



**SIXTH FRAMEWORK PROGRAMME
PRIORITY 2
“Information Society Technologies”**

Project acronym: RUNES

Project full title: Reconfigurable Ubiquitous Networked Embedded Systems

Proposal/Contract no.: IST-004536-RUNES

***D3.5
Interface Description***

Project Document Number: RUNES/D3.5/R-PU¹/v1.0 (official version)

Project Document Date: 23/02/2007

Workpackage Contributing to the Project Document: WP3

Deliverable Type and Security: R-PU

Author(s): Magnus Johansson (editor, cB), Frank Oldewurtel (RWTH), Matthias Wellens (RWTH) and Junaid Ansari (RWTH)

Abstract:

This document describes the interfaces to the lowest layers of the RUNES architecture. These interfaces serve as an abstraction of the platform dependent code.

Keywords: RUNES deliverable, interface, abstraction, platform, link layer, sensor, actuator, virtual operating system

¹ Security Class: PU- Public, PP – Restricted to other programme participants (including the Commission), RE – Restricted to a group defined by the consortium (including the Commission), CO – Confidential, only for members of the consortium (including the Commission)

History

Version	Date	Description, Author(s), Reviser(s)
1.0	26/02/2007	Official version, Magnus Johansson (editor, cB), Frank Oldewurtel (RWTH), Matthias Wellens (RWTH) and Junaid Ansari (RWTH)

Executive Summary

This document defines the interfaces to the lowest layers in the RUNES architecture, handled within work package 3. These interfaces are defined to provide an abstraction of the platform dependent parts of the system. By using the interfaces the higher layer SW can be written in a platform independent way, thus making it easier to port to new platforms.

Contents

	Page
1 Introduction	6
2 Virtual OS interface.....	7
2.1 Data types and constants.....	7
2.2 Interface definitions.....	8
2.2.1 General functions	8
2.2.2 Timer handling.....	8
2.2.3 Message handling	8
2.2.4 Semaphores.....	8
2.2.5 Queues	9
2.2.6 Process handling	9
3 Abstraction Layer Interfaces.....	10
3.1 Link Layer Interface	10
3.2 Sensor Provider Interface	12
3.2.1 Platform interface	14
3.2.2 Sensor interface.....	15
3.2.3 Actuator interface	16
A. Appendix: Example of specific Virtual OS definition (connectBlue Sensor Routing node, ARM7, FreeRTOS)	17
References	23

Figures, Tables

Figure 1 Virtual Operating System.....	7
Figure 2: ULLA architecture.	10
Figure 3: Extension to the ULLA architecture, Sensor Provider Interface.	13

1 Introduction

The purpose of this document is to define the interfaces to the lowest layers in the RUNES architecture. The interfaces are defined to provide an abstraction of the platform dependent parts of the RUNES SW. By defining these interfaces the rest of the RUNES SW can be written without taking different platforms into account. The documents deal with abstraction of the platform resources, operating system, communication links and sensor/actuator representation.

The document begins with a description of the operating system abstraction. It is followed by a description of the platform abstraction layer.

2 Virtual OS interface

Since there are a lot of different operating systems available today, handling heterogeneous operating system might become a large problem if your applications need to be portable to several different platforms. If you write your applications directly interfacing the underlying OS, you have to re-write your code for every OS you need to run your code on top of.

To solve this we have in RUNES chosen to define a *Virtual Operating System*, a common solution to this problem. A virtual operating system is an abstraction layer between the application and the underlying OS. The applications are written using the virtual OS API instead of the actual OS. In the virtual OS the calls are translated to calls to the underlying OS. When you move to a new OS only the translation inside the virtual OS has to be re-written. See Figure 1

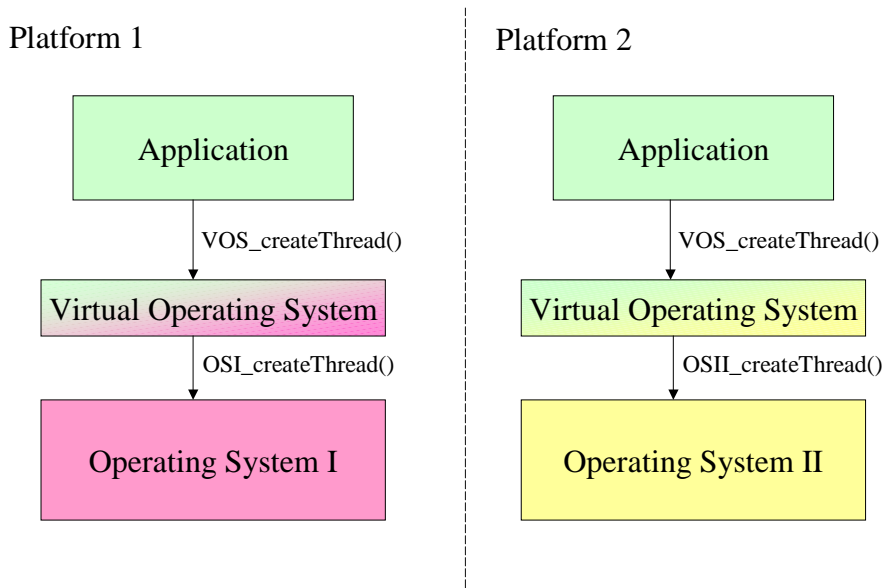


Figure 1 Virtual Operating System

To minimize the maintenance effort for the Virtual OS, the API must be kept as small as possible. We have been very restrictive in adding features to the API. Only the most essential functions of an Operating System have been added to the Virtual OS.

