owa5X Family
OWA5x Software Programming Guide

owasys

Advanced Wireless Devices

Edition: 2025

Document number: BOK-000 5006

Table of Contents

1 INTRODUCTION

This manual explains on one side the basic procedures to login into the system and information on the FW images used by default, and on the other side the purpose of each API and how to start using the available functions in the library of APIs to develop the customer application.

For detailed information about each function see the API documentation. For information about the hardware controlled by these functions, see the Integrators Manual.

2 Connecting owa5X to a PC

2.1 Serial connection

Either Linux OS or Windows OS can be used to connect to the device from a Personal Computer (PC) through serial port. The required configuration parameters are the following:

- Bit Rate: 115200 bps
- Data Bits: 8
- Parity: none
- Bit Stop: 1
- Flow Control: None

It is important to keep a common ground to have a successful communication, apart from the Tx and Rx signals of the debug serial port.

Serial program

Under Windows OS use Windows HyperTerminal, Putty or Teraterm to connect the owa5X to the PC configuring the serial port parameters to the values indicated in previous section.

Under Linux/GNU OS Minicom is the recommended utility to connect the owa5X.

Switch on the owa5X. Once the Kernel is loaded in RAM memory and the system is up, the device waits for the user to enter a valid user name to log in. These are the default credentials:

```
root / root
owasys / temppwd
```

Please consider removing the owasys user or changing its password, and changing the root user password for FS images that are to be used in production as a basic security step.

Once logged in, the user is in the owa5X file system which has the directory structure of a usual Linux distro, on this case Debian.

To transfer a file from the PC to the owa5X, change to /home directory or to the directory where the file is to be stored (cd /home or cd /directory_name), type rz command and choose Transfer-→ Send File... option of the HyperTerminal for example, or Ctrl+A+S → zmodem with Minicom.

To transfer a file from the device to the PC, change to the directory where the file is, then type sz command indicating the name of the file (sz file_name) and choose the Transfer -→ Receive File... option of the HyperTerminal or Ctrl+A+R with Minicom. File transfer protocol is zmodem in both cases, to send and recevie files.

The serial connection is the preferred one for debugging as it will give the option to get into the u-boot prompt, from which it is possible to carry on several processes like for example check the existeing partitions and flash them with FW images.

2.2 Ethernet connections

To communicate with the owa5X a SSH connection can be stablished too using the Ethernet interface, if the unit features this option. The system get the SSH daemon up by default, and its configuration is:

```
IP: 192.168.10.1
Port: 22 (default port of SSH)
```

In order to use this connection, connect to this IP using these credentials:

- user: owasys
- password: temppwd

SSH access is restricted for regular users, edit the file /etc/ssh/sshd_config to change this.

Authentication is only allowed using SSH keys without a passphrase by default.

If you need to use a password for authentication, you must explicitly specify it using the following SSH option:

```
ssh -o PreferredAuthentications=password owasys@192.168.10.1
```

3 Firmware Specifications

3.1 General Description of Firmware

In order to manage the platform resources, a complete library of APIs, for GNSS management, internet connection, management of the interfaces, GSM/GPRS functions and other services is made available. Thus, the developer does not need to consider about low level hardware drivers and protocols, and can focus on the application by means of user-friendly APIs.

3.2 u-boot

The u-boot is the booting system that can be accessed to check and work with different partitions and change the booting sequence.

The Linux is started without waiting any time as by default the variable bootdelay is set to "-2", but in the manufacturing stage this variable is set to 0 so that the user can enter into the u-boot prompt pressing the space bar when powering on the unit.

If the default environment variables are set the bootdelay will be set to "-2" and the access to the u-boot prompt will no longer be possible until this is again changed from the running FS with this command:

fw_setenv bootdelay 0

The u-boot can be customized with the necessary u-boot environment variables, which can help the user for example to recover from unexpected crashes of the FS that prevents it to run**.**

3.2.1 System boot protection

The u-boot includes by default several env variables that makes the system to boot into the recovery FS in case there are too many boots without clearing the counters.

These are the variables in use:

BOOT_MAIN_LEFT=3

BOOT_RECOVERY_LEFT=3

BOOT_ORDER=RECOVERY MAIN

FS_FAILURE_THRESHOLD=3

POWERCUT_FAILURE_THRESHOLD=0x11

POWERCUT_LEFT=0x11

handle_rootfs_redundancy: This variable has the logic to boot into the main or the recovery depending on the LEFT variables that are decremented in every boot. These LEFT variables are reset when the system runs by the rauc service.

3.3 Linux Kernel

This is a Linux standard kernel, version 6.6.25. As this is a standard kernel, PC developed applications are easily made compatible with owa5X platform. Also this kernel can be updated to follow standard kernel revisions.

3.4 File System

The file system is based in a Debian 12 Bookworm distribution.

3.5 Starting to develop the customer application

Once the cross compiler is installed, the user can use this compiler in the desired development environment, for example, VSCode.

To use the available APIs library functions, add the includes to owa5X header files in the application header files and compile the application using the provided cross compiler.

owa5X files to be included in customer header files are:

```
<owa5X/RTUControlDefs.h>
<owa5X/IOs_ModuleDefs.h>
<owa5X/GSM_ModuleDefs.h>
<owa5X/INET_ModuleDefs.h>
<owa5X/GPS2_ModuleDefs.h>
<owa5X/owcomdefs.h>
<owa5X/owerrors.h>
```

All API module libraries have been written in C Language and gcc 9.2 cross-compiler tool, under LINUX Operating system. It is recommended, to avoid problems from the development environment, to use the same language, compiler, and environment when possible.

Usually, a Client application is comprised of a main executable program, and a set of libraries. This set of libraries can be of two types: system libraries, both static and dynamically loaded, and owa5X API module libraries, always used in a dynamic way.

Once an application is defined, the user must select which libraries will be used and get pointers to the needed functions, using them as C language normal functions. For a better use of system resources, it is recommended to unload all functions of a library, as well as the library itself, when it is known that it will not be used any more in the program's scope. Also, it is not necessary to get all functions pointers in a library, if some of them will not be used in the program: it is a waste of memory, and system resources which can be useful for other tasks, and will improve the system behaviour in general.

Another alternative is to use the .h header of the library to compile the program against the library, using the -l option at compilation time.

3.5.1 Available APIs

In this section there is an overview of the available APIs and an explanation of their functionality. For further information about the APIs and their functions, it is recommended to read through Programming Guide and API.

3.5.1.1 API for owa5X Control (RTU)

This API provides functions to enter the low power modes, RTC and standard accelerometer.

3.5.1.2 API for GSM/GPRS

The communication with the GSM module is done with the GSM and GPRS APIs. Sending and receiving SMSs and making dial calls are some of the functions provided.

3.5.1.3 API for GNSS

Getting the time and positioning info are some developed functions included in this API.

3.5.1.4 API for IOs

The functions of this API facilitate the management of owa5X I/Os, UARTs and audio signals.

3.5.1.5 CAN Bus FMS API

There is available a FMS library, that can run in the owa5X under a license. This library provides the most important data in a FMS bus available through a couple of simple functions. This library must be first enabled to use it, so if desired please request this to your distributor.

3.5.1.6 Software Application Notes

In the developers zone some application notes show the usage of the different libraries of Owasys:

- owasys_an3. Use of IO.
- owasys_an5. Use of GNSS.
- owasys_an8. Use of RTU.
- owasys_an24. Use of GSM.

These application notes are also convenient to report a problem with any feature while integrating the user programs. If there is any function that fails in some way, it is best to reproduce it with an application note in order to report it to Owasys (customer_support@owasys.com).

4 Updating owa5X FW

There are two approaches to update the FW of the owa5X:

- Patch. Patch with a delta of changes since previous version. This approach is the more convenient for updates OTA, as the amount of data to transfer will be less.

- Complete image. An image with some or all FW parts: u-boot, kernel and FS. This approach is meant for updates of units located on site, where an operator have physical access to it.

The NAND and the MMC from FW 2.0.0 are divided in these partitions:

NAND 1 GB	Name	Size	Mountpoint
mtd0	u-boot	8 Mb	
mtd1	u-boot-bckp	8 Mb	
mtd2	u-boot-env	1 Mb	
mtd3	u-boot-env.backup	1 Mb	
mtd4	device	16 Mb	/device
mtd5	device.backup	16 Mb	
mtd6	file-systemA	487 Mb	/
mtd7	file-systemB	487 Mb	/

4.1 Flash of complete images

The flashing of the complete images will be using uuu tool, using a USB type A to USB type A cable to connect the owa5X to the PC.

First of all download to the PC the image, flash.bin, uuu script and uuu binary:

- flash.bin

- owa5X_V←version→.img

- owa5X_nand_boot_emmc.uuu

- uuu

Press the space key when powering up the unit to enter the u-boot prompt and execute this command to enter into uuu mode and start listening in the USB interface:

```
u-boot>> run enterflashmode
```

Once the owa5X is in uuu mode run this command in the PC:

```
PC$ sudo ./uuu -v owa5X_nand_boot_emmc.uuu
```

The owa5X_nand_boot_emmc.uuu is a script that is executed at the u-boot level in the owa5X, and it can be edited to flash the desired images.

The default uuu script keeps a copy of hwfile variable to keep the data of hw.json in /device/etc/default/ folder, take care of keeping this in your custom uuu scripts in order to keep this, otherwise it will be lost when upgrading the FW.

4.2 OTA updates

The OwaDM can be used to do OTA updates of either full or delta updates of the FW and the user SW in the owa5X.

To include the OwaDM agent in the system the app-builder can be used, and also to generate the image to be flashed in the unit. With this tool it is also possible to generate the customized FS with the services and files included by the developers.

5 Compiling SW for owa5X

Normally the approach to compile applications for embedded devices has been to use a cross compiler to generate the arm binary in a PC. However and due to the power and resources of the owa5X system it is possible to make direct compilations in the owa5X itself, which provides these advantages:

- Ease of compilation of large libraries with a high amount of dependencies.

- Direct debugging without the need of simulating

5.1 Compiler installation in the owa5X

In the unit install the compiler with these commands from the Debian repositories:

```
apt update
apt upgrade -y
apt install build-essential autoconf automake libtool libdbus-1-dev libev-dev libsystemd-dev git -y
```

If debugging will be used during development using gdb, this can also be installed from the repositories:

```
apt install gdb -y
```

Using an IDE with SSH capability it is possible to debug from the IDE itself in the PC.

5.2 Cross compilation

In the case of needing to cross compile because there is not an accessible owa5X unit at the moment, a cross compiler provided by Owasys can also be used for this purpose. Follow these steps to have it up and running in your PC:

1. Download the crosscompiler from the developers zone

2. Install it, for example the one based in CC12.2 can be installed with this command:

```
./fsl-imx-xwayland-glibc-x86_64-imx-image-multimedia-armv8a-imx8mp-lpddr4-evk-toolchain-6.1-mickledore.sh -d /opt/crosstool/owa5x/CC12.2
```

3. Each time that a program must be compiled, change the environment variables accordingly to compile using this cross compiler. To simplify this setup a link can be created like this:

```
ln -sf /opt/crosstool/owa5x/CC12.2/environment-setup-armv8a-poky-linux
/opt/crosstool/setup-owa5x-12.2_env
```

Then to set up the env variables before building the program use this command:

```
source /opt/crosstool/setup-owa5x-12.2_env
```

6 Owasys systemd services

6.1 system_maintenance

This service runs at boot time and exits after execution.

The user can make use of this service by loading an executable script to "/home/custom_actions.sh". This script can execute the desired commands, for example:

- Reset the bootcount variable if in use.

- Execute an OTA update procedure.

6.2 pmsrv.service

This service must run at all times in the system, as this will be the service communicating with the auxiliary processor in charge of the LPM, RTC, Accelerometer and IO.

This service has a configuration file that can be used to tweak some features of the system:

```
/var/lib/owasys/pmsrv/pmsrv.json
{
    "name": "pmsrv",
    "reset_lines": 20,
    "set_hostname": 1,
    "set_default": 1,
    "set_time": 0,
    "secure_time": 1,
    "temperature": {
        "temp_enable": true,
        "temp_halt": 85,
        "temp_wake": 75,
        "off_timeout": 15
     }
}
```

- reset_lines:

  The number of lines that can be written to the file /var/lib/owasys/pmsrv/pmsrv.info, where it is logged the times when the pmsrv service has been restarted.

- set_hostname:

  (0) Do not set the hostname, the user can set it using the file /etc/hostname.

  (1) Set the hostname from the serial number.

- set_default:

  (0) Set the state for all IO as they were before when booting the system.

  (1) Set the default state for all IO when booting the system.

- set_time:

  (0) Do not set the system time from the one in the RTC HW clock when the system boots up. The script /usr/bin/system_maintenance.sh executed by the

service system_maintenace also sets the system time from the RTC, so comment the "hwclktosys" command from it in order to deactivate completely this functionality.

(1) Set the system time from the one in the RTC HW clock when the system boots up.

- secure_time:

  [1] The pmsrv only restores the RTC clock if the date of the system is older than the one in the RTC

  [0] The system time and date are set from the RTC without any checking.

- temp_enable:
  [true] Board temperature is monitored every minute, logged to syslog and sent to dbus (dbus-monitor –system "member=temperature")
  [false] temperature is not monitored
- temp_halt:
  60..95. If temperature is detected over this value three times in a row, halt signal is sent via dbus and device will go to OFF mode.
- temp_wake:
  55..90. Once the temperature is below this value the system will wake up with wakeup reason "WKUP_TEMPERATURE"
- off_timeout:
  10..600. This is the time in seconds between halt signal and remove of CPU power, in order to give enough time to user services and programs to shut down gracefully.

7 OWASYS API

7.1 How to access owa5X functionality

Depending on the functionality to be accessed the following include files and libraries should be used.

	Include file	Load file	Type
RTU	<owasys/RTUControlDefs.h>	/lib/libRTU_Module.so	Dynamic library
IO	<owasys/IOs_ModuleDefs.h>	/lib/libIOs_Module.so	Dynamic library
GSM	<owasys/GSM_ModuleDefs.h>	/lib/libGSM_Module.so	Dynamic library
GPS	<owasys/GPS2_ModuleDefs.h>	/lib/libGPS2_Module.so	Dynamic library
iNET	<owasys/INET_ModuleDefs.h>	/lib/libINET_Module.so	Dynamic library

7.1.1 Accessing dynamic libraries

There are two ways to access the Owasys libraries from the user program code: using dlopen() and dlsym() or linking against them at compilation time using the -l option.

7.1.1.1 dlopen() and dlsym()

In order to access the owa5X API functionality from the dynamic libraries several steps must be carried out.

First, application should get a handler to the library; for that purpose the system call dlopen() is used. In the example below a handler to the gsm library is got:

```
void* wLibHandle = NULL;

wLibHandle = dlopen("/lib/libGSM_Module.so", RTLD_LAZY);

if (!wLibHandle) {
   printf("No shared library found");
}
```

Once the application has got the handler to the library, it needs to get a reference to the needed functions. For this dlsym() system call is used.

