Product revision: 1.4
2021-03-12

CrossLink TG

Software and Programming Manual

crosscontrol

www.crosscontrol.com

Contents

Revision history

Rev	Date	Author	Comments
1.4	2021-03-12	Finn McGuirk	X

1. Introduction

This manual explains the purpose of each API and how to start using the available functions in the library of APIs to develop an application.

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

1.1. How to access functionality

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

Function	Include file	Load File	Type
RTU	<owa4x/RTUControlDefs.h>	/lib/libRTU_Module.so	Dynamic
IO	<owa4x/IOs_ModuleDefs.h>	/lib/libIOs_Module.so	Dynamic
GSM	<owa4x/GSM_ModuleDefs.h>	/lib/libGSM_Module.so	Dynamic
GPS	<owa4x/GPS2_ModuleDefs.h>	/lib/libGPS2_Module.so	Dynamic
iNET	<owa4x/INET_ModuleDefs.h>	/lib/libINET_Module.so	Dynamic

1.1.1. Accessing Dynamic Libraries

There are 2 ways to access the libraries from the user program code: using dlopen() and dlsym() or linking against them at compilation time.

1.1.1.1. dlopen() and dlsym()

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

First, an 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");
```

1.1.1.2. 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 unit patch in the cross compiler, the unit libraries can be linked at compilation time using the -l option.

```
PC$ arm-linux-gnueabihf-gcc -Wall -mthumb -mthumb-interwork -
D_REENTRANT -oowa4x_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().

2. RTUControl API

2.1. 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.

2.2. 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
...
```

2.3. RTC time

The unit is provided with a **Real Time Clock** (RTC) with calendar options and it is fully programmable using API functions. 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 unit can also retrieve the current **UTC time from the GPS module** through its API.

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 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 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 one retrieved with command “date”
- **hwclktosys**. Sets the system time based on the HW clock. This is done automatically every time the system boots up.

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

2.5. Power Management

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

The signals that can wake up the device include:

MOVEMENT: wakes up if the movement sensor detects any movement.

GSM: wakes up if there is any event data (RING, SMS or COVERAGE events for example) received from the GSM module.

DIN0..9: wakes up if the external DIN0..9 signal changes.

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.

CAN: Wakes up if the unit detect traffic in the CAN1 interface connected to a CAN bus. Any traffic in the bus will wake up the unit, not a specific message can be configured for it.

2.5.1. Standby Mode

This mode allows the user to wake up using any of the events that are able to wake up the device (MOVING, PWRFAIL, CONSOLE, GSM, CAN1, RTC, DIN0, DIN1, DIN2, DIN3, DIN4, DIN5, DIN6, DIN7, DIN8, DIN9).

This mode is the fastest power save mode for waking up. 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 owa4x/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_WKUP_DIN9     (1 << 16)
 #define WKUP_ALL (RTU_WKUP_MOVING | RTU_WKUP_PWRFAIL |
RTU_WKUP_CONSOLE\
| RTU_WKUP_GSM | 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_DIN9)
```

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))
        {
          printf("ERROR Going to Standby Mode\n");
        }
```

It is important to note, that when going to this mode with GSM in the mask, the calling process must be **the same** that the one that started GSM. When the unit is in STANDBY after setting GSM in the mask, any incoming event will wake up the system, for example an incoming call (RING event), SMS arrival, a change in the coverage, etc.
See the API to get an overview of all possible events with GSM.

The Wi-Fi must be turned OFF before entering the Standby mode.

2.5.2. Stop Mode

When the device goes into this state, the 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 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 except GSM signal RXD2, which is not powered in this mode: MOVING, PWRFAIL, CONSOLE, RTC, DIN0, DIN1, DIN2, DIN3, DIN4, DIN5, DIN6, DIN7, DIN8, DIN9.

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");
        }
```

The WiFi must be OFF before entering the Standby mode.

This function will internally call halt on the system, and so there is service called *onhalt* that can be used to execute when 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/debian/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
```

2.5.3. 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 owa4x/RTUControlDefs.h include file.

2.5.4. 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	NO (automatic shut down)	YES (only for Stby - RTU_WKUP_GSM)
GPS	NO (user must shut down)	NO (automatic shut down)	NO
Ethernet	YES	NO (automatic shut down)	NO
CAN	YES	NO (automatic shut down)	NO
RS485	YES	NO (automatic shut down)	NO
RS232	YES	YES	YES (RTU_WKUP_CONSOLE)
Wi-Fi/ Bluetooth	NO (user must shut down)	NO (user must shut down)	NO
Inputs	YES	YES	YES (RTU_WKUP_DIN[0..9])
Outputs	YES	NO (automatic shut down)	NO
uSD	YES	NO (automatic shut down)	NO

2.6. Hardware RTC

The 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 automatically update the system time when the unit reboots or when it comes from any of the power save modes. The system time will never update the RTC. It means that if the application sets the system time the RTC will not be updated.

2.6.1. 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`.

2.6.2. 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.

2.6.3. 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.

2.7. Default Movement Sensor

The 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.

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.

2.7.1. Configuring the Movement Sensor 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 or +/-8G, 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.

2.7.2. 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.

2.7.3. 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.

2.7.4. 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.

2.7.5. Getting the Acceleration Value

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

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 9 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.

2.8. Optional 9 axis Movement sensor

The optional 9 axis sensor offers not only movement acceleration readings, but also gyroscope and magnetometer readings, each with values in 3 axis: X, Y and Z. The readings can be configured at a maximum rate of 476 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 **Error! Bookmark not defined.**.

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
```

An 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.

2.9. 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 battery and the optional battery.

2.9.1. 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);
```

2.9.2. Main 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);
```

2.9.3. Internal temperature

The internal temperature can be obtained with this function.

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

2.10. Timers

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

2.10.1. 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.

