Product revision: 1.0
2018-03-19

crosscontrol

CrossLink TG

Programmer and Software Manual

crosscontrol
www.crosscontrol.com

Contents

Revision history **3**
1. Introduction **4**
1.1. Scope 4
1.2. Accessing CrossLink TG Libraries 4
2. CrossLink TG RTUControl API **5**
2.1. Launching the RTUControl Module 6
2.2. RTC Time 6
2.3. Sleep and useCsleep 7
2.4. Power Management 7
2.5. RTC Hardware 9
2.6. Motion Sensor 10
2.7. Analogue Internal Inputs 11
2.8. Timers 12
3. CrossLink TG IO Modules **14**
3.1. LED Control 14
3.2. Inputs and I/O 15
4. CrossLink TG GSM **16**
4.1. Starting GSM 17
4.2. Handling GSM Events 18
4.3. Making Voice or Data Calls 20
4.4. Subscription to an NE SMS Arrival 22
4.5. Finishing GSM 23
4.6. Audio Parameters and Configuration 23
5. CrossLink TG iNet API **24**
5.1. Starting an Internet Connection 24
5.2. Closing an Internet Connection 25
5.3. Out of GSM Coverage 26
5.4. GSM Functionality in an active INET Session 26
6. CrossLink TG Power Optimization **27**
6.1. Reducing Power Consumption by Switching Off Devices 27
6.2. Low Power Mode 28
7. CrossLink TG GPS API **29**
7.1. Starting GPS 29
7.2. Shutting Down the GPS 30
7.3. Retrieving GPS Position 30
7.4. Change Sensitivity Mode 31
7.5. Change the Dynamic Platform Model 32
7.6. Change Static Hold Mode 33
7.7. Navigation Output 34
8. CrossLink TG CAN **35**
8.1. Creating a Socket 36
8.2. Configuring Can 36

8.3. Receiving CAN frames 36
8.4. Writing CAN Frames 36
8.5. Closing a CAN Channel 37
8.6. CAN-UTILS 37
9. CrossLink TG 37
9.1. Starting and Finalising FMS Library 38
9.2. Retrieving FMS Data 38
9.3. Retrieving TPMS Data 39
9.4. Removing TPMS Data 39
9.5. Retrieving EBS Data 39
10. CrossLink TG RS232 39
10.1. Enabling Serial Interfaces 39
10.2. Working with RS232 40
11. CrossLink TG RS485 40
11.1. Enabling RS485 40
11.2. Working with RS485 40
11.3. Optional Parameters 41
12. CrossLink TG BLUETOOTH 41
12.1. BT Stack 41
12.2. BT Profiles 42
12.3. Bluetooth Programming 44
13. CrossLink TG Wi-Fi 44
13.1. Wi-Fi Activation 44
13.2. Useful Wireless Commands 45
13.3. WPA Supplicant 46
13.4. CrossLink TG Wi-Fi AP 48
14. System Reinitialization 49
14.1. GSM Module 49
14.2. GPS Module 50
14.3. USB and uSD 50
14.4. System Boot 50
15. Watchdog 51
15.1. Hardware Watchdog 51
15.2. Software Watchdog 51
16. Further Reading 54

Revision history

Rev	Date	Author	Comments
Version 1	2019-04-01	Finn Mc Guirk	0

1. Introduction

The CrossLink TG is an advanced Telematics control unit. Custom designed to cope with harsh environments, it sits close to the engine to monitor and control key functions, sift data and transfer KPIs to the cloud.

1.1. Scope

This document is intended for anyone handling the CrossLink TG or developing software for the CrossLink TG. This document is not intended as a complete reference documentation for the device, software, or software development tools. It is intended to introduce the development engineer to the hardware and software, by providing a top-down overview and summary of the device.

The document will give a guide to the structure of each API and how to launch the available functions within the API library for developing applications.