A pointer to a function must be declared as below, for each of the functions that application needs.

```
   int (*FncDIGIO_Set_LED_SW0)( unsigned char)
```

Then application calls **dlsym(),** passing as parameters the handle to the library and the name of the desired function (as it appears in [1]) and casts the result to the function pointer.

```
FncDIGIO_Set_LED_SW0=(int(*)(unsigned char))dlsym(wLibHandle,
“DIGIO_Set_LED_SW0”);
if ( dlerror() != NULL) {
   printf(“No DIGIO_Set_LED_SW0 found..\n”);
}
```

Finally the function is called as is shown below.

```
int  ReturnCode = NO_ERROR;

if ( (ReturnCode = ( *FncDIGIO_Set_LED_SW0)( 1)) != NO_ERROR ) {
   printf(“Error %d in DIGIO_Set_LED_SW0()\n”, ReturnCode);
}
```

All functions return NO_ERROR in case of success; otherwise they return an error code according to the subset of errors defined in the API. By handling these errors the application will improve its tolerance to any fault that might occur.

In case that the application has finished accessing a dynamic library, it might free system resources by removing it from memory. For that purpose system **dlclose()** function is used.

```
 if( (dlclose( wLibHandle) ) != 0) {
    printf( "UnloadExternalLibrary() error\n");
    exit(1);
 }
 printf( "UnloadExternalLibrary() ok\n");
```

7.1.1.2 direct linking

As all the functions are declared in the header files include in the compiler, and the dynamic .so libraries are also included with the owasys patch in the cross compiler, the owasys libraries can be linked at compilation time using the -l option.

```
owa5X$ gcc -Wall -mthumb -mthumb-interwork -D_REENTRANT -oowasys_an3
./*.cpp -ldl -lpthread -lGSM_Module
```

After including the header file of the library, in the example the GSM library, the function can be called directly without having to use dlopen() or dlsym().

8 owa5X RTUControl API

8.1 Introduction

LibRTUControl library provides four main functions available for user applications:

- A system Select control: based in the *Linux Select* daemon, offers a clear and easy way of pooling both synchronous and asynchronous ports to be read. This is needed by other system modules to work (GSM, GPS)
- Functions for handling Power Modes.
- RTC time handling.
- A set of timers available for all kind of uses.
- Accelerometer related functions.
- Internal analog inputs (temperature, Vin, Vbat)

To see the whole set of functions from *RTUControl* library, see the API.

8.2 Starting and finalizing RTUControl module

LibRTUControl library is needed internally by GSM and GPS system modules. For that reason, before using any functionality from those modules, the RTU module must be initialized and started.

Before calling to the RTU initialization and start functions, it is important to wait until all system services are running, pmsrv.service, and this can be checked with the lock file owaapi.lck under /var/lock/ directory. Once this lock file is removed from the filesystem the RTU functions can be called and follow the normal program flow.

```
...

// wait some seconds until owaapi.lck is removed from the file system
for (i=0;i<15;i++)
{
   if (ret = stat("/var/lock/owaapi.lck", &buf) == -1)
   {
      i = 15;
   }
   sleep (1);
}

if (ret == 0)
{
   printf( "Problem with the system initialization\n");
   return 1;
}

if( ( ReturnCode = RTUControl_Initialize(NULL) != NO_ERROR)
{
   printf( "Error %d in RTUControl_Initialize()\n", ReturnCode);
   return 1;
}

if( ( ReturnCode = RTUControl_Start()) != NO_ERROR)
{
   printf( "Error %d in RTUControl_Start()\n", ReturnCode);
   RTUControl_Finalize();
   return 1;
}
```

Another method to wait for this service to run, is to include the dependency in the user application systemd configuration file. Include the option "After" and "ExecStartPre" with a 1 second sleep to wait until the pmsrv.service stabilizes.

/etc/systemd/system/user_application.service

```
[Unit]
After=pmsrv.service
…

[Service]
ExecStartPre=/bin/sleep 1
…
```

8.3 RTC time

The owa5X unit is provided with a **Real Time Clock** (RTC) with calendar options and it is fully programmable using Owasys API functions, as it is not connected to the main processor other methods are not possible, it is not accessible via ioctl functions nor Linux commands.

This clock has a dedicated backup battery, and so keeps the time when the unit is not powered. See Integrators Manual for duration of RTC backup.

Apart from this RTC, the owa5X can also retrieve the current **UTC time from the GPS module** through its API, or synchronize it with a NTP server once the system has connectivity.

Once the time is known, there are four functions inside the RTUControl API for managing the Linux kernel time:

```
int GetSystemTime( TSYSTEM_TIME *wSystemTime);
int SetSystemTime( TSYSTEM_TIME wSystemTime);
```

Using these functions, and a TSYSTEM_TIME type structure, the customer application can get or set the system time at any moment.

The owa5X unit has an internal hardware RTC. The hardware RTC will update the system time when the unit reboots or when it comes from any of the power save modes. The SetSystemTime function does not update the RTC, it only updates the system time. It means that if the application sets the system time and reboots or goes successfully to one of the power save modes the system time will be updated with the time and date in the RTC when the unit wakes up.
To deactivate this feature see Owasys systemd services

To manage the RTC time, which works as a master of the system time, there are two functions:

```
int RTUGetHWTime( THW_TIME_DATE *CurrentTime);
int RTUSetHWTime( THW_TIME_DATE CurrentTime);
```

The system and HW time can be interchanged from the command line simply by calling them. There are two command embedded in the FS, one to set the HW time based in the system time, and one to set the system time based in the HW clock.:

- sysclktohw. Sets the HW time based on the system time, which is the the one got with command "date"
- hwclktosys. Sets the system time based on the HW clock. This is done automatically every time the system boots up.

8.4 *Sleep and usecsleep*

LibRTUControl library provides a function named *usecsleep(int seconds, int useconds)* that user can use instead of Linux sleep, usleep and nanosleep functions.

In the owa5X platform the user can also use any of the Linux sleep calls like sleep(), usleep() or nanosleep().

8.5 *Power Management*

LibRTUControl library provides the functionality required for setting the device into two different power modes: Standby Mode and Stop Mode.

In Stop and Standby modes the signals that can wake up the device are these:

- MOVEMENT: wakes up if the movement sensor detects any movement.
- DIN0..8: wakes up if the external DIN0..8 signal changes.

NOTE: DIN[9-11] can not be used to wake up the system.

- RTC: wakes up if RTC clock reaches a determined time.
- CONSOLE: wakes up if a character is received at RS232 ttyO4 interface in pin RX0.
- PWRFAIL: wakes up if the unit detects that there is no external power or that it is recovered, the change on the level will wake up the unit. Set in the mask RTU_ONLY_POWERUP so that the unit only wakes up with a raising signal in Vin power input.
- GSM: wakes up if there is any event data (RING, SMS or COVERAGE events for example) received from the GSM module.
- CAN: wakes up if the unit detect traffic in any of the CAN interfaces connected to a CAN bus. Any traffic in the bus will wake up the unit, not a specific message can be configured for it.

  To wake up with CAN signal is not needed to have the interface up.

  In the owa5X the controller can also be off, it is not needed to have it on in order to wake up from CAN.
- BLE: wakes up if the unit detects a BLE device in the nearby that has been previously whitelisted.
- Ethernet: wakes up if there is traffic in the Ethernet port.

8.5.1 Standby Mode

This mode allows the user to wake up using any of the events that are able to wake up the device: Moving, Power Fail, Console, GSM, CAN, RTC, DIN0-8, BLE, Ethernet.

This mode is the fastest power save mode for waking up, but the consumption is higher. When the unit returns from the Standby Mode the program execution continues. All the memory and program values are preserved.

The function provided by this library for switching to this state is,

```
int RTUEnterStandby ( unsigned long wMainWakeup, unsigned long
wExpWakeup);
```

The parameter required by this function is a 16 bits mask, wMainWakeup, with the bits of the corresponding events that will wake up the unit set. The second parameter is the mask for an optional expansion board, leave it to 0.

The mask of each event is defined in the owa5X/RTUControlDefs.h include file,

```
   #define RTU_WKUP_MOVING     (1 << 0)
   #define RTU_WKUP_PWRFAIL    (1 << 1)
   #define RTU_WKUP_CONSOLE    (1 << 2)
   #define RTU_WKUP_GSM        (1 << 3)
   #define RTU_WKUP_CAN1RD     (1 << 4)
   #define RTU_WKUP_RTC        (1 << 6)
   #define RTU_WKUP_DIN0       (1 << 7)
   #define RTU_WKUP_DIN1       (1 << 8)
   #define RTU_WKUP_DIN2       (1 << 9)
   #define RTU_WKUP_DIN3       (1 << 10)
   #define RTU_WKUP_DIN4       (1 << 11)
   #define RTU_WKUP_DIN5       (1 << 12)
   #define RTU_WKUP_DIN6       (1 << 13)
   #define RTU_WKUP_DIN7       (1 << 14)
   #define RTU_WKUP_DIN8       (1 << 15)
   #define RTU_REMOVE_VOUT     (1 << 19)
   #define RTU_ONLY_POWERUP    (1 << 20)
   #define RTU_WKUP_BLE        (1 << 21)
   #define RTU_WKUP_ETHERNET   (1 << 22)

     #define  WKUP_ALL             (RTU_WKUP_MOVING  |  RTU_WKUP_PWRFAIL  |
RTU_WKUP_CONSOLE | RTU_WKUP_GSM | RTU_WKUP_CAN1RD\
                                       |  RTU_WKUP_RTC  |  RTU_WKUP_DIN0  |
RTU_WKUP_DIN1\
                                      |  RTU_WKUP_DIN2  |  RTU_WKUP_DIN3  |
RTU_WKUP_DIN4 | RTU_WKUP_DIN5 | RTU_WKUP_DIN6\
                                      |  RTU_WKUP_DIN7  |  RTU_WKUP_DIN8  |
RTU_WKUP_ETHERNET)
```

There are two definitions that do not correspond to wake up signals but they can be used to modify the behavior of this mode:

RTU_REMOVE_VOUT: If this is set in the mask then Vout will be switched off when entering the Low Power Mode.

RTU_ONLY_POWERUP: This can only be set when the signal RTU_WKUP_PWRFAIL is set, and on this case the system will only wake up with raising events of the Vin signal that powers the unit, and the falling events will not wake up the unit.

A small example of how to switch to this mode after being initialized and started is shown below, in this case the unit goes to sleep with MOVING and GSM,

```
            if(RTUEnterStandby(RTU_WKUP_MOVING | RTU_RXD2), 0)
            {
               printf("ERROR Going to Standby Mode\n");
            }
```

8.5.2 Stop Mode

When the device goes into this state, the main CPU and most of the circuitry is switched off, keeping the consumption to a minimum in order of uA.

When the unit comes back from Stop Mode the main CPU restarts, so there will be a delay while the Operating System is loaded, and the user application will restart from the beginning.

The signals allowed to return from this mode are the same as for the Standby Mode.

The function provided by the RTU library to switch into this state is,

```
int RTUEnterStop (unsigned long wMainWakeup, unsigned long wExpWakeup);
```

The first parameter required by this function is a 16 bits mask with the bits of the corresponding events that will wake up the unit set. The mask of each event is defined in the RTUControlDefs.h include file.

The second parameter is the mask for the optional expansion board and it must be set to 0 when this secondary board is not installed.

A small example of how to switch to this mode after being initialized and started is shown below:

```
            if(RTUEnterStop( RTU_WKUP_MOVING | RTU_WKUP_CONSOLE))
            {
               printf("ERROR Going to Stop Mode\n");
            }
```

This function will internally call halt on the system, and so there is service called *onhalt* that can be used to execute externally when this happens, for example to save some logs and sync them, etc.

It follows an example of service /etc/systemd/system/onhalt.service to show how this could be implemented.

```
[Unit]
Description=Log status on shutdown
DefaultDependencies=no
#Before=halt.target shutdown.target reboot.target
Conflicts=reboot.target
Before=poweroff.target halt.target shutdown.target
Requires=poweroff.target

[Service]
Type=oneshot
ExecStart=/usr/bin/touch /home/owasys/file
RemainAfterExit=yes

[Install]
WantedBy=halt.target  shutdown.target reboot.target
```

After saving this service on its place it must be enabled with this command:

```
systemctl daemon-reload && systemctl enable onhalt
```

8.5.3 Wake Up Reason

LibRTUControl library provides the way to know the reason for the wake up of the unit. When the device starts up from one of its low power modes, the user can call the following function,

```
int RTUGetWakeUpReason(unsigned long *WakeUpReason);
```

This function returns WakeUpReason, the 16 bits mask with the event that has woken up the unit.

The mask of each event is defined in the owa5X/RTUControlDefs.h include file.

8.5.4 Compatibility table with low power modes

Before entering standby or stop modes, some considerations must be followed because some modules can not be kept on when going to one of these, or when coming back it may be source of problems.

In the following table, a list of modules is shown, if they can be kept on when entering a power mode, and whether it can be used as wake up interruption.

Functionality	Stby	Stop/Off	Wake up source
GSM	YES	YES	YES (RTU_WKUP_GSM)
GPS	NO (user must shut down)	NO (automatic shut down)	NO
Ethernet	YES	YES	YES (RTU_WKUP_ETHERNET)
CAN	YES	YES	YES (RTU_WKUP_CAN1RD)
RS485	YES	NO (automatic shut down)	NO
RS232	YES	YES	YES (RTU_WKUP_CONSOLE)
WiFi/Bluetooth	YES	YES	YES (RTU_WKUP_BLE)
Inputs	YES	YES	YES (RTU_WKUP_DIN[0..8])
Outputs	YES	YES	NO
Vout	YES	YES	NO (user can remove Vout with **RTU_REMOVE_VOUT**)
uSD	YES	NO (automatic shut down)	NO
USB	YES	NO (automatic shut down)	NO

8.6 Hardware RTC

The owa5X device is equipped with a hardware RTC that will keep the time of the unit. *LibRTUControl* library provides functions for performing the different operations related with this peripheral.

The hardware RTC will update automatically the system time when the unit reboots or when it comes from any of the power save modes, although this feature can be deactivated. The system time will never update the RTC. It means that if the application sets the system time the RTC will not be updated, and this must be done explicitly.

8.6.1 Setting the RTC Time and Date

The library function that sets the RTC time and date is,

```
int RTUSetHWTime(THW_TIME_DATE CurrentTime);
```

The parameter required by this function is a THW_TIME_DATE type struct with its field filled with the current time and date. This struct is as follows,

```
typedef struct
{
   unsigned char sec;
   unsigned char min;
   unsigned char hour;
   unsigned char day;
   unsigned char month;
   unsigned short year;
} THW_TIME_DATE;
```

This type is defined inside the RTUControlDefs.h include file.

From command line the RTC can be set from the system time with the command `sysclktohw`.

8.6.2 Getting the RTC Time and Date

The library's function that gets the RTC time and date is,

```
int RTUGetHWTime(THW_TIME_DATE *CurrentTime);
```

The parameter required by this function is a pointer to a THW_TIME_DATE type struct. This function will return with the CurrentTime struct filled with the time and date of the RTC.

8.6.3 Setting the Wake Up Time and Date

The library's functions that set the wake up time and date are,

```
int RTUSetWakeUpTime(THW_TIME_DATE CurrentTime);
int RTUSetIncrementalWakeUpTime(int Second);
```

The first function sets the time and date for wake up from one of the three low power modes if the bit RTU_WKUP_RTC is set (see the Power Management section).

