ARCHIVE.FSH

From OpenStreetMap Wiki

ARCHIVE.FSH is a binary format used by Raymarine's (http://www.raymarine.com/) chart plotters to save tracks and routes. This page describes the file structures as far as known. Of course, the format is implemented in Raymarine's tools which can be purchased on their web site. The format is further implemented in the GPS Utility (GPSU) (http://www.gpsu.co.uk/) which is closed source as well but a Freeware version is available for download. Unfortunately there is only a Windows version.

The format is partially implemented in the GPLv3 open source Python script fsh2gpx.py (https://github.com/scotte/fsh2gpx) and in the more complete C program parsefsh (http://www.abenteuerland.at/download/parsefsh/) ... wiki page: Parsefsh

The following description uses C data structures defined in **fshfunc.h** parsefsh (http://www.abenteuerland.at/download/parsefsh/) for a more clear explaination.

Please watch the page Talk:ARCHIVE.FSH. I will document important changes and news there.

There are also some notes added at the end of this document which describe the ARCHIVE.FSH format observed from a C90W MFD.

Contents

- 1 Basic Structure
- 2 FSH Blocks
  - 2.1 Track Data
  - 2.2 Track Meta Data
    - 2.2.1 Coordinates of Track Points
  - 2.3 Waypoints
  - 2.4 Routes
    - 2.4.1 Route Part 1
    - 2.4.2 Route Part 2
    - 2.4.3 Route Part 3
  - 2.5 Groups
    - 2.5.1 Group Header
    - 2.5.2 Group GUID and Waypoint List
- 3 Appendix
  - 3.1 C90W MFD notes

Basic Structure

The file is a binary file which is made of a set of fixed length FLOBs (flash objects?), each containing a number of variable length blocks. Each of those blocks has an internal structure dependent on the block type. Typically, there are again some fixed and/or variable length structures within each block. All data types (such as integers) are formatted in **little endian** byte order. The members of the structures are documented as much as known. If it reads "*always*" it is meant "*as observed in the sample files*" which does not really mean "*always*".

Every FSH file is prepended by fixed 28 bytes long header. The member variable *flob* contains the number of FLOBs within the file multiplied by 16. If flob = 0x10 the file contains 1 FLOB, if flob = 0x80 it contains 8 FLOBs.

```
// total length 28 bytes
typedef struct fsh_file_header
{
  char rl90[16];     //!< constant terminated string "RL90
FLASH FILE"
  int16_t flobs;     //!< # of FLOBs, 0x10 (16) or 0x80 (12
8)
  int16_t a;         //!< always 0
  int16_t b;         //!< always 0
  int16_t c;         //!< always 1
  int16_t d;         //!< always 1
  int16_t e;         //!< always 1
} __attribute__ ((packed)) fsh_file_header_t;
```

The file header is followed by a number (see above) of fixed size FLOBs. The total size of one FLOB is exactly 64 kByte (= 0x10000 bytes). Each FLOB starts with a 14 bytes long FLOB header as defined below.

```
// total length 14 bytes
typedef struct fsh_flob_header
{
  char rflob[8];     //!< constant unterminated string "RAY
FLOB1"
  int16_t f;         //!< always 1
  int16_t g;         //!< always 1
  int16_t h;         //!< 0xfffe, 0xfffc, or 0xfff0 (See C9
0W note 1)
} __attribute__ ((packed)) fsh_flob_header_t;
```

FSH Blocks

After the FLOB header a number of variable length blocks follow. Every block is prepended by a fixed length header of type *fsh_block_header_t*. This header in turn contains the length (*len*) of the data following the header. This length field does not include the length of the header. If the length is an odd number, the block is padded at the end by one byte such that the next block header again starts at an even offset. After the last valid block all octets are set to 0xff.

