FALCOM

Transform history binary data in NMEA format

For AVL devices and STEPPII, MAMBO/2

Application Notes

Version: 1.1.0; Last Modified: Wednesday 29 October 2014

Version history:

This table provides a summary of the document revisions.

Number	Author	Changes	Modified
1.1.0	G. Voigt	- Add bugfix for Date number Leap -See NOTE in chapter 3	15.01.2014
1.0.5	F. Beqiri	- Added an example for **Motorway entry** - see table **Table 1.4.** - For AVL devices, the ANA0 and ANA1 values are multiplied by 1000 with an accuracy of 0.001 - see table **Table 1.7.**	04.02.2010
1.0.4	F. Beqiri	- Added chapter 1.3. - Corrected the number of bits in the example Sats of the City entry (from 4 bits to 3 bits). - Updated the internet address for downloading the firmware binary file and Java source code snippets – See chapter 4.3, page 23.	19.03.2008
1.0.3	F. Beqiri	- Corrected description of value 0x10 in the extension entry – see Table 1.7. Its description has changed from Not implemented to Analog inputs values.	19/06/2008
1.0.2	F. Beqiri	- Fix parameter in Tables 1.3, 1.4 and 1.5 changed (0 = invalid or last valid position). Available in the firmware version *2.4.3_rc4* and later. - Added a complete java source code for decoding history data – see chapter 4.3, page 23.	22/03/2007
1.0.1	F. Beqiri	- Dz-position and Sats in Table 1.6 corrected	09/01/2007
1.0.0	F. Beqiri	- Initial version	18/10/2006

Table of contents

Cautions

Information furnished herein by FALCOM is believed to be accurate and reliable. However, no responsibility is assumed for its use. Please, read carefully the safety precautions.

If you have any technical questions regarding this document or the product described in it, please contact your vendor.

General information about FALCOM and its range of products are available at the following Internet address: http://www.falcom.de/

Trademarks

Some mentioned products are registered trademarks of their respective companies.

Copyright

FALCOM GmbH.

No patent liability is assumed with respect to the use of the information contained herein.

Note

Specifications and information given in this document are subject to change by FALCOM without notice.

1 ABOUT THIS DOCUMENT

This application note is relating to the following products: **STEPPII, STEPPIII, FOX3, FOX-IN/EN/LT, BOLERO-LT/LT2, MAMBO** and **MAMBO2**. This document provides all the necessary information to get the GPS data in the right format that has been logged by one of the AVL products.

1.1 Audience

This document is intended for system integrators and application developers.

1.2 How this document is organized

This guide consists of following chapters:

- Chapter **1.3 "Related documents"** gives you general information about how to get the GPS data logged by one of the FALCOM products. It describes also the format of the binary data that can be read (via serial link) or remotely (via a data or TCP connection). It represents some simply examples how to convert this history data into the NMEA format.
- Chapter **3 "Example Decoding History Format 1.0"** gives you step by step instructions how to read out the history from the target device and how to convert the retrieved history binary data into the NMEA format.
- Chapter **4 "Appendix"** gives you some functions and methods needed during converting this binary data.

1.3 Related documents

Refer to the chapter "Related documents" in the:

1. AVL_PFAL_Configuration_Command_Set_x.x.x.pdf
2. MAMBO2PFALCommandsReferenceGuide_x.x.x.pdf
3. Mambo_firmware_2.4.6_user_manual.pdf

2 RETRIEVING HISTORY BINARY DATA

Stored GPS history data into the history space memory can be retrieved either locally (via serial link) or remotely (via a data or TCP connection). When such a connection is already established, to retrieve the History Binary Data you should send two executable command to the STEPPII device.

1. first execute the **GPS.History.SetRead** command and specify the range of history for readout (i.e. mark the complete history for readout),
2. the execute the **GPS.History.Read** command to read the first part of history data,
3. continue executing the **GPS.History.Read** commands to retrieve the next parts of history data – until readout is complete (see PFAL commands for more details).

RECOMMENDATION:

It is not recommended to write positions when reading out history when the history section is filled completely.

In this case the history has to erase old data from time to time in order to write new datasets. Although it is **very unlikely** it might theoretically happen that a record is written directly after a **GPS.History.SetRead** command or when reading out the very first records of the history. This Write Command could then cause a cleanup for the oldest data available. If this data has been selected for readout (and isn't read out yet), the history readout command might stop before the complete selection is read out.

This happens just for the first 64KB of data read out.

If it is not possible (or not desired) to stop history write commands when reading history, the readout algorithm must be extended with:

If **SetRead** returns an approximate length of larger than 65 KB, the amount of data read out should be counted. If Readout ends before the first 64 KB are read out, the complete readout should be repeated (its also possible to specify another timespan – starting at the last correct position read out).

Alternatively: if a history readout doesn't contain the expected timespan (ends much earlier than expected), simply perform this readout again.

Once, the STEPPII receives **GPS.History.Read** command, it starts to transfer the history selected data to the receiving device.
The history data is subdivided in packets, which are constructed in a specific binary format to make possible low-cost solutions where data is downloaded via GSM/GPRS connection.

The common structure of the retrieved data is shown in Table1.1.

	Data Header	Binary data	Additional bytes	End Sequence
	<2_bytes>	<binary history data>	<additional>	<CR><LF>
Example	00 13	1E 00 30 A6 1D F9 1E 56 52 05 E2 E7 25 77 BE F0 00 40 00	00	0D 0A

Table 1.1: The format of the retrieved data.

Each sent TCP packet may include more than one **GPS.History.Read** answer**.** Each Data Header is followed by binary data, additional bytes and terminated by Carriage Return and Line Feed. The Data Header has a fixed length, which consists of 2 Bytes of data, whereas the Binary Data has a variable length. The value of the Data Header determines the length of the binary data, only. In example above, the value "**00 13"** converted to decimal gives a size of **19 Bytes** of Binary Data**.**

Definition: A **packet** is one unit of binary data capable of being routed through networks. To improve communication performance and reliability, the STEPPII device subdivides its history data in packets with a length of 512 bytes and sends them in from packets to the receiving device.

Based on the value of the Data Header the user can re-assemble each packet into the original values, by stripping off the Data Header, reading the number of bytes defined by the Data Header, decoding the binary data as described in tables below and concatenating each decoded values in the correct sequence to get the data in NMEA format required for different applications.