The `RTUSetIncrementalWakeUpTime()` function takes as a parameter the number of seconds the device will be in power save mode. So it will wake up after the time interval specified in the parameter. This function is well suited for scenarios where the unit possibly will not get the date by any means, for lack of GSM and GPS coverage.

8.7 Default Movement Sensor

The owa5X device has a default accelerometer that shows to the customer application if the device has been moved. *LibRTUControl* library provides two functions for getting and resetting the 'MOVED' status, and two functions for configuring and removing the 'MOVED' interruption.

```
int RTUGetMoved(unsigned char *MovedValue);
int RTUResetMoved(void);
```

When the unit has been moved and it is running or in any of the low power modes with the RTU_MOVING bit set, the 'MOVED' status is set. This status continues set until it is cleared (the RTUResetMoved() function is called or the unit goes into one of the two low power modes with the RTU_MOVING bit set for coming back.

8.7.1 Configuring the Movement Sensor interruption

8.7.1.1 Accelerometer HW interruption

This step is mandatory to work with the movement sensor. If the user program will only poll the acceleration values, configure this interruption with high values so that the callback function is never called.

The interruption of the movement sensor may be managed by using a handler function that will be executed when the sensor is moved. The handler can be installed using the following function,

```
int RTU_CfgMovementSensor(unsigned char wScale, unsigned char wLimit,
unsigned char wTime, void(*)(move_int_t));
```

With wScale the range can be modified to either ±2G, ±4G, ±8G and ±16G while with wLimit the threshold may be applied to the sensor, dividing the chosen range by 128. wTime will tell the sensor to interrupt after a certain time has elapsed since the sensor started moving. Finally move_int_t will be the handler that will be executed at a moving interruption.

8.7.1.2 Accelerometer user interruption

For the scenarios where many false positives can arise, there is the possibility of using a different function to get an interruption only after some conditions have been met at HW level. The function to get an interruption on such a way is this one:

```
int RTU_CfgMovementDetection( CONFIG_MOVEMENT_DETECT *wMovementCfg);
```

This function configures the accelerometer with the same parameters as the function RTU_CfgMovementSensor() but adds also these variables:

unsigned short int wMinTimeMovement: Time period in seconds on which the movement must be given under the wMaxTimeBetweenInts criteria to trigger the interruption to the user application.

unsigned short int wMaxTimeBetweenInts: Time period in seconds on which at least one HW movement must be detected.

An example of this functionality can be to set a wMinTimeMovement of 60 seconds and a wMaxTimeBetweenInts of 5 seconds, which means that every 5 seconds there must be at least one HW movement, to have an interruption at user level after 60 seconds since the first HW movement detection.

This scenario is normally used to get an interruption in vehicles, in order to avoid false positives when they are parked and only show the interruption when the vehicle is actually under driving conditions.

8.7.1.3 Removing accelerometer user interruption

Once the accelerometer interruption is no longer used it is important to free the timer used when any of the above functions have been used and a callback function is used to receive the interruptions from the accelerometer.

The interruption and its related internal timer can be free by just calling to the following function:

```
int RTU_RemoveMovementSensor( void);
```

8.7.2 Getting the Movement Sensor Status

After configuring the movement sensor the status can be got.

The library function that gets the movement status is,

```
int RTUGetMoved(unsigned char *MovedValue);
```

This function returns in the MovedValue parameter the 'MOVED' status. If it's a '0' it means that the unit has not been moved from the last time this parameter has been reset. If it contains a '1' it means that the device has been moved.

Please note that by default this sensor is not on. Use RTU_CfgMovementSensor() to configure the settings of the accelerometer, before calling to the RTUGetMoved() function.

8.7.3 Resetting the Movement Sensor Status

The library's function that clears the 'MOVED' status is,

```
int RTUResetMoved(void);
```

This function resets the 'MOVED' status flag.

After performing this action if the sensor detects some movement the value of the 'MOVED' flag is set and it will continue set until this function is called or until the unit switches to one of the power saving modes with the bit RTU_WKUP_MOVING bit set for coming back.

8.7.4 Resetting the Movement Sensor Configuration

The Movement sensor callback function and configuration can be freed with this function:

```
int RTU_RemoveMovementSensor(void);
```

This function will also remove the configuration values, and so this can also be used to change these values while the user application is running, without having to reboot the complete system for this.

8.7.5 Getting the acceleration value

After configuring the movement sensor the acceleration values can be obtained.

The movement sensor registers the acceleration value for X, Y and Z axis. The user program can retrieve the values of these registers in a structure called move_int_t, either after applying the gravity filter or without it. The highest rate to retrieve this information is 50 Hz (for higher rate needs, see optional 6 axis sensor).

The function to get the acceleration values with the gravity filter is,

```
int RTU_GetMovementSensor( move_int_t *pData);
```

The function to get the acceleration values without the gravity filter is,

```
int RTU_GetRawAcceleration( move_int_t *pData);
```

This last function is of help to determine the inclination of the device. If it is well fixed to the vehicle, using the values obtained with this function it is possible to estimate the inclination of the vehicle at any moment.

8.8 Optional 6 axis Movement sensor

Although the optional 6 axis sensor is not controlled with the RTU API it is explained in this section following the description of the default accelerometer that is installed in all units.

The optional 6 axis sensor offers not only movement acceleration readings, but also gyroscope readings, each with values in 3 axis: X, Y and Z. The readings can be configured at a maximum rate of 416 Hz.

The data from this module is interfaced with the industrial IO driver, IIO, which provides a HW ring buffer and events to user space.

As a matter of test the data on this device can be read from the command line with the following commands:

```
cd /sys/bus/iio/devices/iio:device0
echo 1 > scan_elements/in_accel_x_en
echo 1 > scan_elements/in_accel_y_en
echo 1 > scan_elements/in_accel_z_en
echo 1 > scan_elements/in_timestamp_en
echo 4 > buffer/length
echo 1 > buffer/enable
echo 0 > buffer/enable
hexdump /dev/iio\:device0
```

It follows and example of the output of this hexdump command.

```
hexdump /dev/iio\:device0
0000000 04a4 0104 3e49 db71 91fb e610 91d2 0000
```

0000000: sequence number.

04a4: acceleration on X: 0x04a4 * in_accel_x_scale = 1188 * 0.000598 = 0,71

0104: acceleration on Y: 0x0104 * in_accel_y_scale = 260 * 0.000598 = 0,15

3e49: acceleration on Z: 0x3e49 * in_accel_z_scale = 15945 * 0.000598 = 9,5

91fb e610 91d2 0000: uptime in nanoseconds. 0X91d2e61091fb = 160334989005307 nanoseconds

It is important to disable the IIO buffers before rebooting the system, either using the commands or from a c program. Example of commands:

```
cd /sys/bus/iio/devices/iio\:device0
echo 0 > buffer/enable
echo 0 > scan_elements/in_accel_x_en
```

Example of doing it in a C program:

```
/* Enable the buffer */
ret = write_sysfs_int("enable", buf_dir_name, 0);
```

C code to retrieve these data can be found in the iio tools available in the Linux kernel. Contact customer_support@owasys.com to get more information on these tools.

8.9 Analog internal inputs

The different power sources of the unit may be controlled using various functions provided within the RTU library. The power sources are the external Vin, the backup non rechargeable battery and the optional battery.

8.9.1 External power source Vin

The external Vin power source refers to the voltage applied to the V_IN input in the pin 24 of the connector.

```
int RTUGetAD_V_IN(float *ad_v_in);
```

8.9.2 Main optional battery

Depending in the customer's needs, an optional battery may be mounted in the unit. Its voltage level can be measured using the following RTU function.

```
int RTUGetAD_VBAT_MAIN(float *ad_vbat_main);
```

8.9.3 Internal temperature

The internal temperature can be obtained with this function.

```
int RTUGetAD_TEMP(int *ad_temp);
```

The temperature value is also communicated in the system via DBUS with the member name temperature, from command line it can be checked like this:

```
dbus-monitor –system “member=temperature”
```

8.10 Timers

The Linux operating system offers one real timer to use in customer applications. In order to improve it, *owa5X* architecture offers a control service with a set of timers whose minimal resolution is 1 ms.

8.10.1 RTU library timers

IO library offers several functions to handle the timers, as explained below. The minimal resolution for these timers is 10 ms.

The timer has an internal counter which is initialized to the top time value when *OWASYS_GetTimer()* function is called.

The timer waits in *Stopped* internal status until the *OWASYS_StartTimer()* or *OWASYS_RestartTimer()* functions are called. Once either of these functions has been called, the timer switches to *Running* internal status.

If the *OWASYS_StopTimer()* function is called, the timer switches to *Stopped* internal status, maintaining the internal counter to its current value.

If the *OWASYS_StartTimer()* function is called, the timer switches to *Running* internal status again, continuing with the previous value of the internal counter.

If the *OWASYS_RestartTimer()* function is called, the timer switches to *Running* internal status too, but the internal counter is reset to the top time value, beginning to count down the counter from there again.

When starting the timer the user has two options, ONE_TICK and MULTIPLE_TICK.

If ONE_TICK is chosen every time the internal counter reaches 0 value, the timer switches automatically to *Stopped* internal status, maintains the internal counter at 0 value, and the handler specified in the *OWASYS_GetTimer()* function is executed.

If MULTIPLE_TICK is chosen instead, every time the internal counter reaches 0 value, the handler is executed and the counter is automatically restarted.

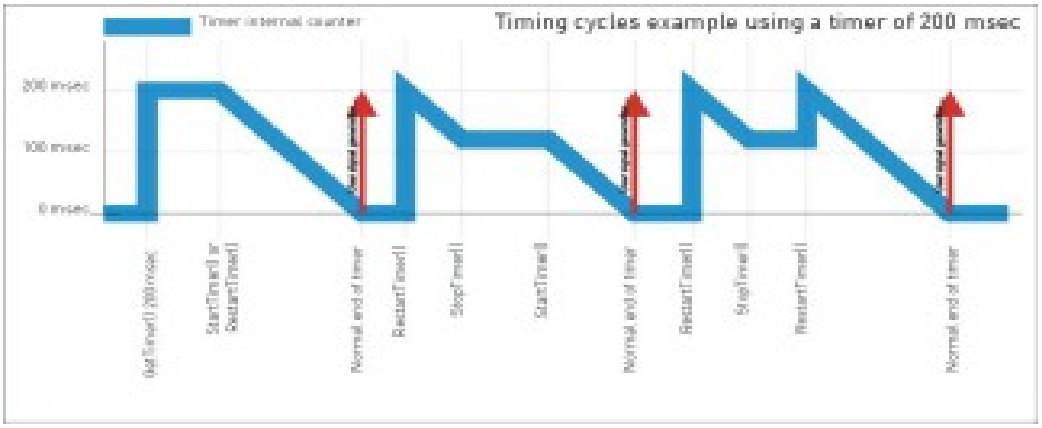
If the *OWASYS_FreeTimer()* function is called, the top value is set to 0 value, and the timer is not available any more.

These changes to internal Status and Counter are shown in next table:

	Internal status	Timer limit	Internal counter
OWASYS_GetTi	Stopped	200 msec	200 msec

	Internal status	Timer limit	Internal counter
mer()			
OWASYS_StartTimer() OWASYS_RestartTimer()	Running	200 msec	200 msec (counting down)
OWASYS_StopTimer()	Stopped	200 msec	x msec
OWAYS_StartTimer()	Running	200 msec	x msec (counting down)
OWASYS_RestartTimer()	Running	200 msec	200 msec (counting down)
OWASYS_FreeTimer()	Stopped	0 msec	0 msec

Below is a scheme of how a 200 milliseconds timer would work:



An **example** of how to use timers is shown below.

First, the application must start the RTU module.

```
if( (ReturnCode = RTUControl_Initialize(NULL)) != NO_ERROR) {
   printf("Error %d in RTUControl_Initialize()...\n", ReturnCode);
   return 1;
}
if( (ReturnCode = RTUControl_Start()) != NO_ERROR) {
   printf("Error %d in RTUControl_Start()...\n", ReturnCode);
   return 1;
}
```

For each timer that the application is going to use, a handling function must be defined. This handling function will be activated every time the timer internal counter reaches 0. For linking this handling function with the timer application, it is only necessary to reference the pointer of a function and pass it as an argument when getting a timer.

```
int  ReturnCode;
unsigned char TimerId;
```

```
……..

if((ReturnCode=OWASYS_GetTimer(      &TimerId,      (void      (*)(unsigned
char))&TimerHandler, 1, 0);)!= NO_ERROR) {
   printf( "   Error %d in TIMERIO_GetTimer()\n", ReturnCode);
   return ReturnCode;
}

if((ReturnCode    =    (*FncTIMERIO_StartTimer)(TimerId,    ONE_TICK))    !=
NO_ERROR) {
   printf( "Error %d in FncTIMERIO_Start()\n", ReturnCode);
   return ReturnCode;
}
```

Every 1 second, the signal assigned to the timer will be raised, and it will be handled by its handler function. If ONE_TICK is chosen, once the handler function has been executed the timer will get stopped, and if the application wants to use it again it must restart it. If MULTIPLE_TICK is chosen instead, the handler function will be executed every time the counter reaches 0 and it will be automatically restarted.

In this example the timer is restarted.

```
void TimerHandler( int wStatus)
{
   int ReturnCode = NO_ERROR;

   if( ( ReturnCode = TIMERIO_RestartTimer(TimerId)) != NO_ERROR) {
      printf("Error %d in FncTIMERIO_RestartTimer()\n", ReturnCode);
   }
}
```

It is highly recommended that the code of timer handlers is as short as possible. Use a flag to tell the main function that the timer signal has been executed so that the main function performs what the customer requires.

Once the application has finished using a timer, it must stop it.

```
TIMERIO_StopTimer(TimerId);
TIMERIO_FreeTimer(TimerId);
```

Finally, the RTU library must be finalized.

```
if( ( ReturnCode = RTUControl_Finalize()) != NO_ERROR) {
      printf("Error %d in RTUControl_Finalize()...\n", ReturnCode);
      return 1;
   }
return 0;
```

8.10.2 Linux timing functionality

In order to schedule timer events, although the LibRTUControl library still provides a set of timers available for all kind of uses, it is also possible to use High Resolution Timers, as the Kernel supports them. This owa5X feature allows the user to request a timer interval in the microsecond range (sub-jiffy timers). Thus, nanosleep, itimers and posix timers provide such high resolution mode without changes to the source code.

It follows an example of a 100 us timer, which periodically calls the function print_time.

```
/* SIGALRM for printing time */
memset (&action, 0, sizeof (struct sigaction));
action.sa_handler = print_time;
if (sigaction (SIGALRM, &action, NULL) == -1)
perror ("sigaction");
/* for program completion */
memset (&sevent, 0, sizeof (struct sigevent));
sevent.sigev_notify = SIGEV_SIGNAL;
sevent.sigev_signo = SIGRTMIN;
if (timer_create (CLOCK_MONOTONIC, &sevent, &timer1) == -1)
perror ("timer_create");
new_value.it_interval.tv_sec = 0;
new_value.it_interval.tv_nsec = 100000; /* 100 us */
new_value.it_value.tv_sec = 0;
new_value.it_value.tv_nsec = 100000; /* 100 us */
if (timer_settime (timer1, 0, &new_value, &old_value) == -1)
perror ("timer_settime");
```

This code must be compiled with -lrt linking option (POSIX.1b Realtime Extensions library).

```
gcc -o hires_timer hires_timer.c -lrt
```

It is necessary to note that High Resolution Timers require system clock interrupts to, temporarily, occur at a higher rate, which tends to increase power consumption.

