



■ **Remote firmware update**

■ **Application Notes**

■

■

■ **For all FALCOM products**

■

operating with firmware version 2.4.0 and later

Table Of Contents

1 INTRODUCTION	4
1.1 SCOPE OF THE DOCUMENT	4
1.2 FEATURES	4
1.3 RELATED DOCUMENTS	4
2 REMOTE UPDATE OPTIONS	5
2.1 WAYS OF COMMUNICATION	5
2.2 UPDATE TYPES	5
2.2.1 How to generate a BIN file from a S-record file	6
2.3 OPTIONS FOR DEVICE CONFIG	7
3 BASIC UPDATE PROCEDURE	8
3.1 RAW FIRMWARE UPDATE	8
3.1.1 Requirements:	8
3.1.2 Starting update mode	8
3.1.3 Transferring firmware data	9
3.1.4 Transferring configuration	9
3.2 COMPRESSED FIRMWARE UPDATE	9
3.2.1 Requirements:	9
3.2.2 Starting update mode	9
3.2.3 Transferring firmware data	10
3.2.4 Transferring configuration	10
4 DETAILED UPDATE PROCEDURE	11
4.1 START UPDATE MODE	11
4.2 ENTER BINARY DATA TRANSFER MODE	11
4.3 SELECT THE FIRST/NEXT SECTOR	11
4.4 WRITING AND VERIFYING DATA FOR THIS SECTOR	11
4.5 EXIT BINARY DATA TRANSFER MODE	12
4.6 FINISH REMOTE UPDATE	12
5 APPENDIX	13
5.1 COMPUTING CHECKSUMS	13
5.2 GENERATING A BINARY CONFIGURATION FILE	14
5.3 GENERATING A COMPRESSED FIRMWARE	15
5.3.1 Without configuration	15
5.3.2 With configuration	15
5.4 HOW TO UPLOAD A-GPS DATA INTO THE GPS RECEIVER	16
5.5 JAVA SOURCE CODE	16

Version history:

This table provides a summary of the document revisions.

Version	Author	Changes	Modified
1.0.5	F. Beqiri	- Added chapter 5.4 - How to upload A-GPS data into the GPS receiver	21/06/2010
1.0.4	F. Beqiri	- Updated internet address for downloading the firmware binary file and Java source code snippets – See chapter 5.5 , page 16	12/02/2008
1.0.3	F. Beqiri	- Changed the internet address for downloading the firmware binary file and Java source code snippets – See chapter 5.5 , page 16	21/01/2008
1.0.2	F. Beqiri	- Added chapter 2.2.1 - how to generate a BIN file from a s-record file. - Added a complete java source code for remote update – see chapter 5.5 , page 16 .	16/03/2007
1.0.1	F. Beqiri	- Updated FW_RAW - see chapter 2.2 . - Updated chapters 3.1.1 and 3.2.1 - Notes are added. - Updated chapter 5.3.1 - Note is added.	23/02/2007
1.0.0	F. Beqiri	- Initial version	19/10/2006

This confidential document is a property of FALCOM and may not be copied or circulated without previous permission.

1 INTRODUCTION

Remote Update allows to upgrade your target device to a new firmware version over the air (*i.e. without the need of a direct serial connection*).

Remote Updates can be performed either using TCP connection or CSD connection (*and via serial line as well – but in this case a regular firmware update is recommended using the **SIRFFlash Tool***)

1.1 Scope of the document

This application note is relating to the following products: STEPPII, STEPPIII, FOX, BOLERO/-LT, MAMBO and MAMBO2.

1.2 Features

- ✓ Data transmission can be interrupted and resumed at any time (i.e. sending PFAL commands).
- ✓ Size restriction for the firmware.
- ✓ Requirement: there is enough space in the on-board memory flash to store the firmware and its update-packet.
- ✓ Firmware data can also be compressed in order to reduce the amount of data to transmit.
- ✓ Checksums of already transmitted data can be read out in order to verify its correctness.
- ✓ During the update process, the device can still be used for other tasks (i.e. executing alarms, sending system messages etc.)
- ✓ History data is not available when the remote update is chosen and will be cleared after starting the remote update. Therefore, it is recommended to read out all desired history data before performing a remote update.
- ✓ Java source code snippets are available for implementing remote update on server side.

1.3 Related documents

- [1] STEPPII_firmware_2.4.xx_user_manual.pdf
- [2] BOLERO_firmware_2.4.xx_user_manual.pdf
- [3] steppIII_fox_bolero_lt_PFAL_Configuration_Command_Set.pdf
- [4] Mambo2_PFAL_Commands_Reference_Guide.pdf

2 REMOTE UPDATE OPTIONS

2.1 Ways of communication

Basically any type of communication (*except SMS*) can be used to perform a remote update.

The communication for remote update is similar to sending PFAL commands. After receiving a single command, the device will create an answer and sends it back. Therefore, it is recommended that the host application sends only one command and waits for its answer before sending another command.

When using TCP as medium for a remote update, it is possible to send several commands at once to reduce the remote update timespan significantly.

Special care has to be taken when transferring the firmware data. Please, refer to chapter 4.3 "[Select the first/next sector](#)" for further details.

2.2 Update types

Two types (methods) are available to send firmware data to the device:

Types	Meaning
FW_RAW	<p>Uncompressed (raw) binary firmware data can be transmitted to the device. To generate a binary file from a s-record file follow steps in chapter 2.2.1, page 6.</p> <p>Advantage:</p> <ul style="list-style-type: none"> ◆ Selecting this type requires no further tools – Only binary files can be used for the update. Note that, a s-record files must be first transformed in a binary file before sending it remotely to the device – see chapter 2.2.1, page 6. → <i>Easy to use</i> <p>Disadvantage:</p> <ul style="list-style-type: none"> ◆ Drawback of using this type is the amount of data, which has to be transferred. It is approximately twice as much as using compressed mode. → <i>Update procedure requires more time (time is doubled compared to FW_CPR)</i> → <i>Costs for data transfer are doubled (compared to FW_CPR)</i>
FW_CPR	<p>Compressed binary firmware data can be transmitted. For more details, see chapter 5.3.</p> <p>Advantage:</p> <ul style="list-style-type: none"> ◆ Less data to transfer as it is compressed (<i>usually this saves up to 50%</i>) → <i>Less costs for data transmission</i> ◆ Remote update procedure is faster <p>Disadvantage:</p> <ul style="list-style-type: none"> ◆ Compression tools (gzip) are required.
AID_raw	<p>An A-GPS file can be transmitted to the device. For more details, see chapter 5.4, "How to upload A-GPS data into the GPS receiver".</p>

For more details, please also refer to the related documents [1] or [2] of the target device accordingly.

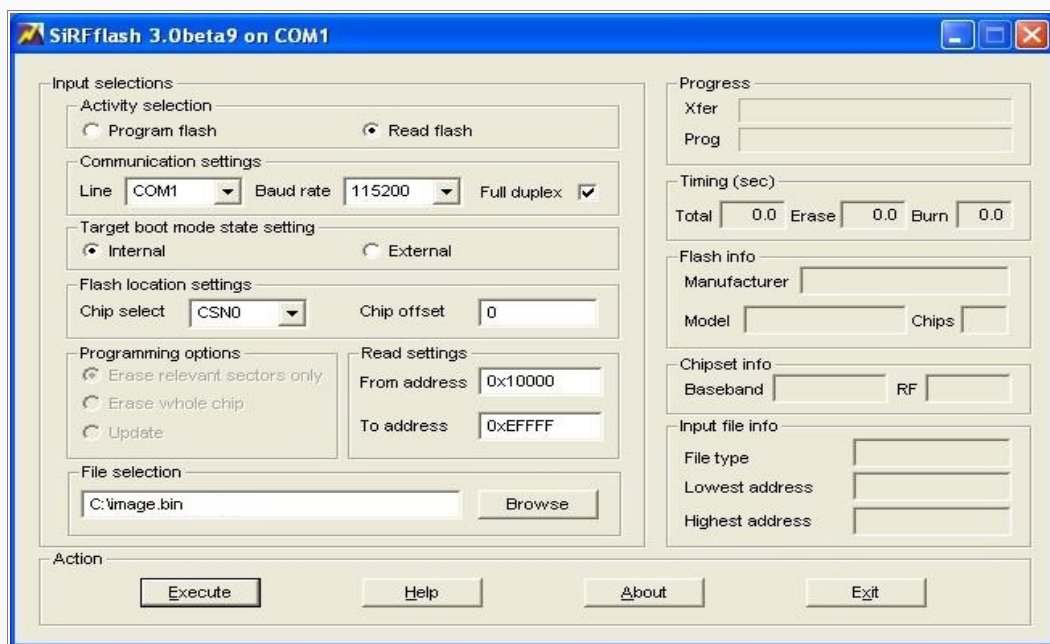
2.2.1 How to generate a BIN file from a S-record file

