

OWASYS owa22x Firmware Description

1 General Description of Firmware

In order to manage the platform resources, a complete library of APIs, for GPS management, Internet connection, management of the interfaces, GSM/GPRS functions and other services is made available. Thus, the developer does not have to care about low level hardware drivers and protocols, but focus on his/her application by means of user-friendly APIs.

The main structure of owa22X software system is the following:



Figure 1.1 owa22X software structure

2 Boot Loader

The boot loader is the booting system that manages the kernel and file system flashing.

3 Linux Kernel



This is a Linux standard kernel, version 2.4.18 with MMU option. As this is an standard kernel, PC developed applications are easily made compatible with owa22A platform. Also this kernel can be updated to follow standard kernel changes and news.

4 owasys File System

This file system follows the conventional structure of a Linux system and is shown as follows:

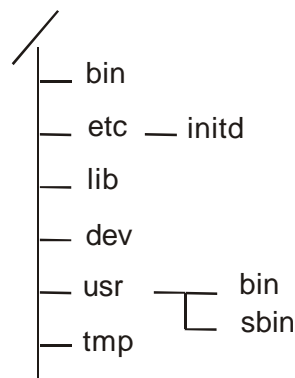


Figure 4.1 File system structure

bin and/sbin: these directories include Linux basic commands for operating system utilities and embedded system shell.

etc: in this directory all configuration files and booting parameters are recorded.

dev: a directory with all peripherals specifications.

tmp: in this directory user can store files during owa22A operation. This information will be lost each time the system boots.

lib: with all Linux standard libraries and APIs libraries.

All this information is stored in Ram memory. As customer has 2Mbytes of free space in RAM memory, files and application can be recorded in this directories structure.

5 User Hard Disk

The user hard disk (flash memory) is mounted in /home directory.:

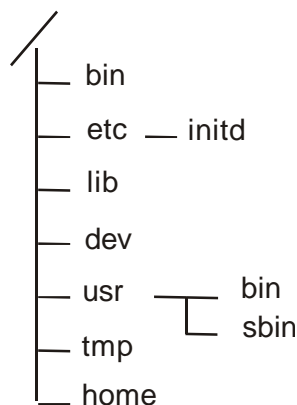


Figure 5.1 User hard disk structure

This directory can be used by user as a 1Mbyte hard disk, so all the stored information in this directory is recoverable when losing device power. In this directory, user can create the desired directory structure and store all kind of files in it.

6 Starting to develop the customer application

Once a cross compiler for ARM microprocessor is installed in a PC with Linux operating system, the user can use this compiler in the desired development environment, for example, **KDeveloper**.

To use the available APIs library functions, add the includes to owa22X files in application header files and compile the application using the provided cross compiler.

owa22X files to be included in customer header files are:

```
<owa2x/RTUControlDefs.h> //Device control
<owa2x/IOs_ModuleDefs.h> //IOs control and management
<owa2x/ GSM_ModuleDefs.h> //GSM managemnet
<owa2x/ GPS_ModuleDefs.h> //GPS control
<owa2x/ INET_ModuleDefs.h> //For Internet connection
<owa2x/ can.h> //CAN bus control
<owa2x/ SerialDefs.h> //Serial ports (RS-232 ans RS485) management
<owa2x/ owa2x_pm.h> //Microprocessor power management
```

All *API module libraries* have been written in C++ Language using the **KDeveloper** environment with *gcc compiler* tool, under *LINUX Red Hat 7.3 Operative System*. It is recommended, to avoid problems from the development environment, to use C or c++ language and the same compiler, and environment when it is possible.

Normally, a Client application is comprised of a main executable program, and a set of libraries. This set of libraries can be of two types: system libraries, both static and dynamically loaded; and *owa22A API module libraries*, always used in a dynamic way.

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

In the example of chapter 7, a simple application is explained. Thus the mechanism of loading and unloading libraries and how to access libraries functions can be understood.

7 Example of a simple application

This is the source code of a simple application. In its main block, it loads an external library (in this case, *RTUControl* module library), gets some pointers to its functions, and unloads the library. Really, this program doesn't do anything useful, but shows the working mechanism that will be the same for all the API module libraries and applications.

```
#include <dlfcn.h> // For use dinamical loading/unloading libraries.
#include <stdlib.h> // For printf() function.
```

```

int main( void)
{
// Variable definitions: [1]
    void *LibHandle = NULL; // Pointer to
store the library handle.

    INT ( *FncGetFirstFreeSignal)( INT *); // Pointer to get
//GetFirstFreeSignal() function.
    INT signal;

// Here loads the library: [2]
    LibHandle = dlopen( "/libs/libRTUControl.so.0.0.0", RTLD_LAZY);
    if( ! LibHandle) {
        printf( "No shared library \"/libs/libRTUControl.so.0.0.0\"
found...");
        return;
    }
// How to get a function pointer: [3]
    FncGetFirstFreeSignal = ( INT ( *) ( INT *)) dlsym( LibHandle,
"GetFirstFreeSignal");
    if( dlerror() != NULL) {
        printf( "No GetFirstFreeSignal () found...\n");
    }
// How to call the function: [4]
    ( *GetFirstFreeSignal>(&signal);
// How to unload the library [5]
    dlclose( wLibHandle);
}

```

There are five different sections in the program (identified by a **[x]** mark):

[1] Variable definition section:

In this section the Client application must define mainly, two sets of variables:

Handles for every API module library to load, each of them with different name.