9 owa5X IOs Module API

9.1 Introduction

LibIOs_Module library manages a resource to access the IOs of the system architecture. The offered services are the following:

- Access for reading/writing digital inputs/outputs.
- Read analog inputs.
- Manage the blinking of the LEDs.
- Switch on and off HW parts: CAN, uSD, WiFi, Bluetooth.

To see the whole set of IO functions included in *IOsModule* library, see the API in the developers area.

9.2 LEDs control

9.2.1 Using the API IO library

Any of the LEDs may be controlled by the user. By default the user has control of all LEDs, and can give control of the yellow LED to GSM and the orange LED to GNSS calling to these functions:

- GNSS control of orange LED

```
if((ReturnCode = DIGIO_Set_PPS_GPS_Input()) != NO_ERROR) //give GNSS
control of orange LED
   printf("Error %d in DIGIO_Set_PPS_GPS_Input()\n", ReturnCode);
```

When the GNSS takes control of the LED, once it gets the time from the satellites, the LED will start blinking at the same rate as the GNSS is configured, which is by default 1 Hz.

- GSM control of yellow LED

```
if((ReturnCode = DIGIO_Set_LED_SW0_Input()) != NO_ERROR) //give GSM
control of yellow LED
   printf("Error %d in DIGIO_Set_LED_SW0_Input()\n", ReturnCode);
```

When the GSM takes control of the yellow LED, these are the blinking patterns for each state:

GSM	Yellow LED
GSM CS data call in progress or established GSM voice call in progress or established UMTS voice call in progress or established - UMTS CS data call in progress	10 ms on / 990 ms off
GSM PS data transfer UMTS data transfer	10 ms on / 1990 ms off

ME registered to a network. No call, no data transfer	10 ms on / 3990 ms off
Limited Network Service (e.g. because no SIM/ USIM, no PIN or during network search)	500ms on / 500 ms off

By default all the LEDs are off when the system of the owa5X boots up.

9.2.1.1 Set LED status

To power on and off the LED only in the point of the source code the user wants, these functions must be used.

- YELLOW LED

```
if((ReturnCode = DIGIO_Set_LED_SW0(1)) != NO_ERROR) //SET ON YELLOW LED
   printf("Error %d in DIGIO_Set_LED_SW0()\n", ReturnCode);
```

- GREEN LED

```
if((ReturnCode = DIGIO_Set_LED_SW1(1)) != NO_ERROR) //SET ON GREEN LED
   printf("Error %d in DIGIO_Set_LED_SW1()\n", ReturnCode);
```

- RED LED

```
if((ReturnCode = DIGIO_Set_LED_SW2(1)) != NO_ERROR) //SET ON RED LED
   printf("Error %d in DIGIO_Set_LED_SW2()\n", ReturnCode);
```

- ORANGE LED

```
if((ReturnCode = DIGIO_Set_PPS_GPS(1)) != NO_ERROR) //SET ON ORANGE LED
   printf("Error %d in DIGIO_Set_PPS_GPS()\n", ReturnCode);
```

9.2.2 Control the LEDs from the user space

The LEDs in the owa5X platform can be controlled from the user space using /sys/class/leds/[1], which also allows to set the timers for blinking. For example:

```
echo 1 > /sys/class/leds/ledsw0\:yellow/brightness
echo timer > /sys/class/leds/ledsw1\:green/trigger
echo 1000 > /sys/class/leds/ledsw1\:green/delay_off
echo 100 > /sys/class/leds/ledsw1\:green/delay_on
```

9.2.3 Boot and system status

The boot process and the systemd services status can be checked with the LEDs if the following u-boot environment variable is set to 1:

[1] https://www.kernel.org/doc/html/latest/leds/leds-class.html

```
fw_setenv ledsbootstatus 1
```

1. U-Boot Phase

The system checks if ledbootstatus = 1.

If 1, the RED LED turns ON.

If 0, the boot process continues to the next phase.

2. Kernel Phase

The system checks the boot command line for ledbootstatus =1.

If 1, the GREEN LED starts blinking.

If 0, the process moves to the next stage.

3. RootFS Phase

The owasysd-led-status.service is executed to check if any critical system service has failed.

If a service has failed, the RED LED turns ON.

If no failures are detected, the system checks the timer status again:

If ledbootstatus = 1, the GREEN LED stays ON.

Otherwise, it proceeds to the next step.

9.3 Handling analog inputs and digital inputs/outputs

9.3.1 Starting the application

Before any function from the IO library is used, the RTU and IO libraries must be started

```
if( (ReturnCode = RTUControl_Initialize(NULL)) != NO_ERROR) {
   printf("Error %d in RTUControl_Initialize()...\n", ReturnCode);
   return 1;
}
if( (ReturnCode = RTUControl_Start()) != NO_ERROR) {
   printf("Error %d in RTUControl_Start()...\n", ReturnCode);
   return 1;
}
if( (ReturnCode = IO_Initialize() ) != NO_ERROR ) {
   printf("Error %d in IO_Initialize()...\n", ReturnCode);
   return 1;
}
if( (ReturnCode = IO_Start()) != NO_ERROR ) {
   printf("Error %d in IO_Start()...\n", ReturnCode);
   return 1;
}
```

9.3.2 Polling digital inputs

In order to get the value of one of the digital input, the corresponding function must be called (see API for the whole set of functions). In the following piece of code, the value of digital input DIN1 is retrieved:

```
int din1;

if( (ReturnCode = (*FncDIGIO_Get_DIN)(1, &din1)) != NO_ERROR)
   {
      printf("Error %d in FncDIGIO_Get_DIN()", ReturnCode);
      return 1;
   }
```

9.3.3 Getting digital inputs value by interrupt

The digital inputs DIN0 to DIN8 can be configured to interrupt the program. DIN[9-11] can neither interrupt the program nor wake up the system from a Low Power Mode.

For these inputs, it is possible to configure an interrupt service on the system in such a way that if a change is given in that digital input the system notifies the application by means of calling a referenced function.

As usual, the IO library functions must be loaded and the IO module must be initialized and started.

The next step is to configure and enable the interrupt service. The handler is called only on edge mode, so it will only be raised every time the digital input changes its value.

The edge can be configured to interrupt when the level goes down, when it goes up or to interrupt in both cases.

In this example, the application will get an interrupt if DIN1 goes to down level.

```
  unsigned char  InNumber;
  unsigned char  EdgeValue;
  unsigned short int NumInts;

    InNumber = 1; // DIN1 interruption
    EdgeValue = 0; // Interruption at low edge
    NumInts = 1; // Interruption every time it goes to low level

       if(  (ReturnCode  =  DIGIO_ConfigureInterruptService(  InNumber,
EdgeValue, (void (*)(input_int_t))&InputIntHandler, NumInts)) != 0)
   {
            printf(  "Error  %d  in  DIGIO_ConfigureInterruptService()\n",
ReturnCode);
   }
```

Warning: for changing interrupt configuration, first remove the interrupt service and then configure the interrupt service again.

The handler function will be called at a frequency specified by the fourth argument of DIGIO_ConfigureInterruptService() function. If the value of NumInts is set to 0, the handler will never be executed, but the number of interrupts will be saved, so they can be obtained at any moment with function DIGIO_GetNumberOfInterrupts()

To change any configuration parameter or to simply free the interruption from the system, OWASYS_RemoveInterruptService() must be called.

```
   DIGIO_RemoveInterruptService( InNumber);
```

Finally, the number of interruptions can be obtained with the following function.

```
unsigned long TotalInts;

DIGIO_GetNumberOfInterrupts( InNumber, &TotalInts);
```

The maximum input frequency for any of the digital inputs is 3 Khz.

9.3.4 Getting analog inputs value

The 4 analog input values are retrieved as a value between 0 and 4095, which must be passed to volts with the following rule:

- Range 0V to 5.12V: 1.25mV per step.
- Range 0V to 30.72V: 7.5mV per step.

```
int AnalogValue;
int AnalogNumber;

AnalogNumber = 0;

ANAGIO_GetAnalogIn( AnalogNumber, &AnalogValue);
```

9.3.5 Changing analog input range

The range of the analog inputs can be configured to work with on of the two possible ranges:

- 0V to 5.12V
- 0V to 30.72V

The range can be changed calling to one internal digital output, as shown in the following example:

```
unsigned char AnalogNumber;
unsigned char AnalogRange;

AnalogNumber = 0;
AnalogRange = 1; // range 1: 0V – 30.72V

DIGIO_Set_ADC_RANGE( AnalogNumber, AnalogRange);
```

10 owa5X GSM API

10.1 Introduction

The GSM API offers programmers all GSM functionality through a set of library functions, not having to worry about commands or the GSM module installed in the unit.

10.2 Starting GSM

GSM library, as well as RTU and IO libraries, exports two functions that must be called if application wants to start using GSM functionality. These are *GSM_Initialize()* and *GSM_Start()*.

But, before the GSM library is started it is compulsory to initialize and start both RTU and IOs libraries in that specific order. This is done as explained in previous sections.

For 4G networks it is also important to activate the PDP context so that the SIM card can register to 4G networks, and start afterwards a data session. Use function *GSM_DefinePDPContext()*.

Once that IO and RTU have been initialized and started, then GSM module must be initialized and started as well. Initialization procedure will be as follows.

```
   // Starting GSM
   memset( &gsmConfig, 0, sizeof( TGSM_MODULE_CONFIGURATION));

   // PIN Introduction
   // User: Introduces PIN Code

   printf( "OWASYS--> Insert PIN Code: ");
   memset( ( void *) &keyEntry, 0, sizeof( keyEntry));
   getEntry( keyEntry);

   if( keyEntry[ 0] == 0){       //<PIN_Code>
      strcpy( ( char*) gsmConfig.wCode, "");
   } else {
      strcpy( ( char*)( gsmConfig.wCode), ( char*) keyEntry);
   }

   gsmConfig.gsm_action = gsm_event_handler;
   // sem init and initialization of events buffer
   InitGsmEventBuffer();
   // GSM Initialize
   if( ( GSM_Initialize ( ( void*) ( &gsmConfig))) > 0){
      exit( 1);
   }

   // GSM Start
   if ( ( ReturnCode = GSM_Start ( )) != NO_ERROR){
          printf(  "OWASYS-->  ERROR  on  GSM  Initialization  (  %d  )\n",
ReturnCode);
      IO_Finalize( );
      RTUControl_Finalize ( );
      exit( 1);
   }
   printf( "OWASYS--> OK on GSM Initialization \n");

   // Subscription to SMS Events. (Initialize module to receive SMSs)
   ReturnCode = 1;
   if( ( ReturnCode = GSM_SMSIndications ( 1)) == NO_ERROR){
      printf( "OWASYS--> OK SMS Indications Active\n");
   } else{
       printf( "OWASYS--> ERROR on SMS Indications Activation ( %d )\n",
ReturnCode);
```

```
   }

   // Subscription to USSD Events.  (Initializa module to receive USSDs)
   if( GSM_SetUSSD ( 1) == NO_ERROR) {
      printf( "OWASYS--> OK USSD notification ENABLED\n");
   }else{
      printf( "OWASYS--> ERROR USSD notification NOT ENABLED\n");
   }

   // Checking if GSM is Active
   GSM_IsActive ( &isActive);
   if( isActive == 1){
     printf("\n********** GSM IS UP AND RUNNING*******\n");
   } else {
     printf("\n********** GSM IS NOOOOOT RUNNING*******\n");
   }
   // Starting GSM Events Attention routine.
   runGSMHandler = TRUE;
   pthread_create( &gsmEvents,NULL,GSMHandleEvents,NULL);
```

In this piece of code

1. PIN is requested and stored in the configuration structure gsmConfig.
2. The handler function pointer is assigned to the gsmConfig.gsm_action structure field. This function is the one that will be called every time an event takes place in the GSM module.
3. A semaphore and an event buffer are initialized. This semaphore synchronizes the main application with a concurrent thread needed for handling asynchronous GSM events.
4. The handler function pointer is assigned to the gsmConfig.gsm_action structure field. This function is the one that will be called every time an event takes place in the GSM module.
5. GSM_Initialize() is called.
6. GSM_Start() is called, if this call success GSM library is ready to accept requests.
7. GSM_IsActive() retrieve whether the GSM module is ready or not. If everything has gone OK, it will return a 1 in its parameter.
8. A separate thread, gsmEvents, is created. This will be in charge of handling the GSM events reported by the library, as later explained. The runHandleEvents flag is set to true, a flag that controls whether the handling events thread is running or not.

10.3 Handling GSM Events

The API describes GSM API functions, type definitions, library reported errors and the GSM events (incoming calls, sms received…)

As explained before, every time one of these events is reported the handler function is executed. This attending routine just sets a flag, which indicates that a GSM event is pending, and signals a semaphore. A different thread, running concurrently, will be in charge of handling these events. If no event is received this thread will be sleeping in a semaphore. When an event is received the gsm_event_handler function will wake the thread up after adding the received event to a buffer.

```
static void gsm_event_handler( gsmEvents_s *pToEvent)
{
   int auxi = (GsmEventsWr+1);
```

```
   GsmEventBuffer[GsmEventsWr] = *pToEvent;
   if( GsmEventsWr == GsmEventsRd) {
      GsmEventsWr = auxi;
   } else {
      if( auxi >= MAX_EVENTS) {
         auxi = 0;
      }
      if( auxi != GsmEventsRd) {
         GsmEventsWr = auxi;
      }
   }
   if( GsmEventsWr >= MAX_EVENTS) {
      GsmEventsWr = 0;
   }
   sem_post( &GsmEventsSem);
}
```

As stated before, the handling of all the GSM events is done by a separate thread. In the GSM starting procedure we have seen the following call.

```
pthread_create( &gsmEvents,NULL,GSMHandleEvents,NULL);
```

The pthread library must be compiled with the user program. In that case the #include ←pthread.h→ and the option –lpthread in the makefile will resolve and link the library. pthread_create and other functions related to the threads construction and destruction are well explained in Linux manual pages.

This launches a new concurrent task for handling the GSM events.

```
void* GSMHandleEvents( void *arg)
{
   gsmEvents_s *owEvents;
   gsmEvents_s LocalEvent;
   //User Vars.
   int   retVal;
   //RING
   unsigned char  ringTimes      = 0;
   //SMS
   int      SMSIndex;
   SMS_s    incomingSMS;
   unsigned char    SMSSize;

   while( runGSMHandler == TRUE){
      sem_wait( &GsmEventsSem);
      if( GsmEventsRd == GsmEventsWr) {
         continue;
      }
      LocalEvent = GsmEventBuffer[GsmEventsRd++];
      owEvents = &LocalEvent;
      if( GsmEventsRd >= MAX_EVENTS) {
         GsmEventsRd = 0;
      }
      switch ( owEvents->gsmEventType){
         case GSM_NO_SIGNAL:
            break;
         case GSM_RING_VOICE:
         case GSM_RING_DATA:
            ringTimes ++;
            if( owEvents->gsmEventType == GSM_RING_DATA){
                printf( "OWASYS--> GSM RING DATA signal Phone Number: %s \
n",
                              owEvents->evBuffer);
```

```
            } else {
                  printf( "OWASYS--> GSM RING VOICE signal Phone Number:
%s \n",
                                owEvents->evBuffer);
            }
            break;
         case GSM_NEW_SMS:
               [...]
               break;
            default:
               printf( "Unknown temperature status\n");
               break;
            }
            break;
         default:
                  printf( "OWASYS--> Signal Event not found ...%d \n",
owEvents->gsmEventType);
      }
   }
   return NULL;
}
```

This function consists of a loop controlled by a flag, runGSMHandler, a boolean global variable. This must be set to true when the event handling thread is created (see GSM starting procedure) and then to false when application has finished with GSM (see GSM finishing procedure).