In order to generate a binary file of your current STEPPII firmware (s-record format), you can simply flash this firmware file to a STEPPII device via serial line and read it out using **SirFlash tool**.

Generating a BIN file in six-step process:


1. Turn OFF the STEPPII
2. Start **SirFlash** program and select the COM port where the STEPPII is connected to.
3. Select the "Read Flash" radio button. Enter **0x10000** in the "From address" and **0xEFFFF** in the "To address".
4. Select the output file by typing the file name (e.g.C:\image.bin) into the **File selection** box or use the **Browse** button.
5. First, turn on the key "Boot", then the **V+** and immediately press the **Execute** button to start flash reading.
6. When reading progress, shown in the **Progress** box., has finished you can then use the output binary file (**image.bin**) for remote update.

Note that, the maximum size of a binary file is $65536*14 - 1$ bytes.



2.3 Options for device config

When setting up the remote update process, it is possible to use the currently stored device configuration later OR to send the new firmware together with another configuration.

 Basically it is recommended to check whether the new firmware runs with an "old" device configuration before the remote update procedure is started. This is usually done on a test device so if the update fails for any reasons (for example, different/new default configuration settings), it can be updated regularly again using a serial connection.

Options	Function
raw_cfg	<ul style="list-style-type: none"> - Erases the device configuration during the update process. A new (uncompressed) configuration can be transmitted via remote update. (see PFAL documentation - select sector for more details) - The current device config can still be used until the remote update is finished ; after finishing, this current configuration will be cleared and replaced by the new one.
compressed_cfg	<ul style="list-style-type: none"> - This option should be used only if "FW_CPR" is selected. - A new configuration is stored within the compressed firmware. - No separate config will be needed. - Therefore, its not allowed to write uncompressed configuration data to the configuration sector if compressed_cfg is selected
current_cfg	<ul style="list-style-type: none"> - Uses the currently stored device configuration later - No separate configuration needs to be specified - An existing compressed configuration would be overwritten, so it is always possible to use this option (even when using FW_CPR) - Using FW_CPR in combination with a compressed firmware which also contains configuration is not recommended, as this additional data isn't required (which means it would lengthen the update process causing additional transmission cost for data which is not required)

For more details, please also refer to the related documents [1] or [2] of the target device accordingly.

3 BASIC UPDATE PROCEDURE

As several configuration options can be specified (which were mentioned above), additional steps might have to be performed. This chapter provides a short overview over what has to be done for which option

3.1 Raw Firmware Update

This chapter describes how to perform a remote update using uncompressed firmware data

3.1.1 Requirements:

- Binary firmware data
 - ✓ For each software version a file "**steppII_XXXXXX.bin**" exists. This file contains the complete binary firmware data.
 - Note** that the length of a binary is maximal $65536 \cdot 14 - 1$ bytes .
- If a new configuration is desired:
 - ✓ A binary file containing this configuration.
 - ✓ For more details please refer to Chapter 5.2 - generating a binary configuration file.

Now detailed steps can be performed which are described within chapter 4.

As chapter 4 just contains a general description, specific settings will be explained here.

3.1.2 Starting update mode

You can start a new update by specifying the PFAL command:

```
PFAL, Sys.RUpdate.Init, FW_raw, new, <size>, <sectors>, <cnf>
```

OR resume a previous remote update by specifying:

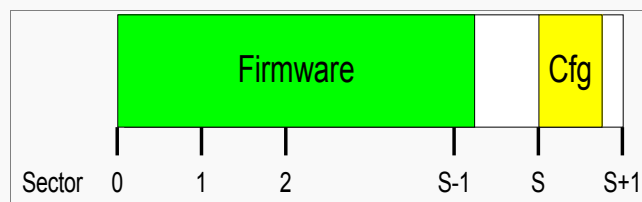
```
PFAL, Sys.RUpdate.Init, FW_raw, resume, <size>, <sectors>, <cnf>
```

Please find more details for this command and the settings "new"/"resume" within PFAL documentation - refer to the related documents [1], [2] or [3] of the target device accordingly.

Parameter	Meaning
<size>	It should contain the number of bytes of the binary firmware data file (marked green inside the illustration below)
<sectors>	It contains the number of sectors used by this firmware. For an uncompressed firmware the number of sectors is the number of bytes divided by 65536 and rounded up. (within the illustration below the number of sectors is marked with S) $\rightarrow \text{<sectors>} = \text{<size>} / 65536 + 1$

The maximal number of sectors allowed is 15.

The following illustration shows the firmware and its configuration. The configuration always uses just a part of a single sector (below 32 KB). Basically, the configuration is located at start of the next sector after the firmware.



After sending the **PFAL, Sys.RUpdate.Init** command, binary transfer mode can be entered (chapter 4.2)

3.1.3 Transferring firmware data

Please refer to chapter 4.3 and 4.4 in order to send firmware data to the device.

3.1.4 Transferring configuration

current_cfg	No configuration needs to be transferred because the current one will be used which is already inside the device. Remote update can be finished now (see chapter 4.6)
raw_cfg	Again please refer to chapter 4.3 and select the configuration sector (value '99'). Perform chapter 4.4 in order to send configuration data to the device.

3.2 Compressed firmware Update

3.2.1 Requirements:

- Binary firmware data
 - ✓ For each software version a file "**steppII_xxxxxx.bin**" exists. This file contains the complete binary firmware data.

Hint: In order to create a binary file of your current STEPPII firmware, you can simply flash this firmware to a device. Now you can read out the binary file using e.g. SirFlash tool. To read out the entire firmware, specify a readout address from 0x10000 to 0xEFFF.
- If a new configuration is desired:
 - ✓ a binary file containing this configuration.
 - ✓ For more details please refer to Chapter 5.2 - generating a binary configuration file.

Please refer to Chapter 5.3 - Generating a compressed firmware

Now detailed steps can be performed which are described within chapter 4.

As chapter 4 just contains a general description, specific settings will be explained here.

3.2.2 Starting update mode

You can start a new update by specifying the PFAL command:

```
PFAL, Sys.RUpdate.Init, FW_cpr, new, <size>, <sectors>, <cfg>
```

OR resume a previous remote update by specifying:

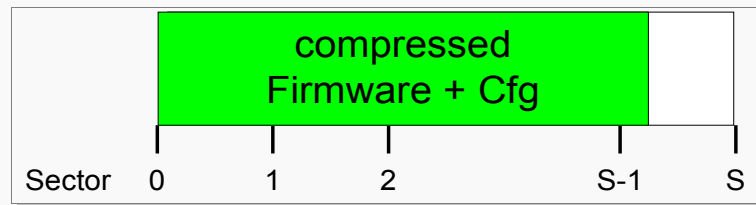
```
PFAL, Sys.RUpdate.Init, FW_cpr, resume, <size>, <sectors>, <cfg>
```

Please find more details for this command and the settings "**new**" / "**resume**" within PFAL documentation - refer to the related documents [1] or [2] of the target device accordingly.

