Broadcom prototype, the WMP control and management interface is implemented in the MAClet manager, while the interface to the upper MAC and higher-layers is implemented in the driver running on the host. The implementation of UPI_R functions can be performed by mainly working on these software modules (MAClet manager and driver), while some modifications can be also performed in the Engine implementations for supporting a broader set of monitored measurements.

Figure 12 shows an overview of the interactions between the software module implementing the adaptation module, the WMP architecture, and the local and global controllers: the intelligence implemented in the control programs is based on the call of generic UPI_R functions, which are mapped into WMP-specific functions defined in the adaptation module. These functions, in turns, interact with the MAClet manager and driver functions, as well as with the WMP shared memory and transceiver registers.

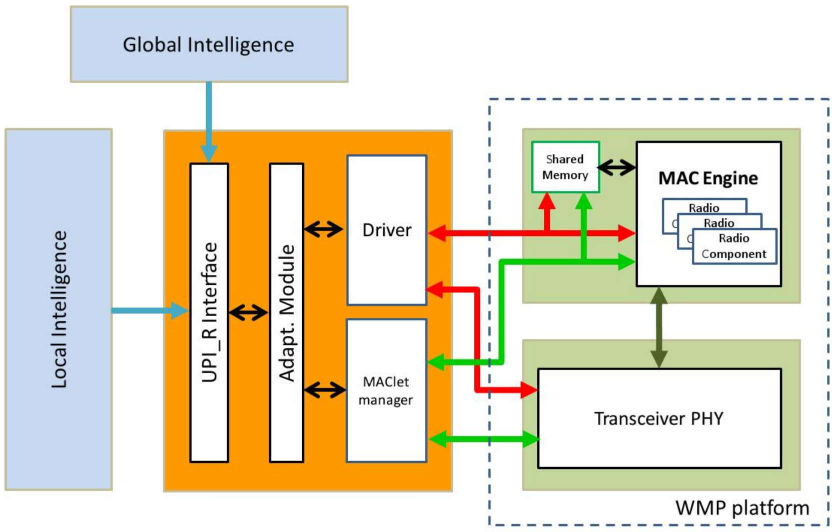

Figure 12 – Adaptation module high-level design for the WMP Broadcom prototype.

Table 21 lists an initial mapping of the UPI_R functions into functions provided by the MAClet manager module, the driver module or both the software modules. Basically, all the functions related to the radio program activation and platform configuration are mapped to MAClet manager functions (e.g. **setActive** and **setRunning**), while the monitoring functions are mapped to both the MAClet manager and the driver.

Table 21 - How UPI_R interfaces are mapped in WMP platform elements from adaptation module

UPI_R interface	**UPI_R INTERFACE MAPPING**	
	Driver	**MAClet manager**
(*getNICs)	x	
(*getNICInfo)	x	x
(*setMonitor)	x	x
(*getMonitor)	x	x
(*defineEvent)	x	x
(*setParameter)		x
(*getParameter)		x
(*inject)		x
(*getInjected)		x
(*setActive)		x
(*getActive)		x
(*getQueue)		x
(*connect)		x
(*disconnect)		x

As far as concerns the radio capabilities, the WMP Broadcom prototype supports only a subset of them related to the hardware capabilities of a standard 802.11b/g PHY.

Table 22 summarizes the supported radio capabilities, by also specifying which architecture element (PHY register, MAC Engine variable or Driver variable) is involved and which software module can configure the different parameters or detect the measurements/events. Note that for the receiver errors both the PHY registers and the MAC Engine variables are involved: the switching of the PHY registers indicate the occurrence of the errors, while the MAC Engine variables count the number of occurrences.