This thread must wait till an event is received, this is done by means of the semaphore call sem_wait(&gsmHandlerSem).

If the semaphore is signaled, then thread continues and looks for the type of event reported from the global buffer of events.

Afterwards, there is a switch containing the different events that the customer application wants to attend plus the actions related to each event.

10.4 Subscription to a New SMS Arrival

The GSM library is ready to receive every event described in the API except for SMS arrival, unless it is explicitly called, which is described within this section.

The GSM transceiver will be configured in the proper way to report those events on doing a call to the `GSM_SMSIndications()` function. If its return code is NO_ERROR, this implies that the subscription has been successful.

The code place where the SMS subscription is made is shown below:

```
    if( ( ReturnCode = GSM_SMSIndications ( 1)) == NO_ERROR){
        printf( "OWASYS--> OK SMS Indications Active\n");
       } else{
        printf( "OWASYS--> ERROR on SMS Indications Activation ( %d )\n",
ReturnCode);
       }
```

The 1 value means ENABLE, and 0 DISABLE.

Once the subscription has been made, the following example shows how to read a new arrived SMS. (This piece of code should be located in the `handleEvents` thread, under the NEW_SMS switch case.)

```
case GSM_NEW_SMS:
{
   int smsIndex;
   unsigned char smsSize;
   SMS_s* readIncomingSMS;

   smsIndex = (char) atoi( owEvents->evBuffer);
   printf( "---> OWASYS DEMO: Entering to read the SMS %d\n",smsIndex);

   readIncomingSMS = ( SMS_s*) malloc( sizeof( SMS_s));

   retVal = ( *FncGSM_ReadSMS) ( readIncomingSMS, &smsSize, smsIndex, 0);

   if( retVal == NO_ERROR)
               printf("   --->   OWASYS   DEMO:   GSM   NEW   SMS:   Received
@:%2.2d/%2.2d/%2.2d,%2.2d:%2.2d\nMessage:%s\n",
            readIncomingSMS->owSCDateTime.day,
            readIncomingSMS->owSCDateTime.month,
            readIncomingSMS->owSCDateTime.year,
            readIncomingSMS->owSCDateTime.hour,
            readIncomingSMS->owSCDateTime.minute,
            readIncomingSMS->owBody);

   free( readIncomingSMS);
}
break;
```

On the arrival of the SMS, the SMS index is stored on the **evBuffer** of the **gsmEvents_s** (**owEvents** variable) structure. Once the user program starts to process the event, loads the **GSM_ReadSMS()** function, and runs it, (the SMS index parameter being the one given by the **evBuffer** field in the **owEvents** structure). If this function succeeds, the incoming SMS is returned in a **SMS_s** structure, all of the SMS relevant fields being available.(See SMS_s type definition in the API)

10.5 Finishing GSM

Once application has done with GSM, application should take the opposite steps to the GSM start, so the process must be GSM finish and library unload, if necessary.

```
   GSM_Finalize ( );

   runGSMHandler = false;

   sem_post( &GsmEventsSem);

   pthread_join( gsmEvents, NULL);

   IO_Finalize( );
   RTUControl_Finalize ( );

   printf( " Ending the GSM Application Note #1\n");
   exit ( 0);
}
```

1. GSM is finalized first, *GSM_Finalize()*
2. The flag that controls that the thread is looping, runHandleEvents, is set to false.
3. The semaphore is signaled to wake up the thread. The thread will exit
4. The thread is joined from the main.

5. Semaphore is destroyed, as we do not need it anymore
6. IO and RTU libraries are finalized

11 owa5X iNet API

11.1 Introduction

The iNet library provides all functions to connect to the Internet by generating an IP routing table and establishing a GPRS session. With this library, the customer does not have to worry about starting the required Internet Protocol sessions to communicate with the GSM module. This is done automatically by high level functions.

11.2 Starting an Internet connection

To establish an Internet connection, RTU, IO and GSM must be first started, as explained in section 10.2.

Once the GSM is initialized and started the Internet session should be started as is shown in the following code.

```
TINET_MODULE_CONFIGURATION    iNetConfiguration;
GPRS_ENHANCED_CONFIGURATION   gprsConfiguration;

sem_init( &iNetHandlerSem, 0, 0);
runiNetHandleEvents = TRUE;
pthread_create(&iNetEvents, NULL, iNetHandleEvents, NULL);
```

All the events are controlled by a created thread that calls the *iNetHandleEvents* function (For more information, see the owasys_an24 Application Note source code).

To call the library function `int iNet_Initialize (void*)`, a *TINET_CONFIGURATION structure* must be initialized with the information needed as shown below.

```
printf ("Insert USER: ");
memset( ( void *) &strEntry, 0, sizeof( strEntry));
getEntry( strEntry);
strcpy( ( CHAR*) gprsConfiguration.gprsUser, ( CHAR*) strEntry);
printf ("Insert PASSWORD: ");
memset( ( void *) &strEntry, 0, sizeof( strEntry));
getEntry( strEntry);
strcpy( ( CHAR*) gprsConfiguration.gprsPass, ( CHAR*) strEntry);
printf ("Insert DNS1: ");
memset( ( void *) &strEntry, 0, sizeof( strEntry));
getEntry( strEntry);
strcpy( ( CHAR*) gprsConfiguration.gprsDNS1, ( CHAR*) strEntry);
printf ("Insert DNS2: ");
memset( ( void *) &strEntry, 0, sizeof( strEntry));
getEntry( strEntry);
strcpy( ( CHAR*) gprsConfiguration.gprsDNS2, ( CHAR*) strEntry);
printf ("Insert APN: ");
memset( ( void *) &strEntry, 0, sizeof( strEntry));
getEntry( strEntry);
strcpy( ( CHAR*) gprsConfiguration.gprsAPN, ( CHAR*) strEntry);
iNetConfiguration.wBearer = INET_BEARER_ENHANCED_GPRS;
iNetConfiguration.inet_action = inet_event_handler;
InitInetEventBuffer();
iNetConfiguration.wBearerParameters = (void*) &gprsConfiguration;

(*Fnc_iNetInitialize)( ( void*) &iNetConfiguration);
ReturnCode = ( *Fnc_iNetStart) ( );
if( ReturnCode != NO_ERROR){
   iNetFinalized       = TRUE;
```

```
      runiNetHandleEvents = FALSE;
      sem_post    ( &iNetHandlerSem);
      sem_destroy ( &iNetHandlerSem);
       printf("OWASYS--> ERROR Initializing the Internet session(%d)\n",
ReturnCode);
   } else {
      iNetFinalized       = FALSE;
      printf("OWASYS--> OK Internet session started\n");
   }
```

The events will be handled with the function inet_event_handler, which pointer is passed in the 'inet_action' configuration structure field. The buffer of inet events is also cleared before the initialization.

Once the iNet module is initialized, it must be started by calling the library function `iNet_Start( void )` function.

The ppp0 interface will get an IP from the operator, the DNS are then configured into */etc/resolv.conf* and the interface is set as default gateway in the routing table with metric 5. This is so to let another gateway being used, for example the one from ethernet with its default metric 0, and then use the ppp0 connection only when there is no other one active.

11.3 Closing the Internet Connection

Finishing the iNet must follow the following steps:

- Call the library function `iNet_Finalize()`.
- Stop the handler of the iNet Module events setting runiNetHandleEvents to FALSE.
- The semaphore is signaled so that the thread stops writing an event.
- Kill the thread that controlled the iNet events calling pthread_exit() from the thread and pthread_join() from the main function.
- Destroy the semaphore.
- Follow the same steps for GSM finishing (See section 10.5).

```
   ( *FnciNet_Finalize) ( );
   runiNetHandleEvents = FALSE;
   sem_post( &iNetHandlerSem);
   pthread_join(iNetEvents,NULL);
   sem_destroy ( &iNetHandlerSem);
```

11.4 Out of GSM Coverage

In case that the GSM module gets out of GSM coverage, the GPRS session is not interrupted.

When the module has not coverage to send data to the network, it stores the data in an internal GSM module buffer and waits till coverage is recovered, then it sends the stored data. The buffer is approximately 1000 bytes long.

In case that the buffer gets full before coverage is recovered there are two different scenarios, depending on whether application is using TCP or UDP protocol:

- The application uses UDP protocol. The application will receive GSM_STOP_SENDING_DATA event when the buffer gets full. In this case, the user application must stop sending data over the GPRS session, because the GSM module will not be able neither to send nor to store them in the buffer. When the GSM coverage is recovered, the application will receive the GSM_START_SENDING_DATA event. At this moment, the GSM module will send all the internally stored data and the application will be able to re-start sending data over the GPRS session.
- The application uses TCP protocol. In this case the application will not receive any event indicating a lost of GSM coverage. Although TCP is a secure protocol with acknowledgement procedure, the application will not realize that the TCP packets do not arrive to the other end of the already established connection. This is due to the TCP protocol congestion control dynamic window. In this scenario, what is called the window and time between retransmission, get bigger and bigger. The application will need another mechanism to realize the loss of coverage.

11.5 GSM functionality in an active INET Session

SMS case:

The SMS will be sent as if no INET session is active (although it is). The whole set of functions is able to be executed while INET session is active.

In the owa5X the voice calls are not possible, the GSM module can only be used for data connectivity and SMS communication.

GPRS coverage lost: In case the GPRS coverage is lost (look at event: GSM_COVERAGE = 0) when using TCP protocol, if the user goes on sending data, these data will be stored in an internal buffer of the GSM peripheral. Once the coverage is recovered these data are sent.

Many users do not want to send non-updated data so to avoid receiving old data, on losing the GPRS coverage the user must not send IP data.

12 owa5X GPS API

12.1 Introduction

The GPS library will give programmers an easy interface, in a high level language, to communicate with a GPS receiver and get positioning information.

12.2 Starting GPS

To start an application wherein GPS is an important part, the GPS must be started.

Prior to loading the GPS, RTU and IO must be initialized and started:

```
#include "owasys/GPS2_ModuleDefs.h"
#include "owasys/pm_messages.h"
#include "owasys/IOs_ModuleDefs.h"
#include "owasys/RTUControlDefs.h"

[...]

// start RTU
  if( ( ReturnCode = RTUControl_Initialize( NULL)) != NO_ERROR) {
      printf("Error %d in RTUControl_Initialize()", ReturnCode);
      return -1;
  }

  if( ( ReturnCode = RTUControl_Start()) != NO_ERROR) {
      printf("Error %d in RTUControl_Start()", ReturnCode);
      return -1;
  }
  // Start IOs
  if( ( ReturnCode = IO_Initialize( )) != NO_ERROR) {
      WriteLog("Error %d in IO_Initialize()", ReturnCode);
      return -1;
  }
  if( ( ReturnCode = IO_Start()) != NO_ERROR) {
      WriteLog("Error %d in IO_Start()", ReturnCode);
      IO_Finalize();
      return -1;
  }
```

Once RTU and Ios have been started, the GPS must be initialized and started:

```
   TGPS_MODULE_CONFIGURATION  GPSConfiguration;
   char                       GpsValidType[][20] = {"NONE", "GPS_UBLOX"};
    char                                GpsValidProtocol[][10] = {"NMEA",
"BINARY"};
   int                        ReturnCode, IsActive;

   memset( &GPSConfiguration, 0, sizeof( TGPS_MODULE_CONFIGURATION));
   strcpy(GPSConfiguration.DeviceReceiverName, GpsValidType[1]);
   GPSConfiguration.ParamBaud = B115200;
   GPSConfiguration.ParamLength = CS8;
   GPSConfiguration.ParamParity = IGNPAR;
   strcpy(GPSConfiguration.ProtocolName, GpsValidProtocol[0]);
   GPSConfiguration.GPSPort = COM4;

        // GPS module initialization.
        if( ( ReturnCode = GPS_Initialize( ( void *) &GPSConfiguration)) !
= NO_ERROR) {
      WriteLog("Error %d in GPS_Initialize()", ReturnCode);
      if (ReturnCode != ERROR_GPS_ALREADY_INITIALIZED) {
          return -1;
```

```
      }
   }
   // GPS receiver startup
       if ( ( ReturnCode = GPS_Start()) != NO_ERROR ) { //start GPS
receiver.
      WriteLog("Error %d in GPS_Start()", ReturnCode);
      if (ReturnCode != ERROR_GPS_ALREADY_STARTED) {
          return -1;
      }
   }
       GPS_IsActive( &IsActive);
       printf("IS ACTIVE(%d)\r\n", IsActive);
   WriteLog("GPS-> Module initialized & started");
```

Note that the COM port in the owa5X changes to COM4 from the COM1 used in the owa4x platform.

Once it is initialized, the GPS should be started: *GPS_Start()*.

If GPS_Start() Function returns an error, the user must finalize the GPS calling *GPS_Finalize()*.

12.3 Finishing GPS

Ending the GPS takes the steps opposite to the GPS start, so the process must be GPS finalized, then IO and finally RTU.

```
  GPS_Finalize();
  RTUControl_Finalize();
  IO_Finalize();
```

First of all, the GPS instance is finalized, followed by the RTUControl instance and the IOs instance respectively.

Warning: Only the GPS operation must be finished in case other modules remain running.

In the GPS Application Note (**owasys_an5**) the complete sample source code for GPS initialization and startup can be found.

12.4 Example of getting GPS position

In order to get the GPS receiver position, the library function *GPS_GetAllPositionData()* must be called:

```
int      ReturnCode;
TGPS_POS CurCoords;

int   ReturnCode = 0;
tPOSITION_DATA LocalCoords;
static int x=0, NumOld = 0;

ReturnCode = GPS_GetAllPositionData( &LocalCoords );
if( ReturnCode != NO_ERROR )
   printf( "Error %d in GPS_GetAllPositionData()...\n", ReturnCode);
else {
```

```
   if( LocalCoords.OldValue != 0){
      NumOld++;
   }
   x++;
   printf("CYCLES(%d)PosValid(%hhu)OLD POS(%hhu)TOTAL(%d),NAV STATUS(%s)\
r\n", x, LocalCoords.PosValid, LocalCoords.OldValue, NumOld,
LocalCoords.NavStatus);
   printf("LATITUDE  --> %02hu degrees %02hhu minutes %04.04f seconds %c
(%.7lf)\r\n", LocalCoords.Latitude.Degrees, LocalCoords.Latitude.Minutes,
LocalCoords.Latitude.Seconds, LocalCoords.Latitude.Dir,
LocalCoords.LatDecimal);
   printf("LONGITUDE --> %03hu degrees %02hhu minutes %04.04f seconds %c
(%.7lf)\r\n", LocalCoords.Longitude.Degrees,
LocalCoords.Longitude.Minutes, LocalCoords.Longitude.Seconds,
LocalCoords.Longitude.Dir, LocalCoords.LonDecimal);
   printf("ALTITUDE(%04.03f),hAcc(%04.01f), vAcc(%04.01f),
Speed(%04.03f), Course(%04.02f)\r\n", LocalCoords.Altitude,
LocalCoords.HorizAccu, LocalCoords.VertiAccu, LocalCoords.Speed,
LocalCoords.Course );
   printf("HDOP(%04.03f),VDOP(%04.01f), TDOP(%04.01f), numSvs(%hhu)\r\n",
LocalCoords.HDOP, LocalCoords.VDOP, LocalCoords.TDOP,
LocalCoords.numSvs );
}
```

If the GPS has not computed a valid position, the *PosValid* field in the structure *tPOSITION_DATA* value is FALSE.