1.2. Accessing CrossLink TG Libraries

Depending on the required functionality that you need to access, the following table shows the files and libraries that may be used.

Libraries

	File	Dynamic Libraries
RTU	<owa4x/RTUControlDefs.h>	/lib/libRTUControl.so
IO	<owa4x/IOs_ModuleDefs.h>	/lib/libIOs_Module.so
GSM	<owa4x/GSM_ModuleDefs.h>	/lib/libGSM_Module.so
GPS	<owa4x/GPS2_ModuleDefs.h>	/lib/libGPS_Module.so
iNET	<owa4x/INET_ModuleDefs.h>	/lib/libINET_Module.so

There are 2 ways to access the dynamic libraries; using dlopen() and dlsym() or by linking to them during compilation.

1.2.1. Using Library Access

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

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

```
void* wLibHandle = NULL;
wLibHandle = dlopen("/lib/libGSM_Module.so", RTLD_LAZY); if (!wLibHandle) {
printf("No shared library found");
}
```

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

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

```
int (*FncDIGIO_SetLED_SW0)( unsigned char)
```

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

```
FncDIGIO_SetLED_SW0=(int(*)(unsigned char))dlsym(wLibHandle,
"DIGIO_SetLED_SW0");
if ( dlerror() != NULL) {
printf("No DIGIO_SetLED_SW0 found..\n");
int ReturnCode = NO_ERROR;
if ( (ReturnCode = ( *FncDIGIO_SetLED_SW0)( 1)) != NO_ERROR ) {
printf("Error %d in DIGIO_SetLED_SW0()\n", ReturnCode);
```

Finally the function is called as shown below.

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

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

1.2.2. Direct Linking

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

```
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, as in the example using GSM library, the function can be called directly without having to use dlopen() or dlsym().

2. CrossLink TG RTUControl API

The Lib LibRTUControl library provides a set of 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.

- Accelerometer related functions.
- Internal analog inputs (temperature, Vin, Vbat)

Other functions are available within the RTUControl library.

2.1. Launching the RTUControl Module

LibRTUControl library is needed internally by the 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, 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 it will 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;
}
```

2.2. RTC Time

The CrossLink TG provides a Real Time Clock with calendar options that is fully programmable. This clock has a dedicated backup battery, and so keeps the time even when the unit is not powered. See the CrossLink TG Technical Manual for further information on the RTC backup.

Apart from this RTC, the CrossLink TG 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 retrieve or set the system time at any moment.

The CrossLink TG unit has an internal RTC. The RTC will update the system time when the unit reboots or when it wakes 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 using the command line protocol. There are two commands embedded in the OS, one to set the HW time based on the system time, and one to set the system time based on the HW clock.:

- sysclktohw. Sets the HW time based on the system time, this can be retrieved with command “date”
- hwclktosys. Sets the system time based on the HW clock. This is done automatically every time the system boots up.

2.3. Sleep and useCsleep

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

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

2.4. Power Management

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

The following signals can wake up the device:

- MOVEMENT: triggers if the movement sensor detects any movement.
- GSM: triggers if there is any event data (RING, SMS or COVERAGE events for example) received from the GSM module.
- DIN0..9: triggers if the external DIN0..9 signal is active.
- RTC: triggers if the RTC clock reaches a determined time.
- CONSOLE: triggers if a character is received at RS232 ttyO4 interface in pin RX0.
- PWRFAIL: triggers if the unit detects that there is no external power.

2.4.1. 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, RXD2, 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 CrossLink TG/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_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)
```

An 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");
}
```

2.4.2. 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.

