Garmin IMG File Format

John Mechalas

29 October 2005

Table of Contents

Changes

A large number of changes and corrections have been made in this version, too many to effectively highlight as that would make the document unreadable. It will be necessary to read the entire document from beginning to end.

Motivation

I'm a tinkerer, which means I like to look under the hood for its own sake, and it was only natural that I'd be intrigued by the file format used by my favorite GPS manufacturer. I also like to share what I know with other, motivated people.

What I quickly learned in the area of IMG parsing was that there are quite a few folks "in the know", many of whom share applications with the Garmin community. There are fewer, however, who share their code, and fewer still who share their knowledge. My goal is to address the latter, in the hopes that more people will be motivated to do as I have…and possibly contribute to this effort in the process. You know who you are. Don't hesitate to write.

Disclaimer

The information contained in this document was gathered from a combination of the sources listed in the Acknowledgements section, from personally created IMG files and IMG files that were freely available without usage restrictions. Note that analysis of licensed IMG files from commercial sources may, depending on your country of origin, constitute a violation of that vendor's license agreement or local laws.

Acknowledgements

Special thanks to:

- Stanislaw Kozicki, creater of cGPSmapper at http://cgpsmapper.com/, who has made it possible for the average user to create IMG files. cGPSmapper-created maps served as the source for this software effort.
- The Karmin project at http://karmin.sourceforge.net/. This effort appears to be stalled or dead, but their source code and documentation (while extremely difficult to parse) was helpful.
- GPSMapEdit at http://geopainting.com/, which is capable of importing IMG files and was helpful in validating the decoding work.
- Mapdekode at http://groups.yahoo.com/group/mapdekode/, which can parse IMG files and has limited IMG file creation capability. It can also parse and build TDB files, allowing the map author to create maps that are viewable in, and downloadable from, MapSource.
- Peter Dekode's IMG specification document for filling in gaps and providing much of the NET structure.

IMG File Format

An individual IMG file is a binary representation of raster map data used by Garmin GPS receivers and the MapSource map utility. It is a closed/proprietary format developed by Garmin.

The IMG format appears to be structured to optimize rendering speed of the GPS device first, and minimize file size second. This is consistent with the fact that GPS receivers generally have low-speed, low-power processors (compared to a personal computer) and therefore benefit from algorithms and data organizations that minimize the work needed to draw the display.

The secondary concern, file size, stems from the limited memory present in most Garmin receivers compared to the sizes of the map files, themselves. The IMG file makes every effort possible to conserve bytes, something that is particularly evident in the section of the file that defines the "text labels" for map elements (streets, points of interest, etc.).

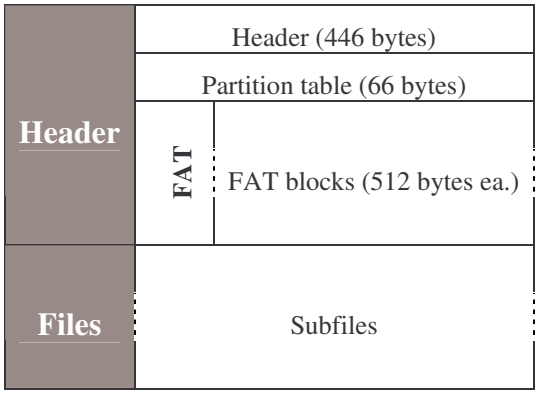
IMG files store words and dwords in a little endian byte order.

Before you can look at an IMG file, you must XOR it with the XOR byte (the first byte in the file). Some maps are not XOR'd (the first byte is 0x00).

File Overview

An IMG file can be thought of as having two parts: a header and a list of files. In this respect, it somewhat resembles a memory-mapped filesystem. In fact, some of the concepts of filesystems can be applied directly to the layout of the file, itself, as Garmin has adopted a few conventions from that arena.

Table 1: IMG File Overview



Of particular interest is what appears to be an actual partition table, recognizable as such by the inclusion of the bytes 0x55 0xAA at file offset 0x0F1E. This placement is obviously deliberate and corresponds to a DOS-style primary disk partition which happens to consists of 512 bytes total, ending in 0x55 0xAA. In a true partition table, the first 446 bytes would be used to hold a boot loader program, but since the IMG file is data and not a program, those 446 bytes are used to store generic information, such as the map name and the date that it was created, as well as some specifics about the physical layout of the file itself.

The *checksum* at 0xF is calculated by summing all bytes, save for the checksum byte itself, in the map file and then multiplying by -1. The lowest byte in this product becomes the checksum byte

at 0xF. Note that this checksum is apparently not validated by MapSource, since you can modify the contents of IMG files directly with a hex editor and they will still work even if the checksum is not updated.

Most notable are the two byte values (at 0x61 and 0x62, here named as E_1 and E_2) which are used to set the block size for the file. They represent powers of two, and the block size is set via the formula $2^{E_1+E_2}$. E_1 appears to always be 0x09, setting the minimum block size to 512 bytes.

Following the partition table is a pointer to the first sub-file in the IMG file, which represents the start of the data section. This offset is used to calculate the length of the FAT (described below).

Another interesting section is the block sequence counter that begins at offset 0x420. Each dword represents a block number in the file, starting at 0 (the 1st byte in the IMG file). This particular block sequence indicates which file blocks are taken up by the IMG file header, and the sequence is padded with 0xFFFF. So, if the sequence count looks like this:

0x0000 0x0001 0x0002 0x0003 0x0004 0x0005 0x0006 0xFFFF …

Then we can conclude that the IMG file header consumes blocks zero through six. Combined with the block size as calculated from E_1 and E_2 we can determine the total number of bytes that the header consumes on disk (the actual header size will likely be less, and is padded to the block-boundary).

IMG Header

Table 2: IMG Header

Abs Offset	Contents	Length (bytes)
0x0	XOR byte (image file is XOR'd with this byte if non-zero)	1
0x1	0x00 …	9
0xA	Update month (0-11)	1
0xB	Update year (+1900 for val >= 0x63, +2000 for val <= 0x62)	1
0xC	0x00 …	3
0xF	Checksum	1
0x10	Signature "DSKIMG\0" 0x44 0x53 0x4b 0x49 0x4d 0x47 0x00	7
0x17	??? 0x02	1
0x18	??? (sectors?) == 0x5F == 0x1C4 0x0004	2
0x1A	??? (heads?) == 0x5D 0x0010	2
0x1C	??? (cylinders?)	2

		0x0020	
0x1E		??? 0x0000	2
0x20		0x00 …	25
0x39		Creation year	2
0x3B		Creation month (0-11)	1
0x3C		Creation day (1-31)	1
0x3D		Creation hour (0-23)	1
0x3E		Creation minute (0-59)	1
0x3F		Creation second (0-59)	1
0x40		??? 0x02	1
0x41		Map file identifier “GARMIN\0”	7
0x48		0x00	1
0x49		Map description, 0x20 padded “…”	20
0x5D		??? (heads?) == 0x1A	2
0x5F		??? (sectors?) == 0x18 == 0x1C4	2
0x61		Block size exponent, E_1	1
0x62		Block size exponent, E_2	1
0x63		??? == (0x1A)*(0x18)*(0x1C)/2^(0x62)	2
0x65		Map name, cont’d (0x20 padded, \0 terminated) “…\0”	31
0x84		0x00 …	314
0x1BE	Partition table	0x00 (boot?)	1
0x1BF		0x00 (start-head?)	1
0x1C0		0x01 (start-sector?)	1
0x1C1		0x00 (start-cylinder?)	1