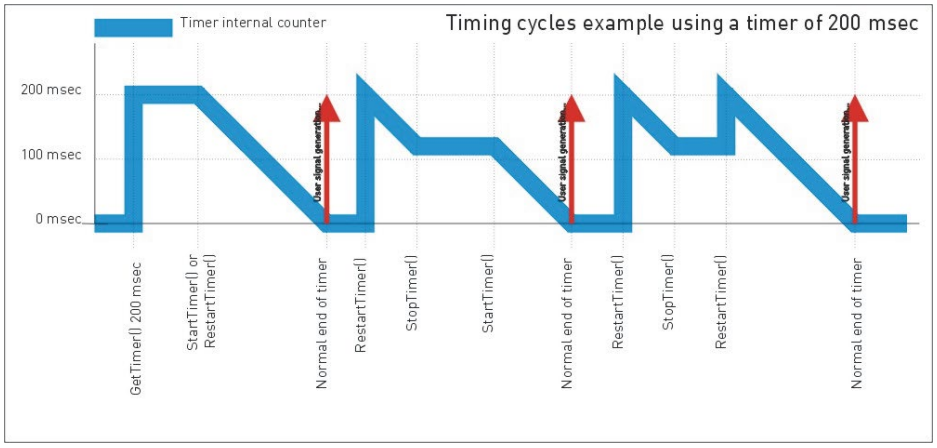
If MULTIPLE_TICK is chosen instead, every time the internal counter reaches 0 value, the handler is executed and the counter is automatically restarted.

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_GetTimer()	Stopped	200 msec	200 msec
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;
```

2.10.2. 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 **Error! Bookmark not defined.**. This unit's 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).

```
arm-linux-gnueabihf-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.

3. IO Module API

3.1. 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, USB, uSD, WiFi, Bluetooth.
- To see the whole set of IO functions included in ***IOsModule*** library, see [1].

3.2. LEDs control

3.2.1. 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 boots up.

3.2.1.1. 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);
```

3.2.2. Using the user space

The LEDs in the platform can be controlled from user space using /sys/class/leds/[1], ([1] https://www.kernel.org/doc/html/latest/leds/leds-class.html) 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
```

3.3. Handling analog inputs and digital inputs/outputs

3.3.1. 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;
  }
```

3.3.2. 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;
  }
```

3.3.3. Getting digital inputs value by interrupt

Any of the digital inputs can be configured to interrupt the program.

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.

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);
```

3.3.4. 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);
```

3.3.5. 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);
```

4. GSM API

4.1. 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.

4.2. 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.

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.

4.3. 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.

There is a sample Note (owa4x_AN24) , which runs the call functions (Dial and Answer) on a different thread, which allows running other GSM actions from this application. Below a more technical explanation is given.

4.4. Making Voice/Data Calls

It is recommended to make a thread where the dial or answer is executed. This enables the main thread to ask for other function to the GSM, such as, GSM_SignalStrength, GSM_SendSMS, etc.

In owa4x_AN24 sample note, commands introduced through the keypad executes the functions. The “help” command lists all of the possibilities. The new thread goal is to leave free the input line and ask for other GSM function.

Below GSM_Dial and GSM_Answer functions are shown on the owa4x_AN24 sample note. As it is observed both functions are executed in a thread. Once every function returns from its execution, the thread is terminated, but in the meantime it is possible to make another GSM function.

Dialing Function:

```
void gsmDial( )
{
  unsigned char     keyEntry[ 128];
  Dial_t    CurrentCall;
  pthread_t  ThDial;

  memset( &CurrentCall, 0, sizeof( Dial_t));
  printf("OWASYS--> Insert Number: ");
  getEntry( CurrentCall.destinationNumber);
  printf( "\n");
  printf("OWASYS--> Type 'd' to make a data call else any key: ");
  getEntry( keyEntry);
  CurrentCall.ActionDial = TRUE;
  CurrentCall.CallType = CALL_TYPE_VOICE;
  if( ( keyEntry [ 0] == 'd') || ( keyEntry [ 0] == 'D')){
    CurrentCall.CallType = CALL_TYPE_DATA;
  }
  pthread_create( &ThDial, NULL, DialingThread, &CurrentCall);
  pthread_detach( ThDial);
  printf( "\n");
}
```

Answering Function:

```
void gsmAnswerCall( )
{
  pthread_t  ThDial;
```

```
  AnswerCall.ActionDial = FALSE;
  pthread_create( &ThDial, NULL, DialingThread, &AnswerCall);
  pthread_detach( ThDial);

}
```

Dial/Answer Thread:

```
void *DialingThread( void *arg)
{
  Dial_t  *Call;
  int     retVal;

  Call = ( Dial_t *) arg;

  callEnd = TRUE;
  if( Call->ActionDial)
    retVal = ( *FncGSM_Dial) ( Call->CallType,(unsigned char *)Call-
>destinationNumber);
  else{
    retVal = ( *FncGSM_AnswerCall) ( );
    if( retVal == NO_ERROR){
      printf( "OWASYS--> OK  Call Answered\n");
      callEnd = FALSE;
    } else {
      printf( "OWASYS--> ERROR Answering Call ( %d )\n", retVal);
    }
    return NULL;
  }

  switch ( retVal){
    case NO_ERROR:
      printf( "OWASYS--> ----------- On Call ------------\n");
      callEnd = FALSE;
      break;
    case GSM_ERR_BUSY:
      printf( "OWASYS--> ----------- Busy ---------------\n");
      break;
    case GSM_ERR_NO_ANSWER:
      printf( "OWASYS--> ----------- No answer ----------\n");
      break;
    case GSM_ERR_NO_CARRIER:
      printf( "OWASYS--> -- Ended call (NO CARRIER) ------\n");
      break;
    case GSM_ERR_NO_DIALTONE:
      printf( "OWASYS--> -- Ended call (NO DIALTONE) -----\n");
      break;
    default:
      printf( "OWASYS--> ----------- Non defined error: %d-\n",retVal);
  }
  return NULL;
}
```

The thread is the process where the GSM library function is called, either GSM_Dial or GSM_AnswerCall.

It is very important the pthread_detach after the new process creation. The goal of this function is that once the thread ends, to save resources from the system the thread must be marked as deleted. This is done by two functions pthread_join, when main process is able to control threads, or pthread_detach, when the process ends asynchronously.

4.5. 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)

4.6. 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

4.7. Audio parameters configuration

There are several audio parameters that can be configured using the file audio.conf stored in /home.

Depending on the model and HW version these parameters will have some differences because of the installed GSM module is also different, so take care of reading the correct parameters according to the model and HW version.

An audio.conf file example is shown below, followed by a description of all the parameters.

```
[OWA_AUDIO_MODEL]4
[HF_TXGAIN]88
[HF_RXGAIN]88,88
[HF_VOLUME]60
[RING_VOLUME]3,2
```

4.7.1. Audio Parameters with HW < R4

An audio.conf file example is shown below, followed by a description of all the parameters.