Parameter	Meaning
<size>	It should contain the number of bytes of the compressed firmware data file (marked green inside the illustration below). This file should also contain the new device configuration if option compressed_cfg has been specified.
<sectors>	It contains the number of sectors used by the uncompressed firmware (only the firmware; configuration is ignored). For an uncompressed firmware the number of sectors is the number of bytes from the uncompressed binary firmware (the bin file) divided by 65536 and rounded up (<i>within the illustration below the number of sectors is marked with S</i>). $\text{<sectors>} = \text{<size>} / 65536 + 1$

The allowed maximal number of sectors is 15.

After sending the command **PFAL, Sys.RUpdate.Init**, binary transfer mode can be entered (chapter 4.2)



3.2.3 Transferring firmware data

Please, refer to chapter [4.3](#) and [4.4](#) in order to send firmware data to the device.

3.2.4 Transferring configuration

current_cfg	No configuration needs to be transferred because the current one will be used which is already inside the device. Remote update can be finished now (see chapter 4.6)
raw_cfg	No further configuration needs to be transferred because it is already included within the compressed file. Remote update can be finished now (see chapter 4.6)

4 DETAILED UPDATE PROCEDURE

This chapter describes detailed steps how to perform remote update.

Further details to commands mentioned below can be found within PFAL documentation at chapter "**Remote Update**".

4.1 Start Update mode

i.e. PFAL, Sys.RUpdate.Init, <desired options>

 **Remark:**

- *No History, will be available from now on; history will be cleared when starting a new update procedure.*
- *Also when resetting the device now, the history feature isn't available, because remote update data could be stored within the device. In order to re-enable History again, remote update must be finished OR a serial update (with option "erase whole flash" must be performed).*

4.2 Enter binary data transfer mode

This mode is required to send new firmware data to the device.

i.e. PFAL, Sys.RUpdate.DataMode, TCP

You are now inside BINARY MODE - no PFAL commands may be entered.

In case a PFAL command has to be send to the device, binary data transfer mode has to be exited by using the command "**Exit Data Mode for Remote Update**" (see PFAL documentation for more details - *refer to the related documents [1] or [2] of the target device accordingly*)

4.3 Select the first/next sector

Once binary mode is entered, new firmware data can be transmitted to the device. Due to the huge size of firmware data, it is always divided into sectors.

Therefore, it is necessary to select a sector before writing data to it.

 **Important note:**

When using TCP as medium for a remote update, it is possible to send several commands at once. As most of binary commands rely on a selected sector, you have to assure that you select the next sector AFTER all previous commands have been successfully performed. Else you might e.g. write data, which belongs to the first sector into the next one if you select another sector before this write command is executed. (This is however a very rare case but would have the consequence of a corrupted firmware ☹ the device won't start until a regular (serial) firmware update has been performed). This binary command is fully described inside PFAL documentation, chapter 2.2.1.2 „Remote update“ - Binary update commands.

4.4 Writing and verifying data for this sector

1. Generally each firmware sector must be divided into smaller blocks (*it is recommended to divide into 64 blocks of 1 KB each*). These blocks can be transferred by a single write command.
 - *See "writing data to sector" inside PFAL documentation chapter Remote update; Binary update commands.*
 - *When using TCP multiple write commands for this sector can be written simultaneously in order to increase download speed. However you should assure that each command is executed correctly before selecting the next sector (mentioned at point 4).*

This confidential document is a property of FALCOM and may not be copied or circulated without previous permission.

2. If a sector doesn't have to be filled completely (i.e. this sector contains trailing **0xFF**'s after its data, it doesn't need to be written. This is usually the case for the last firmware sector and the configuration sector.
3. Query a checksum if sector is complete and compare this checksum with the expected one – they have to match else clear the complete sector and start writing in your data again.
 - *Each sector will contain **0xFF**'s after it is cleared.*
 - *Note that the correct length has to be specified for checksums. Else they might be computed over a complete sector, which also contains trailing "**0xFF**'s".*
4. If the checksums match, you can proceed with selecting the next sector.
5. You may exit and enter binary transfer mode again at any time - but you have to select the sector lately used before continue writing data.

4.5 Exit binary data transfer mode

Once all firmware data has been transmitted (and verified), the binary transfer mode can be exited (which means switching back to PFAL commands). This binary command is fully described inside PFAL documentation, chapter **2.2.1.2** „Remote update“ - Binary update commands.

4.6 Finish remote update

i.e. PFAL,Sys.RUpdate.Finish

This command resets the device and prepares the new firmware to work. Approximately 30-40 seconds later, the new firmware should start if all previous steps were performed correctly (i.e. firmware data was written correctly).

5 APPENDIX

5.1 Computing checksums

The following algorithm can be used to compute checksums for remote update AND self generated configuration files.

- When calling this function, it is recommended to always specify **0** for **start_crc**.
- **p_data** is a pointer which points at the first byte of data (from which a **crc** is computed)
- **len** specifies the number of data bytes.

```
uint16 CRC16 (const uint8 *p_data, uint32 len, uint16 start_crc)
{
    const uint8 *pbyte = p_data;
    while (len--)
    {
        uint8 uIndex = start_crc ^ *pbyte++;
        start_crc = (start_crc >> 8) ^ vCRC16[uIndex];
    }
    return start_crc;
}
```

```
static const uint16 vCRC16[256] =
{ //crc-16 is based on the polynomial x^16+x^15+x^2+1
    0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
    0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
    0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
    0x0A00, 0xCAC1, 0xCB81, 0x0B40, 0xC901, 0x09C0, 0x0880, 0xC841,
    0xD801, 0x18C0, 0x1980, 0xD941, 0x1B00, 0xDBC1, 0xDA81, 0x1A40,
    0x1E00, 0xDEC1, 0xDF81, 0x1F40, 0xDD01, 0x1DC0, 0x1C80, 0xDC41,
    0x1400, 0xD4C1, 0xD581, 0x1540, 0xD701, 0x17C0, 0x1680, 0xD641,
    0xD201, 0x12C0, 0x1380, 0xD341, 0x1100, 0xD1C1, 0xD081, 0x1040,
    0xF001, 0x30C0, 0x3180, 0xF141, 0x3300, 0xF3C1, 0xF281, 0x3240,
    0x3600, 0xF6C1, 0xF781, 0x3740, 0xF501, 0x35C0, 0x3480, 0xF441,
    0x3C00, 0xFCC1, 0xFD81, 0x3D40, 0xFF01, 0x3FC0, 0x3E80, 0xFE41,
    0xFA01, 0x3AC0, 0x3B80, 0xFB41, 0x3900, 0xF9C1, 0xF881, 0x3840,
    0x2800, 0xE8C1, 0xE981, 0x2940, 0xEB01, 0x2BC0, 0x2A80, 0xEA41,
    0xEE01, 0x2EC0, 0x2F80, 0xEF41, 0x2D00, 0xEDC1, 0xEC81, 0x2C40,
    0xE401, 0x24C0, 0x2580, 0xE541, 0x2700, 0xE7C1, 0xE681, 0x2640,
    0x2200, 0xE2C1, 0xE381, 0x2340, 0xE101, 0x21C0, 0x2080, 0xE041,
    0xA001, 0x60C0, 0x6180, 0xA141, 0x6300, 0xA3C1, 0xA281, 0x6240,
    0x6600, 0xA6C1, 0xA781, 0x6740, 0xA501, 0x65C0, 0x6480, 0xA441,
    0x6C00, 0xACC1, 0xAD81, 0x6D40, 0xAF01, 0x6FC0, 0x6E80, 0xAE41,
    0xAA01, 0x6AC0, 0x6B80, 0xAB41, 0x6900, 0xA9C1, 0xA881, 0x6840,
    0x7800, 0xB8C1, 0xB981, 0x7940, 0xBB01, 0x7BC0, 0x7A80, 0xBA41,
```

```

0xBE01, 0x7EC0, 0x7F80, 0xBF41, 0x7D00, 0xBDC1, 0xBC81, 0x7C40,
0xB401, 0x74C0, 0x7580, 0xB541, 0x7700, 0xB7C1, 0xB681, 0x7640,
0x7200, 0xB2C1, 0xB381, 0x7340, 0xB101, 0x71C0, 0x7080, 0xB041,
0x5000, 0x90C1, 0x9181, 0x5140, 0x9301, 0x53C0, 0x5280, 0x9241,
0x9601, 0x56C0, 0x5780, 0x9741, 0x5500, 0x95C1, 0x9481, 0x5440,
0x9C01, 0x5CC0, 0x5D80, 0x9D41, 0x5F00, 0x9FC1, 0x9E81, 0x5E40,
0x5A00, 0x9AC1, 0x9B81, 0x5B40, 0x9901, 0x59C0, 0x5880, 0x9841,
0x8801, 0x48C0, 0x4980, 0x8941, 0x4B00, 0x8BC1, 0x8A81, 0x4A40,
0x4E00, 0x8EC1, 0x8F81, 0x4F40, 0x8D01, 0x4DC0, 0x4C80, 0x8C41,
0x4400, 0x84C1, 0x8581, 0x4540, 0x8701, 0x47C0, 0x4680, 0x8641,
0x8201, 0x42C0, 0x4380, 0x8341, 0x4100, 0x81C1, 0x8081, 0x4040
};