The RUNES virtual operating system is defined for threaded operating systems. The reason is that it is hard to define a common API for both threaded and event driven operating systems. We have chosen to define a threaded API because normally event driven operating systems are used in very resource scarce systems, where you probably cannot afford the overhead of a Virtual OS anyhow.

2.1 Data types and constants

These constants are used to configure the Virtual OS. If you have resource constraints you have to think through the values carefully.

- **OS_MAX_NBR_PROCS.**
Defines the maximum number of processes that can be created.
- **OS_MAX_NBR_TIMERS.**
Defines the maximum number of timers that can be used.

- **OS_MIN_STACK_SIZE.**
Defines how small stack you can use. This sets the lowest level that can be used when you create a new process.

These types are defined to be used in the Virtual OS API.

- **os_Semaphore.**
Used to represent semaphores.
- **os_Queue.**
Used to represent queues.

2.2 Interface definitions

2.2.1 General functions

- **void os_ASSERT(UINT32 c).**
Used for exception handling. If *c* equals 0 the exception handler is called.
- **void os_init(void).**
Init the OS.
- **void os_start(void).**
Start the scheduler. This function ideally never returns.
- **void* os_malloc(UINT32 bytes).**
Allocate *bytes* bytes of memory from the heap. Returns pointer to the allocated memory.
- **void os_free(void* ptr).**
Free previously allocated memory pointed to by *ptr*.
- **UINT32 os_get_clock_time(void).**
Returns the current clock tick count in milliseconds.

2.2.2 Timer handling

- **INT32 os_timer_set(UINT32 pid, INT32 ms, void* msg).**
Set a timer to expire in *ms* milliseconds. When timer expires, the message with the data pointed to by *msg* is sent to the process *pid*. If *pid* == NULL *msg* is sent to the process that called *os_timer_set*. Returns an ID of the timer created.
- **INT32 os_timer_set_periodic(UINT32 pid, INT32 ms, void* msg).**
Same as *os_timer_set()* but creates a periodic timer.
- **void os_timer_cancel(UINT32 id).**
Cancels a previously created timer with the ID *id*.
- **void os_sleep(INT32 ms).**
Puts the current process in sleep for *ms* milliseconds.

2.2.3 Message handling

- **void os_msg_send(INT32 proc_id, void* msg).**
Send a message *msg* to the process with ID *proc_id*.
- **void* os_msg_receive(void).**
Returns a message from the current process' message queue. Blocks if the message queue is empty.

2.2.4 Semaphores

- **void os_semaphore_init(os_Semaphore s).**
Initializes the semaphore *s* of the type *os_Semaphore*.

- `void os_semaphore_post(os_Semaphore s).`
"Give" semaphore *s*.
- `void os_semaphore_try(os_Semaphore s).`
"Takes" semaphore *s*. The call is blocking.
- `INT32 os_semaphore_try_nb(os_Semaphore s).`
Polls, and if available "takes" semaphore *s*. The call is non-blocking, it returns 1 if successful and 0 if not.

2.2.5 Queues

- `os_Queue* os_queue_init(UINT32 len).`
Returns a queue of the length *len* and the type *os_Queue*.
- `void os_queue_send(os_Queue q, void* item).`
Sends the message *item* to the queue *q*.
- `void* os_queue_receive(os_Queue q).`
Returns a message from the queue *q*. The call blocks until a message is available.
- `UINT32 os_queue_nbr_msgs(os_Queue q).`
Returns number of messages in the queue *q*.

2.2.6 Process handling

- `void os_create_process(void *function,
 UINT32 stack_size,
 UINT32 priority,
 UINT8* name).`
Creates a new process with the "main" method *function*, the stack size *stack_size*, the priority *priority* and the name *name*.
- `void os_destroy_process(UINT32 pid).`
Destroys a process with the process id *pid*. All messages in the process message queue are deleted.
- `UINT32 os_get_current_pid(void).`
Returns the process ID of the currently running process.
- `void os_enter_critical(void).`
Disables preemptive context switches.
- `void os_exit_critical(void).`
Re-enables preemptive context switches.
- `void os_disable_irqs(void).`
Disables all interrupts.
- `void os_enable_irqs(void).`
Re-enables all interrupts.