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");
}
```

2.4.3. Wake Up

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 CrossLink TG/RTUControlDefs.h include file.

2.5. RTC Hardware

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

The RTC Hardware 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.

2.5.1. 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 included file.

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

2.5.2. Retrieveing the RTC Time and Date

The library's function that retrieves 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.

2.5.3. 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.

2.6. Motion Sensor

The CrossLink TG has an accelerometer that can be set to display data 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 maintains its 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 return.

2.6.1. Configuring the Motion Sensor

The interruption of the movement sensor may be managed 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 up to 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 for a moving interruption.

2.6.2. Retrieving Motion Sensor Status

The library function that gets the movement status is,

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

This function returns the MovedValue parameter for the 'MOVED' status. If an '0' is returned, that 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.

2.6.3. 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.

2.6.4. Retrieving the Acceleration Value

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 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.

2.7. Analogue 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.

2.7.1. 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);
```

2.7.2. 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);
```

2.7.3. Internal Temperature

The internal temperature can be obtained with this function.

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

2.8. Timers

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

IO library offers several functions to handle the timers, as explained below.

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 again switches to Running internal status, but the internal counter is reset to the top time value, beginning to count down the counter 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.

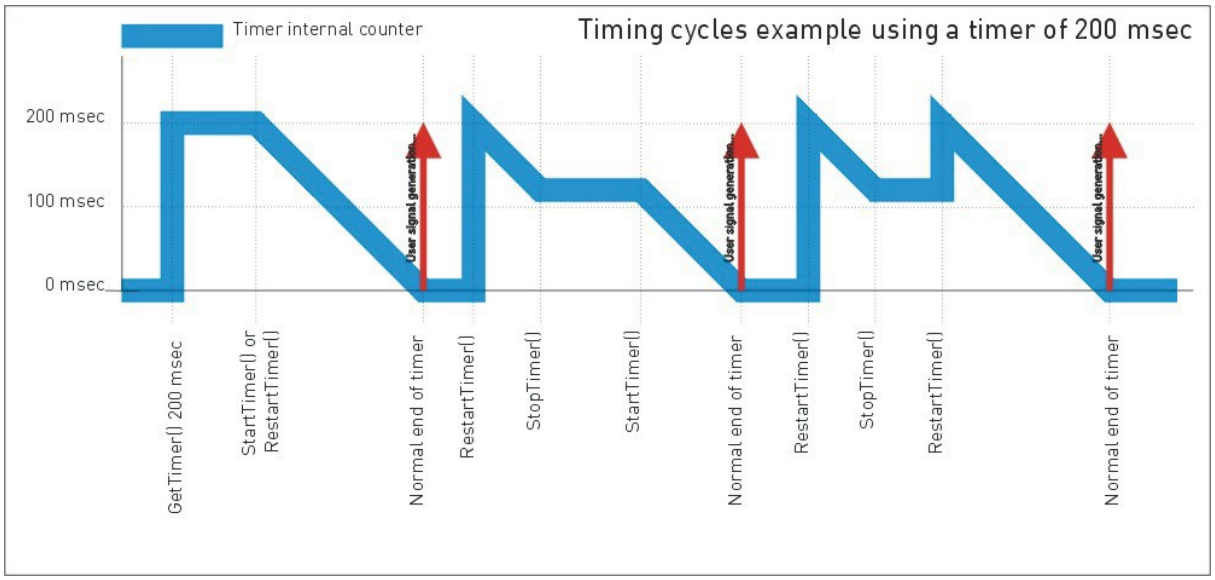
If the OWASYS_FreeTimer() function is called, the top value is set to 0 value, and the timer is not available any more.

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

	Status	Timer Limit	Internal Counter
GetTimer	Stopped	200 msec	200 msec
StartTimer/RestarTtimer	Running	200 msec	200 msec (counting down)
StopTimer	Stopped	200 msec	X msec
StartTimer	Running	200 msec	X msec (counting down

RestartTimer	Running	200 msec	200 msec (counting down)
FreeTimer	Stopped	0 msec	0 msec

For example:



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 should 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_SttartTimer) (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 timer handlers' code 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 the action that is required.

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;
```

3. CrossLink TG I/O Modules

LibIOs_Module library manages a resource to access the I/Os of the system architecture. The offered services include the following:

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

3.1. LED Control

Any of the LEDs may be controlled by the user. By default when GSM is ON it will take control of the YELLOW LED, and when the GPS is ON it will take control of the ORANGE LED. The user can override this default setting and take control of these LEDs as well using I/O functions.