```

5.2 Generating a binary configuration file

Generating a configuration file can also be automatically performed.

Simply create a new (empty) file and append the following data:

```

Byte0: 0x56
Byte1: 0x4E
Byte2: 0x45
Byte3: 0x5F

```

The next 4 bytes must be computed later, after adding the configuration.

Bytes 4, 5 contain the length of the configuration in bytes (*simply store an UINT16 value at byte 4*).

Bytes 6, 7 contain a checksum of the complete textual configuration, which starts at byte 8 (*and has the length specified at Byte 4,5*). Simply store an UINT16 returned from the checksum routine mentioned within chapter 5.1.

Now textual data can be appended at Byte8 - simply append all configuration settings. Note that after each configuration setting a byte with the value *0x00* has to be added.

A short example how 2 configuration settings are appended:

Hex – characters	Corresponding Text
44 45 56 49 43 45 2E 4E 41 4D 45 3D 6D 79 53 74	DEVICE.NAME=mySt
65 70 70 49 49 00 47 50 52 53 2E 41 55 54 4F 53	eppl.GPRS.AUTOS
54 41 52 54 3D 31 00	TART=1.

After appending all configuration settings, the length of this text has to be counted and stored at Byte 4/5.

Using this length, a checksum can be computed (*chapter 5.1*) which is stored at Bytes 6 and 7.

It is recommended to append bytes containing the value *0x00* at end of this file until byte 32767.

The complete configuration file should now have a size of 32 KB.

Hint: To check if the file was generated correctly, simply flash a new firmware on a device (erasing the flash). Before starting this firmware, flash the generated configuration file at position **0xF0000** of the flash (specify this values as chip offset within the SiRFFlash tool)

When starting the firmware and reading out configuration, all your configuration settings should be shown. If only default settings are shown (or possibly the device does not start), the file was corrupt – or flashed on a wrong position inside flash.

5.3 Generating a compressed firmware

5.3.1 Without configuration

Simply compress the binary firmware image (*steppII_xxx.bin*) with *gzip*. The compressed file should have a size of approx. **50%** from the bin file.

In order to create a binary file of your current STEPPII firmware, you can simply flash this firmware to a device. Now you can read out the binary file using e.g. **SirFlash Tool**. To read out the entire firmware, specify a readout address from **0x10000** to **0xEFFFF**.

5.3.2 With configuration

Before compressing the data, a configuration has to be appended to the binary firmware image.

It is critical to append the configuration exactly to the position of the next sector after binary firmware. This is shown in the illustration below

An automatic generation of such a file is quite easy - starting with the binary firmware (*steppII_xxx.bin*), you simply append "0xFF" characters at end of this file until the current sector is filled.

Example:

Assuming that the binary firmware image has a size of **915000** bytes.

$915000 / 65535 = 13,96$ sectors \Rightarrow configuration has to be added at the 14th sector.

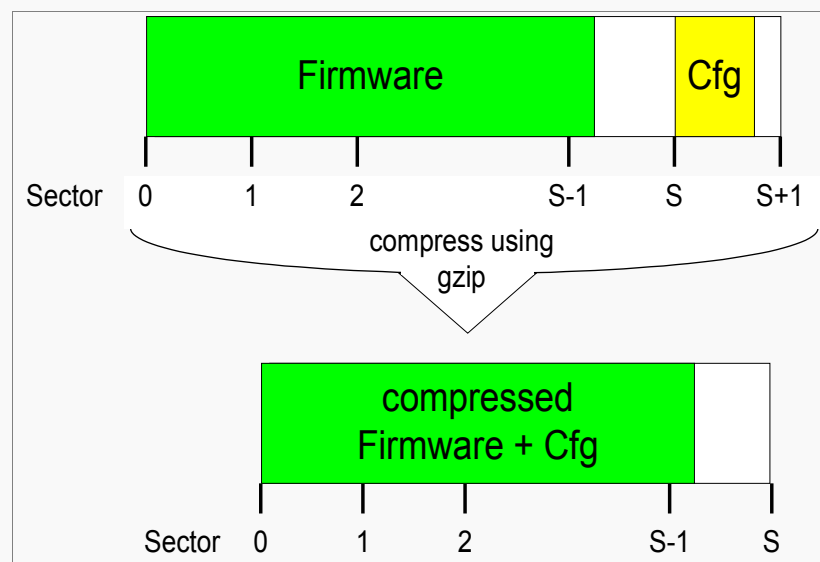
The 14th sector starts at $14 * 65535 =$ the 917504th byte.

This means $917504 - 915000$ (-1 because we want to append already at the 917504th byte – and not after it) \rightarrow 2503 bytes containing **0xFF** would have to be added.

After adding these bytes, the original binary file should have a size of exactly **917504** bytes.

Now simply attach the binary configuration file (see chapter 5.2) and the resulting file can be compressed - which is shown at the following illustration

Note that the value of "S" becomes smaller when the file above is being compressed.

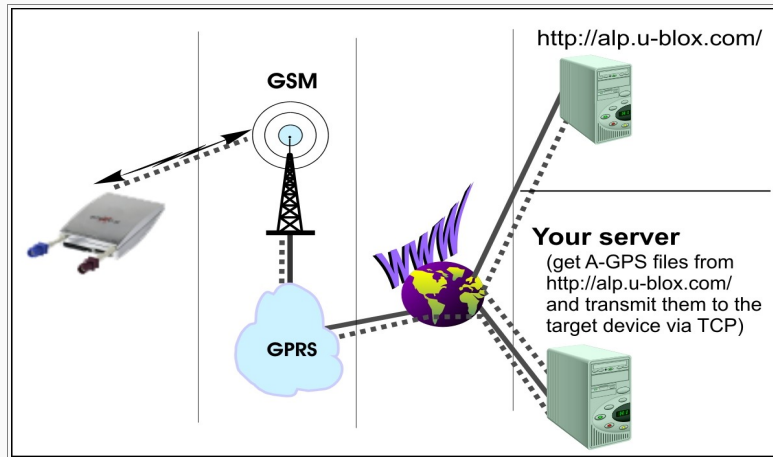


5.4 How to upload A-GPS data into the GPS receiver

Assisted GPS (A-GPS) provides assistance data for GPS calculations within the device using a u-blox GPS receiver. This solution enables A-GPS on your device and provides improved GPS fix times.