Pointers, of the same type as API module library's function, for all the functions to be used in the program scope. It is important to indicate that this pointer NEVER can have the same name as the function to be linked (that is because the compiler can misinterpret about internal name tables). In this example, the function's name is GetFirstFreeSignal(), and the pointer's name is FncGetFirstFreeSignal ().

[2]: Loading the library:

At this point, the required library is loaded. Its full path and name must be passed to *dlopen()* function, and it is recommendable to do a load control test before proceeding to get functions pointers.

[3]: Getting pointers to functions:

Now it is the moment to get a pointer for each library's function which will be used in the program. As it can be seen in the code, it is necessary to do a casting for the type of the pointer filled, and it is necessary to call later to `dLError()` function for checking if the pointer is valid or not (checking pointer against value NULL is not valid, because function number 0 can be got with this function...)

[4]: Calling function's pointer:

This is the way of calling a function's pointer, with the name of the function enclosed in a "`(* ...)`" sequence. Although in this case no return error is checked, it is recommended to check if return code of the function is equal to **NO_ERROR** code, indicating a correct execution of the function.

[5]: Unloading the module library:

To save system resources as handler, memory, etc., once a dynamical library is not going to be used again, it must be unloaded. If system resources are not a problem, simply wait to the end of the program for unloading all libraries.

This is the simplest example of client application. More cases of applications will implicate loading and unloading more than only one library, different sets of functions' pointers, multithread, etc. Practice and developments will indicate which method to follow in each case.

8 Loading the customer application

Once the system has booted, the customer application can be copied to `owa22A /home/` directory using *owa2X Family Management Tool* .

Also, the user can instruct the unit to execute the application automatically during start-up, by including a command in a configuration script located in `/home/`. The name of this script is: *userinit.sh*.

For example, if the name of the application program is *demoapp*, this line must be included in the configuration script:

```
#!/bin/sh
echo
/home/demoapp
or
#!/bin/sh
echo
/home/demoapp&
```

9 Available APIs

In this section there is an overview of the available APIs and an explanation of their functionality.

For further information about the APIs and their functions, it is recommended to read the *owa2X Family Programming Reference Manual* and *owa2X Family Software Programming Guide*.

9.1 API for owa22X Control

This API provides a Select Control function to manage synchronous and asynchronous ports, a centralized system time, and a management of available system signals.

- ? Get System Time: `int GetSystemTime(TSYSTEM_TIME *wSystemTime).`
- ? Sleep the process during the programmed time: `INT usecsleep(INT secs, INT usecs)`

9.2 API for GSM/GPRS

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

- ? Make a call (Voice, data, fax...): `INT GSM_Dial(BYTE wType, UCHAR *wBuffer)`
- ? Configure and initialize iNet module: `int iNet_Initialize(void* wConfiguration)`

9.3 API for GPS ¹

Getting the positioning info and sending NMEA commands through the UART, are some developed functions included in this API.

- ? Get the GPS receiver position (latitude and Longitude): `INT GPS_GetPosition_Polling(TGPS_POS *pCurCoords)`
- ? Get an NMEA standard output message: `INT GPS_GetNMEAMsg(MSGNMEA Nmea, CHAR *pBuffer)`

9.4 API for IOs

The functions of this API facilitate the management of owa22X digital and analog I/Os, audio signals, odometer interface...

- ? Enables/Disables the signaling by interruption service for each digital input signal: `int DIGIO_EnableInterruptService(BYTE wInput, BOOL wStatus, pid_t wProcess)`
- ? Gets the value of the digital input signal IN1: `int DIGIO_GetIN1(BYTE *wValue)`

¹ Only for owa22A.

9.5 Power management driver

The power Management driver controls the operating modes of using the standard Linux driver functions in `/dev/owa2x_pm`.

9.6 CAN Bus Driver

This driver allows the customer to use, the CAN Bus as if it were a serial port of 32 channels. This driver must be loaded dynamically. CAN Bus is used through typical device functions (open, close, write, read and ioctl)

9.7 RS485 and RS232 Driver

RS485 driver is used as if it were a serial port with the option to choose if the device is the master or the slave. RS232 driver is user to control the RS232 interface of the DB9 connector and/or the one in the RJ45 connector. Serial Port API monitors continuously ports listening for incoming data and notifies the user when incoming data have been received from one of these interfaces .

```
? Write a buffer through a RS485 port: INT  
Serial_SendRS485Buffer( BYTE wCOMPort, UCHAR  
*wOutBuffer, UINT *wOutBufferLength)
```

9.8 Software Application Notes

In the development kit CD there are available the source files (.cpp and .h) of some example applications.