By default all the LEDs are off when the CrossLink TG system boots up.

3.1.1. 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. Inputs and I/O

Before any function from the I/O library is used, the RTU and I/O 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;
}
```

3.2.1. Polling Digital Inputs

In order to retrieve the value from the digital input, the corresponding function must be called (see the API for the complete set of available functions). In the following piece of code, the value of the 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;
}
```

3.2.2. Retrieving Digital Input Values through 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.

As usual, the I/O library functions must be loaded and the I/O 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.

For example:

```
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: If a user wants to the change the interrupt configuration, it is first necessary to remove any existing interrupt service before enabling a new service parameters.

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, CrossLink TG_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);
```

4. CrossLink TG GSM

Programmers have access to all levels of the GSM functionality through the library, which contains all necessary commands for the unit.

4.1. Starting GSM

The GSM library, as well as RTU and I/O 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 the RTU and I/Os libraries in that specific order.

Once that I/O 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 the above 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.

4.2. 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 there is 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.

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 signalled, 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 (CrossLink TG_AN24) , which runs the call functions (Dial and Answer) on a different thread, which allows running other GSM actions from this application. A more technical explanation of this process is given in the next section.

4.3. Making Voice or 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.

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 that the pthread_detach after the new process is created. The goal of this function is that once the thread ends, to save resources from the system so 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.

4.4. Subscription to an NE 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 enabled 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);
}
```

1 = ENABLE, 0 = DISABLE.

Once a new subscription has been enabled, the following example shows how to read a newly received 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)

4.5. Finishing GSM

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

```
GSM_Finalize ( ); runGSMHandler = false; sem_post( &GsmEventsSem);
pthread_join( gsmEvents, NULL);

IO_Finalize( ); RTUControl_Finalize ( );

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

1. GSM is finalized first, GSM_Finalize()

2. The flag that controls that the thread is looping, **runHandleEvents**, is set to false.

3. The semaphore is signalled to wake up the thread. The thread will exit

4. The thread is joined from the main.

5. Semaphore is then destroyed, as we do not need it anymore

6. I/O and RTU libraries are finalized

4.6. Audio Parameters and Configuration

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

For example

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

4.6.1. OWA_AUDIO_MODEL

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

4.6.2. [HF_TXGAIN]

MIC sensitivity 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

4.6.3. [HF_RXGAIN]

This controls the DAC amplification in the speaker's 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 can 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 is set to 88,88

4.6.4. [HF_VOLUME]

[0 - 100]

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

Default value is set to 60

4.6.5. [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 is set to 3,2

5. CrossLink TG iNet API

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.

5.1. Starting an Internet Connection

To establish an Internet connection, RTU, I/O and GSM must be first started, as explained previously.

Once the GSM is initialized and started, the Internet session can be started as 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 user created thread that calls the iNetHandleEvents function (For more information, see the CrossLink TG_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, in which the pointer is passed in the 'inet_action' configuration structure field. The buffer of inet events is also cleared before the initialization can take place.

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

5.2. Closing an 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 signalled so that the thread stops writing an event continuously.
- Stop the thread that controlled the iNet events calling pthread_exit() from the thread and pthread_join() from the main function.
- Delete the semaphore.
- Follow the same steps for GSM finishing

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

5.3. Out of GSM Coverage

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

When the module lacks sufficient coverage to send data to the network, it stores the data in an internal GSM module buffer until adequate coverage is recovered, and then it will send the stored data. The buffer is approximately 1000 bytes long.