The A-GPS data is distributed electronically over the Internet and can be downloaded for free of charge from the files present in the u-blox website: <http://alp.u-blox.com/>

The files present in the u-blox website are updated daily and they have differential almanac corrections with a validity of 1 to 14 days.



To implement such an application on your server, you should make a HTTP request to the u-blox server, hold the file you want to transfer to the remote AVL device and then you can transfer that file "AS IS" to the target AVL device using the PFAL commands for remote update:

```
$PFAL,Sys.RUpdate.Init,AID_raw,new,<size>,<sectors>,current_cfg
```

<size> - specifies the length in bytes of the received file from the u-blox server.

<sectors> - specifies the number of sectors that are needed for the received file: 1 sector = 64 KB.

For more details, refer to the chapter 3.1.

5.5 Java source code

The following Java source code snippets are provided to help you easy develop your remote update application in your server. An internal documentation is comprised of comments written within the source code.

The firmware as binary file can be found in the protected download area on the FALCOM's homepage: <http://www.falcom.de/>.

To visit this area, you have to login first (using the log-in data that the FALCOM has provided to you), then go to the section "**Support**", select the product you are using under "**Drivers and Firmware**", finally click on the file e.g. "**steppIII_x.x.x.zip**" to download it - where x.x.x is the firmware version number.

While the Java source code for remote update can be downloaded from: <http://www.falcom.de/uploads/media/>.

```

package de.falcom.snippets;
import java.io.BufferedReader;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.UnsupportedEncodingException;

import Serialio.SerInputStream;
import Serialio.SerOutputStream;
import Serialio.SerialConfig;
import Serialio.SerialPortLocal;

public class UpdateFirmware {

    // -----
    // CRC utilities
    // -----

    /**
     * Table used in CRC16 calculation.
     */
    private static final int[] CRC16_TAB = {
        0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
        0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
        0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
        0x0A00, 0xCAC1, 0xCB81, 0x0B40, 0xC901, 0x09C0, 0x0880, 0xC841,
        0xD801, 0x18C0, 0x1980, 0xD941, 0x1B00, 0xDBC1, 0xDA81, 0x1A40,
        0x1E00, 0xDEC1, 0xDF81, 0x1F40, 0xDD01, 0x1DC0, 0x1C80, 0xDC41,
        0x1400, 0xD4C1, 0xD581, 0x1540, 0xD701, 0x17C0, 0x1680, 0xD641,
        0xD201, 0x12C0, 0x1380, 0xD341, 0x1100, 0xD1C1, 0xD081, 0x1040,
        0xF001, 0x30C0, 0x3180, 0xF141, 0x3300, 0xF3C1, 0xF281, 0x3240,
        0x3600, 0xF6C1, 0xF781, 0x3740, 0xF501, 0x35C0, 0x3480, 0xF441,
        0x3C00, 0xFCC1, 0xFD81, 0x3D40, 0xFF01, 0x3FC0, 0x3E80, 0xFE41,
        0xFA01, 0x3AC0, 0x3B80, 0xFB41, 0x3900, 0xF9C1, 0xF881, 0x3840,
        0x2800, 0xE8C1, 0xE981, 0x2940, 0xEB01, 0x2BC0, 0x2A80, 0xEA41,
        0xEE01, 0x2EC0, 0x2F80, 0xEF41, 0x2D00, 0xEDC1, 0xEC81, 0x2C40,
        0xE401, 0x24C0, 0x2580, 0xE541, 0x2700, 0xE7C1, 0xE681, 0x2640,
        0x2200, 0xE2C1, 0xE381, 0x2340, 0xE101, 0x21C0, 0x2080, 0xE041,
    };

```

```
0xA001, 0x60C0, 0x6180, 0xA141, 0x6300, 0xA3C1, 0xA281, 0x6240,  
0x6600, 0xA6C1, 0xA781, 0x6740, 0xA501, 0x65C0, 0x6480, 0xA441,  
0x6C00, 0xACC1, 0xAD81, 0x6D40, 0xAF01, 0x6FC0, 0x6E80, 0xAE41,  
0xAA01, 0x6AC0, 0x6B80, 0xAB41, 0x6900, 0xA9C1, 0xA881, 0x6840,  
0x7800, 0xB8C1, 0xB981, 0x7940, 0xBB01, 0x7BC0, 0x7A80, 0xBA41,  
0xBE01, 0x7EC0, 0x7F80, 0xBF41, 0x7D00, 0xBDC1, 0xBC81, 0x7C40,  
0xB401, 0x74C0, 0x7580, 0xB541, 0x7700, 0xB7C1, 0xB681, 0x7640,  
0x7200, 0xB2C1, 0xB381, 0x7340, 0xB101, 0x71C0, 0x7080, 0xB041,  
0x5000, 0x90C1, 0x9181, 0x5140, 0x9301, 0x53C0, 0x5280, 0x9241,  
0x9601, 0x56C0, 0x5780, 0x9741, 0x5500, 0x95C1, 0x9481, 0x5440,  
0x9C01, 0x5CC0, 0x5D80, 0x9D41, 0x5F00, 0x9FC1, 0x9E81, 0x5E40,  
0x5A00, 0x9AC1, 0x9B81, 0x5B40, 0x9901, 0x59C0, 0x5880, 0x9841,  
0x8801, 0x48C0, 0x4980, 0x8941, 0x4B00, 0x8BC1, 0x8A81, 0x4A40,  
0x4E00, 0x8EC1, 0x8F81, 0x4F40, 0x8D01, 0x4DC0, 0x4C80, 0x8C41,  
0x4400, 0x84C1, 0x8581, 0x4540, 0x8701, 0x47C0, 0x4680, 0x8641,  
0x8201, 0x42C0, 0x4380, 0x8341, 0x4100, 0x81C1, 0x8081, 0x4040  
};
```

```
/**
```

```
 * Calculates the CRC16 of an array of bytes.
```

```
 *
```

```
 * @param b
```

```
       the bytes.
```

```
 * @param off
```

```
       the start offset in the bytes.
```

```
 * @param len
```

```
       the number of bytes to use.
```

```
 * @return the CRC16 of the bytes.
```

```
 */
```

```
private static int doCRC16(byte[] b, int off, int len) {
```

```
    int crc = 0;
```

```
    for (int i = off; i < off + len; ++i)
```

```
        crc = (crc >>> 8) ^ CRC16_TAB[(crc ^ b[i]) & 0xff];
```

```
    return crc;
```

```
}
```

```
/**
```

```
 * Calculates the CRC16 of an array of bytes cumulatively.
```

```
 *
```

```
 * @param crc
```

```

*           the CRC16 of the bytes used up to now. Use 0 for a new
*           calculation.
* @param b
*           the next group of bytes.
* @param off
*           the start offset in the bytes.
* @param len
*           the number of bytes to use.
* @return the CRC16 of the previous bytes and the bytes passed as
*         parameters to this method.
*/
private static int doCRC16Cumul(int crc, byte[] b, int off, int len) {
    crc = crc & 0xffff;
    for (int i = off; i < (off + len); ++i)
        crc = (crc >>> 8) ^ CRC16_TAB[(crc ^ b[i]) & 0xff];

    return crc;
}

// -----
// byte utilities
// -----

/**
 * The hexadecimal digits.
 */
private final static char[] HEX = {
    '0', '1', '2', '3', '4', '5', '6', '7',
    '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'
};

/**
 * Converts a string to bytes.
 *
 * @param s
 *         the string to convert.
 * @return an array of bytes.
 */
private static byte[] stringToBytes(String s) {
    try {
        return s.getBytes("ISO-8859-1");
    } catch (UnsupportedEncodingException e) {