The history data packet contains a queue of single history entries. Each of this entries can *(but must not necessarily)* have an extension, which is illustrated in the following picture. However, a **packet** may contain up to 4 different entry types.

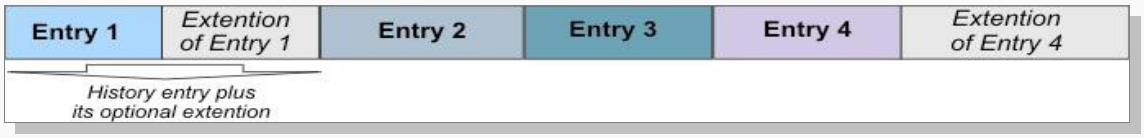

Figure 1: Binary data construction

Each entry includes one type ID and each type ID will give you information about the the data format of this entry .

The very first history entry of this queue is always a **Full entry**, which contains absolute data. So when decoding this entry, the true position, time etc.. is known. Following entries can contain differential positions or timestamps. In order to compute e.g. their absolute position, the last absolute position (from the previous entry) needs to be added to the (delta) position of this entry. The result is an absolute position. Full entries are also found on later position within the queue. This also allows corrupted/incorrect data to resynchronize - all differential entries following a correct full entry will have correct positions.

The entry headers are shown in Table 1.2.

Type ID (binary value) *(the high 2 bits of the first byte)*		**Meaning**
00	absolute position	**Full entry.** The size in bytes of the binary data that follows this value is **15 Bytes**. From this entry, you are able to determine absolute time, X-,Y- and Z-position for other entries.
01	differential positions	**Motorway entry.** The size in bytes of the binary data that follows this value is **9 Bytes.** Time and Position are stored as differential value.
10		**City entry.** The size in bytes of the binary data that follows this value is **6 Bytes**. Time and Position are stored as differential value.
11		**Standing/stationary entry.** The size in bytes of the binary data that follows this value is **4 Bytes** Time and Position are stored as differential value.

Table 1.2: The entry headers for each entry type.

2.1 History format description

The following 4 tables show the format of full, motorway, city and standing entries.

Note that, the format type is always located within the first **2 Bits** (Byte0, bit 7 and 6), so the correct type can figured out easily for any history entry. Each history entry can have an optional extension. If such an extension exists, its data is directly appended on its history entry. Of course this extension data has to be read (or skipped) in order to find the position of the next history entry. Therefore, it is necessary to decode history entries as well as their extensions. Please, refer to chapter **3** on page **14** (Example Decoding History Format 1.0) for more detailed information.

Full entry format						
Contents	**Position**		**Bit length**	**Description**	**Example**	
	Byte	**Bit**			**Bin**	**NMEA**
Entry signs	0	7:6	2	**Full entry.** This entry consists of **15 Bytes** of data.	00	
Sats	0	5:2	4	Determines the number of satellites in use. "**0000**" ≙ 0 satellites have been in use. "**1111**" ≙ more than 8 satellites have been in use.	01 11	7
Fix	0	1	1	Determines the state of GPS fix. It depends on the number of satellites that have been in use (see **Sats**). "**1**" ≙ valid "**0**" ≙ invalid or last valid position.	1	A
Ext	0	0	1	Signifies whether or not extention data is attached in the end of this entry. "**1**" ≙ extension data attached "**0**" ≙ no extension data attached. Please, refer to the **Table 1.7** for more details.	0	No
Speed	1	7:1	7	Determines the speed, in meter per second (m/s), over the ground:	0000 000	0 m/s
Signs	1	0	3	Currently reserved. Do not use these bits.	0	-
	2	7:6			00	
Time[1)]	2:5	-	30	Determines the number of seconds since Saturday/Sunday Midnight UTC (**01/06/1980**). The hexadecimal value must be converted into the DateTime format. Use chapter Convert Time as reference.	**30 A6 1D F9** (hex)	16/11/2005 16:16:57
X-position	6:8	-	24	Determines the value, in meter, of the X-position of the device. The high bit (bit 23) is the sing bit for the X-position, which signifies whether the value is negative or positive. Do not use this bit to the **X**-position value when converting to decimal. *If bit 23="1", then multiply the obtained value of the X-position by –1 (negative value). In this example it is a positive value (bit 23="0"). Multiply the obtained value by 2.*	1E 56 52 (hex)	lat: 50.6733376 lon: 10.9806853 alt: 489.56
Y-position	9:11	-	24	Determines the value, in meter, of the Y-position of the device. The high bit (bit 23) is the sing bit for the X-position, which signifies whether the value is negative or positive. Do not use this bit to the **Y**-position value when converting to decimal. *If bit 23="1", then multiply the obtained value of the Y-position by –1 (negative value). In this example it is a positive value (bit 23="0"). Multiply the obtained value by 2.*	05 E2 E7 (hex)	
Z-position	12:14	-	24	Determines the value, in meter, of the Z-position of the device. The high bit (bit 23) is the sing bit for the Z-position, which signifies whether the value is negative or positive. Do not use this bit to the **Z**-position value when converting to decimal. *If bit 23="1", then multiply the obtained value of the Z-position by –1 (negative value). In this example it is a positive value (bit 23="0"). Multiply the obtained value by 2.*	25 77 BE (hex)	

1) When all x, y, z values has been obtained, finally convert these values (ECEF format) using the function in chapter Convert X,Y,Z positions page 21.

Table 1.3: The binary data format of the full entry.