0x1C2		0x00 (system-type?)	1
0x1C3		0x0F == (0x1a) - 1 (end-head?)	1
0x1C4		0x04 == 0x18 == 0x5F (end-sector?)	1
0x1C5		0x1F == (0x1C) - 1 (end-cylinder?)	1
0x1C6		0x00000000 (rel-sectors?)	4
0x1CA		0x00080000 == (0x63)*2^(0x62)+(0x18)*(0x1C) (number of sectors?)	4
0x1CE		0x00	48
0x1FE		0x55 0xAA	2
0x200		0x00	512
0x400		??? 0x01	1
0x401		??? 0x20 …	11
0x40C		First sub-file offset (absolute)	4
0x410		??? 0x03	1
0x411		0x00 …	15
0x420		Block sequence numbers 0x0000 - ????, 0xFFFF padded (0x420)*(block_size) == (0x40C)	480
0x600	FAT	FAT Length *L* == (0x40c)-0x600	*L* bytes in 512 byte blocks

FAT Block

After the block sequence comes what can be labeled a File Allocation Table, or FAT. The data section of the IMG file contains a number of “sub-files”, and the location and type for each is defined in a series of FAT blocks. Collectively, these blocks make up the FAT itself. Each FAT block is 512 bytes in length and corresponds to a single sub-file.

As with the header, a block sequence counter takes up the final 480 bytes of each FAT block, mapping the sub-file to particular file blocks inside the IMG file, itself. The numbering of these

sequence numbers continues from the header sequence, and from the previous FAT block sequences. This definition implies that an IMG file can never have more than 65,535 blocks, but the use of large block sizes (2048, 4096, etc…) allow the IMG file to hold several gigabytes of data. The only practical limit on IMG file size seems to be the use of 32-bit integers for file offsets, which imposes a hard limit of 4 GB.

Large sub-files, those with more than 240 blocks of data, are spanned across multiple FAT blocks with each sub-file part numbered in a 32-bit integer at offset 0x10. File size is only present in the first of these (part 0x0000).

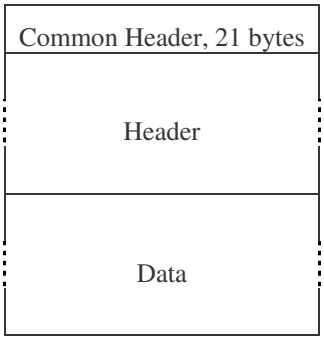
Table 3: FAT Block

Offset (rel)	Contents	Length (bytes)
0x0	Flag 0x01 = true sub-file 0x00 = dummy block (last FAT block in sequence, *IF PRESENT*).	1
0x1	IMG sub-file name (I0012345, etc.)	8
0x9	Sub-file type (RGN, TRE, etc.)	3
0xC	Size of sub-file (bytes), only present if (0x10) == 0x0000	4
0x10	Sub-file part, starting with 0x0000	2
0x12	0x00 …	14
0x20	Block sequence numbers, incrementing from header 0x420 and from previous sub-file sequences. 0xFFFF designates no block.	480

Sub-File Format

Each sub-file consists of three parts: the common header, the personal header and the file data. The common header has the same format for all sub-files while the personal header is dependant upon the file type (RGN, TRE, etc.).

Table 4: Sub-File Format



Common Header

The common file header is defined below.

Table 5: Common Sub-File Header

Offset (rel)	Contents	Length (bytes)
0x0	Header length, HL	2
0x2	Type “GARMIN RGN”, “GARMIN TRE”, etc.	10
0xC	??? 0x01	1
0xD	0x00 for most maps, 0x80 for locked maps (City Nav, City Select, etc.)	1
0xE	Creation year	2
0x10	Creation month	1
0x11	Creation day	1
0x12	Creation hour	1
0x13	Creation minute	1
0x14	Creation sec	1

Data Sub Files

Each sub-file has the same general format but carries a specific type of data. The known file sub-types are:

- RGN. Map elements such as polylines, polygons and points.
- LBL. Labels for map elements, city names, localities, etc.
- TRE. Map structure information that organizes the map elements into a data tree.
- NET. Road network information (intersections, etc)?
- NOD. Routing information?
- MDR. Present when MapSource combines multiple IMG files together into a single map image for uploading to a GPS receiver. Appears to "bind" the individual map files together, possibly to enable cross-IMG searches. Also used in certain map families (City Navigator and City Select) in an "overview" IMG file, generally named ending in "_mdr.img" (e.g., NACSv5_mdg.img), to provide global searches of facilities, addresses, etc.

TRE Sub-File

This file contains the "overview" information for the map elements in the file. It identifies the various zoom/detail levels, the point, line and polygon element types that are present as well as the reference coordinates for the map elements. This file is what the mapping program uses to determine which map elements to display for a particular viewable area at a given zoom level, and can be though of as a "map tree".

The "tree" breaks down the map area into small subdivisions. Each subdivision contains a central reference coordinate, and the map elements are broken out by those subdivisions and referenced relative to that center point.

The reference coordinates in the TRE section are given as 24-bit integers (three bytes). This implies that the maximum granularity for map elements in the image file is approximately .0000214576721 degrees of latitude or longitude. At the equator, a degree of latitude is roughly 365228.16 feet, making the highest resolution for map points about 7.8369461 feet.

Conversion: 1 map unit = $\frac{360}{2^{24}}$ degrees

Note that these are signed integers, so values greater than 0x7FFFFF are negative.

There are numerous versions of the TRE file, each determined by its header length. Known TRE header lengths are: 116, 120, 154 and 188 bytes.

Table 6. TRE Header

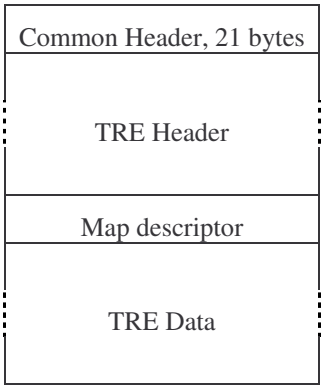
Offset (rel)	Contents	Length (bytes)
0x15	North boundary	3
0x18	East boundary	3

0x1B	South boundary	3
0x1E	West boundary	3
0x21	Offset of map levels section, TRE1	4
0x25	Size of map levels section	4
0x29	Offset of subdivisions section, TRE2	4
0x2D	Size of subdivisions section	4
0x31	Offset of copyright section, TRE3	4
0x35	Size of copyright section	4
0x39	Copyright record size	2
0x3B	0x00 …	4
0x3F	POI display flags Bit 1: transparent map Bit 2: show street before street number Bit 3: show zip before city	1
0x40	???	3
0x43	???	4
0x47	0x0001 ???	2
0x49	0x00	1
0x4A	Offset of polyline overview section, TRE4	4
0x4E	Length of polyline overview section	4
0x52	Size of polyline overview records	2
0x54	???	2
0x56	0x0000 ???	2
0x58	Offset of polygon overview section, TRE5	4
0x5C	Length of polygon overview section	4
0x60	Size of polygon overview records	2
0x62	???	2
0x64	0x0000 ???	2
0x66	Offset of point overview section, TRE6	4
0x6A	Length of point overview section	4
0x6E	Size of point overview records	2
0x70	???	2
0x72	0x0000 ???	2
	TRE Header > 116 bytes	
0x74	Unknown. Map ID?	4

TRE Header > 120 bytes		
0x78	0x00000000 ???	4
0x7C	Offset of section TRE7	4
0x80	Length of section TRE7	4
0x84	Size of TRE7 records	2
0x86	???	4
0x8a	Offset of section TRE8	4
0x8e	Length of section TRE8	4
0x92	???	4
0x96	0x0000 ???	4
TRE Header > 154 bytes		
0x9a	Encrpytion key for map levels section?	20
0xae	Offset of section TRE9	4
0xb2	Length of section TRE9	4
0xb6	Size of TRE9 records	2
0xb8	0x00000000 ???	4

An interesting property of the TRE file is that it also contains a text string in between the file header and the data section. This string appears to describe the map family or the tool used to generate the map. In theory, because this field is not specifically referenced by the map file, it is also not parsed by map viewers and as such could be arbitrarily long and contain arbitrary data. The text string runs from the byte after the header and until the first byte of the data area. It is 0x00-terminated.