3 Abstraction Layer Interfaces

In general the heterogeneity of present networks is constantly increasing leading to more and more complex software modules controlling the respective network interfaces. Additionally, the software interfaces used to configure such devices vary drastically between technologies and also between platforms.

Furthermore, due to a wide range of sensors and actuator types and the functionalities that they may offer, it is not practical to have the same set of application functionalities for all of the sensors. There may be some common functions like turning the sensor on/off, grabbing data etc. but there may be many other functions that differ from sensor to sensor and the type of interface, it is connected to.

Based on the above facts there is a strong need for abstraction layer interfaces within the RUNES project. Especially, the Link Provider Interface and the Sensor/Actuator Provider Interface will be presented within this chapter.

3.1 Link Layer Interface

The different link-layer technologies are unavoidable due to very different requirements for the various RUNES system components. Hence, it is desirable and beneficial to use just a single programming interface for accessing those.

Within the RUNES project we took the decision to use the Unified Link-Layer API (ULLA) from the GOLLUM project. In the following we will explain the purpose and architecture of ULLA and its advantages in the RUNES context briefly again [RUNES-D1.4.2], [RUNES-D1.4.3], [GD24], [GD34]. This is helpful in order to show the rationale and structure of the Sensor Provider Interface which is similar and described in the following section. The whole specification ULLA is based on is freely available from the GOLLUM project website (<http://www.ist-gollum.org>). The interested reader is referred to several GOLLUM project deliverables describing different aspects of ULLA in more detail.

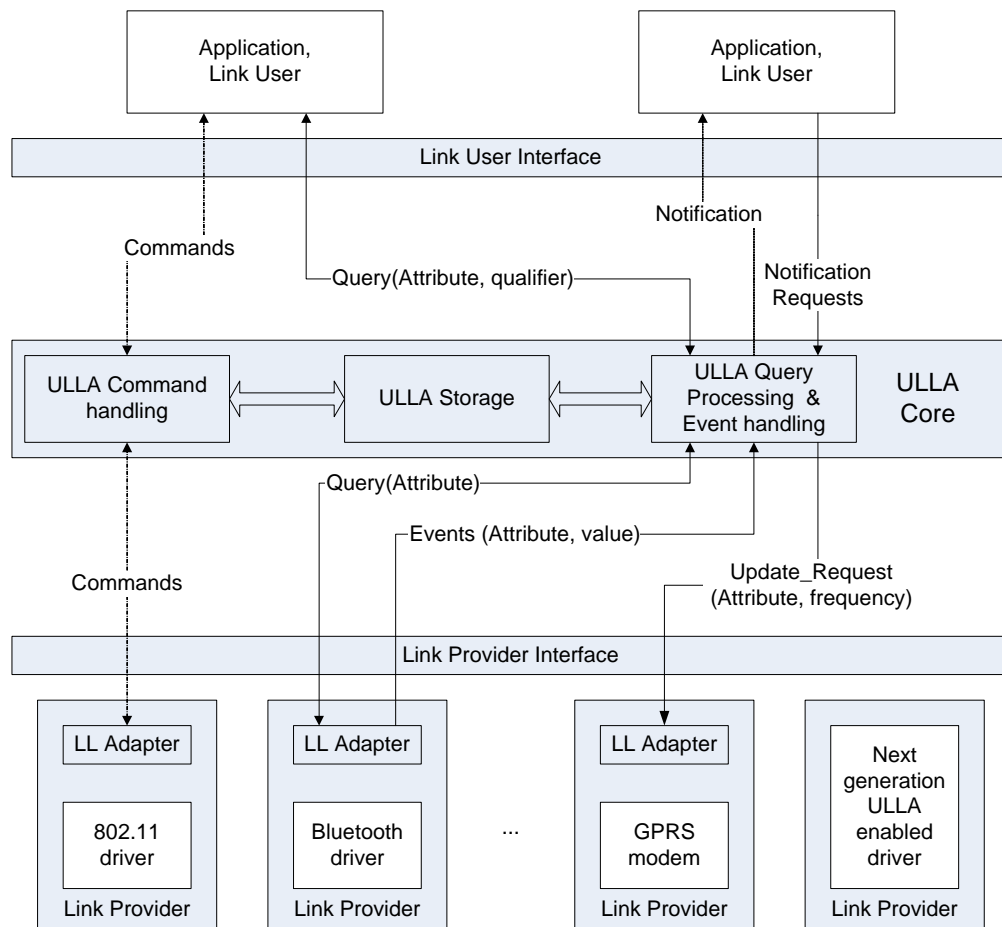


Figure 2: ULLA architecture.

