Mobile: Controlling Mobile Phones and 3G Keys from Erlang

Organisation: Copyright (C) 2019-2021 Olivier Boudeville

Contact: about (dash) mobile (at) esperide (dot) com

Creation date: Sunday, March 3, 2019

Lastly updated: Wednesday, June 16, 2021

Version: 1.0.4

Status: Stable

Dedication: Users and maintainers of the `Mobile` library.

Abstract: The role of the `Mobile` library is to control mobile phones from Erlang. A typical use case is to send SMS from Erlang thanks to a connected 3G USB key.

We present here a short overview of these services, to introduce them to newcomers. The next level of information is either to browse the Mobile API documentation or simply to read the corresponding source files, which are intensely commented and generally straightforward.

The latest version of this documentation is to be found at the official Mobile website (`http://mobile.esperide.org`).
The documentation is also mirrored here.

Table of Contents

Overview

A typical use-case is when one wants to **send SMS from a gateway** (server), for example in order to perform home automation, possibly together with all kinds of fancy personal services (event reminder, UPS notifications, etc.).
For that, in addition to Ceylan-Mobile and its software dependencies, one would need here:

- an **USB-connected 3G device** (typically an older, dedicated phone or, more preferably, a suitable key)
- a **SIM card** enabling the use of the 3G network of a telecom carrier of one's choice
- a proper **operating-system support** thereof (a GNU/Linux box, most probably)

All these points are detailed below.
An alternate, less satisfying solution[1] could be to use the services of a SMS provider through a web gateway; one may look at sms_utils.erl for a basic example of it.
The solution presented here relies on the Gammu library for the actual control of the 3G device, and on Ceylan-Seaplus to make it available to Erlang.
The project repository is located here.

Securing the Prerequisites

Please read the full document prior to making operative choices, since iterations, trials and errors will probably have to be performed (before, hopefully, succeeding ultimately).

Carrier Subscription: Getting a relevant SIM Card first

A 3G device without a companion SIM card would not be of much use.
Finding a good mobile package is quite country-specific. For example French customers might enjoy rather inexpensive, if not free, options.
As we want an automated use of this SIM card (through the 3G device selected in the next section), one should ensure that, in the card's configuration, the request for the PIN code has been disabled.
This can be done by inserting the SIM card in a mobile phone, and, through the settings, disabling once for all said verification. This may also be a good way of checking whether the SIM card works properly before hacking around.

Relying on a Suitable USB Mobile Phone or 3G Key

Basically, one just has to insert the SIM card, connect the device to the computer and start sending SMS to friends.
Well, no. Welcome to a surprising mess instead!

[1] Not nearly as fun: more constraints (ex: credit expiration), probably more expensive, and of course the SMS sending system cannot be used to notify for example that your Internet connection was lost, possibly cut off by burglars...

First of all, as we understand it, mobiles are far less appropriate than 3G keys for this exercice (they are not well supported, they tend to enter various sleep modes), but be reassured that your mileage may vary with keys as well.
In order to select a proper device (phone or key), first of all one should look at the Gammu Phone Database (which, despite its name, lists keys as well).
We tried to favour the *acknowledged* entries in this database, moreover the ones with multiple success reports.
For the record, we selected (before the controversy) only Huawei chips (supposedly becoming, or having been, for better or for worse, a de facto standard) of various quite widespread offers, and bought 4 secondhand 3G keys of different models[2], which we named that way:

- `K3G-1`: a black and orange generic model (no specific brand apparently), labelled HSDPA, with a Huawei E169 chip
- `K3G-2` : white (with a green LED, invisible unless lit), from a former operator, based on a Huawei E170
- `K3G-3` : white, from another former operator, based on a Huawei E172 (labelled as E1752); can host an additional MicroSD card
- `K3G-4` : white, from same former operator as `K3G-2`, based on Huawei E180 (firmware `11.104.16.01.00`), with a rotating USB connection; can also host an additional MicroSD card

To anticipate a bit:

- we have been able to make good use of `K3G-2` (which became our "reference" key, used operationally) and `K3G-4` (kept as a spare key, should the previous one fail)
- we have not been able to durably use `K3G-1` and `K3G-4` (notably: they were regularly not responding anymore after a few, normal interactions)