Table 22 -Where the WMP platform capabilites are located and as they are available

Name	Located			Available by	
	PHY register	MAC Engine variable	Driver Variable	MAClet manager	Driver
IEEE802.11_channel	X			X	X
IEEE802.11_MCS	X				X
IEEE802.11_CCA				X	X
Txpower	X			X	X
TXantenna	X			X	X
RXantenna	X			X	X
TDMA_SuperFrameSize		X		X	
TDMA_NumberOfSyncSlots		X		X	
TDMA_AllocatedSlot		X		X	
CSMA_BackoffValue		X		X	
CSMA_CW		X		X	
CSMA_CWmin		X		X	
CSMA_CWmax		X		X	
CSMA_timeslot		X		X	
IEEE802.11_RSSI	X			X	X
IEEE802.11_SNR	X			X	X
IEEE802.11_busytime	X			X	
IEEE802.11_TXactivity	X			X	
FER			X		X
BER			X		X
IEEE802.11_goodPCLP	X	X		X	
IEEE802.11_badPLCP	X	X		X	
IEEE802.11_goodCRC	X	X		X	
IEEE802.11_badCRC	X	X		X	
IEEE802.11_toolongframe	X	X		X	
IEEE802.11_tooshortframe	X	X		X	
active-function	X	X		X	
ChannelUp	X			X	
ChannelDown	X			X	
QueueOutUp	X			X	
RxEnd	X			X	

IEEE802.11_RxPLCPEnd	X			X	
RxPreambleEnd	X			X	
RxMACHeaderEnd	X			X	
RxErrorBadCRC	X			X	
RxErrorBadPLCP	X			X	
RxErrorQueueOverflow	X			X	
TxErrorQueueUnderflow	X			X	
EndTimer	X			X	
IFSExpired	X			X	
CSMA_BackoffExpired	X			X	
TDMA_SlotStart	X			X	
TDMA_SlotEnd	X			X	
TDMA_FrameStart	X			X	
TDMA_FrameEnd	X			X	

For making the measurements and the program variables available to the Driver or the MAClet manager, the MAC Engine copies their values in a shared memory which can be accessed by both the firmware and the host processes. For efficiency reasons, rather than copying the whole set of parameters, the Engine can write in the shared memory only the measurements and events selected by experimenters by using the UPI_R **setMonitor** function.

Similarly, for providing interrupt signals to the adaptation module and to UPI_R at the occurrence of events, the MAC Engine can be programmed for considering only the events or the aggregation of events (e.g. a number of occurrences overcoming a given threshold) defined by experimenters. Most of the WMP events are signaled by flipping bits in PHY registers, while the counting of event occurrences can be stored in program registers. Interrupt singnals can be associated to the switching of these registers to particular values.

An alternative solution to pass events and measurements between the MAC Engine and the upper elements (Driver and MAClet Manager) will be explored during the implementation phase. Instead of writings and readings in the shared memory, which is collision prone in case of frequent access operations, the MAC Engine can periodically collect statistics and send them to the higher layers by means of signaling frames or piggybacked data added to the reception queue. The use of frame-based communication, opportunely identified with tags, will increase the system flexibility and reduce the number of readings/writings and possible collisions on memory access.

6 Example of UPI_R utilization

6.1 Adapting CSMA contention window

6.1.1 Example Description

This example refers to one of the showcases discussed in D2.2: a given number of radio nodes, employing a contention-based access protocol, coexist in the same environment and an increasing number of greedy traffic flows is activated sequentially. Regardless of the PHY technology (e.g. a low-rate PHY, such as the one supported by TAISC, or an high-rate PHY, such as the one supported by 802.11g/n PHY), as the number of greedy traffic flows gets higher, the aggregated performance of the network degrade because of the increased level of congestion. Indeed, it is well known that CSMA performance can be optimized by tuning the contention window as a function of the number of contending nodes (or equivalently on the collision probability experienced in the network). While standard-based contention protocols use exponential backoff mechanisms with fixed minimum and maximum contention windows, WiSHFUL radio platforms allow to *tune the CSMA contention window* as a function of an estimate of the network load conditions.

6.1.2 Requirements for UPI_R

For supporting the dynamic adaptation of the CSMA contention window, it is required to:

- Retrieve information about the fact that the Radio Program currently active is a CSMA protocol, with tunable contention windows, by calling the UPI_R function responsible of capability discovery;
- Aggregate the low-level measurements of the platform for estimating the network congestion level, by monitoring the number of ACK timeouts and the total number of transmitted frames;
- Configure the Radio Program parameter corresponding to the CSMA contention window as a function of the estimated congestion level.