Table 7: TRE Sub-File Format



Map Levels

Offset 0x21 references the map levels segment of the TRE sub-file. This array of data structures define the “zoom levels” for the image file which tells the map rendering engine what to draw based on the current view area. Levels are numbered from 0 to 15, with the most detailed level

being 0. Typically, an IMG file containing a GPS map would have four map levels numbered from 0 to 3. In MapSource, the overview map is also an IMG file, but its map levels are numbered much higher, usually 12 and above. Presumably, this is to allow for proper transition between the overview and detail maps.

Each map level defines the zoom level, the number of bits per coordinate (a maximum of 24) and the number of "subdivisions" contained in the level. The levels in this section of the IMG file are defined sequentially, starting with the highest (least detailed) level. The structure looks like this:

Table 8: Map Level Definition

Data member	Size (bytes)
Zoom	1
bits 0-3: zoom level (0-15)	
bits 3-6: unknown (always 0?)	
bit 7: inherited	
Bits_per_coord	1
Subdivisions	2

Note that the more bits per coordinate, the more detail is allowed in the map. As mentioned previously, 1 map unit represents $\frac{360}{2^{24}}$ degrees. If fewer bits are allowed per coordinate in a given map level, then the resolution of those objects drops accordingly by powers of two. 23 bits per coordinate is one-half the resolution of 24 bits per coordinate, 22 bits per coordinate is one half of that (and one-quarter of 24), and so on.

Locked maps

If a map is locked, indicated when (0xD) != 0x00 in the subfile header, then the maps level section is encrypted with an unknown algorithm. This renders the map level information essentially unreadable. The data stored at header offset 0x9a is probably related to it.

This encryption is obviously to prevent an unauthorized rendering engine from parsing the object hierarchy of the map. Without the zoom levels and the number of subdivisions in each level, it is impossible to know what objects to display, and when. We can intuit all of this information from the subdivision section, however, so it is still possible to parse the map without examining the level definitions directly. It is, however, added work and generally a pain in the ass.

Subdivisions

This section of the IMG file defines the map subdivisions. Each "zoom level" is broken down into "subdivisions" that actually contain the map elements such as polygons, polylines and so on. The more detailed a particular region of the map is, the more subdivisions it is likely to contain. This structure allows the rendering engine to quickly determine which objects need to be displayed based on the current view area and zoom level.

Subdivisions, like levels, are defined sequentially in the TRE data section, starting with subdivision number 1 which is in the highest (least-detailed) level. Each subdivision defines its

center point, the height and width of the area it covers and the types of objects that are contained within it. The subdivision also holds two more pieces of information that define the map layout.

The first of those is a pointer to a group of subdivisions in the next, more-detailed level (i.e., the level below it). This is apparently a means of increasing display performance when zooming in to a more detailed level: the current subdivision in view quickly tells the rendering engine where the more detailed subdivisions reside so that the CPU doesn't have to waste time searching the boundaries for the right location. Zooming out, however, has no such shortcuts as there is no link from lower levels to higher ones: the rendering engine must re-walk the level tree.

The second piece of data is a terminating flag that indicates the last subdivision definition in the current "chain" of subdivisions. This concept is difficult to explain textually, so instead we'll turn to diagrams:

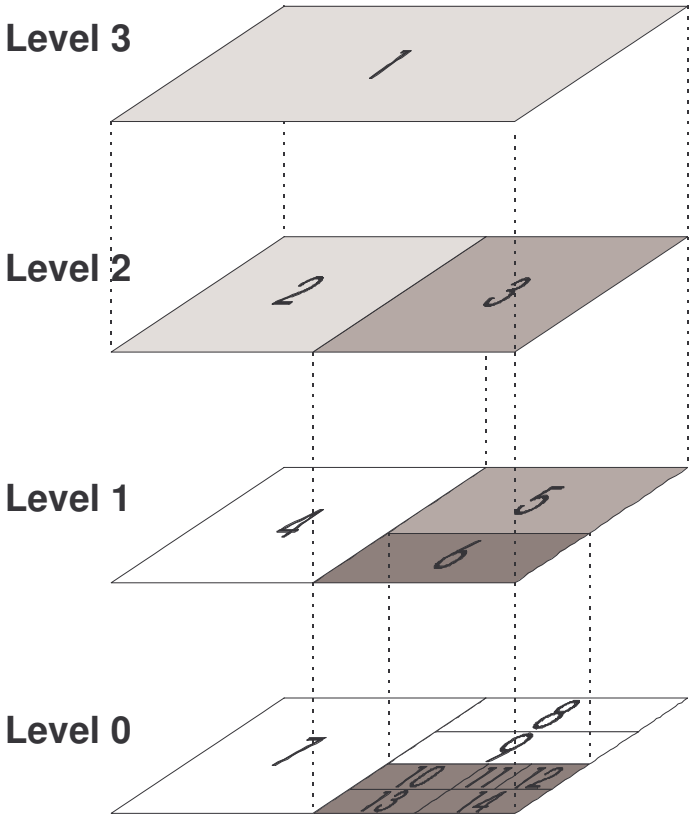


Figure 1: Subdivision Hierarchy

Here, we can see that Level 3, the least detailed level, contains a single subdivision: number 1. This subdivision points to the first subdivision in the next level, which is subdivision number 2. Note however that level 2 contains two subdivisions: numbers 2 and 3, so subdivision number 1 really maps to a group of two subdivisions.

In turn, subdivision number 2 points to subdivision 4 in level 1. Presumably, there is very little detail in this region of the map, so further subdivisions are not necessary. Subdivision 3, however, points to subdivision 5 in level 1. Subdivision 6 completes the area "under" subidivion number 3.

In layer 0, the most detailed layer, subdivision 5 is once again divided into two subdivisions, numbered 8 and 9. Subdivision 6, however, points to subdivision 10, which is one of *five* subdivisions that make up the area "under" 6. Presumably, this is a highly detailed portion of the map, so many subdivisions are needed to hold the map data.

Note that subdivision 8 does not *explicitly* point to subdivision 9, 10 does not point to subdivision 11, 11 does not point to 12, etc. This is what the "terminating" flag is used for in the subdivision data structure. Subdivision number 6 points to subdivision 10, and the map rendering engine reads the subdivisions starting at 10 until it sees a subdivision with a terminating flag (in this case, subdivision 14). It then knows that subdivisions 10-14 all make up the area "under" subdivision 6.

This can be represented as follows:

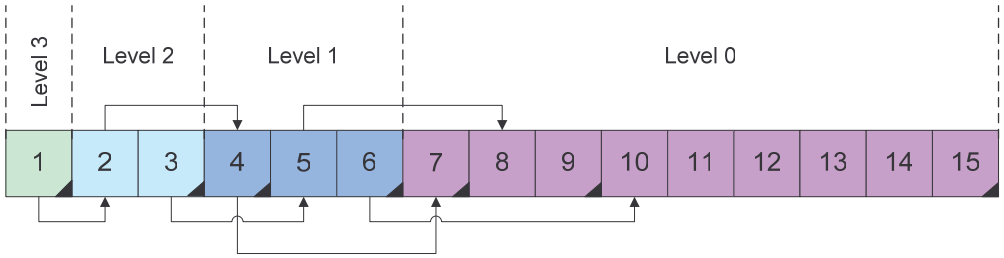


Figure 2: Subdivision Pointer Chain

Here, the arrows represent the pointers from one subdivision to first subdivision in the layer below it, and the black triangles represent the terminating flag in the subdivision chain.