So, from then on, we will mostly consider here the use of `K3G-2`.
Finally, one should ensure that one's 3G device is not locked to a specific network or carrier. In most cases the device will have to be unlocked, so that it can accept SIM cards issued by any operator (and not just the one devices are generally bundled with).
We were told that our 4 keys were unlocked, yet none of them seemed to properly work on Linux (blocking at various steps, like when fetching their IMEI) until we tried to install them on a Windows box and also to unlock them.
Not sure which operation unblocked them, as most of the attempted operations reportedly failed or could not be properly interpreted in terms of result. This online calculator for Huawei chips seemed to work (giving a NCK unlocking code and another one for the flash operation), even if it is difficult to assess whether the use of any actual code really succeeded.
We were not so keen on installing third-party, untrusted software on said Windows box (even installing the driver located on their ROM appearing as a

[2]For a whopping expensive bill of 10 euros.

mass storage is somewhat unpleasant), but flashing tools are required whenever having to unlock.
For them, Sandboxie or similar may be used in order to isolate, at least to some extent, the various software that one may try in one's quest for a correctly-behaving 3G key. At least for us, quite frequently (even without sandboxing) Windows was not even able to detect that such keys were inserted.
More generally, various problems might explain why a 3G key is misbehaving (i.e. does not seem able to operate, at least not durably), including failed unlocking, buggy firmware versions and alike, and other issues discussed at later steps.
Good luck to you!

USB Supply & Connection

Once one managed to correctly put the SIM card in the right format (normal, micro or nano) in said device, a proper USB cable shall be used to interlink, say, the residential server and the 3G phone, whereas a 3G key could be directly connected to a computer port.
A problem might be that the device could end up being insufficiently powered (ex: weaker USB port, longer USB cable). Some people use a separately-powered USB hub, to compensate for computer USB ports that would not be powerful enough. We never experienced that problem, though.

Operating-System Support & Configuration

That's the main part. We prefer using GNU/Linux, typically Arch Linux, taken consistently as a reference here.

Kernel Modules As always, maybe new kernel modules will have to be dynamically loaded; so, should the kernel have been updated since last boot, reboot first, otherwise the (newer) modules will not match the currently running, older kernel.
Before first inserting a 3G device, we advise to record the already loaded kernel modules, in order to detect the additional ones that are needed by said device. For example, as root:

```
$ lsmod > ~/lsmod-before.txt
# Connect 3G device and wait a bit (ex: LED blinking)
$ lsmod > ~/lsmod-after.txt
$ diff ~/lsmod-before.txt ~/lsmod-after.txt
```

In our case, the `option` and `usb_wwan` modules were loaded, so we ensured that, from now then, they were automatically loaded at boot (to avoid that a later kernel update block their loading), by creating a `/etc/modules-load.d/for-3g-keys.conf` file with following content (just one module per line):

```
option
usb_wwan
```

USB Identifiers & Mode Switches The connected key will then appear as a USB device, with a USB ID in the form of `vendor_id:product_id`, which can be for example be obtained thanks to `lsusb`.
For example, at connection, our `K3G-1` key will appear as:

```
Bus 003 Device 096: ID 12d1:141b Huawei Technologies Co., Ltd.
```

Unfortunately, this does not correspond to a (3G) modem, but to a mass storage: most keys will be detected as such (ex: as CD-ROM players), as they comprise a built-in ROM (if not an additional MicroSD slot) where typically the vendor (Windows-only) drivers are located. These drivers, once installed, will switch the operating mode of their key, from mass storage to modem. Here such drivers are of no use, and what we want is to switch the keys to modems.
For that, as root, following package shall be installed first:

```
$ pacman -Sy usb_modeswitch
```

It should install a udev rule file (`/usr/lib/udev/rules.d/40-usb_modeswitch.rules`) suitable for most 3G devices (otherwise you will have to enrich it).
Then the key should be plugged again; the vendor identifier is not expected to change, but the product identifier should, so that the key is now considered as a modem. `journatctl -xe` should allow to check.
For example, once connected, our `K3G-1` key is to spontaneously switch (almost immediately) from the previous:

```
Bus 003 Device 096: ID 12d1:141b Huawei Technologies Co., Ltd.
```

to a newer:

```
Bus 003 Device 060: ID 12d1:1446 Huawei Technologies Co., Ltd. HSPA modem
```