In the case that the buffer reaches capacity before coverage is recovered there are two different resolutions, depending on whether the application is using a TCP or UDP protocol:

- If 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.
- If the application uses TCP protocol the application will not receive any event indicating a lost of GSM coverage. Although TCP is a secure protocol with an 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 longer and longer. The application will require another mechanism to realize the loss of coverage.

5.4. 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 remains on, taking into account the following rules on a call arrival:

If a voice call arrives:

- Hang-up and continue 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:

- Hang-up and go on with the INET session.
- Previously it was required to answer data calls to release an INET session.

- Other events do not release the iNET session, including a GPRS_COVERAGE lost. The INET session is internally maintained until the GSM module buffer is full, some bytes could be sent to the GSM "network".(Remembering 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 á TCP protocol, if the user continues to send data, this data will be stored in an internal buffer of the GSM peripheral. Once the coverage is recovered this data is resent.

Many users may not wish to send un-updated data so as to avoid receiving obsolete data; on losing the GPRS coverage the user may not send IP data.

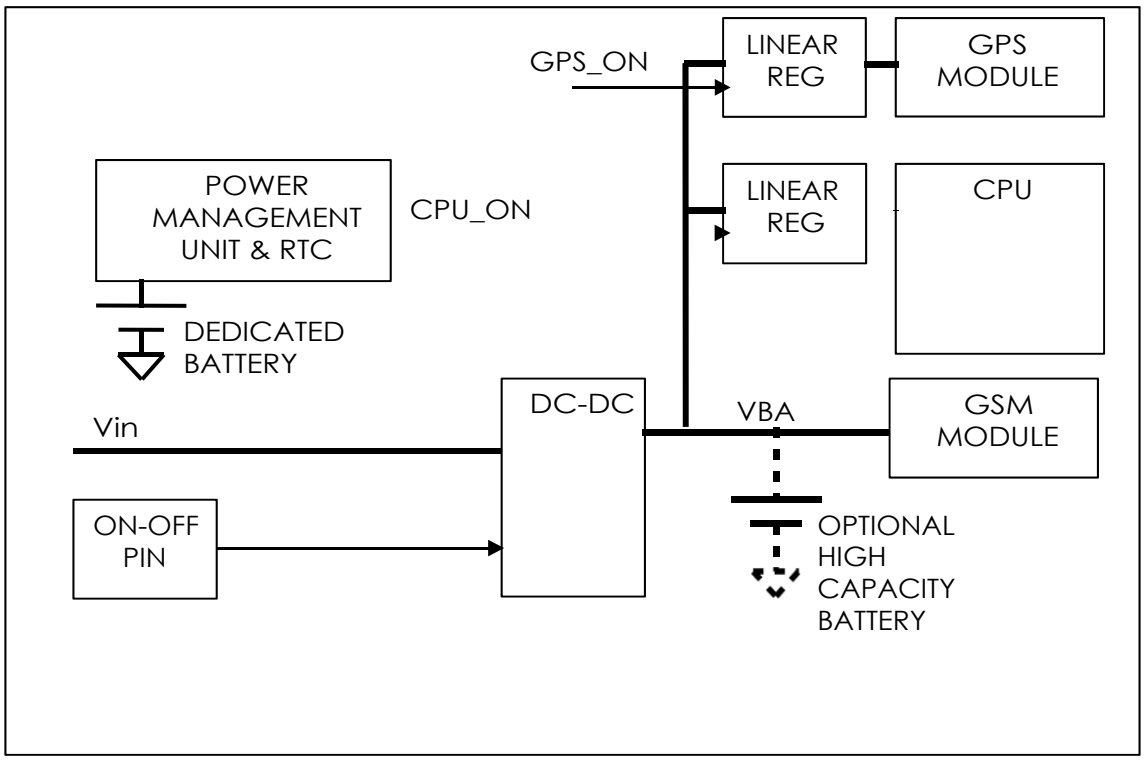
6. CrossLink TG Power Optimization

Power consumption can be reduced in two different methods:

- By reducing microprocessor consumption, in this case customer applications should access the power management functions and select one of the possible low consumption modes.
- By switching off the different modules, for example GSM or GPS.