Please note that all FLOBs in a file may contain block objects even if they are just partially filled. Thus, a parser has always to read all FLOBs.

```
// total length 14 bytes
typedef struct fsh_block_header
{
  uint16_t len;      //!< length of block in bytes excludin
g this header
  uint64_t guid;     //!< unique object identifier
  uint16_t type;     //!< type of block
  uint16_t unknown; //!< (see C90W note 2)
} __attribute__ ((packed)) fsh_block_header_t;
```

Every such block object (i.e. a block, a track, a waypoint,...) within a FLOB has a unique object identifier (*guid*) which is a 64 bit integer number. In several places it is used as a cross reference. As already mentioned, the structure of the block depends on the block *type*. Currently the following block types are known:

- 0x0001 waypoints
- 0x000d track
- 0x000e track meta data
- 0x0021 route (with waypoints)
- 0x0022 group (similar to 0x21, but not a route, instead used to group waypoints)
- 0xffff previous block was the last one

In the following those blocks are described in detail.

Track Data

An FSH block of type 0x0d contains the data of a track segment. This is a list of track points prepended by a fixed header. The fixed header is of type *fsh_track_header_t*. The track header has a total length of 8 bytes and contains the number track points which follow this header. Technically, a track segment cannot contain more than 4680 (= (65536 - 8) / 14) track points.

```
// type 0x0d blocks (list of track points) are prepended b
y this.
typedef struct fsh_track_header
{
  int32_t a;         //!< unknown, always 0
  int16_t cnt;       //!< number of track points
  int16_t b;         //!< unknown, always 0
} __attribute__ ((packed)) fsh_track_header_t;
```

Each track point is a structure of 14 bytes length. It contains the geographic coordinates in ellipsoid Mercator projection (see later), the depth in centimeters and several other fields.

```
// total length 14 bytes
typedef struct fsh_track_point
{
  int32_t north, east; //!< prescaled (FSH_LAT_SCALE) nort
hing and easting (ellipsoid Mercator)
  uint16_t tempr;      //!< temperature in Kelvin * 100
  int16_t depth;       //!< depth in cm
  int16_t c;           //!< unknown, always 0
} __attribute__ ((packed)) fsh_track_point_t;
```

Track Meta Data

The FSH block of type 0x0d has a length of 58 bytes followed by a number of GUIDs. The following structure shows the track meta structure.

```
// total length at least 58 bytes
typedef struct fsh_track_meta
{
  char a;            //!< always 0x01
  int16_t cnt;       //!< number of track points
  int16_t _cnt;      //!< same as cnt
  int16_t b;         //!< unknown, always 0
  int16_t c;         //!< unknown (see C90W note 3)
  int16_t d;         //!< unknown, always 0 (1 in 2nd block
)
  int32_t north_start; //!< Northing of first track point
  int32_t east_start;  //!< Easting of first track point
  uint16_t tempr_start;//!< temperature of first track poi
nt
  int32_t depth_start; //!< depth of first track point
  int32_t north_end;   //!< Northing of last track point
  int32_t east_end;    //!< Easting of last track point
  uint16_t tempr_end;  //!< temperature last track point
  int32_t depth_end;   //!< depth of last track point
  char i;            //!< unknown, 0, 1, or 5;  (see C90W n
ote 4)
  char name[16];     //!< name of track, string not termina
ted
  char j;            //!< unknown, always 0
  uint8_t guid_cnt; //!< nr of guids following this header
  uint64_t guid[];  //!< list of guids to this header belo
ngs to
} __attribute__ ((packed)) fsh_track_meta_t;
```

The *name* of the track may have a maximum length of 16 bytes in which case it might not be \0-terminated although the next value *j* seems to always contain a \0-byte probably to safely terminate the string. A track meta data block is associated with a number (*fsh_track_meta_t.guid_cnt*) of track segments. It is referred to those blocks with the GUIDs of their block headers. The GUIDs are stored in a list after the fixed part of the track meta data header (*fsh_track_meta_t.guid[]*). This allows to have tracks of more than 4680 track points (see *Track Data* above).

Coordinates of Track Points

Track points have coordinates in ellipsoid Mercator coordinates multiplied by a static factor. Please note that this is a different method as used for the coordinates used in waypoints and routes (see there).