Bye bye mass storage, hello modem!
This mode switch can also be done manually, like in:

```
$ sudo usb_modeswitch --verbose -J -v 0x12d1 -p 0x1446
```

`lsusb` would then ultimately report, for `K3G-2`:

```
Bus 002 Device 003: ID 12d1:1003 Huawei Technologies Co., Ltd. E220 HSDPA Modem / E23(
```

Managing /dev/ttyUSB* entries Should the relevant kernel modules be available, at least one entry shall appear as `/dev/ttyUSB*` when a USB 3G device is connected and correctly recognised by the system.
For example, `/dev/ttyUSB0`, `/dev/ttyUSB1`, `/dev/ttyUSB2` and `/dev/ttyUSB3` may appear, sometimes only after a few seconds. Only a subset of them will be useable.
A tests with Gammu will tell them apart.
So, first, that tool shall be installed.
One's distribution should provide it, as it is fairly standard:

```
$ pacman gammu
```

It should notably provide the Gammu library (ex: in `/usr/lib64/libGammu.so.8.1.40.0`) and the various Gammu header (ex: the `/usr/include/gammu/gammu/gammu*.h`).
With this package comes the `/usr/bin/gammu` executable (of course relying on said library), which is useful to test one's configuration.
The executable may read its test configuration from `/etc/gammurc`, whose content may be, for example in order to test whether `/dev/ttyUSB1` (the tty we use for `K3G-2`) is relevant:

```
[gammu]
device = /dev/ttyUSB1
connection = at
logfile = /var/log/gammu-ceylan.log
logformat = textalldate
```

To check whether one's 3G device is supported by the system, one may use:

```
$ gammu --identify
```

Note that each operation is bound to last for a few (around 3-4) seconds before returning.
Hopefully one will not end up with following information returned:

```
Can not access SIM card.
```

or even worse:

```
No response in specified timeout. Probably phone not connected.
```

but, after maybe some trials and errors (start by testing various `/dev/ttyUSB*` devices and `connection` settings), with something like (IDs edited for obvious reasons):

```
Device               : /dev/ttyUSB1
Manufacturer         : Huawei
Model                : E17X (E17X)
Firmware             : 11.304.20.01.00
IMEI                 : XXXXXXXXXXXXXXX
SIM IMSI             : XXXXXXXXXXXXXXX
```

Congratulations, the operating system supports, at least to some extent, your device!
A problem will be afterwards that the numbers involved in the tty pseudofiles are bound to change - based on, notably, the use of the other USB ports.
So a better approach will be to use `udev` in order to give them a stable name, such as `/dev/ttyUSB-my-3G-key`, thanks to a rule typically written in `/etc/udev/rules.d/98-usb-my-3G-key.rules`, whose content would be:

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="12d1", ATTRS{idProduct}=="1446", \
   SYMLINK+="ttyUSB-my-3G-key"
```

Then one should run:

```
$ udevadm control --reload-rules && udevadm trigger
```

One the key is inserted again, it should be available with its new, stable name.
It can be checked more in-depth:

```
$ udevadm info --query=all --name=ttyUSB-my-3G-key
 P: /devices/pci0000:00/0000:00:14.0/usb2/2-3/2-3:1.1/ttyUSB1/tty/ttyUSB1
 N: ttyUSB1
 L: 0
 S: ttyUSB-my-3G-key
 S: serial/by-path/pci-0000:00:14.0-usb-0:3:1.1-port0
 S: serial/by-id/usb-HUAWEI_Technologies_HUAWEI_Mobile-if01-port0
 [...]
```

One may ensure thanks to `fuser` that no component (Network Manager or alike) took control of it:

```
$ fuser -va /dev/ttyUSB-my-3G-key
                     USER        PID ACCESS COMMAND
/dev/ttyUSB1:
```

(as intended, no controller process in the way here)
To interact with such a `/dev/ttyUSB*` file, the user (let's name him `sheldon`) must be in the `uucp` group; so, as root:

```
$ gpasswd -a sheldon uucp
```

And, as `sheldon`:

```
$ newgrp uucp
```

We re-use that group so that this non-privileged user can also write in the Gammu log file we specified; as root:

```
$ touch /var/log/gammu-ceylan.log
$ chgrp uucp /var/log/gammu-ceylan.log
```

This should be sufficient so that `sheldon` is able to send SMS, not involving `root` anymore in the process.

Wrapping-up Telecom Configuration Now, with that user, is time for a bit of configuration before testing.
One may use:

```
$ gammu getsecuritystatus
```

to ensure that no PIN code is required before using the 3G device (hence expecting as answer: `Nothing to enter.`).
Various calls can be made in order to convince oneself that the key operate properly:

```
$ gammu battery
Battery level        : 0 percent
Charge state         : battery connected and is being charged

$ gammu getallsms
0 SMS parts in 0 SMS sequences
```

The SMSC number of the carrier having issued one's SIM card must be set before any actual SMS sending:

```
$ gammu setsmsc 1 "+33695000XYZ"
```

Should this operation fail, it may be a sign that the 3G device is still locked. This can be checked:

```
$ gammu getsmsc
Location             : 1
Number               : "+33695000XYZ"
Default number       : ""
Format               : Text
Validity             : Maximum time
```

Then a SMS can be sent, assuming `TARGET_NUMBER` has been set to some sensible number (like one's mobile), and root is used at first to overcome any permission issue:

```
$ gammu sendsms TEXT ${TARGET_NUMBER} -text "Hello world!"
If you want break, press Ctrl+C...
Sending SMS 1/1....waiting for network answer..OK, message reference=50
```

As the SMSC has just been set previously, one should not get `Failed to get SMSC number from phone`. This can be monitored:

```
$ gammu monitor 1
Press Ctrl+C to break...
Entering monitor mode...

Enabling info about incoming SMS    : No error.
SMS message received
[...]
We already have one pending, ignoring this one!
SMS message received
Enabling info about incoming CB     : Security error. Maybe no PIN?
Enabling info about calls           : No error.
Enabling info about USSD            : No error.
SIM phonebook        :   0 used, 250 free
Dialled numbers      :  10 used,   0 free
Received numbers     :   0 used,  10 free
Missed numbers       :   0 used,  10 free
```

```
Own numbers           :   1 used,   4 free
Phone phonebook       :   0 used, 100 free
Battery level         : 0 percent
Charge state          : battery connected and is being charged
Signal strength       : -51 dBm
Network level         : 100 percent
SIM SMS status        : 9 used, 0 unread, 50 locations
Phone SMS status      : 0 used, 0 unread, 255 locations
Network state         : home network
Network               : 208 15 (XXX Mobile, France), LAC F8F, CID XYZ
Packet network state : home network
Packet network        : 208 15 (XXX Mobile, France), LAC F8F, CID UVW
GPRS                  : attached
Location 4, folder "Inbox", SIM memory, Inbox folder
SMS message
SMSC number           : "+33695000XYZ"
Sent                  : Sat Dec 22 21:22:14 2018 +0100
Coding                : Default GSM alphabet (no compression)
Remote number         : "+XXXXXXX"
Status                : UnRead
[...]
Leaving monitor mode...
```

Then the same could be attempted with this time a non-privileged user (ex: the previous `sheldon` one). If the Gammu `sendsms` command fails with `"Can not open specified file"`, probably that the permissions onto the log file whose path is specified in the Gammu configuration file have not been appropriately updated (see the `uucp` group above).
Once successful, one will be able to send SMS back and forth between the 3G device and "normal" phones:

```
$ gammu getallsms
```

With this first support, one will be able to fight encodings (ex: for special characters), SMS parts (ex: for messages too large for a single SMS) and sequences. MMS should provide a lot of fun too.
Currently, with Ceylan-Mobile one is able to fetch various information from the device, and to send SMS (regular or multipart ones, with GSM 7bit encoding or with UCS-2 one, of various SMS classes), knowing that all settings (except the message itself and the recipient number) can be transparently managed by Ceylan-Mobile. See this example as a first guideline.

Software Prerequisites

Ceylan-Mobile relies on Ceylan-Seaplus, which itself relies on Ceylan-Myriad. All three of them rely on Erlang (for the user API) and on C (for the library driver), which must therefore be both available.
We prefer using GNU/Linux, sticking to the latest stable release of Erlang, and building it from sources, thanks to GNU `make`.

Erlang Environment Refer to the corresponding Myriad prerequisite section for more precise guidelines, knowing that Ceylan-Mobile does not need modules with conditional support such as `crypto` or `wx`.

C Environment One may use a recent enough version of GCC (ex: `pacman -Sy gcc`).

Gammu Conventions The Gammu configuration file will be searched, on POSIX systems, first as `~/.gammurc`, then as `/etc/gammurc`.
For debugging purposes, using the `dummy` driver is quite convenient.
So for example one could have following content for `/etc/gammurc`:

```
[gammu]
model = dummy
connection = none
device = /tmp/gammu-dummy-device
```

Create that directory (as the user to make use of Gammu) first:

```
$ mkdir /tmp/gammu-dummy-device
```

Otherwise you get: `you don't have the required permission.`.
It will populate this directory with data faking a real phone:

```
/tmp/gammu-dummy-device
--- alarm
--- calendar
--- fs
|__ |-- incoming
--- note
--- operations.log
--- pbk
|__ --- DC
|__ --- MC
|__ --- ME
|__ --- RC
|__ |__ SM
--- sms
|__ --- 1
|__ --- 2
|__ --- 3
|__ --- 4
|__ |__ 5
|__ todo
```

Myriad, Seaplus and Mobile Once proper Erlang and C environments are available and Gammu is setup, the Ceylan-Myriad repository should be cloned and built, before doing the same with the Ceylan-Seaplus repository and then this Ceylan-Mobile repository. This can be done directly (manually) or thanks to rebar3 (automatically).
Then one will be able to enjoy using one's mobile from Erlang.

Using Cutting-Edge GIT Once Erlang is available, a manual install should be just a matter of executing:

```
$ git clone https://github.com/Olivier-Boudeville/Ceylan-Myriad myriad
$ cd myriad && make all && cd ..
$ git clone https://github.com/Olivier-Boudeville/Ceylan-Seaplus seaplus
$ cd seaplus && make all && cd ..
$ git clone https://github.com/Olivier-Boudeville/Ceylan-Mobile mobile
$ cd mobile && make all
```

(for OTP compliance, using short names, such as `myriad`, `seaplus` and `mobile`, for clones rather than long ones, such as `Ceylan-Myriad`, `Ceylan-Seaplus` and `Ceylan-Mobile`, is recommended)

Using Rebar3 The usual rebar3 machinery is in place and functional, so the Mobile prerequisites (Myriad and Seaplus) and Mobile itself can be obtained simply thanks to:

```
$ git clone https://github.com/Olivier-Boudeville/Ceylan-Mobile.git mobile
$ cd mobile
$ rebar3 compile
```

Then Mobile and its tests shall be ready for a successful execution. Note that rebar3 is an alternate way of building Mobile, as one may rely directly on our make-based system instead.

Testing Ceylan-Mobile To test the current functional coverage, once the prerequisites (Myriad and Seaplus) and Mobile itself have been secured (for that refer to either Using Cutting-Edge GIT or Using Rebar3), one may run mobile_test.erl; from the root of the `Ceylan-Mobile` clone (once built, and assuming here using the `dummy` Gammu driver - so that the test can be run even if having no 3G device):

```
$ cd test
$ make mobile-test
       Running unitary test mobile_run (second form) from mobile_test mobile.beam
 --> Testing module mobile_test.

 Testing the Ceylan-Mobile service.
 Back-end information: {gammu,{1,40,0}}.
 Device manufacturer: Gammu.
 Device model: Dummy.
 Firmware information: revision is '1.40.0', date is '20150101' and revision number is
 IMEI code: '999999999999999'.
 Hardware information: 'FOO DUMMY BAR'.
 IMSI code: '994299429942994'.
 Signal quality: signal strength is 42 dBm (42%), error rate is 0%.
 [...]
```

One may also have a look at the resulting Seaplus log (ex: `seaplus-driver.27168.log`; timestamps removed for terseness):