Note that the subdivisions in the lowest (most-detailed) level do not point to other subdivisions. This is because there are no "lower" map levels: the lowest level is as detailed as you can get.

The subdivision structure is as follows. Note that it varies in size, depending on whether you are in the lowest/most-detailed level or not. For the lowest level, it is 14 bytes, and for other levels, it is 16:

Table 9: Subdivision Definition

Data member	Size (bytes)
Rgn_data_ptr (offset in RGN sub-file)	3
Obj_types (values are OR'd together)	1
0x10: points	
0x20: indexed points	
0x40: polylines	
0x80: polygons	
Longitude_center	3
Latitude_center	3
Width	2
Bits 0-14: width (in map units)	
Bit 15: terminating flag	
Height	2

Next_level_subdivision	2
1-based index	
Not present in subdivisions in the lowest map level	

Note that the actual size of the area is represented as (width*2 + 1) x (height*2 + 1) map units around the center point.

Each subdivision contains a pointer into the RGN file where the actual map objects are found as well as a flag indicating the kinds of objects that are present.

Polyline, Polygon and Point overviews

These three sections indicate which polylines, polygons and points are present in a given map level. Each section is basically an array of two or three byte data structures indicating the object type and the highest level in which it is found.

Polyline and polygon records are either 2 or three bytes.

Table 10: Polyline and polygon overview records

Data member	Size (bytes)
Polyline or polygon type	1
Maximum level where present	1
Unknown	1
Only present if three-byte records	

Point records always have a three byte structure.

Table 11: Polyline and polygon overview records

Data member	Size (bytes)
Point type	1
Maximum level where present	1
Point subtype	1

LBL Subfile

This file holds the labels for the map objects, address information for POI's as well as indices of defined zip codes, cities, regions and countries. Because LBL holds so many different pieces of data, it has a rather large header with references to several data structures.

Multiple versions of the LBL file exist and they are identified by the header length. Known header lengths are 170, 196, 208 and 236 bytes (the latter seen in marine maps).

Table 12: LBL Header

Offset (rel)	Contents	Length (bytes)
0x15	Data offset, LBL1	4
0x19	Data length	4
0x1D	Data label offset multiplier (power of 2)	1

0x1E	Label coding	1
0x1F	Country defns offset, LBL2	4
0x23	Country defns length	4
0x27	Size of each country record	2
0x29	0x00000000	4
0x2D	Region defns offset, LBL3	4
0x31	Region defns length	4
0x35	Size of each region record	2
0x37	0x00000000	4
0x3B	City defn offset, LBL4	4
0x3F	City defn length	4
0x43	Size of each city record	2
0x45	0x00000000	4
0x49	Offset of section LBL5	4
0x4D	Length of section LBL5	4
0x51	Size of LBL5 records	2
0x53	0x00000000	4
0x57	POI properties offset, LBL6	4
0x5B	POI properties length	4
0x5F	POI properties offset multiplier (power of 2)	1
0x60	POI properties global mask	1
0x61	0x0000	2
0x63	0x00	1
0x64	Offset of section LBL7	4
0x68	Length of section LBL7	4
0x6C	LBL7 section record size	2
0x6E	0x00000000	4
0x72	Zip defn offset, LBL8	4
0x76	Zip defn length	4
0x7A	Size of each zip record	2
0x7C	0x00000000	4
0x80	Highway defn offset, LBL9	4
0x84	Highway defn length	4
0x88	Size of each highway record	2

0x8A	0x00000000	4
0x8E	Exit defn offset, LBL10	4
0x92	Exit defn length	4
0x96	Size of each exit record	2
0x98	0x0000	2
0x9C	Highway data defn offset, LBL11	4
0xA0	Highway data defn length	4
0xA4	Size of each hwy data record	2
0xA6	0x00000000	4
	LBL Header > 170 bytes	
0xAA	???	4
0xAC	???	2
0xB0	Sort descriptor offset, LBL12	4
0xB4	Sort descriptor length	4
0xB8	Offset of section LBL13	4
0xBC	Length of section LBL13	4
0xC0	Record size for section LBL13	2
0xC2	0x0000	2
	LBL Header > 196 bytes	
0xC4	Tide prediction offset, LBL14	4
0xC8	Tide prediction length	4
0XCC	Size of each tide prediction record	2
0xCE	0x0000	2

Many subfiles—as well as the other sections of LBL—reference the labels via an offset in the primary data area. These offsets are multiplied by the power of two at offset 0x1D. Usually, this number is 0, meaning a multiple of 1 (2^0=1) and no change, but large image files may set the exponent to one or more, indicating that label offsets need to be doubled (2^1=2), or quadrupled (2^2=4), etc.

POI properties offsets are similarly modified by the power of two at 0x5F.

Label Encoding

All labels in the LBL file are encoded into a bit-stream that is apparently designed to conserve space by eliminating non-printable characters from the character set. The encoding also allows the addition of special characters for a particular language's character set, specialized symbols on the map screen (such as highway shields), and specialized functions that modify the text in order to, for example, hide abbreviations and suffixes that are useful for searching but not intended for direct display.

Before going into the label encoding, however, it is important to understand the labels themselves and how the GPSr and MapSource render them. For starters, labels are stored in upper-case only, meaning that the encoding does not directly support lower case characters. On the typical Garmin GPS, labels on the map display are shown only in upper case, probably due to limitations in display resolution. MapSource and the search function on GPS units, however, display in mixed case. This is done by automatically shifting from upper case to lower case after the first character in a word, with exceptions for the abbreviations NW, SW, SE and NE. For example, the upper-case label stored as “NW PETTYGROVE AVE” gets displayed as “NW Pettygrove Ave”. There are special codes that can force lower case lettering when its required, but there do not appear to be codes that force upper case. So if, for example, you wanted to name a location in all upper-case, such as “PGE Park”, you are out of luck. The label is stored as “PGE PARK” and gets displayed as “Pge Park”.

There are three kinds of label encodings supported by IMG files. The first is 6-bit encoding and it is used in maps in the United States. Maps requiring more characters, such as those used in other continents, particularly non-English-speaking ones, use either 8-bit or10-bit encodings. The label coding value at offset 0x1E determines the encoding used in the map:

Table 13: Character Encodings

Decimal Value at 0x1E	Encoding
6	6 bit
9	8 bit (yes, I know)
10	10 bit

6 Bit Encoding

In six bit encoding, each character is stored as six bits in a variable-length bit stream. The length of the label is not defined in advance so each label must be parsed six bits at a time until a terminator character is read.

Since a byte consists of eight bits, this implies that each character in the six bit encoding scheme uses up only part of a byte, and that some characters will span two bytes. Reading three bytes will yield exactly four characters.