The Easting in the track point (*fsh_track_point_t.lon*) is a linearly related to the geographic longitude in the way that the integer 0x7fffffff conforms to 180° E.

```
longitude = (double) fsh_track_point_t.lon / 0x7fffffff *
180.0;
```

Calculating the geographic latitude from the Northing found in the track points is an iterative process as usual for reversing ellipsoid Mercator Northing. The value is scaled by the static factor 107.1709342 (defined as the preprocessor macro *FSH_LAT_SCALE*). Please note that this factor was derived empirically. Actually, this is an average value of the minimum 107.1705528 and 107.1720867. Unfortunately, it is not simply a linear function. It seems to be very close to a hyperbolic sine. Thus, *FSH_LAT_SCALE* may slightly change in future.

The reverse Mercator algorithm is well known and documented. The following functions show how it works without explanation, the source code of parsefsh contains some comments. (please look at the according literature [1] (http://mercator.myzen.co.uk/mercator.pdf) [2] (http://www.iogp.org/pubs/373-07-2.pdf)). The function depends on ellipsoid parameters. The Northing within the FSH files seems to be based on the WGS84 reference ellipsoid, hence, ellipsoid_t.a = 6378137 (semi-major axis) and ellipsoid.e = 0.08181919 (eccentricity).[3] (http://www.arsitech.com/mapping/geodetic_datum/)

```
#define MAX_IT 32
#define IT_ACCURACY 1.5E-8
#define FSH_LAT_SCALE 107.1709342

static double phi_rev_merc(const ellipsoid_t *el, double N
, double phi0)
{
  double esin = el->e * sin(phi0);
  return M_PI_2 - 2.0 * atan(exp(-N / el->a) * pow((1 - es
in) / (1 + esin), el->e / 2));
}

static double phi_iterate_merc(const ellipsoid_t *el, doub
le N)
{
  double phi, phi0;
  int i;

  for (phi = 0, phi0 = M_PI, i = 0; fabs(phi - phi0) > IT_
ACCURACY && i < MAX_IT; i++)
  {
     phi0 = phi;
     phi = phi_rev_merc(el, N, phi0);
  }
  return phi;
}
```

```
// ....
double lat = phi_iterate_merc(el, fsh_track_point_t.lat /
FSH_LAT_SCALE) * 180 / M_PI;
```

Waypoints

Different to trackpoints, waypoints are a more detailed description of a position. FSH files may contain three different types of waypoint structures which share a common set of fields. These common fields are found in the structure *fsh_wpt_data_t*. Those three types of waypoints may be found

- within groups of block type 0x22 typed with *fsh_wpt_t*,
- as stand-alone waypoints of block type 0x01 typed with *fsh_wpt01_t*, and
- within routes of block type 0x21 typed with *fsh_route_wpt_t*,

As already mentioned, all three are a superset of *fsh_wpt_data_t*, the route waypoint *fsh_route_wpt_t* is again a superset of type 0x22 *fsh_wpt_t*.

The common waypoint data contains a sub-structure which contains timestamp information. First, a 32 bit long field *timeofday* which represents the time of the day in seconds, and second, a 16 bit long field which is the number of days since the epoch of 1.1.1970 (thus, it will expire at least in June of the year 2149 without using the unused bits of the seconds-field.). The following code line shows how to convert it to a Unix timestamp.

```
time_t t = (time_t) ts->date * 3600 * 24 + ts->timeofday;
```

