ACP Message Library
User Guide

Reference: 13007_ACPLIB_UG
Revision: 1.0
Date: 2009-09-04

Revisions

1.0	2009-09-04	Initial release

Copyright Notice and Trade marks

ACP Message Library

User Guide,

Document Version: 1.0

Document, Release Date, 2009-09-04

EDANTECH (ILWICK S.A.),

Av. Libertador 1807,

11800, Montevideo, URUGUAY

Phone#: +598 2 929 0029

Fax#: +598 2 924 8013

e-mail: info@edantech.com

Contents

1 References

[ACP245] ACP 245 – Application Communication Protocol v 1.2.1
http://www.denatran.gov.br/download/ACP_245_V1_2_1_19_05_09.pdf

[ACPLIB_REF] ACP 245 Message Library API Reference.

2 Introduction

This document will give an introduction the ACP 245 Message Library developed by EDANTECH.

In the next sections, we will give you a general overview of the library, an example showing you how to use it, a description of the target platforms, and a brief presentation of the different ways in which you could use it.

3 Overview of the library

The ACP 245 Message Library is one of the core components of the ACP 245 suite developed by EDANTECH.

It is a portable ANSI C library (working under Linux, Windows, and other operating systems), designed primarily for memory constrained devices and embedded systems, but that can also be used as a server side implementation of the ACP 245 protocol.

The library provides a 100% compliant implementation of the ACP 245 protocol version 1.2.2 message formatting rules, both for Telematic Control Units (TCUs) and Service Operators (SOs). It's also being constantly updated to support new versions of the protocol once they are published by the Brazilian National Transit Department (Departamento Nacional de Trânsito, DENATRAN), which is the organization that defines the ACP 245 protocol. Since our engineers participate on the work groups that are defining the ACP 245 protocol, we often have new version implementations ready even before the new one is published by DENATRAN.

The library has been fully tested and validated by a number of device providers and SO, and it's development follows best practices for software quality assurance.

We are aware that C is not always the language of choice for server side products, and, for that reason, we provide a number of bindings that allow developers to use the library from other languages, either with a specific language binding (for example, for Python), or as a Windows DLL that can be linked from most Windows supported languages.

3.1 Basic API usage

In this section, we will give you a quick overview of the main API functions and a basic C example that shows you how a program that uses our library would look like. The complete API is thoroughly documented on the ACP Message Library Reference document on [ACPLIB_REF].

The library API has been designed to make it very easy to write and read ACP messages, hiding protocol details (as for example the usage of more flags or handling of element lengths). Therefore, the API just exports a small number of functions which makes it simple to understand and use. The

following are the four basic functions available on the API:

- `acp_msg_read_data`: reads an ACP message from a byte array.
- `acp_msg_write_data`: writes an ACP message to a byte array.
- `acp_msg_free`: frees a previously read or created message.
- `acp_msg_init`: Initializes a new ACP message.

The following is an example of code that uses those functions. Don't worry if you don't fully understand it now, you will after reading the explanation that follows the code:

```
#include <stdio.h>
#include "acp245.h"
int main(int argc, char** argv) {
  u8 buf[256];
  acp_msg msg;
  acp_msg msg_read;
  e_ret rc;

  acp_init();

  acp_msg_init(&msg, ACP_APP_ID_ALARM, ACP_MSG_TYPE_ALARM_KA);
  rc = acp_msg_write_data(buf, 256, NULL, &msg);
  if (ACP_MSG_OK == rc) {
    printf("Message written OK.\n");
  }
  rc = acp_msg_read_data(buf, 256, NULL, &msg_read);
  if(ACP_MSG_OK == rc) {
    printf("Message Application Id = %x\n", msg_read.hdr.app_id);
    printf("ACP Message Type is: %x\n", msg_read.hdr.type);
  }
}
```

In the example, we include the `acp245.h` library header which gives access to all the exported functions of the library. Then, we define a buffer to store message data (`buf`) two variables to store ACP 245 messages (`msg` and `msg_read`), and a variable to store library functions return codes (`rc`).

The first step in every application using the library, is to call `acp_init`. This function initializes the library, and must be called just once, at the start of the application, before using any of the other functions. In particular, this function will check that a valid license file is present on the installation directory of the ACP 245 library. There's an alternative version, called `acp_init_opts`, which

allows you to specify the license file explicitly by parameter.

> *Be aware that if you don't have a valid license file, all the functions will return an error code and the example will fail to print anything.*

If you are trying to create a new message to later write it to a buffer, you need to first call the `acp_msg_init` function (which is done on line 11). This function receives as parameters the message, the application ID of the message (in this case Theft Alarm) and the message type (a Keep Alive message). Normally, this function only sets everything to zero and sets the application ID and message type fields of the header to the values given by parameter.

As you probably noticed, in the code we used a pair of constants, defined on the API and documented on the reference, in place of the application ID and message type numbers, but you could use the literal numbers instead if you preferred.