The bits are read from MSB to LSB in each byte, jumping from the LSB of the current byte to the MSB of the next byte, as follows:

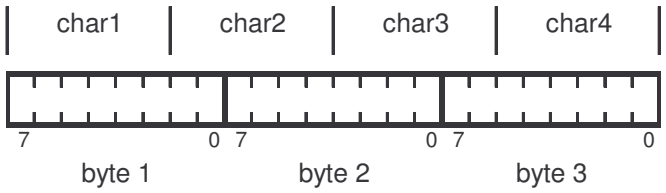


Figure 3: 6 Bit Encoding

Each six bit character has an unsigned value range from 0x00 to 0x3F. Values greater than 0x2F (when the two highest order bits are set) represent a terminating character and denote the end of the label string. The basic character values are presented in the table, below:

Table 14: 6 Bit Basic Characters

Values	*Characters*
0x00	Space
0x01-0x1A	A-Z
0x1B	*Next char is symbol*
0x1C	*Next char is lower case*
0x1D-0x1F	*Special*
0x20-0x29	0-9
0x2A-0x2F	*Special*

Character value 0x1B is something like a “shift” key, denoting that the following character will be a symbol rather than a letter. There is no “shift lock” so each symbol character must pre preceded by the 0x1B code. The symbol character codes are as follows:

Table 15: 6 Bit Symbol Characters

Values	*Characters*
0x00-0x0F	@!"#$%&'()*+,-./
0x1A-0x1F	:;<=>?
0x2B-0x2F	[\]^_

Character value 0x1C denotes that the following character will be in lower case rather than upper case, with the exception of 0x00 which becomes an apostrophe/backtick. Like 0x1B, the 0x1C character has to precede each lower case character.

Table 16: 6 Bit Lower Case Characters

Values	*Characters*
0x00	`
0x01-0x1A	a-z

Character values 0x1D-0x1F denote special actions.

Table 17: 6 Bit Special Actions

Value	*Action*
0x1D	Delimiter between a formal name and its abbreviation
0x1E	Hide the preceding characters and insert a space.
0x1F	Hide the following characters and insert a space

0x1D is seen in country and state/province names where a full name and abbreviation are listed, such as “UNITED STATES[0x1D]US” and “WASHINGTON[0x1D]WA”. Presumably, this allows searches based on both strings.

0x1E and 0x1F are generally used to hide the prefix and suffix in a street name when displayed on the GPSr, such as in the label “NW[0x1E]OAK[0x1F]ST”. Here, the GPS will display only

“OAK” instead of “NW OAK ST”. The goal appears to be to reduce clutter on the map screen, while still storing it in its entirety (so that it can be presented during searches and driving directions). MapSource also appears to display the complete street label, presumably because the PC has a much larger screen and thus can display the full label without cluttering the map.

Character values 0x2A through 0x2F denote special graphics that are used to decorate the labels for major highways.

Table 18: 6 Bit Highway Symbols

Value	*Symbol*
0x2A	Interstate highway shield
0x2B	U.S. highway shield
0x2C	State highway (circle)
0x2D	Canadian national highway, blue and red box
0x2E	Canadian national highway, black and white box
0x2F	Highway with small, white box

8 Bit Encoding

No information.

10 Bit Encoding

No information.

Country Records

Country definition records start at the offset pointed to by LBL offset 0x1F. This is a one-based array of all countries defined in the IMG file (the first country is country #1, the second is #2, etc.). Each country record is generally three bytes long, containing a single pointer to the country’s name in the LBL data section (thus, and offset of 0 would actually be at offset (0x63) in LBL).

Table 19: Country Records

Data member	Size (bytes)
lbl_data_ptr (offset in LBL sub-file data area)	3

Notice that the record does not contain the country number. It is up to the end application to count the country records as it reads them.

Interestingly, the LBL header specifies a size for the country record. It’s not clear what purpose this serves.

Region Records

Region records start at the offset pointed to by LBL offset 0x2D. Each record is generally five bytes long, and like the country records is a 1-based array of regions defined in the IMG file. A region is generally used to define a state or province in a country.

Table 20: Region Records

Data member	*Size (bytes)*
country_index	2
lbl_data_ptr (offset in LBL sub-file data area)	3

The country_index parameter refers to the country defined in the country records sub-section, defined above. The assumption here is that a region belongs to a single country.

Again, the LBL header defines the length of the region record. It's not clear why this record might be variable-length, except possibly to allow a larger integer type for country_index (perhaps in case an IMG file has more than 65,535 countries defined?)

City Records

City records begin at the offset pointed to by LBL offset 0x3B. These are five bytes long.

Table 21: City Records

Data member	Size (bytes)
city_data (see below)	3
city_info	2
Bits 0-13: region_index	
Bit 14: unknown	
Bit 15: point_ref	

This record is more complicated than the region and country records. The city_info parameter dictates whether the first three bytes in the record refer to a label, or to a pair of indices which locate the city's corresponding point in a given subdivision. Presumably this is to allow fast searches of cities. This is determined based on whether or not bit 15 is set in city_info.

If point_ref is *not* set, then city_data is a label data pointer (an offset in the LBL sub-file data area, just as with reigon and country records). The region_index parameter denotes which region the city resides in. As with regions in countries, a city belongs to only one region.

If point_ref *is* set, then city_data is the following structure:

Table 22: City Data Alternate Structure

Data member	Size (bytes)
point_index	1
subdivision_index	2

Subdivision_index states which subdivision the city can be found in. That this is a two-byte integer implies that there can be no more than 65535 subdivisions in a map level. The point_index parameter says which point in the subdivision corresponds to this city. As it is a single byte in length, this implies that cities must appear within the first 256 points defined for a given subdivision, and that a subdivision cannot contain more than 256 cities.

Again, it is not clear why the city record length is defined in the LBL header section since its value is (apparently) fixed.

POI Records

Each POI record is a variable-length record that defines extra properties for points of interest, such as name, address and phone number. The data is compacted using yet another form of encoding in order to minimize storage space. While this incurs a slight performance penalty, it is irrelevant since POI address information is only displayed on user request and thus is not a performance-critical operation.

Parsing the POI properties requires looking at the global mask set in the LBL header at 0x60. This is a bitmask which defines the superset of properties for all POIs. In other words, it specifies which properties can be defined for a given POI.

Table 23: POI Property Mask

Bit	Property
0	has_street_num
1	has_street
2	has_city
3	has_zip
4	has_phone
5	has_exit
6	has_tide_prediction
7	unknown

By default, all POI's have the properties that are set in 0x60, but individual POI definitions can contain a point-specific mask indicating that only a subset of properties are present.

To determine the point-specific POI properties, create a new bitmask by setting bits from lowest to highest for each POI property that is defined in the lobal bitmask at 0x60. For example, if your global bitmask is 0x12 (mask 00010110), then your POIs can at most contain a street, city and phone number. The point-specific properties would then be determined by masking against 0x07 (mask 00000111). Now, bit 0 represents *has_street*, bit 1 becomes *has_city*, and bit 3 is *has_phone*.

Now mask the *property_mask* byte (defined below in Table 24: POI Properties Structure) with the subset bitmask to determine which properties are defined for the POI. using the above example, if *property_mask* were set to 0x05 (binary 00000101), then the POI would have a street and phone number defined, but no city.

Table 24: POI Properties Structure

Data member	Size (bytes)
poi_data	3
Bits 0-21: lbl_data_ptr (offset in LBL sub-file data area)	
Bit 22: unknown	
Bit 23: has_partial_properties (if set, the following byte contains flags that specify the POI's properties as a subset of the global mask)	
property_mask	1

Note: Only present if has_partial_properties *is set. See above for how to process the bitmask.*	
street_number	varies
Note: present only if has_street_number *is set*	
street_lbl_ptr (offset in LBL sub-file data area)	3
Note: present only if has_street *is set*	
city_index	1 or 2
Note: present only if has_city *is set. Length is two bytes if more than 255 cities are defined in LBL.*	
zip_index	1 or 2
Note: present only if has_zip *is set. Length is two bytes if more than 255 zips/postal codes are defined in LBL.*	
phone_number	varies
Note: present only if has_phone *is set*	

Street_number is a variable-length byte stream giving the street number for the POI's postal address. Obviously wanting to save every byte possible, the street number is encoded in base 11 using the lowest seven bits of the value (with one exception: see below). Each base eleven number provides up to two digits of the street address string, with the digit "A" representing a terminator when the second digit in the result is not used. The byte stream starts and terminates with the high bit (bit 7) being set, indicating when to stop reading bytes. This base 11 encoding scheme allows for the representation of all digit pairs from 00 to 99 in each byte.