The value of this field is based on the conditions set for the valid fix with the function GPS_SetFixConfig(). This function takes two arguments, one mask with the status and the horizontal accuracy that must be met to give the fix as valid. See the API to get more information on these conditions.

```
int ReturnCode = 0;
   short int mask;
   unsigned int h_accu;
   char  strEntry[255];

   printf( "Fix Mask >> ");
   memset( ( void *) &strEntry, 0, sizeof( strEntry));
   getEntry( strEntry);
        mask = (short int)strtol( strEntry, NULL, 0);
   printf( "Horizontal Accuracy >> ");
   memset( ( void *) &strEntry, 0, sizeof( strEntry));
   getEntry( strEntry);
        h_accu = atoi( strEntry);

        if( (ReturnCode = GPS_SetFixConfig( mask, h_accu)) != NO_ERROR){
               RES_(  printf(  "Error  %d  in  GPS_SetFixConfig()...\n",
ReturnCode);)
   } else{
      RES_( printf( "GPS_SetFixConfig() OK\n");)
   }
```

12.5 Change Dynamic Platform Model

The Ublox GPS receiver supports different dynamic platform models to adjust the navigation engine to the expected environment. These platform settings can be changed dynamically without doing a power cycle or reset. Setting the GPS receiver to an unsuitable model for the application environment may reduce the receiver performance and position accuracy significantly.

Function to change Dynamic Model:

```
int  ReturnCode = 0;
   char DynamicModel;
   char strEntry[255];

   printf( "Dynamic Model (1-7) >> ");
   memset( ( void *) &strEntry, 0, sizeof( strEntry));
   getEntry( strEntry);
        DynamicModel = atoi( strEntry);
        if( (ReturnCode = GPS_SetDynamicModel(DynamicModel)) != NO_ERROR){
             RES_(  printf(  "Error  %d  in  GPS_SetDynamicModel()...\n",
ReturnCode);)
   } else {
      RES_( printf( "GPS_SetDynamicModel() OK\n");)
   }
```

Where 'DynamicModel' can be: 0=Portable(default value), 2=Stationary, 3=Pedestrian, 4=Automotive, 5=Sea, 6=Airbone<1g, 7=Airbone<2g, 8=Airbone<4g.

The default setting in the owa5X is Dynamic Model=0 (Portable).

- Portable: Applications with low acceleration, e.g. portable devices. Suitable for most situations.
- Stationary: Used in timing applications (antenna must be stationary) or other stationary applications. Velocity restricted to 0 m/s. Zero dynamics assumed.
- Pedestrian: Applications with low acceleration and speed, e.g. how a pedestrian would move. Low acceleration assumed.
- Automotive: Used for applications with equivalent dynamics to those of a passenger car. Low vertical acceleration assumed.
- Sea: Recommended for applications at sea, with zero vertical velocity. Zero vertical velocity assumed. Sea level assumed.
- Airbone ←1g: Used for applications with a higher dynamic range and vertical acceleration than a passenger car. No 2D position fixes supported.
- Airbone ←2g: Recommended for typical airborne environment. No 2D position fixes supported.
- Airbone ←4g: Only recommended for extremely dynamic environments. No 2D position fixes supported.

12.6 Change Static Hold Mode

The Static Hold mode allows the navigation algorithms to decrease the noise in the position output when the velocity is below a configured threshold.

If the speed goes below the defined threshold, the position is kept constant. As soon as the speed rises above twice the value of this threshold, the position solution is released again.

Function to change the Static Hold Threshold:

```
int  ReturnCode = 0;
   unsigned char staticThres;
   char strEntry[255];

   printf( "Static Threshold >> ");
   memset( ( void *) &strEntry, 0, sizeof( strEntry));
   getEntry( strEntry);
   staticThres = atoi( strEntry);
   if( (ReturnCode = GPS_SetStaticThreshold(staticThres)) != NO_ERROR){
```

```
            RES_(  printf(  "Error  %d  in  GPS_SetStaticThreshold()...\n",
ReturnCode);)
   } else{
      RES_( printf( "GPS_SetStaticThreshold() OK\n");)
   }
```

Note of manufacturer: do not set the parameter of the Static Hold Mode too aggressive, as it may degrade the performance of the GPS receiver, when e.g. a vehicle starts moving after a longer stop. A threshold in a range of 0.25 to 0.5 m/s will suit most of the requirements.

The default setting in the owa5X is threshold set to 0m/s.

12.7 Navigation output

The Ublox outputs the navigation data in Geodetic (latitude, Longitude and Altitude) or ECEF coordinate frame.

12.7.1 Get ECEF coordinates

With the Ublox GPS receiver the position ECEF coordinates can be obtained using the following code:

```
int ReturnCode = 0;

   ReturnCode = GPS_GetECEF_Coordinates(&ECEFCoord);
   if( ReturnCode != NO_ERROR )
            RES_(  printf(  "Error  %d  in  GPS_GetECEF_Coordinates()...\n",
ReturnCode);)
   else {
                                                RES_(       printf(       "\
n-------------------------------------------------------\n");)
            printf("\nOWASYS TEST ---> ECEFCoord.Px=%d\n", ECEFCoord.Px);
            printf("OWASYS TEST ---> ECEFCoord.Py=%d\n", ECEFCoord.Py);
            printf("OWASYS TEST ---> ECEFCoord.Pz=%d\n", ECEFCoord.Pz);
            printf("OWASYS TEST ---> ECEFCoord.Vx=%d\n", ECEFCoord.Vx);
            printf("OWASYS TEST ---> ECEFCoord.Vy=%d\n", ECEFCoord.Vy);
            printf("OWASYS TEST ---> ECEFCoord.Vz=%d\n", ECEFCoord.Vz);
      RES_( printf( "-------------------------------------------------------
---\n");)
   }
```

12.7.2 Get Geodetic coordinates

With the Ublox GPS receiver the position Geodetic coordinates can be obtained using the following code:

```
int ReturnCode = 0;

   ReturnCode = GPS_GetGeodetic_Coordinates(&GeoCoord);
   if( ReturnCode != NO_ERROR )
          RES_( printf( "Error %d in GPS_GetGeodetic_Coordinates()...\n",
ReturnCode);)
   else {
               RES_( printf( "\
n-------------------------------------------------------\n");)
```

```
      RES_( printf ("LATITUDE  --> %02hu degrees %02hhu minutes %04.04f
seconds %c\n",

       GeoCoord.Latitude.Degrees, GeoCoord.Latitude.Minutes,
                                                GeoCoord.Latitude.Seconds,
GeoCoord.Latitude.Dir); )
                RES_( printf ("LONGITUDE --> %03hu degrees %02hhu minutes
%04.04f seconds %c\n",

       GeoCoord.Longitude.Degrees, GeoCoord.Longitude.Minutes,

       GeoCoord.Longitude.Seconds, GeoCoord.Longitude.Dir); )
                RES_( printf ("ALTITUDE --> %04.04f meters\n",
GeoCoord.Altitude); )
                RES_( printf ("NAV STATUS --> %s\n",
GeoCoord.NavStatus); )


RES_( printf( "-------------------------------------------------------\
n");)
   }
```

12.7.3 Shared memory information

Once the GPS2 library is started with GPS_Start() function the position information is written every second to shared memory, so it can be retrieved by any program, without having to use any other library function.

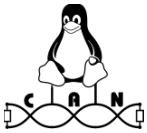
These are the values written to id 10300 of shared memory:

```
[001] -> posValid        00 →  0: no valid 1: valid
[002] -> latitude        43.145690 → latitude
[003] -> longitude       -2.962736 → longitude
[004] -> speed   0.000000 → speed in Km/h
[005] -> course  0.000000 → course over ground
[006] -> altitude        167.952000 → altitude in meters
[007] -> numsatel         00 → number of satellites used to calculate the
position
[008] -> navstatus         00 → 0: NF (No Fix) 1: DR (Dead reckoning only
solution) 2: G2 (Stand alone 2D solution) 3: G3 (Stand alone 3D solution)
4: D2 (Differential 2D solution) 5: D3 (Differential 3D solution) 6: RK
(Combined GPS + dead reckoning)
[009] -> HAccuracy       11597.000000 → Horizontal accuracy
[010] -> VAccuracy       8200.000000 → Vertical accuracy
[011] -> HDilution       99.990000 → Horizontal dilution of precision
[012] -> VDilution       99.990000 → Vertical dilution of precision
[013] -> TDilution       99.990000 → Time dilution of precision
[014] -> positionTS      00000001585237727412 → position timestamp
```

To see the code in C or python to retrieve the information of this shared memory the application note owasys_an12 can be checked, available in the developers zone.

13 owa5X CAN

13.1 Introduction



The CAN bus is accessed as a socket using the SocketCAN implementation on the Linux kernel, see 67 for more information on SocketCAN.

The CAN is accessed as a network interface with name canx. By default the owa5X has 2 CAN interfaces, canfd1 and canfd2, and optionally it can have another 2, canfd3 and canfd4.

The device may be configured and set up on the system for its use in the application. The tool to show and configure the can0 network interface is ip. So for example to configure the CAN bus with a speed of 1 Mbaud the following ip command must be executed in the system.

```
#ip link set can1 type can bitrate 1000000
```

Finally the interface must be set up with the tool ifconfig.

```
#ip link set can1 up
```

To bring down the interface, in order to modify its configuration or to end up with its use:

```
#ip link set can1 down
```

The configuration can also be set with systemd-networkd canfd1.network

```
[Match]
Name=canfd1

[CAN]
BitRate=500K
#FDMode=yes
#DataBitRate=2M
#RestartSec=100ms
#ListenOnly=yes
```

The CAN driver must be then turned on, because at boot time it is not powered in oder to improve the overall consumption of the system. The function DIGIO_Enable_Can() from the IO library must be called in order to do this.

```
wValue = 1;

if (( (ReturnCode = DIGIO_Enable_Can( wValue)) != NO_ERROR )
{
```

```
        printf("ERROR(%d) %s CAN\n", ReturnCode, wValue ? "ENABLE" :
"DISABLE");
}
```

Optionally it can also be switched on with this shell command:

```
# owasys-can 1
```

13.2 Creating a socket

In order to create a socket for communication, the function socket() is used. This function creates a socket and returns a file descriptor which will be used for future references to that socket. In this case, the device file is opened for reading, i.e. receiving messages.

The protocol family used is PF_CAN and CAN_RAW for direct CAN communication.

```
if ((fd = socket(PF_CAN, SOCK_RAW, CAN_RAW)) < 0) {
        perror("socket");
        return 1;
}
```

13.3 Configuring CAN

Before binding the CAN address to the file descriptor the CAN address family AF_CAN is specified and the interface index of the CAN controller is passed to the file descriptor.

```
addr.can_family = AF_CAN;

strcpy(ifr.ifr_name, argv[2]);
if (ioctl(fd, SIOCGIFINDEX, &ifr) < 0) {
        perror("SIOCGIFINDEX");
        return 1;
}
addr.can_ifindex = ifr.ifr_ifindex;

if (bind(fd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        perror("bind");
        return 1;
}
```

13.4 Receiving CAN frames

To receive CAN frames the standard read() function can be used. The data must be received within a can_frame structure.

```
struct can_frame rmessage;

// Packet for receiving
rmessage.can_id = 0;
rmessage.can_dlc = 0;
rmessage.data[0] = 0;
rmessage.data[1] = 0;

// Receive packet
read(fd, &rmessage, sizeof(rmessage));
```

13.5 Bulk read

For CAN interfaces CANFD3 and CANFD4 The CAN messages can be read one by one (default mode), every 16 messages or every 32 messages received in the transceiver. These interruptions can be configured changing a variable in the file */etc/modprobe.d/mcp251xfd.conf*

There are 3 possible options:

mcp251xfd fifo_rx_size=[value]

1: Interrupt when any message is received (default)

16: Interrupt when 16 messages are received

32: Interrupt when 32 messages are received

If the bus load is too high and the timing at which the messages are processed is not critical setting this variable to 16 can decrease the CPU load and improve the overall behavior of the system.

13.6 Writing CAN frames

To send messages through the CAN interface the write() function can be used.

```
struct can_frame frame;

frame.can_id = 0x0;
frame.can_dlc = 4;
frame.data[0] = 0x82;
frame.data[1] = 0x00;
frame.data[2] = 0x01;
frame.data[3] = 0x02;

// Write packet
write(fd, &frame, sizeof(frame));
```

13.7 Closing a CAN channel

Once the application has finished with a CAN channel, it should be closed by calling the function close() passing the file descriptor as argument.

```
if ( close(fd) )
{
   printf("Error closing channel\n");
   exit(-1);
}
```

13.8 can-utils

To test the functionality of the CAN interfaces from the command line, the can-utils tools are a good way of quick testing them.

First install the can-utils with apt, if they are not ready available on the default file system.

```
apt-get install can-utils
```

To test the transmission and reception of the interfaces, a quick test would be to connect CANH1 to CANH2 and CANL1 to CANL2 in the connector, and send a CAN frame from can0 interface to can1 interface.

Set candump to listen in can2 interface, redirecting the output to a file:

```
candump can2 > /tmp/can2.log
```

Send some frames from can1, which should be received in /tmp/can2.log:

```
cansend can1 1F334455#1122334455667788
cansend can1 1F334455#8877665544332211
```

Also it can be of interest to generate random CAN data with "cangen" tool, or to replay real traffic got in a car with "candump" tool, using canplayer. See

```
cansend can1 1F334455#1122334455667788
```

14 owa5X FMS API[2]

14.1 Introduction

FMS library libFMS_Module.so.0.0.1 can be used to get the information from the different FMS messages available in a CAN bus complying with FMS standard, without having to work directly with the CAN interface as shown in the earlier point of this document.

- STATUS

The FMS library makes internal calculations based in the continually received data from the CAN bus in order to offer a status of the vehicle: OFF, STOPPED, IDLING and MOVING.

- FMS

FMS, Fleet Management System, is an interface of data from commercial vehicles. The Owasys FMS library provides a structure where the data is gathered and relayed to the user program.

- TPMS

TPMS, Tire Pressure Monitoring System, covers the data advertised from the tire sensors of some trucks.

- EBS

EBS, Electronically controlled Brake System, gets information from the electronic brakes and their status.

Check the document D16_9904_FMS_Library.pdf to get more information on the API and all the data structures available to retrieve the data from the CAN bus using the Owasys FMS library.

14.2 Starting and finalizing FMS library

The usual procedure of loading the library and calling to the initialization and start functions must be followed before start using the FMS library.