In the following the exact definition of these four structures is depicted.

```
// timestamp used in FSH data, length 6 bytes
typedef struct fsh_timestamp
{
  uint32_t timeofday;  //!< time of day in seconds
  uint16_t date;       //!< days since 1.1.1970
} __attribute__ ((packed)) fsh_timestamp_t;

// common waypoint data, length 40 bytes + name_len + cmt_
len
typedef struct fsh_wpt_data
{
  int32_t north, east; //!< prescaled ellipsoid Mercator n
orthing and easting
  char d[12];         //!< 12x \0
  char sym;           //!< probably symbol

  uint16_t tempr;     //!< temperature in Kelvin * 100
  int32_t depth;      //!< depth in cm
  fsh_timestamp_t ts; //!< timestamp

  char i;             //!< unknown, always 0

  char name_len;      //!< length of name array  (See C90W
 note 5)
  char cmt_len;       //!< length of comment

  int32_t j;          //!< unknown, always 0
  char txt_data[];    /*!< this is a combined string field
. First the name of
                        the waypoint of 'name_len' number
of characters
                        directly followed by a comment of
'cmt_len'
                        characters. Thus, the comment star
ts at 'txt_data' +
                        'name_len'. */
#define NAME(x) ((x).txt_data)
#define COMMENT(x) ((x).txt_data + (x).name_len)
} __attribute__ ((packed)) fsh_wpt_data_t;

// waypoint as used int block type 0x22, 8 bytes + sizeof
fsh_wpt_data_t
typedef struct fsh_wpt
{
  int32_t lat, lon;   //!< latitude/longitude * 1E7
  fsh_wpt_data_t wpd;
} __attribute__ ((packed)) fsh_wpt_t;

// route (0x21) waypoint is simply a GUID followed by the
waypoint data, length
// is 8 bytes + sizeof fsh_wpt = 56 bytes + name_len + cmt
_len
typedef struct fsh_route_wpt
```

```
{
 int64_t guid;    //!< the guid of the waypoint
 fsh_wpt_t wpt;   //!< rest of the waypoint data from abov
e
} __attribute__ ((packed)) fsh_route_wpt_t;

// waypoint 0x01, length 8 bytes  + sizeof wpt_data_t (See
 C90W note 6)
typedef struct fsh_wpt01
{
  int64_t guid;
  fsh_wpt_data_t wpd;
} __attribute__ ((packed)) fsh_wpt01_t;
```

Interestingly, the latitude and longitude is found in two different formats. In the first case (in *fsh_wpt_t*) both are simply scaled by the factor 1E7. Thus, the geographic coordinates in degrees are found by the following formula:

```
double latitude = (double) fsh_wpt_t.lat / 1E7;
```

The second pair of coordinates (*north* and *east*, found in the common data structure *fsh_wpt_data_t*) are in the same format as in the track points (see #Coordinates of Track Points).

Routes

Routes are basically lists of waypoints. Waypoints are similar to track points but contain more information than track points. There are two types of waypoint list blocks: 0x21 and 0x22. At first, the block format of type 0x21 (a route) is described.

The internal block structure is basically split into three parts:

1. A route header and a list of GUIDs (of the waypoints),
2. a 2nd header and a list of additional but currently unknown data,
3. a 3rd header and a list of way points.

Each of those sub-blocks is prepended by a separate header.

Route Part 1

The route header has a variable length since it contains the name of the route. The length of the string is found in the header itself. Please note, that the string is not \0-terminated. The total length of the header is 8 + *name_len* + *cmt_len*. This header is followed by *guid_cnt* GUIDs (each GUID 64 bits, i.e. 8 bytes). These GUIDs are those from the waypoints which are found later in such a block.

```
// route type 0x21
typedef struct fsh_route21_header
{
  int16_t a;         //!< unknown, 0, also 3 observed on C9
0W
  char name_len;     //!< length of name of route
  char cmt_len;      //!< length of comment
  int16_t guid_cnt; //!< number of guids following this he
ader
  uint16_t b;        //!< unknown
  char txt_data[];  /*!< this is a combined string field o
f 'name' and
                      'comment' exactly like it is in the
fsh_wpt_data_t (see
                      there). Use the macros NAME() and CO
MMENT() to retrieve
                      pointers to the strings. */
} __attribute__ ((packed)) fsh_route21_header_t;
```

Route Part 2

Directly after the GUID list a second header with a length of 46 bytes follows:

```
struct fsh_hdr2
{
  int32_t lat0, lon0;  //!< lat/lon of first waypoint
  int32_t lat1, lon1;  //!< lat/lon of last waypoint
  int32_t a;
  //int16_t b;          //!< 0 or 1
  int16_t c;
  char d[24];
} __attribute__ ((packed));
```