```
[OWA_AUDIO_MODEL]4
[HF_TXGAIN]88
[HF_RXGAIN]88,88
[HF_VOLUME]60
[RING_VOLUME]3,2
```

[**AUDIO_MODEL**]

This parameter is set by the library to identify the GSM module in use. Do not change this value.

[**HF_TXGAIN**]

MIC sensitivity. It controls the microphone path amplification.

Parameter [0 - 100]: Microphone gain adjustment in steps. Each step is equal 0.5db starts from -43.5db to 60db (0=-96dB, 1=-43.5db... 99=5.5dB, 100=6.0dB).

Default value = 88

[**HF_RXGAIN**]

It controls the DAC amplification in the speakers signals.

Parameter 1 [0 - 100]: Speaker gain adjustment in steps. Each step is equal 0.5db, starting from -43.5db to 60db (0=-96dB, 1=-43.5db... 99=5.5dB, 100=6.0dB).

Parameter 2 [0 - 175]: Sidetone gain adjustment in steps. Each step is equal 0.5db, starting from -43.5db to 43.5db (0=-96dB, 1=-43.5db ... 174=43.0dB, 175=43.5B).

Default value = 88,88

[**HF_VOLUME]**

[0 - 100]

The volume can be adjusted from 0% to 100%. This value overwrites value programmed in HF_RXGAIN.

Default value = 60

[**RING_VOLUME**]

Volume melody and level for the ring signal.

Parameter 1 [0 - 8]: Type of ring tone.

- 0 = Mutes the played tone.
- 1 = Sequence 1

.....

- 8 = Sequence 6

Parameter 2 [0 - 4]: Volume of ring tone, varies from 0 dB to mute

- 0 = Mute
- 1 = Very low
- 2 = Low
- 3 = Middle
- 4 = High

Default value = 3,2

4.7.2. Audio parameters with HW >= R4

An audio.conf file example is shown below, followed by a description of all the parameters.

```
[OWA_AUDIO_MODEL]5
[HF_TXGAIN] 25905,16345
[HF_RXGAIN]25905,0
[HF_VOLUME]3
[RING_VOLUME]1
```

[**AUDIO_MODEL**]

This parameter is set by the library to identify the GSM module in use. Do not change this value.

[**HF_TXGAIN**]

MIC sensitivity. It controls the microphone path amplification.

Parameter 1 [0-65535]: Indicates uplink codec gain and the range is 0-65535. (NOT USED)

Parameter 2 [0-65535]: Indicates uplink digital gain and the range is 0-65535.

Default value = 25905,16345

[**HF_RXGAIN**]

It controls the DAC amplification in the speakers signals.

Parameter 1 [0-65535]: Indicates downlink digital gain.

Parameter 2 [0-65535]: Indicates the configured side tone gain.

Default value = 25905,0

[**HF_VOLUME**]

[0 - 5]

The volume can be adjusted from 0 to 5.

Default value = 3

[**RING_VOLUME**]

Volume melody and level for the ring signal.

Parameter 1 [0 - 2]: Type of ring tone.

0 = Disable ring tone.

1 = Enables Nokia ring tone.

2 = Enable ring tone.

Default value = 1

5. iNet API

5.1. 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.

5.2. Starting an Internet connection

To establish an Internet connection, RTU, IO and GSM must be first started, as explained in section 4.2.

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 owa4x_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.

5.3. 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 4.6).

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

5.4. 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.

5.5. 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 but GSM_Dial, which requires the end of the INET session, before doing any outgoing call.

Call arrives:

If an INET session is up and running and a call arrives, the INET session keeps on, taking into account following rules on a call arrival:

If a voice call arrives:

Hangup and go on with the INET session.

Answer and after hanging up the call (no matter which side releases the call) the INET session is continued.

If a data call arrives:

Hangup and go on with the INET session.

To answer data calls is needed to release INET session, previously.

Other events do not release the INET session, including a GPRS_COVERAGE lost. The INET session is internally maintained and until GSM module buffer is full some bytes could be sent to the GSM “network”.(Remember that the GSM terminal is part of the GSM network)

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.

6. Power Optimization

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.

6.1. Reducing consumption by switching off devices

The unit 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:

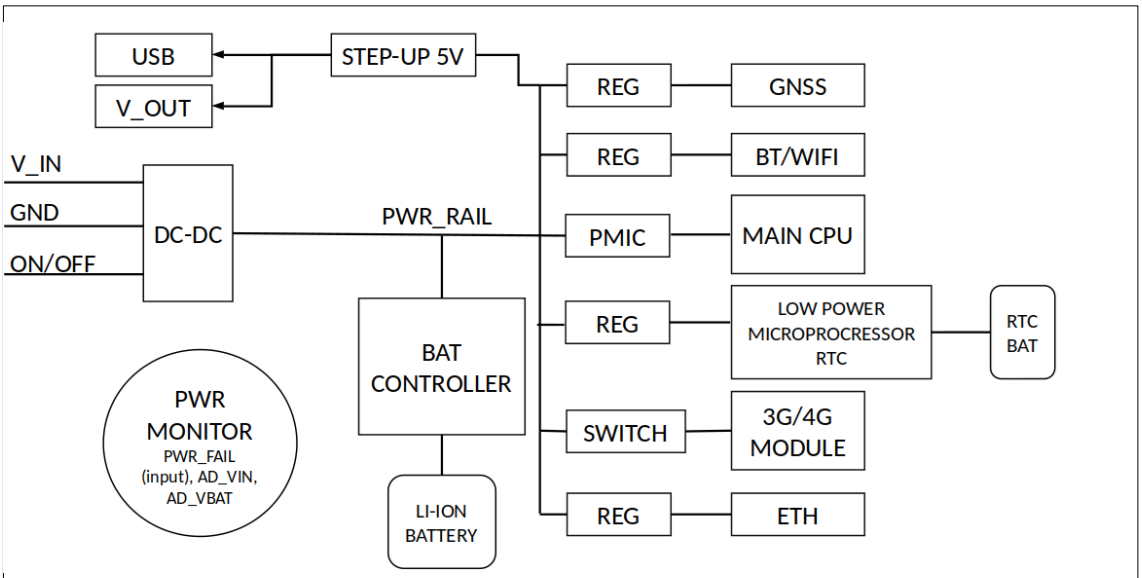


Figure 1: Block Diagram

6.1.1. Main CPU

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

6.1.2. 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
```

6.1.3. 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
```

6.1.4. 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)
```

6.1.5. Wi-Fi / Bluetooth

The Wi-Fi 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)
```

6.1.6. 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
```

6.2. How to get the lowest consumption in low power mode

To decrease the power 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 10 mA at 24V.