```

```

        return new byte[0];
    }
}

/**
 * Converts an array of bytes to a string.
 * @param bytes the array of bytes to convert.
 * @return the string.
 */
private static String bytesToString(byte[] bytes, int off, int len) {
    try {
        return new String(bytes, off, len, "ISO-8859-1");
    } catch (UnsupportedEncodingException e) {
        return "";
    }
}

/**
 * Converts a number to its hexadecimal representation.
 *
 * @param num
 *         the number to convert.
 * @param bits
 *         the number of bits to use.
 * @return the hexadecimal representation of the number.
 */
private static String numToHex(int num, int bits) {
    if (bits < 1)
        return "";
    StringBuilder s = new StringBuilder();
    int n = bits / 4;
    int b = bits % 4;
    if (b > 0)
        s.append(HEX[num >>> 4 * n & (1 << b) - 1]);
    for (--n; n >= 0; --n)
        s.append(HEX[num >>> 4 * n & 0xf]);
    return s.toString();
}

// -----
// firmware update

```

```

// -----
/**
 * The maximum length of the firmware binary.
 */
public static final int MAX_FIRMWARE_LEN = 0xf0000;

/**
 * An exception to indicate that the firmware update failed.
 */
public static class FirmwareException extends Exception {

    /**
     * The serial version UID.
     */
    private static final long serialVersionUID = 8004211078749528029L;

    /**
     * Creates a firmware exception with the specified detail message.
     *
     * @param message
     *         the detail message.
     */
    public FirmwareException(String message) {
        super(message);
    }

    /**
     * Creates a firmware exception with the specified detail message and
     * underlying cause.
     *
     * @param message
     *         the detail message.
     * @param cause
     *         the underlying cause.
     */
    public FirmwareException(String message, Throwable cause) {
        super(message, cause);
    }
}

/** The thread in which the update takes place. */

```

```
private Thread updateThread;

/** Whether the operation succeeded. */
private boolean success;

/** The error message of the finished operation. */
private String error;

/**
 * The fraction done up to this point (0.0 means just started, 1.0 means
 * ready).
 */
private double fractionDone;

/** The input buffer contains bytes received from the device. */
private StringBuilder inputBuffer;

/** Whether the connection is over TCP. */
private boolean paramTCP;

/** The firmware binary to upload. */
private byte[] paramFirmware;

/** The number of sectors the uncompressed firmware binary will take. */
private int paramSectors;

/** Whether the firmware binary to upload is compressed. */
private boolean paramCompressed;

/** The size of the blocks to upload. */
private int blockSize;

/** The command timeout in milliseconds. */
private static int timeout;

/** An answer id used during uploading. */
private byte answerId;

/** The current sector being uploaded. */
private int currentSector;

/** The device output stream. */
```

```

private OutputStream deviceOS;

/** The device input stream. */
private InputStream deviceIS;

/** Flag used for closing the read thread. */
private boolean closeReadThread;

/** The read thread reads deviceIS into the input buffer. */
private Thread readThread;

/**
 * Creates a new UpdateFirmware object.
 */
public UpdateFirmware() {
    inputBuffer = new StringBuilder();
    answerId = (byte) (Math.random() * 256);
}

/**
 * Starts a firmware update.
 *
 * @param is
 *         the input stream of the device.
 * @param os
 *         the output stream of the device.
 * @param tcp
 *         whether the connection is over TCP.
 * @param firmware
 *         the firmware binary to upload.
 * @param sectors
 *         the number of sectors used by the uncompressed firmware. Each
 *         sector is 65536 bytes long. This is equal to the length of the
 *         uncompressed binary divided by 65536 and rounded up to the
 *         nearest integer. It can be calculated as
 *         <code>sectors = (uncompressedLength - 1) / 65536 + 1</code>;.
 * @param compressed
 *         whether the firmware is compressed using GZIP compression.
 * @throws IllegalStateException
 *         if an update is in progress.
 */
public synchronized void startUpdate(InputStream is, OutputStream os, boolean tcp,

```

```

        byte[] firmware, int sectors, boolean compressed) {
    if (updateThread != null)
        throw new IllegalStateException();
    fractionDone = 0;
    deviceIS = is;
    deviceOS = os;
    paramTCP = tcp;
    paramFirmware = firmware;
    paramSectors = sectors;
    paramCompressed = compressed;

    // For TCP use a block size of 1024.
    blockSize = paramTCP ? 0x400 : 0xf80;

    // For TCP, the timeout is 10 s, otherwise use 5 s.
    timeout = paramTCP ? 10000 : 10000;

    // Clear the input buffer.
    eraseInputBuffer();

    // Create and start the working thread.
    updateThread = new Thread() {
        public void run() {
            try {
                System.out.println("update Start...");
                startReadThread();
                threadRun();
                error = "Success.";
                success = true;
            } catch (FirmwareException e) {
                error = e.getMessage();
                success = false;
            }
            synchronized (UpdateFirmware.this) {
                System.out.println(" updateThread = null ");
                updateThread = null;
                stopReadThread();
                eraseInputBuffer();
                UpdateFirmware.this.notifyAll();
            }
        }
    };
};

```

```

        updateThread.start();
    }

    /**
     * Gets the fraction done up to this point
     *
     * @return the fraction of the process which is ready. 0.0 means that the
     *         process has just started, and 1.0 means that the process is
     *         ready.
     */
    public double getFractionDone() {
        return fractionDone;
    }

    /**
     * Returns <code>>true</code> if the process is ready.
     *
     * @return <code>true</code> if the process is ready.
     */
    public synchronized boolean isReady() {
        return updateThread == null;
    }

    /**
     * Returns <code>true</code> if the process was completed successfully and
     * <code>false</code> if an error occurred. This method must be called
     * after the process is completed.
     *
     * @return <code>true</code> if the process was completed successfully and
     *         <code>false</code> if an error occurred.
     * @throws IllegalStateException
     *         if the process is not ready.
     */
    public synchronized boolean success() {
        if (!isReady())
            throw new IllegalStateException();
        return success;
    }

    /**
     * Returns the error message. This method must be called after the process
     * is completed.

```

```

*
* @return the error message.
* @throws IllegalStateException
*         if the process is not ready.
*/
public synchronized String getError() {
    if (!isReady())
        throw new IllegalStateException();
    return error;
}

/**
* Blocks until the update process to complete.
*/
public synchronized void waitReady() {
    while (!isReady()) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
}

/**
* Blocks until the update process is complete, or a timeout occurs.
*
* @param timeout
*         the maximum number of milliseconds to wait.
* @return <code>>true</code> if the update process is complete.
*/
public synchronized boolean waitReady(long timeout) {
    if (!isReady()) {
        try {
            wait(timeout);
        } catch (InterruptedException e) {
        }
    }
    return isReady();
}

/**
* Performs the firmware update.

```

```

*
* @throws FirmwareException
*         if an error occurs.
*/
private void threadRun() throws FirmwareException {
    // Clear the input buffer.
    eraseInputBuffer();

    // Initialize the update.
    StringBuilder cmd = new StringBuilder();
    cmd.append("$PFAL,SYS.RUpdate.Init,");
    cmd.append(paramCompressed ? "FW_cpr" : "FW_raw");
    cmd.append(",new,").append(paramFirmware.length);
    cmd.append(",").append(paramSectors).append(",current_cfg\r\n");
    System.out.println("Timeout..." + timeout);
    String response = doCommand(cmd.toString(), "SYS.RUpdate.Init", timeout);
    if (response == null || response.toUpperCase().indexOf("SUCCESS") < 0)
        throw new FirmwareException("Cannot initialize update.");
    fractionDone = 0.01;

    // Initialize the data mode.
    cmd.delete(0, cmd.length()).append("$PFAL,SYS.RUpdate.DataMode,");
    cmd.append(paramTCP ? "TCP\r\n" : "Serial\r\n");
    response = doCommand(cmd.toString(), "SYS.RUpdate.DataMode", timeout);
    if (response == null || response.toUpperCase().indexOf("SUCCESS") < 0)
        throw new FirmwareException("Cannot initialize data mode.");
    fractionDone = 0.02;

    // Upload the firmware binary.
    uploadFirmware();

    // Complete the firmware update.
    response = doCommand("$PFAL,SYS.RUpdate.Finish\r\n",
        "SYS.RUpdate.Finish", timeout);
    // The finish command should return no response.
    if (response != null)
        throw new FirmwareException("Error finishing update.");
    fractionDone = 1.00;
}

/**
 * Uploads the firmware binary.