Example for Full entry *(this example is also represented in diagrammatic form, see figure 2 below)*							
Example (hex)	**1E 00 30 A6 1D F9 1E 56 52 05 E2 E7 25 77 BE** (15 Bytes, see Full entry in Table 1.1)						
	Byte 0	**Byte 1**	**Byte 2**	**Byte 3..5**	**Byte 6..8**	**Byte 9..11**	**Byte 12..14**
Divided bytes	**1E**	**00**	**30**	**A6 1D F9**	**1E 56 52**	**05 E2 E7**	**25 77 BE**
Converted in binary	**0001 1110**	**0000 0000**	**0011 0000**	**1010 0110 0001 1101 1111 1001**	**0001 1110 0101 0110 0101 0010**	**0000 0101 1110 0010 1110 0111**	**0010 0101 0111 0111 1011 1110**

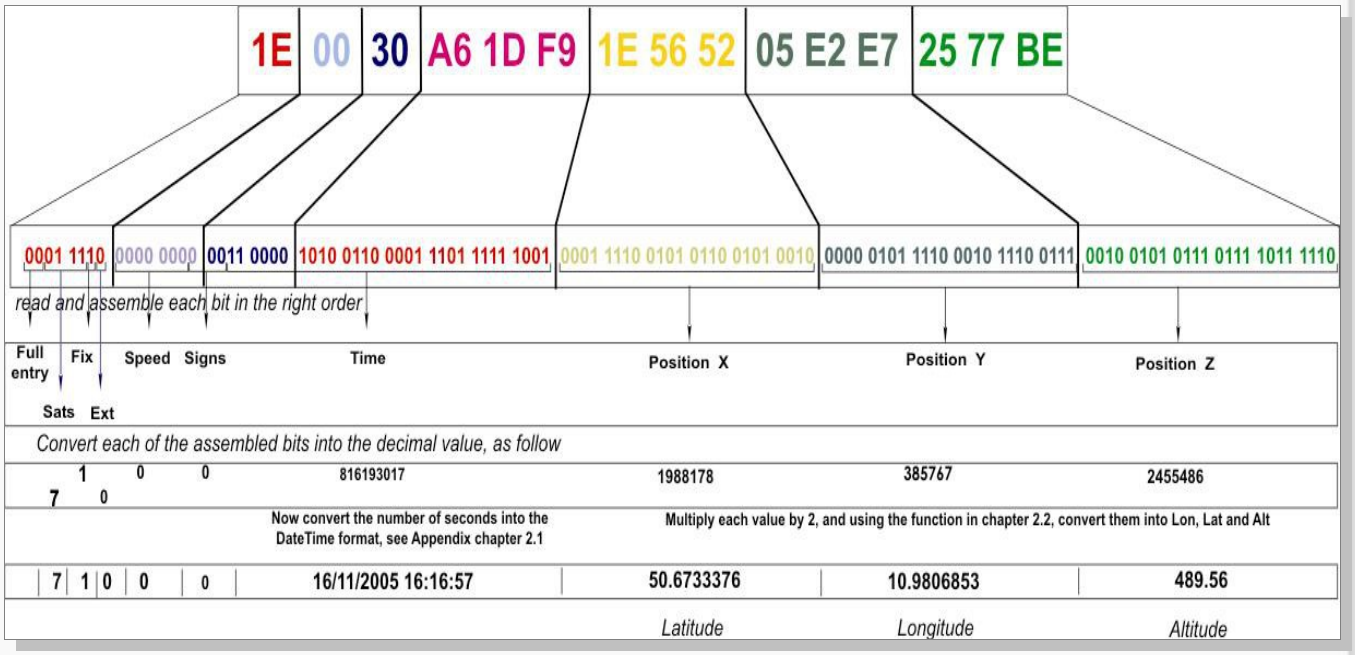

Figure 2: Example represented in diagrammatic form

Motorway entry format									
	Byte 0	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**	**Byte 6**	**Byte 7**	**Byte 8**
HEX	**62**	**01**	**C2**	**70**	**05**	**40**	**0C**	**80**	**0D**
BIN	0110 0010	0000 0001	1100 0010	0111 0000	0000 0101	0100 0000	0000 1100	1000 0000	0000 1101
Counting	$bit^7 \leftarrow bit^0$	$bit^7 \leftarrow bit^0$	$bit^7 \leftarrow bit^0$	$bit^7 \leftarrow bit^0$	$bit^7 \leftarrow bit^0$	$bit^7 \leftarrow bit^0$	$bit^7 \leftarrow bit^0$	$bit^7 \leftarrow bit^0$	$bit^7 \leftarrow bit^0$

Contents	**Position**		**Bit length**	**Description**	**Example**	
	Byte	**Bit**			**Bin**	**NMEA**
Entry type	0	7:6	2	Motorway entry**.** This entry consists of **9 Bytes** of data.	01	
Sats	0	5:2	4	Determines the number of satellites in use. "**0000**" ם 0 satellites have been in use. "**1111**" ם more than 8 satellites have been in use.	1000	8
Fix	0	1	1	Determines the state of GPS fix. It depends on the number of satellites that have been in use (see **Sats**). "**1**" ם valid "**0**" ם invalid or last valid position.	1	1
Ext	0	0	1	Signifies whether or not extention data is attached in the end of this entry. "**1**" ם extension data attached "**0**" ם no extension data attached. Please, refer to the **Table 1.7** for more details.	0	0
Speed	1	7:1	7	Determines the speed, in meter per second (m/s), over the ground:	0000 000	0
Time[1)] dt	1 2 3	0 7:0 7:5	12	Determines the number of seconds that have been passed from the number of seconds of the **last entry**. This value has to be added to the Time of the **last entry** to obtain the number of seconds since Saturday/Sunday Midnight UTC (**01/06/1980**). Then convert it into the DateTime format, use chapter Convert Time as reference.	111000010011	3603
dx-position	3 4 5	4:0 7:0 7:6	15	It determines the distance, in meter, of the X - position of the device from the **last entry**. The sign bit is the bit 4 of byte 3. Do not use this bit to the **dx**-position value when converting to decimal. *If bit 4="1", then multiply the obtained value of the X-position by –1. In this example it is a negative value (bit 4="1").* Thereafter, multiply that value by 2 and add the result to the X-position of the **last entry.**	1 10101	-21
dy-position	5 6 7	5:0 7:0 7	15	It determines the distance, in meter, from the Y-position value of the device from the **last entry**. The sign bit is the bit 5 of byte 5. Do not use this bit to the **dy**-position value when converting to decimal. *If bit 5="1", then multiply the obtained value of the Y-position by –1. In this example it is a negative value (bit 5="0").* Thereafter, multiply that value by 2 and add the result to the Y-position of the **last entry**.	0 00 00000001 1001	25
dz-position	7 8	6:0 7:0	15	It determines the distance, in meter, from the Z-position value of the device from the **last entry**. The sign bit is the bit 6 of byte 7. Do not use this bit to the **dz**-position value when converting to decimal. *If bit 6="1", then multiply the obtained value of the Z-position by –1. In this example it is a negative value (bit 6="0").* Thereafter, multiply that value by 2 and add the result to the Z-position of the **last entry**.	0 00 00000000 1101	13

[1)] *when all **dx, dy, dz** values has been obtained, then convert them using the function in chapter Convert X,Y,Z positions page **21**.*

Table 1.4: The binary data format of the motorway entry.