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 0.34 mA at 24V.

GSM can not be set as signal to wake up.

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
- WiFi/BT off
- GSM off
- GPS off
- SD off
- USB off
- Ethernet off
- CAN off
- 9 axis off
- Vout on, unless RTU_REMOVE_VOUT signal is set in the mask when entering into Off mode.

6.3. 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.

7. GPS API

7.1. 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.

The GPS library has changed from the one used in previous platforms, and it is called libGPS2_Module.so.0.0.1. Most of the functions are the same as with the old library, but please check the API to make sure when migrating a program from the owa3x platform to the platform.

7.2. 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 "owa4x/GPS2_ModuleDefs.h"
#include "owa4x/pm_messages.h"
#include "owa4x/IOs_ModuleDefs.h"
#include "owa4x/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 = COM1;

            // 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");
```

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()*.

7.3. 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, the GPS instance is finalized, followed by the RTUControl instance and the IOs instance respectively.

Only the GPS operation must be terminated in the case of other modules being required to continue operation.

In the GPS Application Note (**owa4x_AN5**) the complete sample source code for GPS initialization and startup can be found.

7.4. 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");)
  }
```

7.5. Change Sensitivity mode

There is a GPS library function to configure the sensitivity setting on the Ublox 6 GPS receiver for acquisition and tracking modes.

During the calculation of a position fix, the GPS receiver must acquire and track GPS signals. Acquisition is the process of “locking” onto the GPS signal. Tracking is the process of maintaining a lock on the previously acquired signal.

Tracking and Acquisition algorithms have an associated sensitivity level at which GPS signals can be detected with a sufficiently high confidence level.

On Ublox GPS module, higher sensitivity can be reached by extending the integration time of the GPS signal. This means that higher sensitivity is a trade off versus the time it takes to detect a GPS signal.

The Ublox GPS Technology allows the sensitivity of the receiver to be modified in three modes. Namely, they are:

- Normal. (trade off between sensitivity and time to acquire).

- Fast Acquisition (optimized for fast acquisition, at the cost of 3dB less sensitivity than with “normal” setting.

- High Sensitivity (optimized for higher sensitivity, i.e. 3dB more sensitive than the “normal” setting, at the cost of longer startup times).

Function to change Sensitivity Setting:

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

  printf( "GPS mode (0=normal,1=fast,2=high) >> ");
  memset( ( void *) &strEntry, 0, sizeof( strEntry));
  getEntry( strEntry);
```

```
          gpsmode = atoi( strEntry);
          if( (ReturnCode = GPS_SetGpsMode(gpsmode)) != NO_ERROR){
            RES_( printf( "Error %d in GPS_SetGpsMode()...\n",
ReturnCode);)
  } else {
    RES_( printf( "GPS_SetGpsMode() OK\n");)
  }
```

Where ‘GPSmode’ can be: 0=Normal, 1=Fast Acquisition, 2=High Sensitivity.

The default setting in the unit is High Sensitivity Mode.

Note of manufacturer: Use ‘Fast Acquisition’ mode if the C/No ratio of the strongest SV exceeds 48dBHz. In the case the C/No value of the strongest SV is below 45dBHz, use ‘High Sensitivity mode’.

7.6. 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 unit 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.

7.7. 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 unit is threshold set to 0m/s.

7.8. Navigation output

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

7.8.1. 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");)
  }
```

7.8.2. 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");)
  }
```

7.8.3. Shared memory information

Since version 1.0.9 of the GPS2 library the position information is written every second to shared memory, so it can be retrieved by any program, without having to use the GPS library functions.

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

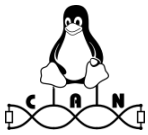
[011] -> HDilution 99.990000 → Horizontal dilution of precision

[012] -> VDilution 99.990000 → Vertical dilution of precision

[013] -> TDilution 99.990000 → Time dilution of precision

[014] -> positionTS 00000001585237727412 → position timestamp

8. CAN



The CAN bus is accessed as a socket using the SocketCAN implementation on the Linux kernel, see **Error! Bookmark not defined.** for more information on SocketCAN.

The CAN is accessed as a network interface with name canx. By default the unit has 2 CAN interfaces, can1 and can2, and optionally it can have another 2, can3 and can4.

The CAN driver must be first 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");
}
```

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
```

8.1. 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;
}
```

8.2. 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;
}
```

8.3. 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));
```

8.4. 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));
```

8.5. 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);
}
```

8.6. 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.

```
root@arm:/#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:

```
root@arm:/#candump can2 > /tmp/can2.log
```

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

```
root@arm:/#cansend can1 1F334455#1122334455667788
root@arm:/#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

```
root@arm:/#cansend can1 1F334455#1122334455667788
```

9. FMS API

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: 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: Tire Pressure Monitoring System, covers the data advertised from the tire sensors of some trucks.
- EBS: Electronically controlled Brake System, gets information from the electronic brakes and their status.

9.1. 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 unit is being connected. These values are the bitrate, the CAN interface on the unit 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);
                        exit(0);
            } else {
                        printf( "OWASYS--> OK in FMS_Start\n");
            }

            […]

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

9.2. 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));
```

9.3. 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));.
```

9.4. 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);
```

9.5. 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));
```

10. RS232

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

There are three possible serial interfaces in the UART1 (ttyO1), UART4 (ttyO4) and UART5 (ttyO5). The UART4 can be used as a full RS232 serial interface, or it is also possible to use only Tx and Rx leaving the rest of the signals free for the other UARTs. This UART4 is the one set for debugging purposes, to logging in the system or to enter into the bootloader prompt.

The UART5 Tx and Rx signals are multiplexed with the CTS and RTS signals of the UART4. These are the two choices available,

- 2 serial ports: UART4 with HW flow control with TXD4, RXD4, RTS4, CTS4. UART1 with TXD1 and RXD1. In this case UART5 can not be used.
- 3 serial ports: UART4 with TXD4 and RXD4. UART1 with TXD1 and RXD1. UART5 with TXD5 and RXD5.

10.1. Enabling serial interfaces

The UART4 is active by default with all its signals available. As UART5 is multiplexed with UART4 signals RTS and CTS, in order to work with this UART5 it must be enabled first.

This code shows how to enable the UART5.

```
int retVal;

retVal = DIGIO_Enable_Uart5(1);
if( retVal != NO_ERROR ) {
  printf("ERROR %d enable Uart5\r\n", retVal);
} else {
  printf("Enable UART5 OK\r\n");
}
```

10.2. 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 owa4x_AN3 to get further information about the use of these functions.