The ULLA architecture is shown in Figure 2. It consists of three major parts and the interconnecting interfaces. If an application wants to use ULLA it registers as Link User (LU). Here, application is not seen only as entity working on OSI layer 7 but any entity sitting above layer two. LUs communicate with ULLA core through the Link User Interface, the upper part of the overall API. The lower part, the Link Provider interface, connects the ULLA core with various Link Providers (LPs), which are abstractions of the network interface cards. LPs and the respective device drivers might directly support the LP interface or could, during the transition phase, being extended by Link Layer Adapters (LLAs), which are software modules implementing the LP interface by using proprietary legacy device driver functions.

The main data unit, ULLA works with, are links, which are registered with ULLA core by the LP after initially detecting them.

The data model used and the ULLA framework follow an object-oriented approach. As one of the main design objectives for ULLA was technology-independence several general so-called attributes, such as rxBitRate, localL2address, or rxSignalStrength, are covered by the main base class ullaLink, which has to be supported by each registered link. The usage of attributes from such a mandatory base class enables comparison of links independently of the underlying technology. Following the object-orientation links can support further more specific classes, starting from classes for specific standards, such as 80211Link or the wsnLink for wireless sensor networks. These classes cover technology-specific attributes and can be used to configure or monitor characteristics specific for one technology. Classes for standard-amendments and even vendor-specific classes complete the class hierarchy. Although we call this object-orientation the idea of inheritance is not applied here avoiding interdependencies between different classes. A similar hierarchy is maintained for classes describing LPs because several attributes are not specific for the link but for the LP, such as the sleep cycle.

The ULLA framework offers three main features to registered LUs:

- Queries
- Notifications
- Commands

Querying is done by using a flexible query language called ULLA Query Language (UQL), which is a subset of the well-known SQL. An example query looks like:

```
SELECT id, rxBitRate FROM ullaLink
```

As result the LU will get all registered links with their identifiers and the current receive bitrate.

Using similar queries LUs can register for notifications upon certain events:

```
SELECT id, rxBitRate FROM ullaLink WHERE (rxBitRate < 1000000 AND id = 4)
```

Using such a query for a notification would mean that ULLA core will fire the notification when the receive bitrate of link with identifier 4 falls below 1 Mbps. In addition to such event-based notifications LUs can also register periodic notifications which would be fired a configurable number of times each time after the given period elapsed. This feature is a powerful tool when certain characteristics should be monitored over time.

If attributes resemble the member variables in the object orientation commands will be the member methods allowing LUs to trigger certain actions such as scanning for available links or connecting. If an LU requests a command ULLA core will forward it towards the respective LP after successful syntax-check.

As can be seen from the query examples the Link User Interface is mostly working with classes although its scope can be narrowed using WHERE-conditions as part of the queries. In contrast, the LP interface is based on single links and attributes. ULLA core will update its values in the ULLA storage by using the LP interface and retrieve single attribute values from the LPs. As such update processes might be complicated LUs can specify validity as part of the WHERE-clause describing the requirements in terms of up-to-dateness of the requested information. Only if the values available in the ULLA storage do not fulfil these limits ULLA core will fetch attribute updates from the LPs.

In the RUNES' context, several further ULLA features are important especially for wireless sensor networks (WSNs) and resource limited devices:

In addition to Local LUs, running locally on a wireless sensor node, in the following called mote, LUs can also access ULLA remotely, thus called Remote LUs. Such Remote LUs run on gateway devices and can address single motes using the correct IpId (Link Provider identifier) as part of the WHERE-clause.

The already introduced validity is very important in case of embedded devices because the time-scale of interest is often much larger so that a bit older link layer information is still far up-to-date enough so that additional measurement tasks do not have to be performed. Careful usage of the validity-parameter thus enables resource saving applications. Another issue related to runtime efficiency is the fact that motes do not have to run a UQL-parser because a text-replacing tool can be used before compiling the initial source code. The tool searches for UQL-queries, parses those and replaces them with resource-optimized structures. This way, the application developer can still use the convenient UQL-interface but the load of parsing these queries is not put to the motes.

Existing implementations running on laptops down to tiny motes show the scalability but prove also that ULLA does not require an enormous amount of resources. Link User as well as Link Provider interface, which together form the Unified Link-Layer Interface, are both completely technology- and platform-independent. If the application developer is aiming at highly portable code he can limit himself to attributes offered by the base classes. Such applications can run over different wireless technologies and on different platforms but still do efficient cross-layering and be link-aware. Finally, flexibility is clearly enhanced compared to existing interfaces because the application developer can freely choose its own queries to gather exactly the amount of information required in each special case. Taking into account the presented extensions for embedded networks it can be concluded that ULLA is also appropriate for RUNES as it offers great deal compared to other existing interfaces but still considers resource-saving aspects important in the embedded world.

3.2 Sensor Provider Interface

The ULLA approach is not limited to link layer information. The architecture is flexible enough so that it can be used for other types of data. As an example, the GOLLUM project defined few new classes to enable LUs to access sensing information from wireless sensor network nodes.