City entry format						
Contents	**Position**		**Bit length**	**Description**	**Example**	
	Byte	**Bit**			**Bin**	**NMEA**
Entry type	0	7:6	2	**City entry.** This entry consists of **6 Bytes** of data.	10	
Sats	0	5:3	3	Determines the number of satellites in use. *"**000**" ↵ 0 satellites have been in use.* *"**111**" ↵ more than 8 satellites have been in use.*	-	-
Fix	0	2	1	Determines the state of GPS fix. It depends on the number of satellites that have been in use (see **Sats**). *"**1**" ↵ valid* *"**0**" ↵ invalid or last valid position.*	-	-
Ext	0	1	1	Signifies whether or not extention data is attached in the end of this entry. *"**1**" ↵ extension data attached* *"**0**" ↵ no extension data attached.* Please, refer to the **Table 1.7** for more details.	-	-
Speed	0 1	0 7:4	5	Determines the speed, in meter per second (m/s), over the ground:	-	-
Time dt	1 2	3:0 7:3	9	It represents the number of seconds that have been passed from the number of seconds of the **last entry**. This value has to be added to the Time of the **last entry** to obtain the number of seconds since Saturday/Sunday Midnight UTC (**01/06/1980**). Then convert it into the DateTime format, use chapter Convert Time as reference.	-	-
dx-position[1)]	2 3	2:0 7:2	9	It determines the distance, in meter, from the X-position value of the **last entry**. The sign bit is the bit 2 of byte 2. Do not use this bit to the **dx**-position value when converting to decimal. *If bit 2="1", then multiply the obtained value of the X-position by –1.* Thereafter, multiply that value by 2, and add the result to the X-position of the **last entry.**	-	-
dy-position[1)]	3 4	1:0 7:1	9	It determines the distance, in meter, from the Y-position value of the **last entry**. The sign bit is the bit 1 of byte 3. Do not use this bit to the **dy**-position value when converting to decimal. *If bit 1="1", then multiply the obtained value of the Y-position by –1.* Thereafter, multiply that value by 2 and add the result to the Y-position of the **last entry**.	-	-
dz-position[1)]	4 5	0 7:0	9	It determines the distance, in meter, from the Z-position value of the **last entry**. The sign bit is the bit 0 of byte 4. Do not use this bit to the **dz**-position value when converting to decimal. *If bit 0="1", then multiply the obtained value of the Z-position by –1.* Thereafter, multiply that value by 2, add the result to the Z-position of the **last entry**.	-	

[1)] *when all **dx, dy, dz** values has been obtained, then convert them using the function in chapter Convert X,Y,Z positions page **21**.*

Table 1.5: The binary data format of the city entry.

Standing/stationary entry (*example*)				
Example (hex)	**F0 00 40 00**			*(4 Bytes, see record in Table 1.1)*
	Byte 0	**Byte 1**	**Byte 2**	**Byte 3**
Split bytes	**F0**	**00**	**40**	**00**
Converted on binary	**1111 0000**	**0000 0000**	**0100 0000**	**0000 0000**

Standing/Stationary entry format						
Contents	**Position**		**Bit length**	**Description**	**Example**	
	Byte	**Bit**			**Bin**	**NMEA**
Entry type	0	7:6	2	**Standing/stationary entry.** This entry consists of **4 Bytes** of data.	11	
Sats	0	5:4	2	Number of satellites in use. "**00**" = 0 satellites have been in use (GPS-fix invalid). "**01**" = min. 3 satellites have been in use (GPS-fix valid). "**10**" = min. 5 satellites have been in use (GPS-fix valid). "**11**" = min. 7 satellites have been in use (GPS-fix valid). In this example the number of satellites in use is 3.	**11**	**3, 7 or more**
Fix	-	-	0	Determines the state of GPS fix. It depends on the number of satellites that have been in use (see **Sats**). "**1**" = valid "**0**" = invalid or last valid position.	-	**V**
Ext	0	3	1	Signifies whether or not extention data is attached in the end of this entry. "**1**" = extension data "**0**" = no extension data. Please, refer to the **Table 1.7** for more details.	**0**	**No**
Speed	0	2:0	3	Determines the speed, in meter per second (m/s), over the ground:	**000**	**0 m/s**
Time dt	1	7:0	12	It represents the number of seconds that have been passed from the number of seconds of the **last entry**. To obtain the number of seconds since Saturday/Sunday Midnight UTC (**01/06/1980**), This value has to be added to the Time of the **last entry**. Here is the calculation: **816193017 + 4 = 816193021.** Then convert it into the DateTime format, use chapter Convert Time as reference.	0000 0000	**16/11/2005 16:17:01**
	2	7:4			**0100**	
dx-position	2	3:0	4	It determines the distance, in meter, from the X-position value of the **last entry**. The sign bit is bit 3 of byte 2. *If bit 3="1", then multiply the obtained value of the X-position by –1.* Thereafter, multiply that value by 2, add to the x-position of the **last entry** and then convert it using the function in chapter Convert X,Y,Z positions.	**0000**	**lat: 0.0 lon: 0.0 alt: 0.0**
dy-position	3	7:4	4	It determines the distance, in meter, from the Y- position value of the **last entry**. The sign bit is the bit 7 of byte 3. *If bit 7="1", then multiply the obtained value of the Y-position by –1.* Thereafter, multiply that value by 2, add to the Y-position of the **last entry** and then convert it using the function in chapter Convert X,Y,Z positions.	**0000**	
dz-position	3	3:0	4	It determines the distance, in meter, from the Z-position value of the **last entry**. The sign bit is the bit 4 of byte 3. *If bit 3="1", then multiply the obtained value of the Z-position by –1.* Thereafter, multiply that value by 2, add to the Z-position of the **last entry** and then convert it using the function in chapter Convert X,Y,Z positions.	**0000**	

[1] *when all **dx, dy, dz** values has been obtained, then cnvert them using the function in chapter Convert X,Y,Z positions page **21**.*

Table 1.6: The binary data format of the Standing/stationary entry.