As an example, the following byte stream would decode to the street address "15856":

0x90 0x5d 0xCC

The high bit is set in 0x90 and 0xCC, representing the terminators for the byte stream. Masking off, bits 0-6 yields the following bytes:

0x10 0x5D 0x4C

The corresponding base-11 values become:

15 85 6A

The "A" is base-11 for the number 10, denoting an empty/unused digit place and giving us the final address string of "15856".

If the first byte of *street_number* is not a base-11 terminator, meaning the high bit (bit 7) is not set, then *street_number* is actually a pointer to the data section of the LBL sub-file where the street number is stored as a text label. This technique is used when the street address includes a non-numeric component, such as an apartment or suite reference. In these cases, the first byte of the stream becomes bits 17-24 while the next two bytes are read as a little-endian word.

For example, if we were given the data stream:

0x01 0x4E 0xFB

we would see immediately that the first byte does not have the high-bit set, so this is not a base-11 number. Instead, we have a pointer into the LBL data section at offset 0x01FB4E.

The resulting labels will almost always contain a special string that is expanded using the following substitution table:

Value	*Substitution*
-<space> -0	#
-1	APT
-2	BLDG
-3	DEPT
-4	FL
-5	RM
-6	STE
-7	UNIT

The special strings are also a delimiter of sorts, indicating that the following text should be appended to the street name. For example, the label:307-24

Would translate to:

307 *streetname*, BLDG 4

Street_lbl_ptr is a pointer to the street's name in the data section of the LBL sub-file. *City_index* and *zip_index* are indices to the city and zip records, stating which city and postal code the POI resides in (there is a many-to-many relationship between cities and zip codes in some countries, including the U.S.).

The phone number is encoded in the same manner as the street address (base 11, yielding two digits per byte) with the base 11 value of 10 representing either a) a separator or dash if in the middle of the digit sequence, or b) a terminator if at the end.

Zip Records

The zip records begin at the offset pointed to by 0x72. Each record is a three-byte pointer to a string in the label data area that contains a zip or postal code. They are laid out as a 1-based array, similar to country, region and city records.

Highway Records, Exit Records and Highway Data Records

These records were apparently used by the old *Roads and Recreation* maps to describe services available at various highway exits, rest areas and so on.

They are not currently described in this document.

RGN Subfile

The RGN subfile defines the map elements, consisting of points, indexed points, polylines and polygons. While the overall layout of the subfile is relatively simple, the data structures contained inside are just the opposite: in an effort to save literally every possible bit of memory (yes, that's *bit* and not *byte*), Garmin has devised a complicated data compaction scheme that squeezes polylines and polygons into as few bytes as possible without having to resort to data compression. This provides a nice tradeoff between data size, working memory size and performance.

The RGN header is very simple, containing only two fields after the common header:

Table 25: RGN Header

Offset (rel)	Contents	Length (bytes)
0x15	Data offset	4
0x19	Data length	4

That's it. As the RGN data section immediately follows the header, the data offset at 0x15 generally matches the header length in the common header.

Layout

The data area is much messier than the header. While it has a structure, there are no cues in the RGN header for decoding it. This information is, instead, stored in the subdivision definitions, found in TRE. Each subdivision contains two pieces of information that are necessary for decoding the RGN structure: the first is an offset into RGN indicating where the elements of that subdivision are defined, and the second is the types of elements that are present in that subdivision. Without both of these pieces of info, we don't know where to begin parsing elements, where to stop and which subdivision they belong too.

Thus, the RGN data area can be thought of as an array of data segments, with each segment corresponding to exactly one subdivision. Each data segment begins at the offset defined by the subdivision and ends at the next subdivision offset. The last subdivision stops at the end of the RGN data block.

The data layout looks like this:

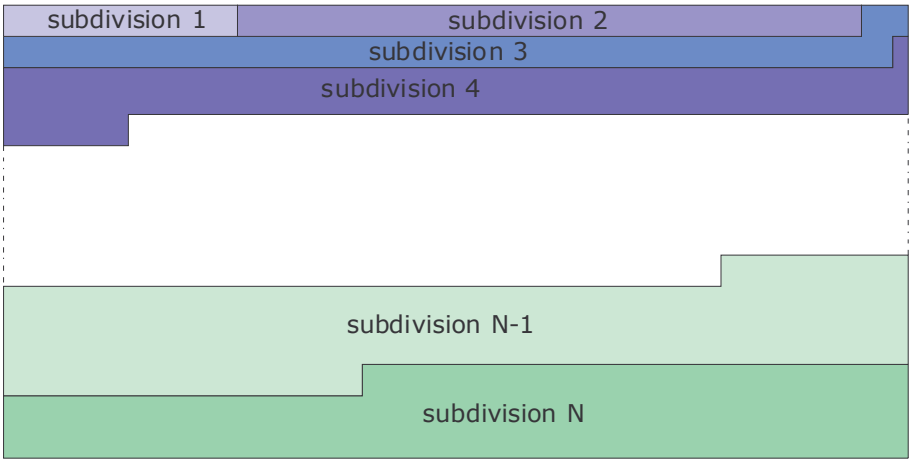


Figure 4: RGN Data Segments

Each data segment is broken up into two pieces: the first is a set of pointers to the element groups in the data segment, and the second is the element groups, themselves.

The elements are grouped by type and are always presented in the same order: points, indexed points, polylines and then polygons. Any of these element groups may be absent (not all subdivisions have all element types), but the order is still the same. For example, if there are no indexed points in a particular subdivision, then the order would be points, polylines, polygons.

The pointers at the beginning of the data segment point to the element groups, and like the element groups they come in the same order with one exception: there is no pointer to the first element group. This appears to be a space-saving device in the map file. To find the first element group, you simply read past the set of pointers.

A visual representation of the RGN segment is shown here:

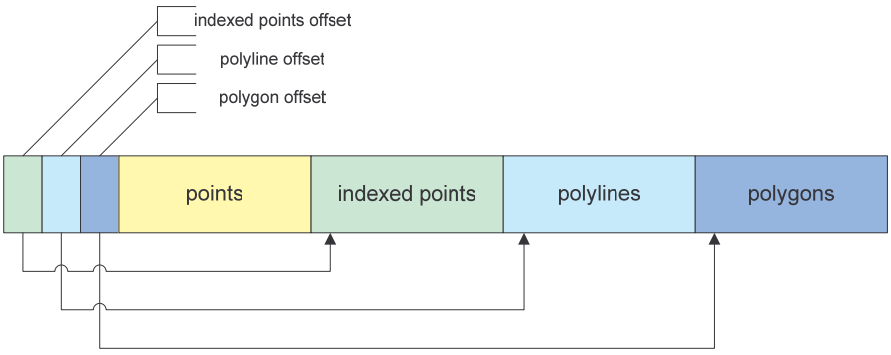


Figure 5: RGN Data Segment

As you can see, there is no pointer to the “points” element group, which is the first set of element types in the RGN data segment. If there were no “points” or “indexed points” element groups, then we would see the following:

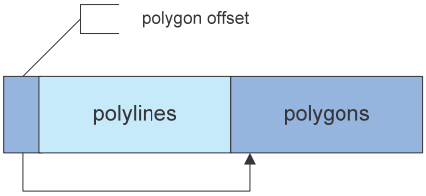


Figure 6: RGN Data Segment with Missing Element Groups

In the above figure, there are only polylines and polygons in the data segment (a typical arrangement for high-zoom/low-detail map levels) so only one pointer is necessary: the one that shows the start of the polygon group.