In the RUNES project, we leverage this work and the flexibility of the ULLA framework. We extend the rudimentary work on the sensing data model to a more complete representation of sensing information. Combined with periodic or event-based notifications these extensions can easily be used to monitor the environment or implement systems warning the user when certain conditions are sensed.

In addition to the data model, which was considerably extended with new classes for sensing information, we present an architecture adaptation to extend the terminology from the link layer focus followed by the GOLLUM project also to embed sensor devices. Analogue to the Link Provider Interface we define a Sensor/Actuator Provider Interface (SAPI), which enables ULLA core to access sensing information. This section presents the details of the new SAPI. Afterwards we also describe the newly introduced classes. The major classes are in particular the platform class, the sensing class and the actuating class which are used to provide information via the Sensor/Actuator Provider Interface.

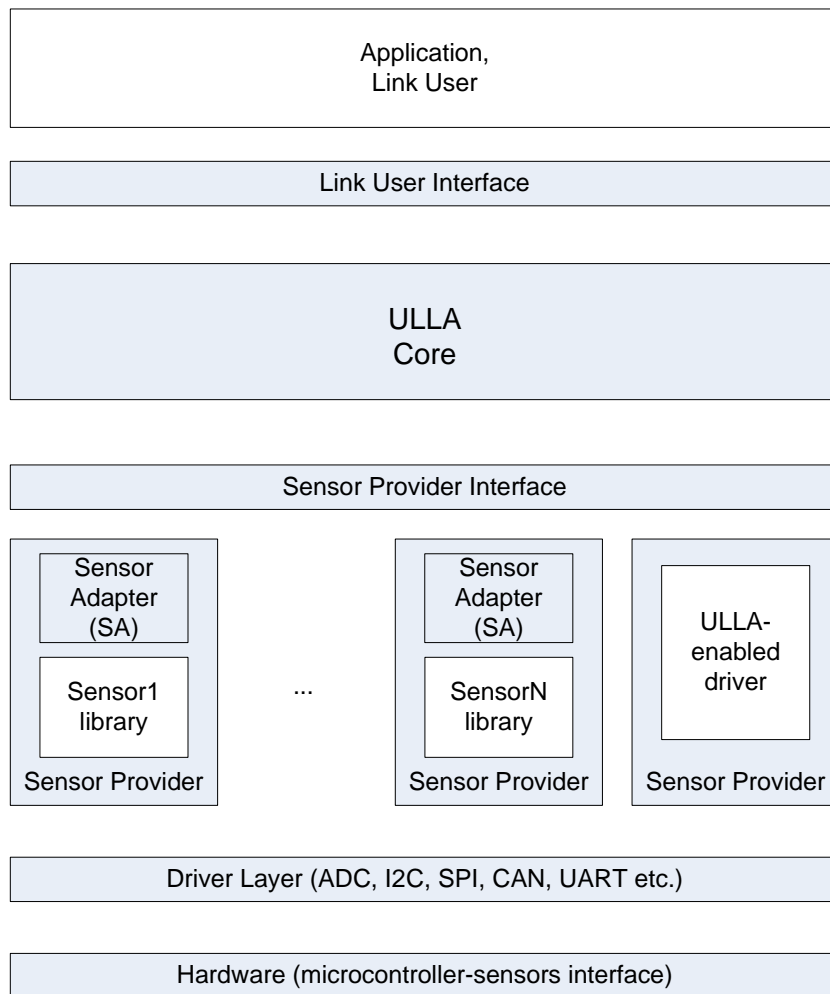


Figure 3: Extension to the ULLA architecture, Sensor Provider Interface.

The Sensor/Actuator Provider Interface developed in RUNES, depicted in Figure 3, enables LUs to access platform, sensor and actuator information in a unified way. The LU interface did not need to be changed because the introduction of new classes to the ULLA data model is enough to enable access to all information. However, instead of using the id or the lpId as primary keys for the ULLA storage database tables we introduce the **saId** (sensor actuator identifier) and the **sapId** (sensor actuator provider identifier). All features offered via the LU interface are exactly the same as defined for link information. From functionality point of view also the SAPI does not differ from the LPI, so that for simplicity reasons we did not include the interaction between modules to Figure 3.

Sensor providers (SPs) can either contain a sensor adapter (SA) and a proprietary sensor library or natively support the SPI. In the former case the SA wraps existing interfaces and implements the SAPI towards ULLA core. The SAPI does not differ from the LPI from implementation point of view because the same function calls will be used. However, in order to stress the different type of handled data we introduced the new names.

In contrast to the link layer case where link providers access the firmware, which interfaces with the radio hardware, SPs interact with the driver layer, which controls the hardware modules actually transferring and converting the sensed information. Typical examples are the analog to digital converters or the I2C bus.

The actual usage of the SAPI is transparent to LUs because they still use the well known LU interface, no matter whether they access ULLA locally or remotely. As long as they are aware of the new classes they can benefit from the SPI, using for example the following query

SELECT sapId, ram, flash FROM platform

To gather hardware information about the sensor node.