FMS_Initialize() function takes three values to start the communication as expected for the bus where the owa5X is being connected. These values are the bitrate, the CAN interface on the owa5X that is being used and a flag, enable_log, that tells the library whether to log debug traces in /home/fms.log.

```
   #include FMS_Defs.h

   […]

   if( ( ReturnCode = FMS_Initialize(&FmsConf)) != NO_ERROR){
              printf( "OWASYS--> ERROR in FMS_Initialize( %d )...\n",
ReturnCode);
              exit(0);
       } else {
              printf( "OWASYS--> OK in FMS_Initialize\n");
   }
       if( ( ReturnCode = FMS_Start ()) != NO_ERROR) {
              FMS_Finalize( );
              printf( "OWASYS--> ERROR in FMS_Start( %d )...\n",
ReturnCode);
```

[2]This Owasys API library is **optional**, contact your distributor to know how to activate it.

```
                exit(0);
        } else {
                printf( "OWASYS--> OK in FMS_Start\n");
        }

        […]

    // Ending FMS library
    FMS_Finalize( );
    printf( "OWASYS--> OK Ending the FMS Test Application\n");
```

14.3 Retrieving FMS data

The FMS data is received in a structure, which also contains different structures with each possible data that can be received from the FMS standard. The structure can be filled calling to a FMS library function, as in the following example.

```
int retVal;
fms_data_t ActualFMS;

memset( &ActualFMS, 0, sizeof(fms_data_t));
retVal = FMS_GetFMSData ( &ActualFMS, sizeof(fms_data_t));
```

See the API information on all the available data that can be obtained with this function.

14.4 Retrieving TPMS data

The TPMS data is also received in another structure, which also contains different structures with each possible data that can be received from the TPMS.

```
int retVal, x, y = 0;
tpms_data_t ActualTPMS;

memset( &ActualTPMS, 0, sizeof(tpms_data_t));
retVal = FMS_GetTPMSData ( &ActualTPMS, sizeof(tpms_data_t));
```

See the API to get information on all the available data that can be obtained with this function.

14.5 Remove TPMS data

The TPMS data should be removed from the structure when physically a tire or a group of tires are no longer in their former place, for example after changing the trailer.

```
int retVal = 0;
unsigned char axle = 1;
unsigned char tire_left = 7;
unsigned char tire_right = 9;

retVal = FMS_RemoveTPMSData ( axle, tire_left);
[...]
retVal = FMS_RemoveTPMSData ( axle, tire_right);
```

See the API or the document D16_9904_FMS_Library_P1B.pdf to get information on all the available data that can be obtained with this function.

14.6 Retrieving EBS data

The EBS data is received in a structure, which also contains different structures containing the data from the braking system of the truck. The structure can be filled calling to a FMS library function, as in the following example.

```
int retVal;
ebs_data_t ActualEBS;

memset( &ActualEBS, 0, sizeof(ebs_data_t));
retVal = FMS_GetEBSData ( &ActualEBS, sizeof(ebs_data_t));
```

See the API to get information on all the available data that can be obtained with this function.

15 owa5X RS232

15.1 Introduction

The serial interfaces must be programmed using the standard Linux functions.

There are two possible serial interfaces in the owa5X in the UART1 (ttymxc0) and the UART2 (ttymxc1). The UART1 can be used with HW flow control. This UART1 is the one set for debugging purposes, to log in the system or to enter into the u-boot prompt.

15.2 Working with RS232

As there are no Owasys API functions the programmer have to use the standard Linux functions to configure and work with the serial interfaces:

- open() to get the file descriptor of the serial port.
- tcgetattr() to get the present configuration of the port.
- tcsetattr() to set a new configuration to the port.

See the code of application note owasys_an3 to get further information about the use of these functions.

16 owa5X RS485

16.1 Introduction

The owa5X can mount a RS485 serial interface available at the two pins where normally go the UART2 (ttymxc1).

16.2 Working with RS485

To configure the UART2 as 485, get the TIOCGRS485 configuration and change the needed flags as shown in the example.

```
  FD_485 = open( UART_RS485, O_RDWR | O_NOCTTY | O_NONBLOCK );
   if( FD_485 < 0) {
      printf("open %s error %s\n", UART_RS485, strerror(errno));
      return -1;
   }

   ReturnCode = tcgetattr(FD_485, &oldtio);
   if( ReturnCode < 0 ) {
      printf("tcgetattr error %s\n", strerror(errno));
      close(FD_485);
      FD_485 = -1;
      return -1;
   }

   newtio = oldtio;
   newtio.c_cflag |= B9600 | CS8 | CLOCAL | CREAD | CRTSCTS;
   newtio.c_iflag = IGNPAR;
   newtio.c_oflag = 0;    // no output modes
   newtio.c_lflag = 0;    // no canonical, no echo, ...
   newtio.c_cc[VMIN] = 0;
   newtio.c_cc[VTIME] = 0;

        ReturnCode = cfsetospeed(&newtio, B9600);
        if (ReturnCode < 0) {
      printf("Could not set output speed!\n");
      close(FD_485);
      FD_485 = -1;
               return -1;
        }
        ReturnCode = cfsetispeed(&newtio, B9600);
        if (ReturnCode < 0) {
      printf("Could not set input speed!\n");
      close(FD_485);
      FD_485 = -1;
               return -1;
        }
   tcflush  ( FD_485, TCIOFLUSH);
   tcsetattr( FD_485, TCSANOW, &newtio);
if (ioctl (FD_485, TIOCGRS485, &rs_old485conf) < 0) {
      printf("Error getting 485 configuratiosn\n");
        }
        rs485conf = rs_old485conf;
        // Enable RS485 mode:
        rs485conf.flags |= SER_RS485_ENABLED;
//      rs485conf.flags &= ~(SER_RS485_ENABLED);

        // Set logical level for RTS pin equal to 1 when sending:
        //rs485conf.flags |= SER_RS485_RTS_ON_SEND;
        // or, set logical level for RTS pin equal to 0 when sending:
        rs485conf.flags &= ~(SER_RS485_RTS_ON_SEND);

        //* Set logical level for RTS pin equal to 1 after sending:
        rs485conf.flags |= SER_RS485_RTS_AFTER_SEND;
        //* or, set logical level for RTS pin equal to 0 after sending:
```

```
        //rs485conf.flags &= ~(SER_RS485_RTS_AFTER_SEND);

        if (ioctl (FD_485, TIOCSRS485, &rs485conf) < 0) {
      printf("Error setting 485 configuratiosn\n");
        }
   printf(" %s configured succesfully.\n", UART_RS485);
   return 0;
}
```

16.3 OPTIONAL STEP

There are also some parameters that can be tweaked, like the time to wait before and after transmitting a message in the bus, that can be of interest in some cases. By default both timings are set to 1 ms.

```
  /* Set rts delay ms before send, if needed: */
  rs485conf.delay_rts_before_send = 1;
  /* Set rts delay ms after send, if needed: */
  rs485conf.delay_rts_after_send = 1;

  if ( (retVal = ioctl (FD_485, TIOCSRS485, &rs485conf)) < 0) {
     printf("Error setting RS485, retVal(%d), errno(%d)(%s)\n", retVal,
errno, strerror(errno));
  return -1;
  }
```

17 owa5X BLUETOOTH[3]

17.1 Introduction

The bluetooth module integrated in the owa5X model is a v5 low power consumption module that makes possible the bluetooth communication in a piconet. Its class 1 with 9 dBm output allows a communication range of up to 100 meters in an industrial environment (with line of sight). For BLE usage the output power is 8 dBm.

17.2 BT stack

The BT software stack is based in the BlueZ protocol stack which offers to the

programmer a modular implementation with a complete abstraction of the HW level.

To turn on the Bluetooth module use the enable IO library function.

ReturnCode = DIGIO_Enable_Bluetooth (1);

A shell command to switch BT on can also be used for testing and debugging purposes.

Use the command with argument 1 to switch the BT on.

```
# owasys-bt-wifi 1
```

Use the same command with argument 0 to switch the BT off.

```
# owasys-bt-wifi 0
```

[3] Only for owa44-BT with Bluetooth option

18 owa5X WiFi

18.1 Introduction

The WiFi peripheral that the owa5X family integrates is handled using the standard methods in Linux.

When the device boots up the WiFi is off by default. To use the WiFi device it must be switched on first with the IO function *DIGIO_Enable_Wifi()*, and after its use it can be switched off with the same function.

After switching on the WiFi peripheral the user can make use of the following tools to manage the wireless communication:

- Linux Wireless: The Linux Wireless tools provide a set of console commands that can be used from the host platform in order to access a Wi-Fi device through the Wireless Extension API from the command line. Most of the configuration of the wireless device can be done with the provided iw command.
- WPA Supplicant: The WPA supplicant is an open source Linux application that is used in Wi-Fi client stations to implement key negotiation with a WPA Authenticator, and it may control the roaming and IEEE 802.11 authentication/association of the WiFi driver.

18.2 WiFi switch on and off

Before the use of the WiFi peripheral it is required to switch it on with the enable IO function.

```
ReturnCode = DIGIO_Enable_Wifi(1);
```

After working with WiFi if the user does not need it any longer it can be switched off in order to decrease the consumption of the owa5X.

```
ReturnCode = DIGIO_Enable_Wifi(0);
```

A shell command can also be used to enable or disable WiFi from the command line, which can be of help for testing and debugging purposes. To switch the WiFi on call the command with argument set to 1.

```
owasys-bt-wifi 1
```

To switch off the WiFi, use the same command with argument set to 0.

```
owasys-bt-wifi 0
```

18.2.1 Automatic WiFi start

In order to have WiFi started automatically when the system boots up, a systemd service can be configured in the unit. For example the following service waits until pmsrv service is running and then after 5 seconds the WiFi module is switched on.

/etc/systemd/system/wifistart.service:

```
[Unit]
After=pmsrv.service

[Service]
Type=oneshot
ExecStartPre=/bin/sleep 5
ExecStart=/usr/bin/owasys-bt-wifi 1
ExecStartPost=/bin/sleep 5

[Install]
WantedBy=multi-user.target
```

To enable this service:

```
root@arm:~# systemctl enable wifistart
```

To disable this service:

```
root@arm:~# systemctl disable wifistart
```

19 owa5X Power Optimization

owa5X Power consumption might be reduced by means of two different strategies:

- by reducing microprocessor consumption, in this case customer application should access the power management functions and set one of the possible low consumption modes.
- by switching off the different devices, i.e. GSM, GPS, etc.

See the Integrators Manual for power consumption in various modes.

19.1 Reducing consumption by switching off devices

The owa5X incorporates a power management unit with various outputs for the different functional blocks. These blocks can be shut down under software control to enable the various power modes.

The following diagram shows the power management architecture:

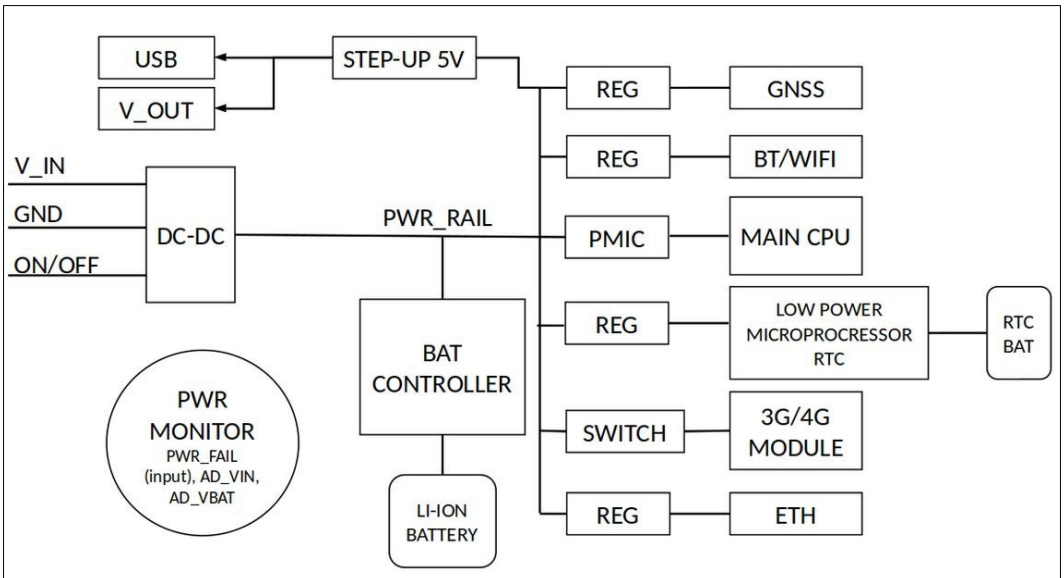


Figure 1: Owa5 Block Diagram

19.1.1 Main CPU

This block includes the CPU and memory and is powered OFF in both Standby and Off modes.

19.1.2 GSM Module

The GSM module is switched on when application calls GSM_Start() function. On the other hand it is switched off when the application calls GSM_Finalize(). Before calling GSM_Finalize, remember to finish the GPRS session in case it is established.

```
int GSM_Initialize(void* wConfiguration)
int GSM_Start(void) -> SWITCHES ON THE GSM MOODULE
```

```
int GSM_Finalize(void)-> SWITCHES OFF THE GSM MODULE
```

19.1.3 GPS Module

The GPS module can be switched off by calling its API function GPS_Finalize().

```
int GPS_Initialize(void* wConfiguration)
int GPS_Start(void) -> SWITCHES ON THE GPS MOODULE
int GPS_Finalize(void)-> SWITCHES OFF THE GPS MODULE
```

19.1.4 CAN

The CAN driver can be switched off and on using a IO library function. It is suggested to switch it off if not in use.

```
int DIGIO_Enable_Can (char wValue)
```

From the shell it can be switched off with this command:

```
owasys-can 0
```

19.1.5 WiFi / Bluetooth

The WiFi and Bluetooth module can be switched off and on using a IO library function. It is suggested to switch it off if not in use.

```
int DIGIO_Enable_Wifi (unsigned char wValue)
```

19.1.6 USB

The USB can be switched off and on writing on a file in the filesystem.

```
echo 0 > /sys/kernel/debug/musb-hdrc.0/softconnect
echo 1 > /sys/kernel/debug/musb-hdrc.0/softconnect
```

19.2 How to get the lowest consumption in low power mode

To decrease the owa5X consumption to the minimum customer application must follow these steps:

- If GSM module is switched on then switch it off
- If GPS module is switched on then switch it off
- If the CAN is switched on then switch it off
- If there is any optional feature like BT or WiFi, switch them off
- Switch off leds with DIGIO_Set_LED_SWx(0) function.
- Set the unit into Standby or Off mode. The main differences between these 2 modes:

➔ Standby mode:

Consumption TBD.

GSM can be also set as signal to wake up.

The execution continues instantly with the next instruction.

```
int RTUEnterStandby(unsigned long wMainWakeup, unsigned long wExpWakeup)
```

➔ Off mode:

Consumption TBD.

The complete system reboots after waking up, so that the user application will take about 40 seconds to execute from the beginning.

```
int RTUEnterStop(unsigned long wMainWakeup, unsigned long wExpWakeup)
```

By default the IO are kept at the same state when the unit goes to Off mode, the user application should deactivate the outputs that will no longer be used during the time that the device is in Off mode.