11. RS485

The unit has got a RS485 serial interface available at two pins of the connector. It is placed in the UART2 of the system which corresponds to ttyO2.

11.1. Enabling RS485

The RS485 is switched off by default so it must be enabled first before using it.

```
ReturnCode = DIGIO_Enable_RS485 ();
```

Please note that this function will disable the Kline serial interface of the unit as both Kline and RS485 are multiplexed in the same UART2.

11.2. Working with RS485

By default this UART2 is configured as 485, and so no configuration is needed to work with it. The transmission or reception states are automatically controlled by the Linux driver.

```
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 |= B115200 | CS8 | CLOCAL | CREAD;
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, B115200);
        if (ReturnCode < 0) {
  printf("Could not set output speed!\n");
  close(FD_485);
  FD_485 = -1;
                  return -1;
        }
        ReturnCode = cfsetispeed(&newtio, B115200);
        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);
printf(" %s configured succesfully.\n", UART_RS485);
```

11.3. OPTIONAL STEPS

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;
 }
```

The UART can also be used as RS232 if needed. To use it as RS232 the enabling flag must be unset in the mask.

```
 /* Unset the RS485 enabling flag, if needed: */
 rs485conf.flags &= ~(SER_RS485_ENABLED);

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

12. KLINE

The platform supports 2 optional Kline interfaces that can be easily configured using the same standard Linux functions that are used to configure and work with the serial ports.

The **first Kline** is bidirectional and it works with the UART2 which corresponds to ttyO2 in pin 33 of the machine connector.

The **second Kline** is unidirectional as it only receives data from the bus. It works with the UART1 which corresponds to ttyO1 in pin 23 of the machine connector, which is multiplexed with DIO7.

12.1. Enabling the Kline

The first Kline interface is switched off by default, so it must be enabled first before using it.

```
ReturnCode = DIGIO_Enable_KLINE (1);
```

Warning: Please note that this function will disable the RS485 serial interface of the unit as both Kline and RS485 are multiplexed in the same UART2.

12.2. Configuring the Kline

Both Kline interfaces are configured as serial ports using the standard termios structures, with the exception of the termios option c_oflag and the speed.

The option c_oflag is used for the first Kline to activate or deactivate the initialization message and if this must be a slow or a fast initialization.

```
#define KL_INIT     0x80000000 /* Selects init sequence */
#define KL_SLOW      0x40000000 /* Selects SLOW init sequence
```

The speed must be configured at 10400 bps by passing B50 to termios option c_cflag. The following code shows how to configure these parameters to finally apply them with the function tcsetattr().

```
newtio.c_cflag = B50 | CS8 | CLOCAL | CREAD;
newtio.c_iflag = 0;
newtio.c_oflag = KL_INIT; // With initialization
newtio.c_oflag |= KL_SLOW; // With slow initialization
newtio.c_lflag = 0;
newtio.c_cc[VMIN]=0;
newtio.c_cc[VTIME]=0;

tcflush(fd, TCIFLUSH);
tcsetattr(fd,TCSANOW,&newtio);
```

After opening the interfaces and changing their configurations they can be used as any other serial interface, using the standard read() and write() functions to communicate with other Kline devices. See application note owa4x_AN9 to get an example of their use.

13. BLUETOOTH

The integrated bluetooth module is a v4.2 low power consumption module that makes possible the bluetooth communication in a piconet. Its class 1 with 7dBm output allows a communication range of up to 100 meters in an industrial environment (with line of sight).

13.1. 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.

```
# Start_BT_WiFi 1
```

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

```
# Start_BT_WiFi 0
```

13.2. BT profiles

At the time of writing this guide there are 4 profiles supported in the owa44-BT: Serial Port (SPP), OBEX Object Push (OPP), OBEX File Transfer (FTP) and Network Access Point (PAN). The get all the available profiles type this command

```
$ sdptool browse
```

13.2.1. Serial Port (SPP)

This profile is based on the RFCOMM protocol. It emulates a serial cable to provide a simple substitute for a RS232 connection.

In the owa44-BT the rfcomm utility may be used to start serial connections with BT devices in the neighborhood.

To connect to a PC with bluetooth using rfcomm first the RFCOMM device must be bind to the PC in channel 1.

```
$ rfcomm bind 0 00:25:56:DF:C5:22 1
```

0 → device /dev/rfcomm0

00:25:56:DF:C5:22 → BT address of the PC

1 → Channel number 1

If this command is successful the owa44-BT is bound to the PC.

```
$ rfcomm show all
rfcomm0: 00:25:56:DF:C5:22 channel 1 clean
```

By trying to access the device /dev/rfcomm0 the connection is with the bounded device is made. Following with the connection at the PC example the device can be accessed with the cat command at the shell prompt of the owa44-BT.

```
$ cat /dev/rfcomm0
```

The PC will show a message to pair the owa44-BT, by default the configured pin is BlueZ, and once the connection is established the connection can be tested with hyperterminal. Open the port available for the bluetooth interface with the usual configuration parameters (speed 115200, no parity and no flow control).

To connect the other way around, from the PC to the owa44-BT, the owa44-BT may be set in listening mode. Then the connection may be started from the PC, and characters may be sent from the PC to the owa44-BT with hyperterminal and opening the rfcomm0 device.

```
$ rfcomm listen /dev/rfcomm0 1 &
// Start connection at the PC
$ cat /dev/rfcomm0
```

13.2.2. OBEX Object Push (OPP)

This profile is used to facilitates the exchange of binary objects between devices using bluetooth, the communication protocol is implemented over RFCOMM, and this service in particular is used to exchange objects like business cards that are normally text files with information about a contact, but that can also be calendar items, notes or messages.

The first thing to do is to start the obexftpd daemon in the owa44-BT.

```
$ obexftpd -c /tmp -b9 &
```

In the example above /tmp is set as default base directory to receive the objects from the PC. With the option -b9 we indicate to communicate using channel 9.

Once the daemon is running the object is sent from the PC by selecting the OBEX Object Push profile.

13.2.3. OBEX File Transfer (FTP)

This profile is like the OPP profile but for transferring ordinary files from a PC to a owa44-BT. Like with the OPP profile the obexftpd daemon must be started first.

```
$ obexftpd -c /tmp -b9 &
```

Opening the FTP profile in the PC opens a exchanging directory where files can be transferred.