The following query

```
SELECT saId, temp_value FROM sensing WHERE temp_value < 1.5
```

could be used with an event-based notification to implement driver warning from black ice in a car. The notification feature is very powerful and can considerably lower the development complexity of monitoring applications, typical use cases of embedded networks.

The SAPI extension enables LUs to access sensing information but also information describing the sensors in a technology independent way. LUs do not have to take care of the detailed hardware interfaces as long as appropriate sensor adapters are available. Summarizing, the SPI also offers several of the ULLA benefits and enables efficient application development also in the area of embedded sensing.

There are varying nature of the functionalities associated with different types of sensors and actuators. SAPI provides all the supported functionalities by the underlying devices with a unified abstraction.

If a new sensor library is added, it registers itself to the ULLA core and then ULLA core is intelligently able to handle the directions/redirections of the requested functionalities from the application user depending upon the attributes. If a wrong or unsupported query is generated (or an unsupported functionality is requested from a sensor or actuator), ULLA core replies back with query unsupported reply. ULLA core comes to know about the supported functionalities during the sensor library registration process.

Any sensor or actuator will be connected to one (or more) of these hardware interfaces and will be communicating to the CPU (microcontroller) through these. The driver layer provides the read/write and I/O control facility to various hardware interfaces (e.g. SPI, ADC, I2C, CAN etc). A Sensor library (e.g. Sensor1 library, Sensor2 library etc.) implements the supported functions of a particular sensor or actuator. The Sensor library relies on the driver layer to communicate with the sensors. Sensor libraries provide the device specific functionalities and sensor adapter (SA) makes it look the way it is understood by the ULLA core.

In order to further elaborate our design, let us consider an example of two temperature sensors; one having an I2C based digital interface and another having an analog interface. Both the sensors provide temperature readings. SAPI provides the same functional API for getting temperature readings. ULLA core directs the corresponding functions in the two Sensor libraries for obtaining temperature readings. Further down, the two libraries use different driver layer functionalities for communicating with two different types of temperature sensors; ADC and I2C. For instance if the precision level of the ADC based temperature sensor cannot be varied, the Query processing unit in the ULLA core will notify the application user that this functionality is not supported.

In order to follow this flexible architecture, we propose to use three base classes; namely the platform class, the sensor class and the actuator class. The three classes also possess an inherent logical and natural grouping.

3.2.1 Platform interface

The platform class contains all the OS dependent functionalities. In the following tables, we list the common attributes and commands associated with this class. Following the design principle of ULLA, the attributes and commands listed here are extensible.

Attribute	Rationale
Vendor	Describes the vendor of the platform
ID	Identification of the platform
Version	Version number of the running OS
AvailableRAM	Bytes of available RAM resource
AvailableProgMem	The remaining program memory.
AvailableExternalMem	The number of bytes of the external memory resource which is left to utilize.
TotalRAM	The total physical size of the RAM
TotalProgMem	The total physical size of the program memory
TotalExternalMem	The total size of the external physical memory attached to a device
PowerMode	

Command	Purpose
WriteExternalMem[n]	This command is used to write n-bytes to the external memory attached to a device at the specified address.
ReadExternalMem	This command is used to read n-bytes from the external memory from the specified address.
setPowerMode	

3.2.2 Sensor interface

All the sensors have a common feature of providing sensory data. The resolution, type and units of the sensor readings may be different from sensor to sensor. In the following, we list some of the attributes and commands associated with the sensor class. Each specific sensor class containing sensor library functionalities will be derived from this class.

Attribute	Rationale
Vendor	Describes the vendor/manufacturer of the sensor
ID	Sensor ID is used to identify a particular sensor
Version	This describes the version number of the sensor
ValueType	The type of the value returned by the sensor (bool, unit8,int8, float, double etc)
ValueSize	The number of size of the data returned by the sensor
Units	This attribute describes the units of the sensory data
Resolution	This attribute defines the resolution of the data
SamplingRate	SamplingRate describes the rate at which the sensor needs to provide the measurements.
State (On/Off)	This attribute describes whether a sensor is switched on or off.

Command	Purpose
Turn On/Off	This command is used to switch on a particular sensor
Reset	This command is used to reset a particular sensor
Calibrate	Some sensors support the feature of software controlled calibration or the sensor library supports a calibration feature, which will be exercised by the

	Calibrate command
SetSamplingRate	This command is used to set the SamplingRate at which the sensor needs to grab the data
GetData	This command is used to get sensory readings from a particular sensor.

3.2.3 Actuator interface

There are varying nature of controlling needs and this leads to a wide range of actuating systems with different needs and capabilities. In order to support a unified abstraction as viewed in SAPI, we suggest to have an actuator base class. Each of the actuator specific class will be derived from this base class. Some of the systems just support sensing capabilities and some have the controlling features and some systems consist of both, sensors and actuators. Keeping the actuator class decoupled from sensing class makes the system design simpler and more flexible.