After creating a message, you can set the fields you like (depending on it's application ID and message type), and then you can write it to a buffer using `acp_msg_write_data`. This function receives the buffer where you want to write the message (`buf`), the maximum size of this buffer, for security (`256` in this case), an optional variable to store the number of bytes written to the buffer (in this case is not given, that's why it's `NULL`), and the message to write (`msg`). The function returns if the message was correctly written or an error that indicates why the message couldn't be written. If everything went OK, the buffer now contains the binary representation of the ACP message sent by parameter.

Then on line 16, we read the buffer, that contains the message we wrote on line 12, by calling `acp_msg_read`. This function receives the buffer to read from (`buf`), the size of the buffer (256), an optional variable to store the number of bytes read from the buffer (`NULL` in this case), and the message where to store the read message.

In those few lines, we created, wrote and read an ACP message, which is basically for what you will be using the library.

As you can see, there are no many secrets to use the API. The most complex part is knowing the structure if the ACP message variable. If you noticed, on line 18 and 19, to access the application ID and message type fields of the message header, you must write `msg.hdr.app_id` and `msg.hdr.type` respectively. The same applies to other fields, for example, on a Function Command message, to access the TCU manufacturer identifier of the ACP 245 Version Element, you would have to write `msg.data.func_cmd.version.tcu_manufacturer`, a somewhat long line. We have tried to keep the names and field navigation as intuitive as possible, but even so, the message structure of the messages is also fully documented on the API specification.

3.2 Notes on the design of the library

There are a number of design decisions that we took while developing the library. In this section, we discuss some of them in order to give you a better understanding of how it works. You can skip this section if you just want to use the library.

For embedded devices, is often preferable to have predictable and bounded memory consumption

than using dynamic memory allocation, deep buried in internal functions. Dynamic memory allocation makes it harder to know how much memory will a program consume, and can also cause memory exhaustion because of memory fragmentation over a long period of time or when there isn't much memory available. Also, memory handling functions are sometimes different on each target platform (`malloc` is *not* universal).

For that reason, the library tries to avoid using dynamic memory allocation whenever possible, and provides ways to hook a custom memory allocator when there are no other choices (for example, to handle variable length strings or binary data).

That is why functions like `acp_msg_init` do not actually allocate memory for a message, but instead expects to receive an already allocated message (usually defined as a local variable, and allocated on the stack, or as an static variable, allocated on the BSS segment). You can allocate the message however you want, the library will not take that choice for you.

The `acp_msg_init` function doesn't do much actually, it just zeroes the message. Its designed just in case we need to perform some other initialization in the future, so we require you to use it now (we don't want to tell you *then* that you have to change your code to use a new version).

In the same way, `acp_msg_free` does not free the allocated message passed by parameter. It only frees internal fields of the message in case they have been allocated by the library (for example, for variable length strings).

You can't normally hook a custom made allocator to the ACP library. To do so, you need a set of libraries developed by EDANTECH called `e_libs`. These libraries allows a programmer to use the ACP library in advanced ways, hooking a memory allocator, or using a secure `e_buff` on the message read/write functions. We don't provide this library packed with the ACP library by default, because it would require you to learn a much more complex API, and you will probably don't need it for the most common cases.

The same arguments for memory consumption predictability hold for the decision of using a single `acp_msg` structure, which includes an `union` of all the possible types of ACP messages. This causes every message (without considering variable length fields) to use the same amount of memory, which doesn't sound very reasonable if you consider that in ACP there are some small messages (like a Keep Alive) and others that are thrice as large.

But, if you need to calculate a bound for memory consumption, you have to think of the worst case scenario. In the worst case, all message will be as large as they are supported by the library. So, from our point of view, in an embedded device, if you waste some memory bytes, but in exchange you avoid dynamic memory allocation and improve the performance of the application, that is a plus.

On the server side, we consider that this is not a problem, since wasting 100 bytes per message in a server with xGB of RAM memory it's not an issue (or starts being an issue when handling over 100000 messages at a time, at which point per message performance is probably more important).

4 Known limitations

4.1 New elements and fields

At the time, the library is able to parse every valid ACP 245 message, as defined on [ACP245] (otherwise, it's a bug, so please inform us). However, this doesn't mean it can *interpret* every valid message.

ACP 245 is an extensible protocol, new fields and new elements are supposed to appear in the future. If the library finds these extended elements, it will just ignore them. A new ACP 245 field or element won't break the library, but a new version of it will be needed if you want to access them.

4.2 Element truncation

Also, the library does not allow you to write every possible valid ACP 245 message. ACP 245 truncation rules allows every element to be truncated at arbitrary positions (for example, you could send a Version Element with only two bytes, truncating the software and hardware release fields). The library does not allow you to truncate elements at every possible position, but does provides way of truncating the most common cases. For example, it's possible to truncate the Location Element so it doesn't include the Previous GPS Raw Data, Delta Area Coding and Dead Reckoning elements.