2.2 Description of the Extention entry

These extensions can be optionally appended after any history entry. They contain additional information, which couldn't be stored within the entry itself. The format of this extension is equal for each type of history entry (whether its a full, motorway, city or standing record). However the true contents of the extension can vary from history entry to history entry (i.e. one record contains additional data for IO-states, another contains user text etc...)

Extension entry						
Entry headers	**Read Byte**	**Example (hex value)**		**Description**		
<1_byte>	1	0000 001=0x01 (1 x 2 = 2)		The first byte of this entry defines the length of the data followed the first byte <1_byte>. Once the first byte is read and converted in hexadecimal value, to get the number of bytes attached in the end of the extension entry**,** you have to multiply the result of <1_byte> by 2. In our example the length of the **extension entry** is 4 bytes.		
<2_byte>	2	0000 0100=0x04		The second byte of the extension entry signifies which kind of messages includes the **Extension entry.** Once the second byte is received and it is converted in hexadecimal value, the result must be compared with one or more values listed in table below. In this example the result is "**0x04**".		
Extension binary data value	**Read Byte**	**Read Byte Range**	**Read Bit position**	**Description**	**Example**[1)] **Bin**	**NMEA**
0x01	0	0:1	7:0	It shows the state of the provided inputs of the STEPPII device (10010110 = IN7... IN0 - means **IN7,4,2,1** are HIGH)		
	1		7:0	It shows the state of the provided outputs of the STEPPII device. (0000110 = OUT7... OUT0 - means OUT**2,1** are HIGH)		
0x02	0	0:9		It shows the field strength of GSM state (usually 0-99 decimal)		
	1:2			It shows the local area code of GSM (in decimal)		
	3:4			It shows the cell ID of GSM (in decimal)		
	5			It shows the final state machine	For debug purposes only.	
	6			It shows the GSM call state		
	7			It shows the GSM registration state		
	8			It shows the number of incoming SMS (in decimal)		
	9			It shows the number of outgoing SMS (in decimal)		
0x04	0	0:7		It shows the GPRS state	For debug purposes only.	
	1			It shows the PPP state		
	2			It shows the TCP state		
	3			It shows the main task state		
	4:7			It shows the system life time in milliseconds		
0x08	0..3-X	3-X		Reserved (empty)		
0x10	0 1 2 3	0:3		Analog inputs values (for STEPPII the ANA0 and ANA1 values are multiplied by 10 => accuracy 0.1 while for STEPPIII are multiplied by 1000 => accuracy 0.001). Byte **0** – High byte **ANA0**; Byte **1** – Low byte **ANA0** Byte **2** – High byte **ANA1**; Byte **3** – Low byte **ANA1** ** Hint: For BOLERO, the number of bytes to read is: **0** ... **7**.* *Byte **0** – High byte I/O**0**; Byte **1** – Low byte I/O**0*** *and so on ....* *Byte **6** – High byte I/O**3**; Byte **7** – Low byte I/O**3***		
0x20	0	x:x+1		It shows the length of text (maximum: 0xFF characters)		
	1 - x			It shows the specified text		
0x40	0			It shows the state of the areas **0-7** (inside or outside of an area). (10010110 = AREA7... AREA0 - means inside **AREA7,4,2**)		
	1			It shows the state of the areas **8-15** (inside or outside of an area). (00001100 = AREA15... AREA8 - means inside **AREA10,11**)		
	2			It shows the state of the areas **16-23** (inside or outside of an area). (00000000 = AREA23... AREA16 - means outside these areas)		
	3			It shows the state of the areas **24-31** (inside or outside of an area). (00000000 = AREA31... AREA24 - means outside these areas)		
0x80	-	-	-	reserved		

[1)] *See chapter 3 page 14*

Table 1.7: The binary data format of the extension entry.

3 EXAMPLE DECODING HISTORY FORMAT 1.0

This chapter will geive you a complete Example for manually decoding History entries (plus their extensions).

3.1 Algorithm

- **Writing History records:**
  - ✔ Perform a **GPS.History.Write** command (additional data can be specified if desired– see PFAL commands for more details).
- **Readout:**
  - ✔ Perform a **GPS.History.Setread** command to specify the range of history for readout (i.e. mark the complete history for readout).
  - ✔ Perform a **GPS.History.Read** command to read the first part of history data
  - ✔ Continue performing **GPS.History.Read** commands to retrieve the next parts of history data – until readout has completed (see PFAL commands for more details)

3.2 Example:

The following example also contains history extensions which are also decoded.

Background information:

- History extensions are not necessarily within a history – they are an option, which can be used to attach additional data to a single history record.
- If a record contains extension data or it cannot be determined by reading the **"extension"** bit of this record. Please, see History format documentation for more details.

3.2.1 Writing data to history

Lets fist assume we wrote 2 history records using the following command:

```
$PFAL,gps.history.write,20,"user txt time=&(TIME) date= &(date)"
```

This means each history record will have additional user data attached – showing the current system time and date (usually time and date is also decoded within history, so it makes no sense to add such information as it only increases the size of our records – but here it is a good way showing the time and date of a record).

3.2.1.1 1st Record

Our current position is:

```
lat:      50.6733666,
lon:  10.9806515,
alt:  483.43,
time: d=28. m=09.y=2006,12:26:09
fix:  1,
sats: 8,
speed:0
```

ECEF coordinates of this positions divided by 2 are:

```
X:        1E55C8,
Y:      5E2CB,
Z:      257715,
```

The time converted to hexadecimal format is :

```
0x324681E1
```

Having debug information enabled allows to compare this data with the records read out – the data should be the same ...

If debug information is enabled, the following entry can be seen:

```
$DBG1207|#1>WriteHistory: 28. 9.2006,12:26: 9 pos:0x40110000 fix:1 lat:50.6733666,
lon:10.9806515, alt:483.43, format:HIST_RECORD_
$DBG1207|#2>FULL
```

→ a full record is written at the history, which is the normal case – each history starts with a full record. After a full record, smaller records can be written to history, which saves additional space.