To start the action from the owa44-BT unit instead, the command obexftp can be used like it follows.

```
$ obexftp -b 00:25:56:DF:C5:22 -p file
```

-b <bt_address> is the address of the PC we want to send the file.

-p <file> is the file to “put” in the PC.

To get a file from the PC to the owa44-BT the option -g can be used. The file in the PC must be in the bluetooth exchange directory.

```
$ obexftp -b 00:25:56:DF:C5:22 -p file
```

-g <file> is the file to “retrieve” from the PC.

13.2.4. Network Access Point (PAN)

BNEP (Bluetooth Network Encapsulation Protocol) is required for Bluetooth PAN and, unlike in the owa3x, the module ***bnep*** will be automatically loaded when needed.

Furthermore, the BlueZ version 5 stack architecture, which is available in the unit, has introduced some changes. For example, there is no longer *pand* utility available. Hence, to easily create a PAN bridge, installing the ***bluez-tools*** package is advisable.

```
apt-get install bluez-tools
```

This package contains a set of tools to manage bluetooth devices:

- bt-adapter allows the user to change adapter properties (e.g., Name, Discoverable, Pairable, etc) and discover remote devices.
- bt-agent manages incoming requests (e.g., request of pincode, of authorize a connection/service request, etc.).
- bt-device allows the connection to remotes devices by their MACs and service discovery.
- bt-network manages network services (client/server). For example, it allows the user to register server for the provided UUID.

Therefore, for example, the following commands configure an access point:

```
bt-agent -c NoInputNoOutput     # Set agent's input/output capabilities
bt-network -s nap pan0         # Create a Bluetooth NEP PAN
bt-adapter --set Discoverable 1   # Switch the adapter to discoverable
```

Moreover, it would be necessary to assign IP addresses, either static or dynamic ones and enable IP forwarding on the unit:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

13.3. Bluetooth programming

The bluetooth communication can be also managed coding the functionality within the customer application itself. The BlueZ stack offers a series of functions that can be used to search for bluetooth devices, connect and communicate with them. These functions may be found in the source code of the bluez stack.

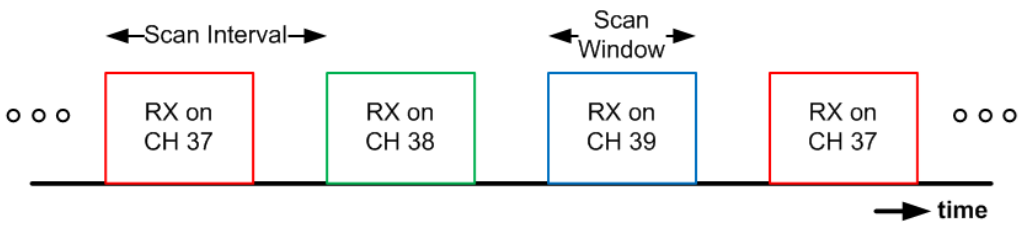
13.4. BLE scanning with WiFi as AP

If Bluetooth is used simultaneously with the Wi-Fi, some recommendations must be followed to avoid disruptions in the communication when scanning for BLE devices. The default BLE scan timing is to listen for 10ms every 10ms. With a single antenna setup as with the installed module, this will create contention with the beacons emitted by the access point, depending on timings some or all will be dropped. The Wi-Fi clients will in turn disconnect because they do not see beacons anymore.

The solution is to modify the BLE scan timing parameters to leave more time for WLAN. The *hcitool lescan* command sets those parameters to the default values before starting scanning, these values must be modified directly in the hcitool.c source code.

```
//uint16_t interval = htobs(0x0010); // default value 10 ms
uint16_t interval = htobs(0x0080);  // value set to 80 ms
uint16_t window = htobs(0x0010);
```

Setting the interval value to 80 ms will give the needed time to the WiFi part to work correctly.



14. Wi-Fi

The Wi-Fi peripheral that the unit integrates is handled using the standard methods in Linux.

When the device boots up the Wi-Fi is off by default. To use the Wi-Fi 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 Wi-Fi 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 Wi-Fi driver.

14.1. Wi-Fi switch on and off

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

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

After working with Wi-Fi if the user does not need it any longer it can be switched off in order to decrease the consumption of the owa44-Wi-Fi.

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

To enable the mlan0 interface the command ip may be used.

```
root@arm:~# ip link set mlan0 up
```

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

```
# Start_BT_WiFi 1
```

To switch off the Wi-Fi, use the same command with argument set to 0.

```
# Start_BT_WiFi 0
```

14.1.1. Automatic Wi-Fi start

In order to have Wi-Fi 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 Wi-Fi module is switched on.

/etc/systemd/system/wifistart.service:

```
[Unit]
After=pmsrv

[Service]
Type=oneshot
ExecStartPre=/bin/sleep 5
ExecStart=/usr/bin/Start_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
```

14.2. Wireless useful commands

Some useful commands are in the default file system in order to interact with the WiFi module and its functionality. This section will show examples of how to perform the following operations using these commands:

- Enable the WiFi network interface
- Connect to an unprotected network
- Assign an IP address using static configuration or DHCP.
- Verify the connection status with the ping command.
- Disconnect from the network.
- Connect to a WPA2 encrypted network.
- Disable device power save mode.

All the steps and expected output is provided in detail below. See the end of this section for a complete list of supported commands. For complete documentation on these commands, see **Error! Bookmark not defined.**.

14.2.1. iw

To get information abou the status of the interface at any moment, use this command.

```
root@arm:~# iw mlan0 info
```

To scan the SSID around the unit.

```
root@arm:~# iw mlan0 scan
```

This command shows the APs with useful information like the ESSID, the MAC address, signal level, encryption mode and the channel used.

This command can be used with its option *connect* to connect to an open AP.

```
root@arm:~# iw mlan0 connect OWABS
```

With this command the mlan0 will associate with the AP. Check with the *info* option that the interface has connected successfully. Once the connection is established the interface mlan0 configuration can be done statically or dynamically.

To configure a static IP ip command can be used.

```
root@arm:~# ip addr add 192.168.1.100/24 dev mlan0
```

If the AP has got a DHCP server a good idea is to use the DHCP client of the unit to configure the connection.

```
root@arm:~# dhclient mlan0
```

To disconnect from the network use option *disconnect*.

```
root@arm:~# iw mlan0 disconnect
```

With *iw* a power save mode can also be set. In power save mode, the device will sleep until either the host request to transmit data or until there is buffered incoming data to fetch from the access point. In power save mode, the throughput performance will be slightly degraded and the response latency will depend on the access point beacon interval and DTIM parameters. In most cases the access point is configured with 100 ms beacon interval and a DTIM interval of 1; this results in responses time around 100 ms. Depending on the application, the power consumption of the WiFi device can be heavily reduced.