This 2nd header contains mostly unknown data except the first four integers which represent the latitude and longitude of the first and the last waypoint. According to the [fsh2gpx.py https://github.com/scotte/fsh2gpx] there should be a timestamp within the header but this could not be reproduced yet. After this 2nd header a list of probably some additional waypoint information follows. I call them *"points"*. Each point has a length of 10 bytes. The number of points is equal to the number of GUIDs which is found in the first route header *fsh_route21_header_t*.

```
struct fsh_pt
{
  int16_t a;
  int16_t b;          //!< depth?
  int16_t c;          //!< always 0
  int16_t d;          //!< in the first element same value l
ike b
  int16_t sym;        //!< seems to be the symbol
} __attribute__ ((packed));
```

Route Part 3

Behind the list of points a simple 4 byte header follows which contains the number of waypoints which follow this header. As far as observed yet the number of waypoints within the route21 header is equal to the number waypoint within this header.

```
struct fsh_hdr3
{
  int16_t wpt_cnt;  //!< number of waypoints
  int16_t a;        //!< always 0
} __attribute__ ((packed));
```

Then a list of waypoints follow which are of type *fsh_route_wpt_t* as defined above in Section #Waypoints.

Groups

Groups are another form of waypoint list, in which the user can group related waypoints together.

A group is simply:

1. A group header, followed by
2. a list of GUIDs (of the waypoints), followed by
3. a list of waypoints

Group Header

The group header has a variable length since it contains the name of the group. The length of the string is found in the header itself. Please note, that the string is not \0-terminated. The total length of the header is 8 + *name_len*. This header is followed by *guid_cnt* GUIDs (each GUID 64 bits, i.e. 8 bytes). These GUIDs are those from the waypoints which are found immediately after the GUID list.

```
// group type 0x22
typedef struct fsh_group22_header
{
  int16_t name_len; //!< length of name of route
  int16_t guid_cnt; //!< number of GUIDs in the list follo
wing this header
  char name[];      //!< unterminated name string of lengt
h name_len
} __attribute__ ((packed)) fsh_group22_header_t;
```

Group GUID and Waypoint List

Directly after the group header is a list of GUIDs, of length guid_cnt.

Immediately following the GUID list is a list of waypoints of type *fsh_wpt_t* (see #Waypoints, including the excellent description of the lat/lng and northing/easting formats.).

Appendix

C90W MFD notes

These observations were made using the ARCHIVE.FSH file created by a C90W MFD running v1.13 firmware, and modified files that were read back into the C90W.

1. The C90W requires all but the last flob with data to have fsh_flob_header.h set to 0xfff0, the last with data as 0xfffc, and the empty ones completing the file set to 0xfffe. Presumably this is to make it easy to extend the file; the ARCHIVE.FSH file format is cumulative and new saved blocks are appended to the existing ones in the last flob with data in the file.
2. fsh_flob_header.unknown appears to be a status flag for the block. The value is observed to be 0x4000; if the block is deleted using the MFD, the block is not physically removed, rather this field is set to 0x0000.
3. fsh_track_meta.c,d actually appear to be a single int32_t that represents the length of the track in meters. d*2^16+c comes close to my calculated lengths for the tracks.
4. fsh_track_meta.i is interpreted as a colour code. The C90W displays the tracks in colour as follows: 0 - red, 1 - yellow, 2 - green, 3 - blue, 4 - magenta, 5 - black
5. The C90W won't load a waypoint group if any waypoint name is longer than 16 characters.
6. The C90W interpretation of the northing and easting in a simple waypoint (block type 0x0001) is somehow tied the guid associated with it. You can't just create an arbitrary guid; it has to be the one generated by the C90W in the first place for that waypoint if it is to display correctly. This is not true for the waypoint group, block type 0x0022, possibly because it uses the lat/lon directly.

Categories: File format | Openseamap