The debug information also shows the hexadecimal values of this history entry (shown as hexadecimal values):

```
DBG1203|WriteHistory: write 0x3C bytes at at 0x40110000: [Hex Info of 60 Bytes]
$DBG1203|[string+0]:"#.2F...U....w.. (user txt time=12:"
$DBG1203|[Byte 0]:23,00,32,46,81,E1,1E,55,C8,05,E2,CB,25,77,15,16,20,28,75,73,65,72,20,74,
78,74,20,20,74,69,6D,65,3D,31,32,3A,
$DBG1203|[string+36]:"26:09 date= 28.09.2006.."
$DBG1203|[Byte 36]:32,36,3A,30,39,20,64,61,74,65,3D,20,32,38,2E,30,39,2E,32,30,30,36,00,FF,
```

(the last 0xFF should be ignored right now – its not part of the message)

3.2.1.2 2nd Record

```
$DBG1207|#1>WriteHistory: 28. 9.2006,12:26:10 pos:0x4011003b fix:1 lat:50.6733666,
lon:10.9806515, alt:483.43, format:HIST_RECORD_
$DBG1207|#2>STANDING
$DBG1203|WriteHistory: write 0x32 bytes at at 0x4011003a: [Hex Info of 50 Bytes]
$DBG1203|[string+0]:"..... (user txt time=12:26:10 date"
$DBG1203|[Byte
0]:00,F8,00,10,00,16,20,28,75,73,65,72,20,74,78,74,20,20,74,69,6D,65,3D,31,32,3A,32,36,3A,31,30,
20,64,61,74,65,
$DBG1203|[string+36]:"= 28.09.2006.."
$DBG1203|[Byte 36]:3D,20,32,38,2E,30,39,2E,32,30,30,36,00,FF,
```

(the first 0x00 and the last 0xFF should be ignored right now – its not part of the message)

3.2.2 Read out the history

3.2.2.1 SetRead - mark history for reading

Lets assume we want to read out the complete history and therefore send the command:

```
$pfal,gps.history.setread,all
```

The device answers:

```
$<GPS.History.SetRead>
$107 Bytes (0 KB) estimated for readout inside history
$SUCCESS
$<end>
```

So we have approximately **107 Bytes** to expect for readout data *(the true amount of data can be a bit lower than this estimated value, its a first approximation only)*

3.2.2.2 Read out binary history

After sending the command **PFAL,Gps.History.Read** the device sends the binary data within its answer (*if there are more than 510 bytes to read out, a **512 byte** package is shown for each read command - so several read commands are neccessary to read out the selected history).*

Here is a hexadecimal notification of the received command.

When reading out you should assure that your terminal application does not make any conversion upon the received binary characters – else history conversion will fail *(as the data gets corrupted).*

Binary information is shown in colors here - grey for length information (always the first **2 bytes**), The first record is marked as green – the **2nd** one as yellow.

```
24 3C 47 50 53 2E 48 69 73 74 6F 72 79 2E 52 65 $<GPS.History.Re
61 64 3E 0D 0A                                  ad>
00 6B 23 00 32 46 81 E1 1E 55 C8 05 E2 CB 25 77
15 16 20 28 75 73 65 72 20 74 78 74 20 20 74 69  (user txt ti
6D 65 3D 31 32 3A 32 36 3A 30 39 20 64 61 74 65  me=12:26:09 date
3D 20 32 38 2E 30 39 2E 32 30 30 36 00 F8 00 10  = 28.09.2006
00 16 20 28 75 73 65 72 20 74 78 74 20 20 74 69  (user txt ti
6D 65 3D 31 32 3A 32 36 3A 31 30 20 64 61 74 65  me=12:26:10 date
3D 20 32 38 2E 30 39 2E 32 30 30 36 00 0D 0A     = 28.09.2006
24 72 65 61 64 6F 75 74 20 63 6F 6D 70 6C 65 74  $readout complet
65 64 0D 0A                                     ed
24 53 55 43 43 45 53 53 0D 0A                   $SUCCESS
24 3C 65 6E 64 3E 0D 0A                         $<end>
```

The Length (*marked grey in the readout*) gives information about how many binary bytes will follow – this length should be used to read out the data regardless of its contents. after the number of bytes have been read out, the application can switch back to normal readout mode.

The length is **006B** bytes (hexadecimal notation) – this means **107 Bytes** will follow

3.2.2.2.1 Decoding the first record:

23 00 32 46 81 E1 1E 55 C8 05 E2 CB 25 77
15 16 20 28 75 73 65 72 20 74 78 74 20 20 74 69
6D 65 3D 31 32 3A 32 36 3A 30 39 20 64 61 74 65
3D 20 32 38 2E 30 39 2E 32 30 30 36 00

First **2 bits** of the first byte (**Bits 7 and 6**) are used to determine the used record format:

- the first Byte is **0x23** → ***00***10 0011 *in binary notation,*
- the first **2 bits** are **00** → *documentation of full record format is used to decode this information.*

Full record format::

Next 4 bits are the number of satellites used for navigation when creating this record:

1st Byte : **0x23** → 00**10 00**11 → *8 sats were used*

next bit shows fix information (valid gps fix or not):

1st Byte : **0x23** → 0010 00**1**1 → *valid gps fix*

next bit shows if this record contains an extension:

1st Byte : 0x23 → 0010 001**1** → *extension is contained (this comes into consideration later when this full record format is completely decoded – the extension is directly appended - here the extension exists, so we have to decode then the following bytes as an extension of this record)*

next **7 bits** shows the speed of the device:

2nd Byte : **0x00** → **0000 000**0 → *speed= 0m/s*

next **3 bits** are used for signs - they are reserved for test purposes only and can be ignored:

2nd Byte : **0x00** → 0000 000**0**
3rd Byte : **0x32** → **00**11 0010

next **30 bits** are used for time information

3rd Byte : 0x32 → 0011 0010
4th Byte : 0x46 → 0100 0110
5th Byte : 0x81 → 1000 0001
6th Byte : 0xE1 → 1110 0001
→ *time is 0x324681E1*

next **24 bits** are used for position x: (the first bit is the *sign*)

7th Byte : 0x1E → 0001 1110
8th Byte : 0x55 → 0101 0101
9th Byte : 0xC8 → 1100 1000
→ *x: 0x1E55C8 (sign is 0 → positive value)*

next **24 bits** are used for position y: : (the first bit is the sign)

10th Byte : 0x05 → 0000 0101
11th Byte : 0xE2 → 1110 0010
12th Byte : 0xCB → 1100 1101
→ *y: 5E2CB (sign is 0 → positive value)*

next **24 bits** are used for position z: : (the first bit is the sign)

13th Byte : 0x25 → 0010 0101
14th Byte : 0x77 → 0111 0111
15th Byte : 0x15 → 0001 0101
→ *z: 0x257715 (sign is 0 → positive value)*

(if sign would be '1' then multiply the value with [-1])

Please refer to chapter **Appendix** – converting positions into LAT/LON/ALT coordinates.