Device power save mode can be enabled with the *iw* command:

```
root@arm:~# iw mlan0 set power_save on
```

As a drawback, data may be lost and the latency may increase significantly in power save mode. When using the unit as a gateway, using the WiFi in hotspot mode, it would be better to set off this power save mode.

```
root@arm:~# iw mlan0 set power_save off
```

The current power management configuration can be shown by issuing *iw* again:

```
root@arm:/etc# iw mlan0 get power_save
Power save: off
```

14.3. WPA SUPPLICANT

The WPA supplicant can be configured to control the roaming and IEEE 802.11 authentication/association of the WiFi. The configuration is usually performed in a configuration file, e.g. /etc/wpa_supplicant.conf. It is also possible to directly issue commands to the WPA Supplicant, using a dedicated shell command, wpa_cli. The usage of wpa_cli is out of the scope of this document, but is described in detail in the WPA supplicant documentation **Error! Bookmark not defined.**.

This section will show examples of how to perform the following operations using WPA Supplicant.

- Connect to an unprotected network
- Connect to a WPA-PSK network with TKIP encryption
- Connect to a WPA2 network with CCMP encryption
- Connect to a WPA or WPA2 network with either TKIP or CCMP encryption

14.3.1. Connect to an unencrypted network

To simply instruct the WPA Supplicant to connect to any unencrypted network with ssid OWASYSBS, the following configuration file should be enough:

```
ctrl_interface=/var/run/wpa_supplicant
network={
ssid="OWASYSBS"
key_mgmt=NONE
}
```

The path to the configuration file and the interface name (mlan0) should then be passed as parameters when starting the WPA Supplicant:

```
$ wpa_supplicant -imlan0 -c /etc/wpa_supplicant.conf -B
```

- -imlan0 to use interface owl0
- -c option to tell the command what configuration file look at
- -B to run in background

For detailed information on how to configure and run the WPA supplicant, see the WPA supplicant documentation **Error! Bookmark not defined.**.

The WPA Supplicant will now periodically scan for networks until one that matches the configuration is found. Once found, a connection will be established. The WPA Supplicant will also handle reconnect if the connection is lost.

Note that the WPA Supplicant configuration can hold several networks and the WPA Supplicant will choose and roam amongst them. However, most importantly, the WPA supplicant implements the key negotiation with a WPA Authenticators.

14.3.2. Connect to a WPA-PSK network with TKIP encryption

To connect to a network using WPA key management and TKIP encryption, the following network configuration can be specified in the configuration file:

```
network={
ssid="OWASYSBS"
key_mgmt=WPA-PSK
group=TKIP
pairwise=TKIP
proto=WPA
psk="owasyswirelesskey"
}
```

The key configured on the access point should be owasyswirelesskey. To force the WPA Supplicant to re-read its configuration file wpa_cli can be used.

```
$ wpa_cli reconfigure
```

The interface configuration may be done in the same way as with open connections, checking the status with *iw* command and getting an IP with *dhclient* or setting it statically with *ip*.

14.3.3. Connect to a WPA2 network with CCMP encryption

To connect to a network using the WPA2 protocol and CCMP encryption, the following network configuration can be specified in the configuration file:

```
network={
ssid="OWASYSBS"
key_mgmt=WPA-PSK
group=CCMP
pairwise=CCMP
proto=WPA2
psk="owasyswirelesskey"
}
```

14.3.4. Connect to a WPA or WPA2 network with either TKIP or CCMP encryption

Note that several encryption parameters can be specified on a single line, allowing connections to a specific ssid using a range of encryption methods. The configuration file below should allow connections to the OWASYSBS access point regardless of whether the WPA or WPA2 protocol is used or whether CCMP or TKIP is used for pairwise and group key encryption. The actual encryption method used will be the most secure one that is supported by the access point.

```
network={
ssid="OWASYSBS"
key_mgmt=WPA-PSK
group=TKIP CCMP
pairwise=TKIP CCMP
proto=WPA WPA2
psk="owasyswirelesskey"
}
```

14.3.5. Automatic WiFi connection to an AP

The Wi-Fi mlan0 interface can be set up automatically at every reboot using the file *'/etc/network/interfaces'*.

It follows a configuration example where the interface is configured when the interface gets up, using a wpa.conf configuration file with the authentication data to connect to the desired AP essid.

'example of /etc/network/interfaces'

```
...
auto mlan0
allow-hotplug mlan0
iface mlan0 inet manual
      wireless-power off
#      wireless-frag 1024
      wpa-roam /etc/wpa_supplicant/wpa.conf

iface somename inet dhcp
   metric 2
```

'example of /etc/wpa_supplicant/wpa.conf'

```
ctrl_interface=/var/run/wpa_supplicant
ctrl_interface_group=root
update_config=1
ap_scan=1

#keep scanning when disconnected from the AP
autoscan=periodic:10
#autoscan=exponential:2:60
disable_scan_offload=1

network={
   ssid="ssidname"
   #psk="password"

psk=02ec875fc5fab3ac6efe6edec3b84baa8e156d42ac6b0d7b50621d02b0c1ed85
   id_str="somename"
}
```

14.4. WI-FI AP

The Wi-Fi module can be set as master to work as AP, so that Wi-Fi clients, up to 10 clients, can connect to the unit working as AP. To set the unit on this mode, first some tools must be installed, and after their configuration set them to work.