This arrangement begs the question, "how does one know how many elemnt types are present in the RGN data segment?" The answer lies in the TRE sub-file where the subdivision definitions are kept. If you recall from Table 9: Subdivision Definition, each subdivision definition states what element types are present in the subdivision. Thus, you know how many pointers are present (the number of object types – 1) as well as which element groups they refer to.

Each element pointer is a two-bytes (a single word) in length. It's an offset referenced *from the beginning of the segment* (i.e., the first pointer is at offset 0x0):

Table 26: RGN Element Pointer

Data member	*Size (bytes)*
Offset	2
Note: relative to start of the RGN segment	

Since the pointers allow for only a single word length for element groups, this implies that there can be no more than 65535 bytes of data immediately before the last element group in the data segment (effectively forcing the map generator to use smaller subdivisions if this limit is reached in dense map areas).

Parsing each element group requires that you start at the offset for the element group, and then parse data until you reach one of the following:

1. the offset of the next element group
2. the beginning of the next RGN data segment, or
3. the end of the RGN data area (an issue when you are parsing the last data segment in the file)

Points and Indexed Points

Points and indexed points share the same structure. The difference between them is that an indexed point is one that is indexed for finds.

Table 27: Point Data Structure

Data member	*Size (bytes)*
point_info	1
Bits 0-7: point_type	
Bit 8: has_subtype	

lbl_data_ptr (offset in LBL)	3
longitude_delta	2
latitutde_delta	2
subtype	1
Present only if has_subtype is set	

The latitude and longitude are given as deltas from the center point of the subdivision. Note that these values must be left-shifted by 24 - bits_per_coord based on the level containing the subdivision. For example, if the subdivision were in level one, and level one defined 21 bits per coordinate in TRE, then you would left-shift by 24-21= 3.

The *subtype* member is only present if the *has_subtype* bit is set in *point_info*. This subtype is generally used to set a specific classification of the point type for searches (such as searching for a specific restaurant cuisine, like American, French, etc.).

Polylines and Polygons

The data structures are very similar for polylines and polygons, since graphically the latter is just the former with a fill. Operationally, polylines have some extra data fields to account for such things as road direction and road networks, where applicable.

The basic structure is as follows:

Data member	*Size (bytes)*
poly_type	1
Bits 0-6: polyline or polygon type	
Bit 7: direction indicator (polylines only)	
Bit 8: two_byte_len	
label_info	3
Bits 0-21: label_offset (usually in LBL, but see below)	
Bit 22: extra_bit	
Bit 23: data_in_NET	
longitude_delta	2
See Points and Indexed Points	
latitutde_delta	2
See Points and Indexed Points	
bitstream_len	1 or 2
	(2 if two_byte_len is set)
bitstream_info	1
Bitstream	*bitstream_len*

Polylines and polygons represent one of the most complicated data structures in the IMG file, and show the great lengths that Garmin goes to in order to conserve every possible bit in a map image.

The first six bits of *poly_byte* determine what kind of polyline or polygon is being described. If the object is a polyline, then bit 6 indicates whether or not it is directional, referring to a one-way street (this information probably was used by older maps that did not supply autorouting data, but still needed to indicate to the user that a particular street was one-way). The highest order bit, bit 7, defines whether the *bitstream_len* parameter is one byte or two bytes (a word) in length. A bitstream that has more than 255 bytes will require a word-length *bitstream_len* value.

The *label_info* member is a 3-byte pointer to the label associated with the poly object. The first 22 bits represent the offset in the LBL sub-file. If the object is a polyline, and the map has autorouting information, then bit 23 might be set, indicating that *label_info* is actually a pointer into the road definitions section of the NET sub-file. This section of NET provides pointers to the object's label or labels in LBL and is described in a later section.

Bit 22 in *label_info* is *extra_bit*, indicating that there is one more bit per coordinate in the bitstream than defined by *bitstream_info*.

The starting point of the polyline and polygon are defined by *longitude_delta* and *latitude_delta*, just as in a point or indexed point record. They must be shifted by 24 minus the number of bits per coordinate for the level containing the subdivision.

The successive points in the poly object are compacted into the bitstream defined by *bitstream_len*, *bitstream_info* and the bitstream itself, *bitstream*.

Bitstream parsing

Bitstream parsing is one of the most tedious and convoluted tasks in decoding an IMG file.

Each point in a poly object is defined relative to the previous point. The IMG file encodes these relative "deltas" into a bitstream, designed to minimize the amount of storage required to define the points. The assumption here is that each delta between one point and the next is likely to be very small in comparison to the extents of the subdivision, so don't waste whole bytes or words when only a few bits will do.

For example, if the delta between points was 3 map units, this number could be stored as the binary value 11, using only two bits out of eight possible in a byte. If all the deltas in the poly object were values of three or less, then we could squeeze two points into each byte (each point requires two deltas, one for latitude and one for longitude). Compare this to using one byte for each delta, and this represents a compaction of 8:1.

For larger deltas, the savings may not always be as significant, but could represent the difference between having to use a two-byte word and a single byte, and even in large polyobjs, saving two or three bits can add up over time. A typical map might have hundreds of polylines and polygons, each having anywhere from a handful to several dozen or more points. Just a few bits multiplied out a thousand or more times starts to add up to a noticeable savings in data size.

One can imagine that Garmin employs complex optimization schemes to determine how many bits per coordinate point are required to define a poly object, whether or not an object should be split into two pieces (to allow for smaller bits-per-coordinate values) and even how many subdivisions should be present in a map level. Thus, while defining map objects is fairly easy at a conversational level, the process of encoding those into the Garmin IMG format is non-trivial, particularly if you are attempting to generate the smallest possible map file.

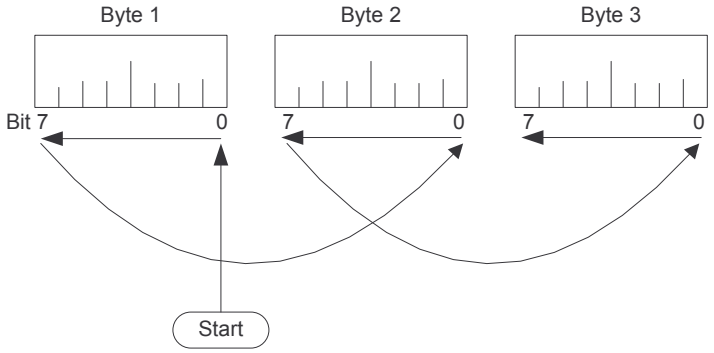
The first parameter, *bitstream_len* defines how many bytes are present in the bitstream. It is either a byte or a word in length, depending on whether or not the bit *two_byte_len* is set in *polytype*.

The second parameter, *bitstream_info*, defines the *base* number of bits used for the latitude and longitude deltas in the bitstream. Note that this is a base number: it will be used in calculating the final number of bits per coordinate later on.

Data member	*Size (bytes)*
bitstream_info	1
Bits 0-3: base bits per longitude	
Bits 4-7: base bits per latitude	

The base number of bits per latitude, or y-component, are stored in the high bits of the byte, while the base number of bits per longitude, or x-component, are stored in the low bits. Thus, if you have the value 0x37, then that translates to binary 00110111, telling us that the base bits per coordinate are 3 for latitude and 7 for longitude.