6.1.3 Control Program Pseudo-Code

The first operation is the identification of the radio capabilities of the NIC on node:

```
/* Check NIC capabilities */
nic_list UPI_R.getNICs();
radio_info UPI_R.getNICInfo(nic_list[1]);
radio_prg UPI_R.getRunning(nic_list[1]);
print info (i.e. radio_prg_id, platform, NIC_t, etc.)
```

From the output, the local intelligence retrieves the description of the capabilities of this NIC, on the radio program currently active, on the platform type, etc. In this example, the NIC capabilities are stored in xml, such as the following:

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

<capability id="7">
    <category>parameter</category >
    <name>TXantenna</description>
    <radio_program>ALL(PHY)</radio_program >
    <description>Antenna selected for transmission</description>
</capability>

<capability id="17">
    <category>measurement</category >
    <name>TX_frames</ name >
    <description>Number of transmitted frames</description>
</capability>

<capability id="3">
    <category>event</category >
    <name> QUEUE-OUT-UP </ name >
    <description> Triggered when the frame is injected into the HW platform
physical queue from the upper MAC </description>
```

```
</capability>

...

</ wishful >
```

Now the local controller monitors periodically the number of ACK timeouts and the total number of transitted frames as a measure to detect congestion. If the ratio between the two is higher than 5%, it enlarges the congestion window value, while if this ratio is lower than 0.5%, it reduces it.

```
      /* Monitoring of parameters */

      //initialize parameters to be periodically checked
      ack_timeout=0;
      tx_frames=0;
      CW=CWmin;
      monitor_list  check_monitors={  monitor_t{  ACK_timeout;  0;},
monitor_t{ TX_frames; 0;}};
      //every 10s check timeout/transmitted ratio
      while true {
            sleep 10s;
            check_monitors=UPI_R.getMonitor(nic, check_monitors);
            ack_timeout = check_ monitors[1].value;
            tx_frames = check_ monitors[2].value;
            if(ack_timeout/tx_frames>=0.05)
                  CW=(CW*2>CWmax)? CW*2 : CWmax;
            else if(ack_timeout/tx_frames<=0.005)
                  CW=(CW/2<CWmin)? CW/2 : CWmin;
            UPI_R.setParameter(nic, { param_t{ CSMA_CW; CW; }});
      }
```

6.2 from CSMA to TDMA

6.2.1 Example Description

Also this example refers to the showcases discussed in D2.2: an high number of radio nodes, employing a contention-based access protocol, coexist in the same environment with greedy traffic flows and high collision rates. Being the traffic scenario predictable (the flows are greedy and the number of contending nodes is fixed), the network efficiency can be improved by adopting a TDMA access scheme rather that a contention-based protocol. This is possible on WiSHFUL radio platforms by exploiting the UPI_R interface and the TDMA Radio Program (available for all the platforms Iris, TAISC and WMP). For this purpose, the platform-dependent TDMA Radio Program must be loaded on each node, the program must be configured with an equal super frame size (number of slots), a different slot for each node needs to be allocated and a common temporal signal to all the nodes for synchronizing the start of TDMA frames must be provided.

6.2.2 Requirements for UPI_R

To support switching the Radio Program from CSMA to TDMA, it is required to:

- Retrieve information about the fact that the Radio Program currently active on the nodes is a CSMA protocol, and that the TDMA Radio Program is loaded or available on the WiSHFUL repository.

- Retrieve information about the TDMA configuration parameters, which include the frame size, the slot to be allocated and the start of a new frame;
- Exploit low-level measurements to estimate the number of nodes in the network by monitoring the different MAC addresses of transmitted frames or the collision probability experienced by each node;
- Configure the TDMA Radio Program parameters corresponding to the TDMA frame size and allocated slot on each node;
- Activate the TDMA Radio Program on each node and ensures that the start of the TDMA frame is synchronized among all the network nodes.

6.2.3 Control Program Pseudo-Code

A simplified pseudo-code example of the UPI_R calls implementing the CSMA/TDMA switching is illustrated below.

The first operation is the identification of the radio capabilities of each node:

```
        /* Check NIC capabilities */
        nic_list UPI_R.getNICs();
        radio_info UPI_R.getNICInfo(nic_list[1]);
        print  radio_info  (i.e.  NIC_t;  event_list;  monitor_list;
param_list;)
```

The condition for informing the global controller about the need to switch to TDMA can be based on several observations. In our example, we assume that it depends on the observation of high contention levels in terms of high values of the contention window:

```
        /* Local monitoring of parameters */

        //initialize parameters to be periodically checked
        CW=CW_MIN;
        param_list check_params={ param_t{ CSMA_CW; null; }};
        //every 0.5s check if congested (i.e. if CW>=128)
        while true {
                sleep 0.5s;
                check_params=UPI_R.getParameter(nic, check_params);
                CW = check_params[1].value;
                if(CW>=128)
                        send alert to global controller;
        }
```

The controller is then responsible for verifying that a TDMA Radio Program is active on each node. If this is not the case, such a program must be loaded by using the UPI_R management functions.

Once loaded, the TDMA radio program can be configured with the desired frame size and with a different slot allocated to each node. Finally, since the program natively provides the frame synchronization function among multiple nodes, it is possible to activate the program and make it running on each node.

```
        /* Switching do TDMA */

        // TDMA parameters: 10ms SuperFrame, 2 slots, 1 allocated to
the nic
```

```
    param_list tdma_params={ param_t{ TDMA_SuperFrameSize;
10000},         param_t{TDMA_NumberOfSyncSlots; 2},
          param_t{ TDMA_AllocatedSlot; 1} };
    UPI_R.setParameter( nic, tdma_params);

    //run TDMA radio program (must be previously activated)
    UPI_R.setRunning(nic, tdma_pgm);
```

7 Conclusion

In this deliverable, starting from the presentation of the programmable radio architectures and prototypes available in WiSHFUL (namely, the Iris, TAISC and WMP architectures), we have described the first UPI_R specification. The UPI_R interface has been conceived for offering a **unified interface to experimenters** willing to work on heterogeneous radio platforms and for enabling the definition of **platform-independent adaptation logics** of the MAC/PHY stack. An initial design of the adaptation modules required for mapping the UPI_R interface into platform-specific function calls and an analysis of the capabilities supported by each platform are also presented in this document.

From a methodological point of view, the UPI_R interface has been specified by abstracting the Iris, TAISC and WMP architectures into a general radio architecture, whose behavior is characterized by the platform **radio capabilities** and loaded **radio program**. The radio capabilities have been conveniently abstracted into a set of configurable parameters, detectable events and monitored measurements. The radio programs specify the logic for driving the hardware platforms and implementing lower-MAC protocols, modulation/demodulation schemes or other processing operations on the hardware platform (e.g. spectrum scanning schemes, interference estimation schemes and localization schemes). Radio capabilities and radio programs represent the main data structures used for the definition of UPI_R functions.

For validating the thoroughness of the UPI_R specifications, we considered several use cases dealing with MAC/PHY reconfigurations, by mainly referring to the D2.2 showcases. Two specific examples have been discussed at the end of the document, in terms of general UPI_R requirements and specific UPI_R calls that can be exploited for their implementation. In the first example, a CSMA radio program is adapted to the network congestion state by tuning the contention window as a function of the number of competing nodes. A different adaptation, based on the switching from a radio program to another, where the MAC scheme is changed from CSMA to TDMA in case of greedy traffic flows and high contention levels. Although these preliminary validations have shown that the current UPI_R specification allows dealing with the considered adaptation problems, we expect that some minor refinements could be necessary during the implementation phase (that will last for the next six months of the project).

8 References

[1] The GNURadio Software Radio, http://gnuradio.org/trac

[2] WARP, http://warp.rice.edu/trac

[3] USRP. The universal software radio peripheral, http://www.ettus.com

[4] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, G. M. Voelker, “Sora: High Performance Software Radio Using General Purpose Multi-core Processors”, NSDI 2009.

[5] M. C. Ng, K. E. Fleming, M. Vutukuru, S. Gross, Arvind, H. Balakrishnan, “Airblue: A System for Cross-Layer Wireless Protocol Development”, ACM/IEEE Symp. on Architectures for Networking and Communications Systems (ANCS) 2010.

[6] S. Heath, “Mircoproecssor Architectures, Second Edition: RISC, CISC and DSP”, ISBN-13: 978-0750623032

[7] “rm090”, http://www.rmoni.com/en/products/hardware/rm090

[8] “proflex01”, http://www.lsr.com/embedded-wireless-modules/zigbee-module/proflex01-r2

[9] “TSCH or 802.15.4e”, https://tools.ietf.org/html/draft-ietf-6tisch-tsch-06

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

[11] 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.