In Off mode and for the rest of peripheriques, these should be switched off by the user program in a controlled way, but in any case this is their state when entering Stop mode:

- Battery charge off
- GPS off
- SD off
- 6 axis off
- USB on, unless RTU_REMOVE_VOUT signal is set in the mask when entering into Off mode.
- Vout on, unless RTU_REMOVE_VOUT signal is set in the mask when entering into Off mode.

19.3 Monitoring Power Sources

It is recommended to monitor the Vin external power source, and go to Off mode if its level decreases too much.

In case of installing an optional battery, it is important to monitor also its level and go to Off state if it goes as low as 3.5V. The unit will start working with the optional battery as soon as the external voltage is under 9V.

See functions RTUGetAD_V_IN() and RTUGetAD_VBAT_MAIN() to get more information on how to get these values.

20 System Reinitialization

If a problem occurs where parts of the platform stop responding or are functioning incorrectly, then each part can be reinitialized as follows.

20.1 GSM Module

Close and initialize the GSM module as follows:

```
GSM_Finalize();

//Initialize GSM

if((ReturnCode = GSM_Initialize((void*)&Configuration))!= NO_ERROR)
{
   printf( "Error %d in GSM_Initialize()...\n", ReturnCode);
}

if( ( ReturnCode = GSM_Start()) != NO_ERROR )
{
   printf( "Error %d in GSM_Start()...\n", ReturnCode);
}
```

If this does not resolve the problem then perform a system boot as described below.

20.2 GPS Module

Stop and start GPS module as follows:

```
GPS_Finalize();

//Initialize GPS

if((ReturnCode = GPS_Initialize((void*)&Configuration))!= NO_ERROR)
{
   printf( "Error %d in GPS_Initialize()...\n", ReturnCode);
}

if( ( ReturnCode = GPS_Start()) != NO_ERROR )
{
   printf( "Error %d in GPS_Start()...\n", ReturnCode);
}
```

If this does not solve the problem then perform a system boot as described below.

20.3 System Boot

This will completely restart the whole system and is similar to removing power. During system boot the GSM and GPS modules are also reinitialized.

It is important to shut down properly all user system services and the HW modules before restarting the system.

21 Watchdog

21.1 Support for hardware and software watchdogs

There are two levels of watchdog functionality. By using *systemd* , the owa5X provides full support for hardware watchdogs (as exposed in /dev/watchdog to userspace), as well as software watchdog support for individual system services.

21.2 Hardware watchdog

Leveraging the CPU hardware watchdog exposed at /dev/watchdog, if enabled, systemd will regularly ping the watchdog hardware. If systemd or the kernel stops, the watchdog will generate a hardware reset.

This type of watchdog can be enabled by setting the following options (they default to 0, i.e. no hardware watchdog use) in /etc/systemd/system.conf:

```
RuntimeWatchdogSec  = X       // Set it to a value like 20s and the
                              // watchdog is enabled. After 20s of no
                              // keep-alive pings the hardware will reset
                              // itself.

ShutdownWatchdogSec = Y       // It sets a timer to force reboot if
                              //  shutdown hangs, adding extra
                              // reliability to the system reboot logic
```

The watchdog behavior can be checked by executing some of the following commands:

```
strace -p1 -s 500 -tt -eioctl -v
tail -f /var/log/syslog | grep watchdog
```

21.3 Software watchdog

systemd exposes a watchdog interface for individual services so that they can also be restarted if they hang. This software watchdog logic can be configured for each service in the ping frequency and the actions to take.

To enable the software watchdog logic for a service, it is sufficient to set the *Type* option to ***notify*** in the unit file.

```
# location: /etc/systemd/system/
[Unit]
…
[Service]
…
# In case if it gets stopped, restart it immediately
Restart     = always

# With notify Type, service manager will be notified
# when the starting up has finished
Type        = notify

# Since Type is notify, notify only service updates
# sent from the main process of the service
NotifyAccess= all
```

NotifyAccess option can be "none", "main", or "all:

- *none* (default): ignores all messages.
- *main*: *systemd* will listen to notifications only from the main process.

- *all*: *systemd* will listen to notifications from forked processes.

To configure the timeout of an application wherein a watchdog is to be loaded, it is necessary to set the ***WatchdogSec*** option in the systemd unit definition. Moreover, certain options in the unit definition allow the user to configure whether the service shall be restarted and how often, and what to do if it then still fails:

- To enable automatic service restarts on failure set *Restart=on-failure* for the service.
- To configure how many times a service shall be attempted to be restarted use the combination of *StartLimitBurst* and *StartLimitInterval* which allow you to configure how often a service may restart within a time interval. If that limit is reached, a special action can be taken, which is configured with *StartLimitAction:*
  - *none* (default): the service simply remains in the failure state without any further attempted restarts.
  - *reboot:* reboot attempts a clean reboot.
  - *reboot-force:* it will not actually try to cleanly shutdown any services, but immediately kills all remaining services and unmounts all file systems and then forcibly reboots.
  - *reboot-immediate*: it does not attempt to kill any process or unmount any file systems. Instead it just hard reboots the machine without delay (like a reboot triggered by a hardware watchdog).

Below, an example unit file will automatically be restarted if it has not pinged systemd for longer than 30s or if it fails otherwise. If it is restarted this way more often than 4 times in 5min action is taken and the system quickly rebooted, with all file systems being clean when it comes up again.

```
[Unit]
Description=My owa5X daemon

[Service]
ExecStart=/usr/bin/myowa5Xd

WatchdogSec=30s               // The desired failure latency: it will
                              // cause the service to enter a failure
                              // state as soon as no keep-alive ping is
                              // received within the configured interval.

Restart=on-failure            // Application reset

// RestartSec=                // Configures the time to sleep before
                              // restarting a service (as configured with
                              // Restart=). Takes a unit-less value in
                              // seconds, or a time span value such as
                              // "5min 20s". Defaults to 100ms.

StartLimitInterval=5min
StartLimitBurst=4
StartLimitAction=reboot-force // system reset as a fallback measure in
                              // response to multiple unsuccessful
                              // application resets
```

21.3.1 Software implementation overview

This section uses C program examples to illustrate how to add watchdog logic to individual services. However, the so-called *systemd notification protocol (sd_notify)* is also implemented in other programming languages (e.g., *sdnotify* implementation in Python).

The watchdog functionality is accessible for the user application by using *libsystemd*, which, after including the header file of the library, can be dynamically linked at compilation time using the -lsystemd option. Legacy code can be patched to support the systemd watchdog logic pointed out above as shown below.

```
#include <systemd/sd-daemon.h>
```

Before performing any actions over the watchdog mechanism, the user must get the watchdog timer by reading the WATCHDOG_USEC environment variable without error.

```
// watchdog initialization
char *wdtTimer = getenv("WATCHDOG_USEC");
   if(!wdtTimer) {
      /* No WATCHDOG_USEC set */
      …
   }

// It is possible to reset WATCHDOG_USEC value during runtime
// through the following code
sd_notify(0, "WATCHDOG_USEC=20000000")
```

Then, the watchdog must be initialized, and the service should call *sd_notify* regularly (e.g., every half of the interval) with "WATCHDOG=1". If the watchdog is not restarted before WATCHDOG_USEC secs then the service resets.

```
// watchdog initialization

   if (sd_notify(0, "READY=1")<0){
      printf("Systemd WD startup error");
   }
…

// watchdog restarting
   if (sd_notify(0, "WATCHDOG=1")<0) {
      printf("Error notifying watchdog service");
   }
```

Option "STOPPING=1" allows the service to tell the service manager that it is beginning its shutdown.

```
// Stopping the service
   if (sd_notify(0, "STOPPING=1")<0){
      printf("Error stopping watchdog service");
   }
```

More information about *systemd* watchdogs can be found in 67

21.3.2 Reboot

It follows an example in C code based on the reboot() system call:

```
…
#include <linux/reboot.h>
#include <sys/reboot.h>
#include <signal.h>
```

```
#endif
…
   sync();
   if(reboot(LINUX_REBOOT_CMD_RESTART) != 0) {
         /* Reboot failed */
         …
   }
```

22 HW variables

It is important to know the type of HW on which the program is running, as there are different variants of products and sometimes it is needed to have the custom applications ready for each variant that is going to be used on field. For this purpose a file is generated during the manufacturing or the units that describes all the HW features available on the device, and it is saved in */etc/default/hw.json*

Flashing all partitions with the uuu script will remove the hw.json file in /etc/default folder, either do not delete the NAND.u-boot-env and NAND.u-boot-env.backup partitions where a copy of hw.json is kept with the variable hwfile or save a copy of this file in your PC previously, so that it can be recovered afterwards.

22.1 /etc/default/hw.json

This file consists of a list of HW features as json variables, with their corresponding values, that can be a string, a boolean or an integer.

It follows an example of this file.

```
{
    "manufacturer": "Owasys",
    "model": "OWA547W",
    "revision": "P1A",
    "serial_number": "__SERIAL_NUMBER__",
    "6_axis": true,
    "accelerometer": true,
    "ethernet": true,
    "ibutton": true,
    "micro_sd": true,
    "tpm": true,
    "wifi_bt": true,
    "gnss": {
        "model": "M8J"
    },
    "memory": {
        "flash": {
            "nand": "1GB",
            "emmc": "8GB"
        },
        "ram": "2GB"
    },
    "cellular": {
        "model": "EG25G",
        "2g": true,
        "3g": true,
        "4g": true,
        "mff2": true
    },
    "can": {
        "1": true,
        "2": true,
        "3": true,
        "4": true
    },
    "ain": {
        "0": true,
        "1": true,
        "2": true,
        "3": true
    },
```

```
    "dio": {
        "0": true,
        "1": true,
        "2": false,
        "3": false,
        "4": true,
        "5": true,
        "6": true,
        "7": true,
        "8": true,
        "9": true,
        "10": true,
        "11": true
    },
    "usb": {
        "1": true
    },
    "rs485": {
        "1": false
    },
    "rs232": {
        "1": true,
        "2": true
    }
}
```

23 PMSRV SERVICE

In the system this service must be always running, as it is in charge of the communication between the system and the auxiliary processor in order to perform a series of tasks:
- Set the unit in a Low Power Mode
- Get or set digital IOs
- Set the RTC time
- Control the default accelerometer

This service uses the configuration file in */etc/pmsrv.json* to control its behavior regarding certain aspects. It follows an example of this configuration file:

```
{
   "name": "pmsrv",
   "reset_lines": 20,
   "set_hostname": 1,
   "set_default": 1,
   "set_time": 0,
   "secure_time": 1,
   "temperature": {
      "temp_enable": true,
      "temp_halt": 85,
      "temp_wake": 75,
      "off_timeout": 15
    }
}
```

Description of the variables:

"name": Name of the service
"reset_lines": Number of lines to log with reset times of this service
"set_hostname": 1 → Set the serial number as hostname at every boot
0 → Do not set the the hostname
"set_default": 1→ Set default values in IOs
0 → Do not set default values in IOs
"set_time": 1 → Set the RTC time as system time when the system boots up
0 → Do not set the RTC time as system time when the system boots up
"secure_time": 1 → The pmsrv only restores the RTC clock if the date of the system is older than the one in the RTC
0 → The system time and date are set from the RTC without any checking.
"temp_enable": true → Board temperature is monitored every minute, logged to syslog and sent to dbus (dbus-monitor –system "member=temperature")
false → temperature is not monitored
"temp_halt": 60..95. If temperature is detected over this value three times in a row, halt signal is sent via dbus and device will go to OFF mode.
"temp_wake": 55..90. Once the temperature is below this value the system will wake up with wakeup reason "WKUP_TEMPERATURE"
"off_timeout": 10..600. This is the time in seconds between halt signal and remove of CPU power, in order to give enough time to user services and programs to shut down gracefully.

24 Useful tips

Please check the tutorials in our developers zone, there the user can find solutions to different problems that normally arise when working with the owa5X device.

https://developers.owasys.com/tutorial-list

24.1 Logs

Different logs are generated under /var/log directory, and it is important to control the size that these logs will take in the NAND flash, which could fill up the whole memory if they are not controlled in some way. This control can be maintained with these tools:
- logrotate
- Variables SystemMaxFileSize, SystemMaxFiles in /etc/systemd/journald.conf

24.2 System hardening

Before sending the unit into production there are some steps that should be taken in order to harden the system and securize it as much as possible to external intrusions.

Here it follows a checklist with some points to take into account:

- [ ] Set u-boot env **boot_sequence=0** to avoid executing `owaX-boot.script` from uSD or USB

- [ ] Set u-boot env **ledsbootstatus=0** to avoid LEDs switching on (optional)

- [ ] Change root password

- [ ] Remove or change password for Debian user

- [ ] Check date and time configuration. Is your system time restored after reboot? Is owasysd-timesync enabled? Is the time "UTC" (good practice) or local (bad practice)?

- [ ] Disable password login for SSH

- [ ] Disable root access for SSH

- [ ] Ensure U-boot is disabled or password protected

- [ ] Disable serial console, or configure it with a long password and long time between retries

- [ ] Disable root login directly

- [ ] Do not allow login as Debian or Root

- [ ] Disable Ethernet if not used

- [ ] Disable USB if not used

25 REFERENCES

Ref.	Doc. Number	Description
[1]	BOK_000_5002	owa5X Family **Integrator Manual**
[2]	BOK_000_5009	owa5X Family **API**
[3]	can.txt	http://www.kernel.org/doc/Documentation/networking/can.txt
[4]	BlueZ	http://bluez.sourceforge.net/
[5]	Linux Wireless	https://wireless.wiki.kernel.org/en/users/Documentation
[6]	WPA supplicant	https://w1.fi/wpa_supplicant/
[7]	Systemd	https://www.freedesktop.org/wiki/Software/systemd/
[8]	Watchdog	http://0pointer.net/blog/projects/watchdog.html
[9]	IIO driver	https://wiki.analog.com/software/linux/docs/iio/iio
[10]	Linux timing functionality	https://www.kernel.org/doc/Documentation/timers/hrtimers.txt
[11]	U-boot recovery tool	https://www.denx.de/wiki/DULG/UBootBootCountLimit
[12]	Jounal of systemd	http://0pointer.de/blog/projects/journalctl.html

26 History

Rev.	Changes
A	- First release
B	- WKUP_DIN9 removed as this is not valid for the owa5X platform 8 - GNSS COM port is COM4 35 - Ethernet signal added as wake up source 8 - Partition layout updated 9 - Cross compiler usage 11 - Warning added to check for hw.json before reflashing the system 62
C	- PMSRV service description included 64 - Changed command to enter uuu flashing mode 9 - Updated functionality during iNet sessions, no audio calls are possible 34 - Internal temperature communicated also via DBUS 17
D	- secure_time parameter of pmsrv.json explained 23 - Bluetooth IO function described 17
E	- Power Management CAN interface description 8 - Netbeans replaced for VSCode as recommended IDE 7 - Updated partition layout table 9
F	- ledbootstatus described 22 - Removed GPS_SetGpsMode() function description, this function is deprecated - Added Bulk Read option with canfd3 and canfd4 43 - WiFi standard commands removed - System hardening tips included 66 - 6 axis maximum sampling value corrected to 416 Hz 15 - Digital inputs DIN[9-11] can not interrupt 24 - Power management DIN[9-11] can not wake up 8
G	- Add systemd-network configuration example for canfd interface 41 - User debian replaced with owasys - U-boot env redundancy variables added