In the following, we list the attributes and commands associated with the actuator class in the SAPI framework.

Attribute	Rationale
Vendor	Describes the vendor/manufacturer of the actuator
ID	Sensor ID is used to identify a particular actuator
Version	This describes the version number of the actuator

Command	Purpose
Start	To start an action
Stop	To stop an action

3.2.3.1 Continuous

There may be actuators that are continuous in nature. For instance, an actuator may be to inject fuel in the combustion chamber of an engine. This is a continuous process and for this purpose the maximum, minimum and instantaneous value attributes are associated.

Attribute	Rationale
MaxValue	The maximum value attribute associated with a certain actuator.
MinValue	The minimum value attribute associated with a certain actuator.
Value	The instantaneous value attribute associated with a certain actuator.
Unit	The units of expressing the value

3.2.3.2 Boolean

There are actuators that act in a binary fashion. For instance, an automatic door will either be open or closed, the CD drive is ejected or inserted. For these types of actuators, a binary attribute is associated.

Attribute	Rationale
State (Value)	On/Off

A. Appendix: Example of specific Virtual OS definition (connectBlue Sensor Routing node, ARM7, FreeRTOS)

```

#ifndef _OS_H_
#define _OS_H_

/*=====
 * INCLUDES
 *=====*/
#include "FreeRTOS.h"
#include <stdlib.h>
#include "types.h"
#include "semphr.h"
#include "queue.h"
#include "task.h"

/*=====
 * DEFINES
 *=====*/
#define OS_MAX_NBR_PROCS      10
#define OS_MAX_NBR_TIMERS    15
#define OS_MIN_STACK_SIZE    128

/* Processor stack sizes. The ABT stack is used when a [pd]abt occurs,
 * IRQ stack by interrupts handlers and the SVC stack during the initial
 * startup before the FreeRTOS scheduler is started.
 *
 * All sizes are in words!
 */
#define OS_UND_STACK_SIZE    1
#define OS_ABT_STACK_SIZE    32
#define OS_FIQ_STACK_SIZE    1
#define OS_IRQ_STACK_SIZE    64 /* @@cb PH: Was 256 before adjustment */
#define OS_SVC_STACK_SIZE    128 /* @@cb PH: Was 256 before adjustment */
#define OS_SYS_STACK_SIZE    1

#define os_ASSERT(c)          if (!(c)) { OS_ERRORHANDLER(__FILE__, __LINE__); }
#define OS_PROCESS_FUNCTION(func) void func(void)
#define OS_CREATE_PROCESS(func, stack, prio) \
    os_create_process(func, stack, prio, #func)

#define LED1_ON                GPIO0_IOSET = (1 << 27)
#define LED1_OFF               GPIO0_IOCLR = (1 << 27)
#define LED2_ON                GPIO0_IOSET = (1 << 4)
#define LED2_OFF               GPIO0_IOCLR = (1 << 4)
#define LED3_ON                GPIO0_IOSET = (1 << 21)
#define LED3_OFF               GPIO0_IOCLR = (1 << 21)

#ifndef NDEBUG
#define OS_ERRORHANDLER(a, b)  os_error(a, b);
#else
#define OS_ERRORHANDLER(a, ...) LED3_ON
#endif
#endif

```

FP6 IP "RUNES" – D3.5 Interface Description

```

#define HTONL(x) ( (UINT32)((x & 0x000000FF) << 24) | \
                  (UINT32)((x & 0x0000FF00) << 8) | \
                  (UINT32)((x & 0x00FF0000) >> 8) | \
                  (UINT32)((x & 0xFF000000) >> 24) )

#define HTONS(x) ( (UINT16)((x & 0x00FF) << 8) | \
                  (UINT16)((x & 0xFF00) >> 8) )

#define BYTES2IP(a, b, c, d) HTONL( ( (UINT32)((a & 0xFF) << 24) | \
                                     (UINT32)((b & 0xFF) << 16) | \
                                     (UINT32)((c & 0xFF) << 8) | \
                                     (UINT32)(d & 0xFF) ) )

/*=====
 * TYPES
 *=====*/
typedef void* os_Semaphore;
typedef xQueueHandle os_Queue;

/*=====
 * FUNCTIONS
 *=====*/
/*-----
 * Init the OS
 *-----*/
void os_init(void);

/*-----
 * Start the scheduler. This function ideally never returns.
 *-----*/
void os_start(void);

/*-----
 * Allocate args bytes of memory from the heap. Returns pointer to the
 * allocated memory.
 *-----*/
#define OS_malloc(args)    pvPortMalloc(args)
void* os_malloc_real(UINT32 bytes);
// #define os_malloc(args)  os_malloc_real(args)

/*-----
 * Free previously allocated memory pointed to by args
 *-----*/
#define OS_free(args)      vPortFree(args)
void os_free_real(void* ptr);
// #define os_free(args)   os_free_real(args)

/*-----
 * Set a timer to expire in ms milliseconds. When timer expires, the message
 * with the data pointed to by msg is sent to the process pid. If pid == NULL
 * msg is sent to the process that called OS_timer_set. Returns
 * an ID of the timer created.
 *-----*/
INT32 os_timer_set(UINT32 pid, INT32 ms, void* msg);