```
[debug] Starting Seaplus session...
[debug] Starting the Seaplus C driver, with a buffer of 32768 bytes.
[trace] At start-up: currently allocated blocks: 0; length of freelist: 0.
[trace] Driver started.
[debug] Starting Gammu.
[debug] Executing get_backend_information/0.
[debug] Executing get_device_manufacturer/0.
[debug] Executing get_device_model/0.
[debug] Executing get_firmware_information/0.
[debug] Executing get_imei_code/0.
[debug] Executing get_hardware_information/0.
[debug] Executing get_imsi_code/0.
[debug] Executing get_signal_quality/0.
[...]
[debug] Stopping Gammu.
[debug] Stopping the Seaplus C driver.
[trace] At stop: currently allocated blocks: 0; length of freelist: 0.
[debug] Stopping Seaplus session.
```

Note

Mobile is built and tested at each commit through continuous integration, and the same holds for its prerequisites (Myriad and Seaplus), so in terms of usability, some confidence exists. However such test-beds lack an actual 3G key and often Gammu, so these tests cover mostly the build of Ceylan-Mobile.

Some SMS-related General Information

The text to be sent as a SMS must be somehow encoded in messages. Either the default, very limited alphabet of 7bit encoding can be used, and then a single, regular SMS will contain up to 160 characters (knowing that the `|^€{}[]\` characters will have to be escaped and thus will count for 2 characters with this encoding), or at least one character does not belong to that alphabet and then the Unicode UCS-2 encoding will have to be used, and then only 70 characters will fit in that SMS.
Should the message be longer than what a single SMS can carry for the relevant encoding, a multi-part SMS shall be used: the text will be split into as many SMS as needed (at least, up to 255 of them, each with a reduced per-SMS payload due to an UDH header being needed; with the 7bit encoding: 153 characters per SMS; with UCS-2: 67 of them), and they will be sent as separate SMS. The receiver is expected to decode these headers, reassemble the messages correctly and present them as if they were a single, longer SMS.
Ceylan-Mobile automatically detects the relevant encoding and type (single/multiple) parts needed; the user just has to specify the text message that shall be sent.
See also a Free Online SMS Length Calculator.

Issues & Planned Enhancements

The coverage of the Gammu APIs could be increased (not specifically tricky, just time-consuming).
Notably:

- a check whether phone needs to enter some PIN could be added
- a support to accept/deny the receiving of SMS could be done based on the SMSC and/or mobile number of the sender
- auto-hang up should a call be made to the 3G device (rather than letting the caller leave a message in the voice mail, if any)
- SMS delivery reports could be requested and checked.

Support

Bugs, questions, remarks, patches, requests for enhancements, etc. are to be sent through the project interface, or directly at the email address mentioned at the beginning of this document.

Ceylan-Mobile Inner Workings

Mobile relies on:

- libGammu (GPLv2), for the actual mobile phone support
- Ceylan-Seaplus (LGPLv3), for the integration of the previous library to Erlang

Ceylan-Mobile links directly to (lower-level) Gammu library services, instead of using the Gammu SMSD daemon, as it provides similar features, such as driving the mobile-side operations and polling it for incoming events.
Ceylan-Mobile respects the way Gammu searches for, and reads, its configuration file (no change needed, the same configuration can be used on the command-line and with Ceylan-Mobile).
Internally, Gammu uses state machines.
Not specifically used/supported: WAP, FM stations, GPRS access points, MMS, SyncML, phonebooks, calendars, alarms, TO-DO lists, notes, profiles, chats, voice mailboxes, vCards, security (PIN, PIN2, PUK, PUK2), ringtones, JAD files, voice call management, cell broadcast, USSD, callbacks, backups, etc.; inspiration could be found in `gammu/smsd/core.c` (ex: `SMSD_ReadDeleteSMS/1`).
Please feel free to enrich Ceylan-Mobile!
A source of inspiration has also been python-gammu.

A Few Information Pointers

About 3G Devices

- USB 3G Modem, by Arch Linux
- USB_ModeSwitch, by Gentoo Linux
- About Huawei E173D (and Linux)
- in French: with a Raspberry Pi

About Gammu

- libGammu C API
- dummy driver

About SMS

- IMSI: identifier of a SIM card, i.e. a 64-bit field designating a user (*International Mobile Subscriber Identity*)
- SMSC: SMS operator gateway (*Short Message Service Center*)
- UDH: optional binary SMS header (*User Data Header*)
- SMS class

Troubleshooting Telecom-level Issues

Best is to test various keys on various USB ports of various computers running various operating systems, possibly with various SIM cards. Ultimately some combination may work.
On GNU/Linux, being root and monitoring the system and Gammu logs (and/or using the `--debug-file` Gammu command-line option) should certainly help.

Please React!

If you have information more detailed or more recent than those presented in this document, if you noticed errors, neglects or points insufficiently discussed, drop us a line! (for that, follow the Support guidelines).

Ending Word

Have fun with Ceylan-Mobile - but do not spam people!