The full record format is analyzed - it has an extension (extension bit was 1), so the extension follows now:

1st byte of the extension contains length information of this extension (the number of bytes which were appended to this record). Note that this value has to be multiplied with 2.

1st Byte: 0x16 → 0x16*2 = 0x2C = 44 bytes.

So **44 bytes** were appended to the first record *(the length byte itself is also counted)*

Hint: **If extensions should be ignored by the application no further decoding needs to be done - simply ignore the next X bytes after the record (in this case 44 bytes) . after these X bytes, the next record will follow.**

2nd byte of the extension contains the types of extensions which will follow.

In case several types are appended, the extension having the least index comes first (*i.e. IO comes before User data always*).

2nd Byte: **0x20** → *only user defined data follows (see documentation for a complete list of extensions)*

User data:

3rd Byte *(the first byte of the user data section)* contains the length of user data which follows

3rd Byte: 0x28 → 40 *Bytes of user data are appended*

now 40 bytes user data follows

(as hexadecimal notification):

75 73 65 72 20 74 78 74 20 20
74 69 6D 65 3D 31 32 3A 32 36
3A 30 39 20 64 61 74 65 3D 20
32 38 2E 30 39 2E 32 30 30 36
now 1 byte is left which contains to the extension (specified by the length byte - 1st byte of extension)
00

as no other extensions were set *(type 20 says only user text)*, this byte should be ignored. The reason for this byte to happen is that the complete extension length was multiplied by 2 *(which means only equal numbers can act as length - within this example, the number of bytes within this extension is odd - so 1 fill byte had to be added)*

The first record *(and its extension)* is now completely decoded - after it starts the **2nd** record which is described in the next sub-chapter.

3.2.2.2.2 Decoding the 2nd record:

Decoding this record works basically similar to decoding the first record.

F8 00 10
00 16 20 28 75 73 65 72 20 74 78 74 20 20 74 69 (user txt ti
6D 65 3D 31 32 3A 32 36 3A 31 30 20 64 61 74 65 me=12:26:10 date
3D 20 32 38 2E 30 39 2E 32 30 30 36 00 0D 0A

first 2 bits of the first byte (Bits 7 and 6) are used to determine the used record format

The first Byte is:

0xF8 → 1111 1000 *in binary notation*

the first **2 bits** are **11** → documentation of standing record format is used to decode this information.

Standing record format:

Next **2 bits** are the number of satellites used for navigation when creating this record

1st Byte : 0xF8 → 1111 1000 → *3 → minimal 7 sats are used for navigation*

next **bit** shows if this record contains an extension

1st Byte : 0xF8 → 1111 1000 → *extension is contained (this comes into consideration later when this record format is completely decoded – the extension is directly appended - here the extension exists, so we have to decode then the following bytes as an extension of this record)*

next **3 bits** shows the speed of the device

1st Byte : 0xF8 → 1111 1000 → *speed= 0m/s*

next **12 bits** are used for time information

2nd Byte : 0x00 → 0000 0000
3rd Byte : 0x10 → 0001 0000 → *time is 0x01 (its a delta time - so 1 has to be added to the previous timestamp)*

next **4 bits** are used for position x: (the first bit is the sign)

3rd Byte : **0x10** → 0000 0000 → *dx: 0x0 (delta position – has to be added to previous x value)*

next **4 bits** are used for position y: (the first bit is the sign)

4th Byte : 0x10 → **0000** 0000 → *dy: 0x0 (delta position – has to be added to previous y value)*

next **4 bits** are used for position z: (the first bit is the sign)

4th Byte : 0x10 → 0000 **0000** → *dz: 0x0 (delta position – has to be added to previous z value)*

(if sign would be '1' then multiply the value with (-1))

The standing record format is analysed - it has an extension (extension bit was 1), so the extension follows now:

1st byte of the extension contains length information of this extension *(the number of bytes which were appended to this record)*. **Note that,** this value has to be multiplied with **2**.

1st Byte: 0x16 → 0x16*2 = 0x2C = 44 bytes.

So 44 bytes were appended to the first record (the length byte itself is also counted)

Hint: **If extensions should be ignored by the application no further decoding needs to be done - simply ignore the next X bytes after the record (in this case 44 bytes) . after these X bytes, the next record will follow.**

2nd byte of the extension contains the types of extensions which will follow.

In case several types are appended, the extension having the least index comes first (i.e. IO comes before User data always).

2nd Byte: 0x20 → *only user defined data follows (see documentation for a complete list of extensions)*

User data:

3rd Byte *(the first byte of the user data section)* contains the length of user data which follows>

3rd Byte: 0x28 → 40 *Bytes of user data are appended*

now **40 bytes** user data follows

(as hexadecimal notification):

```
75 73 65 72 20 74 78 74 20 20
74 69 6D 65 3D 31 32 3A 32 36
3A 31 30 20 64 61 74 65 3D 20
32 38 2E 30 39 2E 32 30 30 36
```

now 1 byte is left which contains to the extension (specified by the length byte - 1st byte of extension)
00

as no other extensions were set *(type 20 says only user text*), this byte should be ignored. The reason for this byte to happen is that the complete extension length was multiplied by 2 *(which means only equal numbers can act as length - within this example, the number of bytes within this extension is odd - so 1 fill byte had to be added)*

The second record *(and its extension)* is now completely decoded - after it a third record could start.

4 APPENDIX

4.1 Convert Time

4.1.1 Seconds into Date & Time

The number of seconds that have passed since Saturday/Sunday Midnight UTC (01/06/1980*), with time zero begging this midnight. Used with GPS Week Number to determine a specific point in GPS Time. The value in seconds must be converted into the DateTime format.