```

FP6 IP "RUNES" – D3.5 Interface Description

```
/*-----  
 * Same as OS_timer_set() but creates a periodic timer.  
 *-----*/  
INT32 os_timer_set_periodic(UINT32 pid, INT32 ms, void* msg);  
  
/*-----  
 * Cancels a previously created timer with the ID id.  
 *-----*/  
void os_timer_cancel(UINT32 id);  
  
/*-----  
 * Puts the current process in sleep for ms milliseconds.  
 *-----*/  
void os_sleep(INT32 ms);  
  
/*-----  
 * Send a message to the process with ID proc_id.  
 *-----*/  
void os_msg_send(INT32 proc_id, void* msg);  
  
/*-----  
 * Send a message to the process with ID proc_id, USE ONLY FROM ISR  
 *-----*/  
INT32 os_msg_send_from_isr(INT32 proc_id, void* msg, INT32 prev_woken);  
  
/*-----  
 * Receives a message from the current process' message queue. Blocks if  
 * queue is empty.  
 *-----*/  
void* os_msg_receive(void);  
  
/*-----  
 * Returns the process ID of the currently running process.  
 *-----*/  
UINT32 os_get_current_pid(void);  
  
/*-----  
 * Returns the current clock tick count in milliseconds.  
 *-----*/  
UINT32 os_get_clock_time(void);  
  
/*-----  
 * Initializes the semaphore s of the type OS_Semaphore  
 *-----*/  
#define os_semaphore_init(s) vSemaphoreCreateBinary(s)  
  
/*-----  
 * "Give" semaphore s  
 *-----*/  
void os_semaphore_post(os_Semaphore s);  
  
/*-----  
 * "Takes" semaphoer s. Blocking  
 *-----*/
```

FP6 IP "RUNES" – D3.5 Interface Description

```
void os_semaphore_try(os_Semaphore s);

/*-----
 * Polls and if available "takes" semaphore s. Non-blocking. Returns 1 if
 * successful, 0 if not.
 *-----*/
INT32 os_semaphore_try_nb(os_Semaphore s);

/*-----
 * Returns a queue of the length len and the type OS_Queue
 *-----*/
#define os_queue_init(len) xQueueCreate((portBASE_TYPE) len, sizeof(void*))

/*-----
 * Sends the message item to the queue q
 *-----*/
void os_queue_send(os_Queue q, void* item);

/*-----
 * Returns a message from the queue q. Blocks until a message is available
 *-----*/
void* os_queue_receive(os_Queue q);
UINT32 os_queue_nbr_msgs(os_Queue q);

/*-----
 * Creates a new process with the "main" method function, the stack size
 * stack_size and the priority priority
 *-----*/
void os_create_process(void *function, UINT32 stack_size, UINT32 priority,
                      UINT8* name);

/*-----
 * Destroys a process with the process id pid. All messages in the process
 * message queue are deleted.
 *-----*/
void os_destroy_process(UINT32 pid);

/*-----
 * Macro to disable preemptive context switches
 *-----*/
#define os_enter_critical() taskENTER_CRITICAL()

/*-----
 * Macro to re-enable preemptive context switches
 *-----*/
#define os_exit_critical() taskEXIT_CRITICAL()

/*-----
 * Macro to disable all interrupts
 *-----*/
#define os_disable_irqs() taskDISABLE_INTERRUPTS()

/*-----
 * Macro to re-enable all interrupts
 *-----*/
```

FP6 IP "RUNES" – D3.5 Interface Description

```
#define os_enable_irqs()  taskENABLE_INTERRUPTS()

void os_bluetooth_reset(UINT32 action);
void os_avr_reset(UINT32 action);

#ifdef NDEBUG
void os_error(void* file, INT32 line);
#endif

#endif // _OS_H_
```


References

[GD24] GOLLUM deliverable D2.4, “Final Architecture and API”, available from <http://www.ist-gollum.org> [Cited on 13th September 2006], August 2006.

[GD34] GOLLUM deliverable D3.4, “API Guidebook”, available from <http://www.ist-gollum.org> [Cited on 13th September 2006], August 2006.

[RUNES-D1.4.2] F. Oldewurtel (Editor), “RUNES Deliverable 1.4.2: Joint Design, Architecture and Protocols Handbook for WPs”, October 2006.

[RUNES-D1.4.3] F. Oldewurtel (Editor), “RUNES Deliverable 1.4.3: Updated specific architecture and component definition”, February 2007.