```

```

*
* @throws FirmwareException
*     if an error occurs.
*/
private void uploadFirmware() throws FirmwareException {
    int nSectors = (paramFirmware.length - 1 >> 16) + 1;
    currentSector = 0;
    int attempts = 0;

    // Write the sectors.
    while (currentSector < nSectors) {
        // Calculate the sector position, sector length, and number of parts
        // to split the sector in.
        int secPos = currentSector << 16;
        int secLen = 0x10000;
        if (secLen + secPos > paramFirmware.length)
            secLen = paramFirmware.length - secPos;
        int secParts = (secLen - 1) / blockSize + 1;

        // Select the sector.
        binSelectSector();

        int sectorCRC = 0;
        for (int secPart = 0; secPart < secParts; ++secPart) {
            int partPos = secPart * blockSize;
            int partLen = blockSize;
            if (partLen + partPos > secLen)
                partLen = secLen - partPos;
            // Write this part of the current sector.
            sectorCRC = binWrite(partPos, partLen, sectorCRC);
            fractionDone = 0.03 + 0.96 * (secPos + partPos + partLen)
                / paramFirmware.length;
        }

        // Match the sector CRC.
        int crcSecLen = secLen;
        if (crcSecLen == 0x10000)
            --crcSecLen;
        if (binCRCMatch(crcSecLen, sectorCRC)) {
            // CRC match, start next sector.
            ++currentSector;
            attempts = 0;
        }
    }
}

```

```

        } else {
            // CRC mismatch, try again or fail.
            if (++attempts > 2)
                throw new FirmwareException("CRC mismatch on sector "
                    + currentSector + ".");

            binEraseSector();
        }
    }

    // Exit binary mode.
    binExit();
}

/**
 * Exits the data mode.
 *
 * @throws FirmwareException
 *         if an error occurs.
 */
private void binExit() throws FirmwareException {
    byte[] packet = {
        (byte) 0xfc,
        4,
        0,
        ++answerId,
        0,
        (byte) 0xec
    };
    String response = doCommand(packet, timeout);
    if (response == null || response.toUpperCase().indexOf("SUCCESS") < 0)
        throw new FirmwareException("Cannot exit from data mode.");
}

/**
 * Selects the sector.
 *
 * @throws FirmwareException
 *         if an error occurs.
 */
private void binSelectSector() throws FirmwareException {
    byte[] packet = {

```

```

        (byte) 0xfc,
        5,
        1,
        ++answerId,
        1,
        (byte) currentSector,
        (byte) 0xec
    };
    String response = doCommand(packet, timeout);
    if (response == null || response.toUpperCase().indexOf("SUCCESS") < 0)
        throw new FirmwareException("Cannot select sector " + currentSector + ".");
}

/**
 * Uploads a part of the current sector. If in TCP mode, this method does
 * not wait for a response.
 *
 * @param partPos
 *         the offset of the part in the current sector.
 * @param partLen
 *         the length of the part
 * @param sectorCRC
 *         the CRC of the sector before this part.
 * @return the CRC of the sector up to and including this part.
 * @throws FirmwareException
 *         if an error occurs.
 */
private int binWrite(int partPos, int partLen, int sectorCRC)
    throws FirmwareException {
    int dataLen = partLen + 4;
    int lenField = dataLen < 128 ? 4 : 5;
    int packetLen = lenField + (lenField < 128 ? 2 : 3);

    byte[] packet = new byte[packetLen];
    int i = 0;
    packet[i++] = (byte) 0xfc;
    if (lenField < 128) {
        packet[i++] = (byte) lenField;
    } else {
        packet[i++] = (byte) (lenField >>> 7 & 0x7f | 0x80);
        packet[i++] = (byte) (lenField & 0x7f);
    }
}

```

```

packet[i++] = 2;
packet[i++] = ++answerId;
if (dataLen < 128) {
    packet[i++] = (byte) dataLen;
} else {
    packet[i++] = (byte) (dataLen >>> 7 & 0x7f | 0x80);
    packet[i++] = (byte) (dataLen & 0x7f);
}
packet[i++] = (byte) (partPos >>> 8 & 0xff);
packet[i++] = (byte) (partPos & 0xff);
// Copy the part of the sector into the packet.
System.arraycopy(paramFirmware, currentSector * 0x10000 + partPos,
    packet, i, partLen);
int packetCRC = doCRC16(packet, i, partLen);
boolean fillsSector = partLen + partPos == 0x10000;
int newSectorCRC = doCRC16Cumul(sectorCRC, packet, i,
    partLen - (fillsSector ? 1 : 0));
i += partLen;
packet[i++] = (byte) (packetCRC >>> 8 & 0xff);
packet[i++] = (byte) (packetCRC & 0xff);
packet[i++] = (byte) 0xec;

if (paramTCP) {
    doCommand(packet, 5);
} else {
    String response = doCommand(packet, timeout);
    if (response == null || response.toUpperCase().indexOf("SUCCESS") < 0)
        throw new FirmwareException("Cannot write to sector " + currentSector + ".");
}
return newSectorCRC;
}

/**
 * Checks the CRC of the current sector up to the specified position.
 *
 * @param pos
 *     the position of the last byte to check.
 * @param crc
 *     the expected CRC value.
 * @return <code>true</code> for a match, <code>false</code> for a
 *     mismatch.
 * @throws FirmwareException

```

```

*           if an error occurs.
*/
private boolean binCRCMatch(int pos, int crc) throws FirmwareException {
    byte[] packet;
    packet = new byte[] {
        (byte) 0xfc,
        6,
        3,
        ++answerId,
        2,
        (byte) (pos >>> 8 & 0xff),
        (byte) (pos & 0xff),
        (byte) 0xec };
    int crcTimeout = timeout;
    // If TCP is being used, the write commands may still be unreceived,
    // so a larger timeout is required.
    if (paramTCP)
        crcTimeout = 90000; // allow for about 1 Kb/s
    String response = doCommand(packet, crcTimeout);
    if (response == null || response.toUpperCase().indexOf("SUCCESS") < 0)
        throw new FirmwareException("Cannot verify sector " + currentSector + ".");
    String reqCRCString = "$" + numToHex(crc, 16) + "\r\n";
    return response.toUpperCase().indexOf(reqCRCString) >= 0;
}

/**
 * Erases the current sector.
 *
 * @throws FirmwareException
 *         if an error occurs.
 */
private void binEraseSector() throws FirmwareException {
    byte[] packet = {
        (byte) 0xfc,
        4,
        4,
        ++answerId,
        0,
        (byte) 0xec
    };
    String response = doCommand(packet, timeout);
    if (response == null || response.toUpperCase().indexOf("SUCCESS") < 0)

```

```

        throw new FirmwareException("Cannot erase sector " + currentSector + ".");
    }

    /**
     * Clears the input buffer.
     */
    private void eraseInputBuffer() {
        synchronized (inputBuffer) {
            inputBuffer.delete(0, inputBuffer.length());
        }
    }

    /**
     * Executes a text-mode command.
     *
     * @param cmd
     *         the command to execute.
     * @param ans
     *         the answer name.
     * @param timeout
     *         the timeout in milliseconds.
     * @return the response of the command
     * @throws FirmwareException
     *         if an error occurs.
     */
    private String doCommand(String cmd, String ans, int timeout)
        throws FirmwareException {
        eraseInputBuffer();
        try {
            deviceOS.write(stringToBytes(cmd));
        } catch (IOException e) {
            throw new FirmwareException("Cannot send command " + cmd
                + " to device.", e);
        }

        String tStart = ("${<" + ans + ">\r\n");
        String tEnd = "${<end>\r\n";

        return getResponse(tStart, tEnd, timeout);
    }

    /**