To see which elements or fields can be truncated, you should check for the existence of a `presence` field on the element. This field (or fields called `<field_name>_present`) allows you to truncate a element by setting this field to a specific value (documented on the reference, but most of the times, to the constant `ACP_EL_NOT_PRESENT`). You can expect the library to allow you to read and write, at least, every possible message conforming to the [ACP245] section 13.

4.3 Limits on the number of elements

As explained on section 3.2, to avoid dynamic memory allocation, the length of some elements are constrained to store a reasonable maximum of values. As explained previously, if a message has more values than the supported ones, they will be skipped but the message will be parsed anyway.

The following are the current limits:

- Maximum number of breakdown status fields: 5
- Maximum number of satellites ID on location element: 12
- Maximum number of location delta items: 10

Other limits, implicit in the ACP 245 specification, are also enforced (for example, a maximum length of 65535 bytes for each ACP message), but in those cases the message parsing will fail since the message does not conform to the specification.

5 Installation and usage

The library comes packaged in a way suitable to your target platform.

For Windows, we provide an typical installer that will install all the required files on your hard drive.

For Linux, we generally provide an RPM package for Red Hat based distributions (Fedora, CentOS, RHEL, etc.), but we can also provide you a package for other distributions, upon request.

Depending on your development environment, you will need to point your project includes and library search paths to the installed directory, and add the required linker flags to link your code against the library. If you need assistance to configure you environment to use the library, please feel free to contact us.

5.1 Target platforms

We provide binary versions of the library for Windows (as a DLL and as an static library), Linux (as a shared dynamic library) and Wavecom OpenAT (as a binary plug-in). The release package includes the library, it's headers, and all the relevant documentation to be able to compile and link your own code with it.

The library has been designed to be easily ported to other platforms, so we can provided versions for other operating systems on per-case basis. Please, feel free to contact us through EDANTECH web page for additional information.

5.2 Using the library from other languages

The library can be used from C or C++ code without any additional software.

For other languages, if they allow you to access a C shared library directly (for example, a DLL on Windows), you can link against the library without modifications. However, be aware that you will also need to adapt the language internal data types to the ones used by the library.

Alternatively, we provide bindings to other languages separately. At this time, the only currently supported language is Python, but we are working on C# and Java versions also.

These bindings allow you to use the library from your preferred language more naturally than calling a linked C function, since we take the work to wrap and adapt the API interface so it looks and feel like it was designed for the given language. For example, on the Python version, to create a Keep Alive message and get its binary representation you can just write:

```
AlarmKA().as_bytes()
```

or:

```
buffer = msg_write(AlarmKA())
```

Which is much shorter than the C alternative.

5.3 Using the library on Windows platforms

We provide an installer that will guide you through the installation process. By default, the software will be installed to `c:\Program Files\Edantech\ACP245`, where `Program Files` is the language specific folder for program files.

After installing the software, two environment variables will be defined:

- `E_ACP245_PATH`: references the ACP 245 installation directory.
- `E_ACP245_LICENSE`: references the location of the license file.

You can use `E_ACP245_PATH` in your projects to include the required header files and libraries to compile and link an application. We provide a sample application for Visual Studio C++ 2008 that uses this variable to define the additional include directories (`$(E_ACP245_PATH)/include`) and included libraries (`$(E_ACP245_PATH)/acp245.lib`). Alternatively, you can just copy all the headers and libraries to the project directory and reference them directly from your project.

To run the application, you will need to include the `acp245.dll` in its working directory. In our sample application, we use '`xcopy`' to copy the DLL from the installation directory to the project working directory. You can do the same, or, alternatively, copy the DLL directly into your project directory by hand.

5.3.1 Installing a license file

Instructions will be provided when you purchase the library, but you can install a license file by copying the file to the library installation directory under the name '`license.sig`'.

If `acp_init()` returns `ACP_INIT_OK`, it means that the license file is installed in the right place, and it's valid.

6 Contacting support

Support is provided under the following conditions:

- An incident must be reported by the Customer to Edantech using email, phone or an alternative mechanism proposed by Edantech (ie. Support. Web site). Contact information will be provided on the commercial proposal.

  Email contact information: support@edantech.com
- After the incident is reported, Edantech is bound to answer and solve the incident in the terms specified on the commercial proposal (Supplier Statement of Work or Services Contract document).
- Edantech will only resolve incidents relating to the system described on this document or on the base software used on a Edantech installed equipment required to support the system (operating system, database systems, etc.).
- All support will be provided remotely, by establishing a SSH connection to the equipment running the system.
- Once the equipment is installed at the Customer site, network and hardware issues that impede Edantech for connecting to the server must be fixed by the Customer (by reinstalling or replacing the equipment, or fixing their network configuration).