6.1. Reducing Power Consumption by Switching Off Devices

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



6.1.1. CPU System

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 the 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 just in case it has been 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 I/O library function. It is suggested to switch it off when not in use.

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

6.1.5. Wi-Fi and 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)
```

6.2. Low Power Mode

To decrease the CrossLink TG 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 Wi-Fi, switch them off
- Switch off leds with DIGIO_SetLED_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 a signal to initiate 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 cannot be set as a signal to initiate wake up.

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

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

7. CrossLink TG GPS API

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

7.1. Starting GPS

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

Prior to loading the GPS, RTU and I/O 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 I/Os 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"};           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().

7.2. Shutting Down the GPS

Ending a GPS session takes the steps opposite to the GPS start sequence, so the process must be GPS finalized, then I/O and finally RTU.

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

First of all, the GPS instance is finalized, followed by the RTUControl instance and the I/Os instance respectively.

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

7.3. Retrieving GPS Position

In order to retrieve the GPS receiver position, the library function GPS_GetAllPositionData() must be first 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 will show as 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 masked with the status and the horizontal accuracy that must be met to give the fix as valid. See the API to get further information on the details of 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");)
}
```

7.4. Change Sensitivity Mode

There is a GPS library function set 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 first 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 acquisition times).

Function to change the 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 CrossLink TG is set to High Sensitivity Mode

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'.

7.5. Change the 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 a 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 CrossLink TG for the Dynamic Model is set to **0** (Portable).

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

7.6. Change Static Hold Mode

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

If the speed drops below the defined threshold, the position is kept constant. When the speed rises to a value above double the threshold, the position solution is released again.

Function to change the Static Hold Threshold is as follows:

```
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");)
}
```

Do not set the parameter of the Static Hold Mode too aggressive, as it may degrade the performance of the GPS receiver, when, for example, 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 application's requirements.

The default setting in the CrossLink TG threshold is set to 0m/s

7.7. Navigation Output

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

7.7.1. Retrieve 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");)
}
```

7.7.2. Retrieve 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"
);)
}
```

8. CrossLink TG CAN

The CAN bus is accessed as a socket using the SocketCAN implementation on the Linux kernel. The CAN is accessed as a network interface with the name canx. By default the CrossLink TG has 2 CAN interfaces, can0 and can1.

The CAN driver must be activated first, because at boot time it is not powered in order to improve the overall power consumption of the system. The function DIGIO_Enable_Can() from the I/O 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 both 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 can0 type can bitrate 1000000
```

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

```
#ip link set can0 up
```

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

```
#ip link set can0 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. However, 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 rmessage;

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 for quickly testing the interfaces.

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

```
root@arm:/#apt-get install can-utils
```

To test the transmission and reception of the interfaces, one can employ a quick test by connecting CANH1 to CANH2 and CANL1 to CANL2 in the connector, and sending a CAN frame from can0 interface to can1 interface.

Set candump to listen in can1 interface, redirecting the output to a file:

```
root@arm:/#candump can1 > /tmp/can1.log
```

Then send some frames from can0, which should be received in /tmp/can1.log:

```
root@arm:/#cansend can0 1F334455#1122334455667788
root@arm:/#cansend can0 1F334455#8877665544332211
```

9. CrossLink TG

Fleet Management System (FMS) library libFMS_Module.so.0.0.1 can be used to retrieve information from the different FMS messages available in a CAN bus complying with the FMS standard, without having to work directly with the CAN interface.