First enable the Wi-Fi module with IO function DIGIO_Enable_Wifi(), or from the command line with "Start_BT_WiFi"

```
# Start_BT_WiFi 1
```

The tools that must be installed are "**hostapd**" to set the Wi-Fi as AP and "**dnsmasq**" to serve network configuration with DHCP.

```
# apt-get install hostapd dnsmasq
```

After installing them, the configuration must be edited, so these services must be stopped.

```
# systemctl stop hostapd dnsmasq
```

The interface that is going to be used in master mode is “uap0”. The hostapd service and the IP that will have can be edited in /etc/network/interfaces configuration file.

```
auto uap0
iface uap0 inet static
hostapd /etc/hostapd/hostapd.conf
address 192.168.1.1
netmask 255.255.255.0
```

Then edit the configuration file of hostapd service, in **/etc/hostapd/hostapd.conf**.

```
ctrl_interface=/var/run/hostapd
ctrl_interface_group=0
interface=uap0
driver=nl80211
ssid=owasys
ignore_broadcast_ssid=0
hw_mode=g
channel=6
macaddr_acl=0
auth_algs=1
wpa=2
wpa_passphrase=owasys123456789
wpa_key_mgmt=WPA-PSK
rsn_pairwise=CCMP
```

In the configuration example, the AP is called “owasys” with password “owasys123456789”. The hw_mode is set to g in this example, but it could also be set to any other available, a/b/g/n or ac, using ieee80211 parameter

It follows an example of how to configure it with *hw_mode=a* and *ieee80211ac=1*, which works at 5 GHz and can be interesting for some scenarios where there are too many 2.4 GHz bands in use, and the distances are rather short and with good visibility:

```
ctrl_interface=/var/run/hostapd
ctrl_interface_group=0

ssid=owasys
wpa_passphrase=owasys123456789
wpa=2
wpa_key_mgmt=WPA-PSK
```

```
rsn_pairwise=CCMP

interface=uap0
driver=nl80211

hw_mode=a
ieee80211d=1
ieee80211h=1
ieee80211n=1
require_ht=1
ieee80211ac=1
require_vht=1

country_code=US
vht_oper_chwidth=1
channel=36
vht_oper_centr_freq_seg0_idx=42
```

To change the mode, channel, encryption and other features, see the hostapd full description in https://w1.fi/hostapd/

In order to provide network configuration with DHCP, dnsmasq configuration file must be edited to be in accordance with the one chosen for the interface uap0, in this example the IP 192.168.1.1 in the network 192.168.1.x. The configuration file of dnsmasq is in **/etc/dnsmasq.conf**

```
interface=uap0
except-interface=eth0
dhcp-range=192.168.1.2,192.168.1.20,12h
```

Now that everything is ready, the networking service must be restarted, which will get the uap0 interface up and with it the hostapd service, and then the dnsmasq service.

```
root@arm:/home/debian# systemctl restart networking
[80805.422247] get_channel when AP is not started
[80805.456922] get_channel when AP is not started
[80805.470097] get_channel when AP is not started
[80805.496504] IPv6: ADDRCONF(NETDEV_UP): uap0: link is not ready
[80805.623568] wlan: Starting AP
[80805.706199] wlan: AP started
[80805.709319] IPv6: ADDRCONF(NETDEV_CHANGE): uap0: link becomes ready
[80805.737823] Set AC=3, txop=47 cwmin=3, cwmax=7 aifs=1
[80805.756575] Set AC=2, txop=94 cwmin=7, cwmax=15 aifs=1
[80805.776528] Set AC=0, txop=0 cwmin=15, cwmax=63 aifs=3
[80805.796424] Set AC=1, txop=0 cwmin=15, cwmax=1023 aifs=7
root@arm:/home/debian# systemctl start dnsmasq
```

This is valid to connect to the unit from a mobile phone or tablet and to open a web site from an HTTP service running in the unit itself. If the purpose is to get the unit working as router to get internet access using another interface, ppp0 from 3G module or eth0 from Ethernet cabled connection, a couple of commands are yet needed.

To enable the routing, uncomment this line in /etc/sysctl.conf

```
net.ipv4.ip_forward=1
```

To enable the change in this configuration file, execute this command:

```
# sysctl --system
```

Finally, enable the iptables rule to forward the traffic to eth0 interface:

```
# iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

15. 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.

15.1. 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.

15.2. 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.

15.3. USB and uSD

uSD can be switched off and on with the following IO function:

Switch the SD off:

```
ReturnCode = DIGIO_Set_SD_Card(0)
```

Switch the SD on:

```
ReturnCode = DIGIO_Set_SD_Card(1)
```

The USB however, can be switched on and off writing on the following file of the file system.

Switch the USB off:

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

Switch the USB on:

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

15.4. 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.

16. Watchdog

16.1. Support for hardware and software watchdogs

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

16.2. 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
```

16.3. 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 owa4x daemon

[Service]
ExecStart=/usr/bin/myowa4xd

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
```

16.3.1. 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");
  }
```

16.3.2. 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 */
            …
  }
```

17. Useful tips

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

- logrotate
- Variables *SystemMaxFileSize, SystemMaxFiles in /etc/systemd/journald.conf*

18. Technical support

Additional sources of information are available on the CrossControl support site:
https://crosscontrol.com/support/
You will need to register to the site in order to be able to access all available information

Contact your reseller or supplier for help with possible problems with your device. In order to get the best help, you should have access to your device and be prepared with the following information before you contact support.

- The part number and serial number of the device, which you can find on the brand label.
- Date of purchase, which can be found on the invoice.
- The conditions and circumstances under which the problem arises.
- Status indicator patterns (i.e. LED blink pattern).
- Prepare a system report on the device, using CCSettingsConsole (if possible).
- Detailed description of all external equipment connected to the unit (when relevant to the problem).

19. Trademarks and terms of use

All trademarks sighted in this document are the property of their respective owners.

CCpilot is a trademark which is the property of CrossControl.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

CC Linux is an officical Linux distributuition pursuant to the terms of the Linux Sublicense Agreement

Microsoft® and Windows® are registered trademarks which belong to Microsoft Corporation in the USA and/or other countries.

Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Qt is a registered trademark of The Qt Company Ltd. and its subsidiaries.

CrossControl is not responsible for editing errors, technical errors or for material which has been omitted in this document. CrossControl is not responsible for unintentional damage or for damage which occurs as a result of supplying, handling or using of this material including the devices and software referred to herein. The information in this handbook is supplied without any guarantees and can change without prior notification.

For CrossControl licensed software, CrossControl grants you a license, to under CrossControl’s intellectual property rights, to use, reproduce, distribute, market and sell the software, only as a part of or integrated within, the devices for which this documentation concerns. Any other usage, such as, but not limited to, reproduction, distribution, marketing, sales and reverse engineering of this documentation, licensed software source code or any other affiliated material may not be performed without the written consent of CrossControl.

CrossControl respects the intellectual property of others, and we ask our users to do the same. Where software based on CrossControl software or products is distributed, the software may only be distributed in accordance with the terms and conditions provided by the reproduced licensors.

For end-user license agreements (EULAs), copyright notices, conditions, and disclaimers, regarding certain third-party components used in the device, refer to the copyright notices documentation.