PLEASE NOTE: the DATE/TIME variable is a 30-bit integer value which runs over every 34 years. The current time window is 14.01.2014 13:37:04 to 2044 ! For Date values smaller than 946339200 add 1073741824 to get the correct Date/Time.

For example:

- First, convert the binary data to hexadecimal value, which is **30 A6 1D F9** (see Table1.3). The converted hexadecimal value must be converted to the decimal value, which results the number of seconds that have been passed since 01/06/1980: **816193017**.
  1.1. Obtain the number of seconds required for Time format: **816193017 mod**** 60 = **57** ss.
- Obtain the number of minutes: (**816193017 – 57**) **/** 60 = **13603216** minutes.
  2.1. Obtain the number of minutes required for Time format: **13603216 mod 60** = **16** mm.
- Obtain the number of hours: (**13603216 – 16**) **/**60 = **226720** hours.
  3.1.Obtain the number of hours required for Time format: **226720 mod 60** = **10** hh.
- Obtain the number of days left from 06/01/1980 : (**226720 – 40**) **/**24 = **9446** days.
- Add the number of days to 06.01.1980: **06/01/1980 + 9445** = **13/11/2005**.
- Finally, obtain the DateTime value: (**13/11/2005 + 40/**24 + **16**/24/60 + **57**/24/60/60) =**16/11/2005 16:16:57.**

* In your application this value must be defines as a constant expression, because it may change in the feature.

** The **mod** operator returns the remainder obtained by dividing its operands.

4.2 Convert X,Y,Z positions

4.2.1 Defines

```
#define NLMAJA  ( 6378137.0 )
#define NLASQR  ( NLMAJA * NLMAJA )
#define NLFLAT  ( 1.0 / 298.2572235630 )
#define NLESQR ( NLFLAT * (2.0 - NLFLAT) )
#define NLOMES ( 1.0 - NLESQR )
#define NLEFOR ( NLESQR * NLESQR )
#define NLMINB  (NLMAJA * (1.0-NLFLAT)) /* len of semi minor axis of ref  ellips*/
#define PI    ( 3.14159265358979323384 )
#define RADIANS_PER_DEGREE    ( PI / 180.0 )
#define DEGREES_PER_RADIAN    ( 180.0 / PI )
#define METERS_PER_NAUTICAL_MILE ( 1853.32055 )
#define LAT_METERS_PER_DEGREE   ( METERS_PER_NAUTICAL_MILE * 60.0 )
```

4.2.2 Types

```
typedef struct
        {
        double lat;
        double lon;
        double alt;
        } t_LTP;
typedef struct
        {
        double x;
        double y;
        double z;
        } t_ECEF;
```

4.2.3 Function ECEF → LTP (latitude, longitude, and altitude)

Coordinates of STEPPII's position in ECEF (meters). The Earth-centered Earth-Fixed (ECEF) is a Cartesian coordinate system with its origin located at the center of the Earth. The coordinate system used by GPS to describe 3-D location. For WGS-84 (World Geodetic System 1984) reference ellipsoid. ECEF coordinated have the Z-axis aligned with the Earth's spin axis, The X-axis through the insertion of the Prime meridian and the Equator and the Y-axis is rotated 90 degrees East of the X-axis about the Z-axis. In order to convert the ECEF coordinates obtained from the history binary data into latitude, longitude, and altitude, use this function.

```
void CGPSTools::ConvertECEFToLTP(t_ECEF *pecef, t_LTP *plla)
  {
        double majA = NLMAJA;
        double minB = NLMINB;
        double majAA = NLMAJA * NLMAJA;
        double minBB = NLMINB * NLMINB;
        double cosLat, sinLat, p=0.0, esq, epsq, theta, lat, lon;
        if ( pecef->x == 0.0 && pecef->y == 0.0 )   // on the axis
        {
        //umDebugPrintf("NavConvertECEFToLTP: Failure exactly on the AXIS!");
        // We are sitting EXACTLY on the earth's axis.
        // Probably at the center or on one of the poles.
        if (pecef->y < 0.0)
        lat = -PI / 2;                          // alt above north pole
        else
        lat = PI / 2;                           // alt above south pole
        lon = 0.0;                              // as good as any other value
        }
        else
        {
        p = sqrt (pecef->x*pecef->x + pecef->y*pecef->y);
        theta = atan (pecef->z * majA/(p * minB));
        esq = 1.0 - minBB / majAA;
        epsq = majAA / minBB - 1.0;
        lat = atan ((pecef->z + epsq * minB * pow (sin(theta),3)) / (p - esq        * majA * pow (cos(theta),3)));
        lon = atan2 (pecef->y, pecef->x);
        }
        cosLat = cos(lat);
        sinLat = sin(lat);
        plla->lat = lat;
        plla->lon = lon;
        plla->alt = p / cosLat - (majAA / sqrt(majAA * cosLat*cosLat + minBB    * sinLat*sinLat));
```

```
        plla->lat *= DEGREES_PER_RADIAN;
        plla->lon *= DEGREES_PER_RADIAN;
    }
```

4.2.4 Function LTP (latitude, longitude, and altitude) → ECEF

This transformation can be used to test or calculate back the results

```
void ConvertLTPToECEF(t_LTP *plla, t_ECEF *pecef)
{
  // convert lat,lon from decimal degree into radiant
plla->lat= plla->lat*PI /180;
plla->lon= plla->lon*PI /180;
slat = sin(plla->lat);
clat = cos(plla->lat);
r = NLMAJA / sqrt(1.0 - NLESQR * slat * slat);
pecef->x = (r + plla->alt) * clat * cos(plla->lon);
pecef->y = (r + plla->alt) * clat * sin(plla->lon);
pecef->z = (r * NLOMES + plla->alt) * slat;
          }
```

Example:
lat=50.673333,
lon=10.980722,
alt=490.3 m
Result: X: 0x3CACA6 (= 3976358) Y: 0xBC5D1 (= 771537)
(divided by 2, it should be equal to e.g.history positions read out from history)

4.3 Java source code

The following java source code snippet is provided to help you easy develop your application for decoding history data. An internal documentation is comprised of comments written within the source code.

Java source code snippet **"Java source code snippet for decoding history"** is also available for download at the following internet address: http://www.falcom.de/uploads/media/