- STATUS
- The FMS library makes internal calculations based on the continually received data from the CAN bus in order to provide a base status of the vehicle: OFF, STOPPED, IDLING and MOVING.
- FMS
- FMS is an interface of data from commercial vehicles. The CrossLink TG FMS library provides a structure where the machine data is gathered and relayed directly to the user program.
- TPMS
- TPMS, Tire Pressure Monitoring System, covers the data advertised from the tire sensors of some trucks.
- EBS

- EBS, Electronically controlled Brake System, recieves information from the electronic brakes and can relay their status.
- Check the document D16_9904_FMS_Library.pdf to get more information on the API and all the data structures available to retrieve the data from the CAN bus using the CrossLink TG FMS library.

9.1. Starting and Finalising 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 requires three values to start the communication as expected for the bus where the CrossLink TG has been connected. These values are the bitrate, the CAN interface on the CrossLink TG that is being used and a flag or 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 data variable 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 an additional structure, which also contains various structures for each possible data set 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. Removing TPMS Data

The TPMS data should be removed from the structure when a tire or a group of tires are physically no longer in their former or original place, for example after changing a 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 an additional structure, which also contains various structures containing the data set from the braking system of the vehicle. The structure can be filled by 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. CrossLink TG RS232

The serial interfaces must be programmed using standard Linux functions.

There are three possible serial interfaces in the CrossLink TG 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 to the user,

• 2 serial ports: UART4 with HW flow control with TXD4, RXD4, RTS4, CTS4. UART1 with TXD1 and RXD1. In this case UART5 cannot 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 first enabled.

```
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

Programmers have to use the standard Linux functions to configure and work with the serial interfaces on the CrossLink TG:

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

11. CrossLink TG RS485

CrossLink TG has a RS485 serial interface available at two pins on 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 before using it.

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

11.2. Working with RS485

The standard Linux RS232 functions are used in order to open, read and write to a file descriptor.

As the RS485 is semi-duplex, RS485 has to drive the bus into the 2 possible states: transmitting or receiving. This is done automatically by the Linux driver, so the user can simply read and write in the file descriptor as with a RS232 port. The only need is to set CRTSCTS flag in the mask passed in the configuration.

```
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 | CRTSCTS; newtio.c_iflag
= IGNPAR;
newtio.c_oflag = 0;     // no output modes newtio.c_lflag = 0;          // no
canonical, no echo, ...
newtio.c_cc[VMIN] = 0;
newtio.c_cc[VTIME] = 0;

ReturnCode = cfsetospeed(&newtio, 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 Parameters

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;
}
```

12. CrossLink TG BLUETOOTH

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

12.1. 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 I/O library function.

```
ReturnCode = DIGIO_Enable_Bluetooth (1);
```

12.2. BT Profiles

Supported profiles include; BT: Serial Port (SPP), OBEX Object Push (OPP), OBEX File Transfer (FTP) and Network Access Point (PAN). To retrieve all the available profiles type this command:

```
$ sdptool browse
```

12.2.1. 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 device 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 a connection with the bounded device is made. Following with the connection of the PC example the device can be accessed with a cat command at the shell prompt of the device.

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

The PC will show a message to pair the device, 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 device, the device can be set to listening mode. Then the connection may be started from the PC, and characters may be sent from the PC to the device with hyperterminal and opening the rfcomm0 device.

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

12.2.2. OBEX Object Push (OPP)

This profile is used to facilitate 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 can include calendar items, notes or messages.

Begin by starting the obexftpd daemon in the device.

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

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

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

12.2.3. OBEX File Transfer (FTP)

To transfer ordinary files, from a PC to the device, the obexftpd daemon must be started.

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

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

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

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

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

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

To get a file from the PC to the device the option -g can be used. The file on 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 “get” from the PC.

12.2.4. Network Access Point (PAN)

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

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 pincode, authorize a connection/service request, etc.).
- bt-device allows the connection to remotes devices by their MAC address and service discovery.
- bt-network manages network services (client/server). For example, it allows the user to register server for the provided UUID.

The following commands can 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 may be necessary to assign IP addresses, either static or dynamic ones and enable IP forwarding on the CrossLink TG:

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

12.3. Bluetooth Programming

The Bluetooth communication can be also managed by 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.

13. CrossLink TG Wi-Fi

The Wi-Fi peripheral that the device integrates is handled using the standard methods within 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 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.

13.1. Wi-Fi Activation

Before the use of the Wi-Fi peripheral can be utilised, it is required to switch it on with the enable I/O function.

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

After working with the Wi-Fi module, if the user no longer needs it for services, it can be switched off in order to decrease data usage.

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

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

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

13.2. Useful Wireless Commands

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

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

13.2.1. iw

To get information about 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, MAC address, signal level, encryption mode and the channel used.

This command can be used with the 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 information 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 connected to a DHCP server, it is a good idea to use the DHCP client to properly configure the connection:

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

To disconnect from the network use the disconnect option:

```
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 a host request to transmit data event 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 both 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 response times of around 100 ms. Depending on the application, the power consumption of the Wi-Fi device can be heavily reduced.

Device power save mode can be enabled using the iw command:

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

In power save mode data may be lost and the latency may increase significantly. When using the CrossLink TG as a gateway or using the Wi-Fi in hotspot mode, it is advised to avoid this mode and set it to off.

```
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
```

13.3. WPA Supplicant

The WPA supplicant can be configured to control the roaming and IEEE 802.11 authentication/association of the Wi-Fi. 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 beyond the scope of this document.

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

13.3.1. Connecting to an open network

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

```
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 is set to tell the command what configuration file to look at
- -B is set to run in the background

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 reconnection 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 WPA Authenticators.

13.3.2. Connecting 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 may 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.

13.3.3. Connecting 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"
}
```

13.3.4. Connecting 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"
}
```

13.4. CrossLink TG Wi-Fi AP

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

First enable the Wi-Fi module with I/O 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 the tools, 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 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 the user may 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 this configuration example, the AP is called “owasys” with password “owasys123456789”. 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, the 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 tools and settings arr prepared, the networking service must be restarted, which will set 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 CrossLink TG from a mobile phone or tablet and to open a web site from an HTTP service running in the device itself. If the purpose is to get the CrossLink TG working as router to provide internet access using another interface, ppp0 from 3G module or eth0 from an Ethernet cabled connection, a few commands are still needed.

To enable routing on the device, 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
```

14. System Re-initialisation

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

14.1. GSM Module

Close and boot 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 complete system reboot.

14.2. GPS Module

Stop and boot the 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 resolve the problem then perform a complete system reboot.

14.3. 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 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.auto/softconnect
```

Switch the USB on:

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

14.4. 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 reinitialised.

The safest way to restart the unit is by starting a watchdog and then not setting it to cause the restart.

15. Watchdog

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

15.1. 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
```

15.2. 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 simply 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. Certain options in the unit definition allow the user to configure settings, such as 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 allows the user 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 is an example unit file which 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 a 5min period a new action is taken and the system is quickly rebooted, with all file systems being clean when it boots 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
```

15.2.1. 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 the time of compilation using the -lsystemd option. Legacy code can be patched to support the systemd watchdog logic pointed out above as is shown below.

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

Before performing any actions over the watchdog mechanism, the user must retrieve 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 initialised, 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 the set WATCHDOG_USEC time elapses then the service will reset.

```
// 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");
}
```

15.2.2. Reboot

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 */
…
}
```

16. Further Reading

LinX Software Suite	- LinX Getting Started Guide
Watchdog	- http://opointer.net/blog/projects/watchdog.html
Systemd	- https://www.freedesktop.org/wiki/Software/sys temd/
WPA supplicant	- https://w1.fi/wpa_supplicant/
Linux Wireless	- https://wireless.wiki.kernel.org/en/users/Documentation
BlueZ	- http://bluez.sourceforge.net/
CAN	- http://www.kernel.org/doc/Documentation/networking/can.txt