```

```

* Executes a binary command.
*
* @param packet
*     the command packet.
* @param timeout
*     the timeout in milliseconds.
* @return the response of the command
* @throws FirmwareException
*     if an error occurs.
*/
private String doCommand(byte[] packet, int timeout)
    throws FirmwareException {
    int off = (packet[1] & 0x80) == 0x80 ? 1 : 0;
    int cmdId = packet[2 + off] + 0x100 & 0xff;
    int ansId = packet[3 + off] + 0x100 & 0xff;

    eraseInputBuffer();
    try {
        deviceOS.write(packet);
    } catch (IOException e) {
        throw new FirmwareException("Cannot send binary command " + cmdId
            + " to device.", e);
    }

    String tStart = "$<" + (cmdId < 10 ? "0" : "") + cmdId + ">\r\n";
    String tEnd = "$<end:" + (ansId < 10 ? "0" : "") + ansId + ">\r\n";

    return getResponse(tStart, tEnd, timeout);
}

/**
* Waits for a response and extracts it from the input buffer.
*
* @param start
*     the start string of the response.
* @param end
*     the end string of the response.
* @param timeout
*     the timeout in milliseconds.
* @return the response from the device enclosed by the start and end
*     strings.
*/

```

```

private String getResponse(String start, String end, int timeout) {
    if (timeout == 0)
        return null;
    String uStart = start.toUpperCase();
    String uEnd = end.toUpperCase();
    synchronized (inputBuffer) {
        long time = System.currentTimeMillis() + timeout;
        long remaining = timeout;
        while (remaining > 0) {
            String uInput = inputBuffer.toString().toUpperCase();
            int i = uInput.indexOf(uStart);
            int j = -1;
            if (i >= 0) {
                i += uStart.length();
                j = uInput.indexOf(uEnd, i);
            }
            if (j >= 0)
                return inputBuffer.substring(i, j);

            try {
                inputBuffer.wait(remaining);
            } catch (InterruptedException e) {
            }
            remaining = time - System.currentTimeMillis();
        }
        return null;
    }
}

/**
 * Starts the read thread. This thread reads the device input stream and
 * writes the input in the input buffer.
 */
private void startReadThread() {
    closeReadThread = false;
    readThread = new Thread() {
        public void run() {
            byte[] buffer = new byte[1024];
            for (;;) {
                synchronized (UpdateFirmware.this) {
                    if (closeReadThread) {
                        readThread = null;
                    }
                }
            }
        }
    };
}

```

```

        UpdateFirmware.this.notifyAll();
        return;
    }
}
try {
    int n = deviceIS.read(buffer);
    if (n > 0) {
        synchronized (inputBuffer) {
            inputBuffer.append(bytesToString(buffer, 0, n));
            inputBuffer.notifyAll();
        }
    }
} catch (IOException e) {
    closeReadThread = true;
}
}
};
readThread.start();
}

/**
 * Stops the read thread.
 */
private void stopReadThread() {
    synchronized (this) {
        closeReadThread = true;
        while (readThread != null) {
            readThread.interrupt();
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
    }
}

private static void showUsage() {
    System.out.println("Usage: java UpdateFirmware serial-port firmware-file-path compressed");
    System.out.println("           params: port -----> if updating over serial port enter COMx
           where x = port number");
    System.out.println("           params: firmware-file-path ---> Path to the Firmware file.");
}

```

```

        System.out.println("
                                params: compressed ---> whether the firmware is compressed
                                using GZIP compression." +
                                "\r\n Enter true if compressed else false");
    }
// -----
// The main function.
// -----

public static void main(String[] args) {
    String filename = null;
    String tPortname = null;
    boolean compressed = false;
    boolean tTCP = false;
    if( args.length != 3 ){
        showUsage();
        return;
    }

    System.out.println("port: " + args[0]);
    System.out.println("path: " + args[1]);
    System.out.println("compression: " + args[2]);
    try
    {
        tPortname = args[0].toUpperCase();
    } catch (Exception tEx){
        showUsage();
        return;
    }
    try
    {
        filename = args[1]; //"C:\\firmware\\steppII_2.4.3_rc4.bin.gz";
    } catch (Exception tEx){
        showUsage();
        return;
    }

    try
    {
        if(String.valueOf(args[2]).equals("true"))
            compressed = true;
        if(String.valueOf(args[2]).equals("false")){

```

```

        compressed = false;
    }else{
        System.out.println("Parameter "+args[2] +" seems to be wrong !!!");
        showUsage();
        return;
    }
} catch (Exception tEx){
    showUsage();
    return;
}

int sectors = 14;

byte[] firmware = null;
int length = 0;
// First read the file into the array firmware[]
try {
    File firmwareFile = new File(filename);
    if (!firmwareFile.isFile())
        throw new IOException("File not found.");
    long lengthL = firmwareFile.length();
    if (lengthL > MAX_FIRMWARE_LEN)
        throw new IOException("Firmware file too long.");
    length = (int) lengthL;

    ByteArrayOutputStream baos = new ByteArrayOutputStream(length / 2);
    FileInputStream fis = new FileInputStream(firmwareFile);
    int bufSize = 4096;
    byte[] bytes = new byte[bufSize];
    for (int i = 0; i < length; i += bufSize) {
        int size = bufSize;
        if (length - i < size)
            size = length - i;
        if (fis.read(bytes, 0, size) != size)
            throw new IOException("Unable to read bytes.");
        baos.write(bytes, 0, size);
    }
    firmware = baos.toByteArray();
} catch (IOException e) {
    System.out.println("Error reading firmware file.");
    e.printStackTrace(System.out);
    System.exit(1);
}

```

```

}

// Open the device connected to the serial port.
OutputStream deviceOS = null;
InputStream deviceIS = null;
SerialPortLocal serialPort = null;
if(tPortname != null){
    try {
        SerialConfig serialConfig = new SerialConfig();
        serialConfig.setPortName(tPortname);
        serialConfig.setBitRate(SerialConfig.BR_57600);
        serialConfig.setParity(SerialConfig.PY_NONE);
        serialConfig.setDataBits(SerialConfig.LN_8BITS);
        serialConfig.setStopBits(SerialConfig.ST_1BITS);
        serialConfig.setHandshake(SerialConfig.HS_NONE);
        serialConfig.setTxLen(1024);
        serialPort = new SerialPortLocal(serialConfig);
        serialPort.setDTR(true);
        deviceOS = new SerOutputStream(serialPort);
        deviceIS = new SerInputStream(serialPort);
    } catch (IOException e) {
        System.out.println("Error opening serial port. "+ tPortname);
        e.printStackTrace(System.out);
        System.exit(1);
    }
}

// Start the update.
UpdateFirmware uf = new UpdateFirmware();
uf.startUpdate(deviceIS, deviceOS, tTCP, firmware, sectors, compressed);

// Display status throughout process.
System.out.print("0: 0%");
long tStart = System.currentTimeMillis();
while (!uf.waitReady(1000)){
    StringBuilder s = new StringBuilder();
    s.append("\r").append((int) Math.floor(uf.getFractionDone() * 100));
    s.append("% Done in ").append((System.currentTimeMillis() - tStart) / 1000).append(" [ sec ]");

    System.out.print(s.toString());
}
System.out.println();

```

```
if (uf.success()) {
    System.out.println("SUCCESS after "
        + (System.currentTimeMillis() - tStart) / 1000 + " s.");
} else {
    System.out.println("ERROR after "
        + (System.currentTimeMillis() - tStart) / 1000 + " s.");
    System.out.println(uf.getError());
}

// Close the serial port.
try {
    deviceIS.close();
    deviceOS.close();
    serialPort.close();
} catch (IOException e) {
}
}
}
```