The bitstream follows the *bitstream_info* parameter and is *bitstream_len* bytes long. The bytes in the bitstream are read from "left to right", but the bits are read from LSB to MSB ("right to left"):



The very first bit of the bitstream (byte 0, bit 0) determines whether or not all the longitude, or x-component, deltas are same-signed. If the bit is cleared, then the longitude values all vary in sign. If the bit is set, then the sign of all the longitude components is determined by the next bit, bit 1. If bit 1 is set, then the signs are all nagative. If the bit is cleared, the signs are all positive.

The next bit (either bit 1 or bit 2) does the same for the latitude, or y-component, deltas. The sign is determined exactly in the same manner as the longitude/x-component.

Immediately following these special bits are the bits representing the delta pairs, themselves. Each delta pair has the longitude followed by the latitude. The number of bits for each "half" is determined using the following algorithm.

1. Start with the number 2. All deltas have at least two bits per coordinate.
2. Take the base number of bits per coordinate as determined from *bitstream_info.*
    a. If this value is <= 9, then just add the value

b. If this value is >9, then add the quantity (2*value – 9). E.g., if the value is 12, then add 15 (2*12-9 = 15).

3. If the sign of the component is variable, add 1.
4. If *extra_bit* is set in *label_info*, add 1.

The final value represents the number of bits per coordinate (the range of values is 2 to 25, once all possible modifiers are taken into account). This value must be calculated for both the longitude and latitude components.

Finally, the bits in each coordinate delta pair are presented in little-endian format. Each resulting "word" represents delta value.

If the sign of the delta is the same for all coordinates, then you simply take the positive or negative of that value as appropriate. If the sign is allowed to vary, then more special processing is required, described below.

Before we go too far, however, let's take a look at a simple example:

Take the following polyline object definition:

```
05 40 07 00 bc 01 85 00 03 57 6d 12 0a
```

This represents an arterial road (based on the type of 0x05). In *label_info*, which is 0x000740, the *extra_bit* field is clear as is the *two_byte_len* field. Our first latitude and longitude points are 0x0085 and 0x01bc, respectively, relative to the center of the subgroup.

The bitstream is 0x03 bytes long and *bitstream_info* is set to 0x57. In binary, this value is 01010111, giving a base number of 7 bits per longitude coordinate and 5 bits per latitude coordinate.

The first byte in the bitstream is 0x6D, which corresponds to the binary value 01101101. Bit 0 in this sequence is set, indicating that the longitude deltas always has the same sign. The next bit, bit 1, is clear, so that sign is always positive.

The next bit, bit 2, is set, so the latitude deltas also all have the same sign. The next bit, bit 3, is set, so the latitude components are always negative.

The number of bits per latitude and longitude component are:

Latitude = 2 + 5 (base) + 0 (sign always the same) + 0 (*has_extra_bit* not set) = 7

Longitude = 2 + 7 + 0 + 0 = 9

Thus, each latitude delta has 7 bits and each longitude delta has 9 bits.

Returning to the bitstream, the hex values are:

```
6d 12 0a
```

This corresponds to the following bit stream:

```
01101101 00010010 00001010
```

We skip bits 0-3 in the first byte since those were used to determine our signs. Processing

bits from LSB to MSB, bytes from left to right, and filling in bits from LSB to MSB in each word, we get the following sequence of binary values. Note that longitudes are read before latitudes:

```
100100110 1010000
```

The leftover bits in byte 3 (bits 4-7) are not used.

This gives a decimal value of 294 for the longitude and -80 for the latitude (remember that our latitude values are always singed negative).

If Sign Varies

If the sign of the longitude or latitude (or both) varies, then an extra bit is added to each delta to provide the sign value for that delta. The sign is always the highest-order bit. If it's set, then the value is negative and if it's clear, the value is positive.

Negative values are calculated according to convention for representing signed integers. E.g., the word 1110 value would be -2 (110 = 6, 1000 = 8, result= 6 – 8 = -2) and word value 11010 would be -6 (1010 = 10, 10000 = 16, result= 10 – 16 = -6).

If, however, the value has *only* the "sign" bit set (with all other bits 0), then we have *yet another* a special case. In this case, we take 1 minus the unsigned value of the word and then add the signed value of the *next* word. This lets us achieve the possibility or larger negative values than a single word would allow.

Once again, it is easier to follow this through example.

Take the following polyline object definition:

```
08 00 00 00 d0 01 e6 fe 03 01 ab 7a b1
```

This represents a highway ramp with no label. In *label_info*, which is 0x0000000, the *extra_bit* field is clear as is the *two_byte_len* field. Our first latitude and longitude points are 0xfee6 and 0x01d0, respectively, relative to the center of the subgroup.

The bitstream is 0x04 bytes long and *bitstream_info* is set to 0x01. In binary, this value is 00000001, giving a base number of 1 bits per longitude coordinate and 0 bits per latitude coordinate.

The first byte in the bitstream is 0xab, which corresponds to the binary value 01100011. Bit 0 in this sequence is set, indicating that the longitude deltas always have the same sign. The next bit, bit 1, is also set, so that sign is always negative. The next bit, bit 2, is clear, so the latitude deltas have a varying sign. The number of bits per latitude and longitude component are:

Latitude = 2 + 0 (base) + 1 (sign varies) + 0 (*has_extra_bit* not set) = 3

Longitude = 2 + 1 + 0 + 0 = 3

Thus, each latitude delta has 3 bits and each longitude delta has 3 bits.

Returning to the bitstream, the hex values are:

```
ab 7a b1
```

This corresponds to the following bit stream:

```
10101011 01111010 10110001
```

We skip bits 0-2 in the first byte since those were used to determine our signs. Processing bits from LSB to MSB, bytes from left to right, and organizing bits from little-endian bit order, we get the following sequence of binary values. Note that longitudes are read before latitudes:

```
101 010 101 111 010 100 101
```

This tells us that we have three coordinate pairs. The latitude values have a variable sign and the longitudes are always negative. Thus, the first delta pair is:

Lat= 2, Long = -5

In the second pair, the sign bit is set in the latitude component, giving us a negative value:

Lat= -1, Long= -5

In the last pair, we reach the special case where only the sign bit is set for the latitude component (binary value 100). In this special case, we take 1 – 4 = -3 (unsigned binary 100 = 4) and then add the value of the next word (binary value 101). The sign bit is set, so this next word has a value of -3, giving us a total final value of -6. Our final delta pair then is:

Lat= -2, Long= -6

Once all the coordinate deltas have been calculated, they, like the first point, must be left-shifted by 24 – bits_per_coord for the level containing the subdivision to obtain the final coordinate deltas.

NET Subfile

The NET subfile contains road information for those maps products which support geocoded street addresses and multiple labels for road segments. It also contains pointers into the NOD subfile in maps that support autorouting.

There are multiple versions of NET, determined by the header length. Known header lengths are 55, 59 and 67 bytes.

Table 28: NET Header

Offset (rel)	Contents	Length (bytes)
0x15	Road definitions offset, NET1	4
0x19	Road definitions length	4
0x1D	Road definitions offset multiplier (power of 2)	1
0x1E	Segmented roads offset, NET2	4
0x22	Segmented roads length	4
0x26	Segmented roads offset multiplier? (power of 2)	1
0x27	Sorted roads offset, NET3	4
0x2B	Sorted roads length	4
0x2F	Sorted roads record size *Always 0x0003?*	2
0x31	0x00000000	4
0x35	Unknown Always 0x01?	1
0x36	???	1
	NET Header > 55 bytes	
0x37	???	4
	NET Header > 59 bytes	
0x3b	Unknown Always 0x00000001?	4
0x3f	0x000000	4

Offset multipliers

As in LBL, 0x1D and 0x26 indicate power of two multiplier to offsets referenced into their respective sections of NET.

Road Definitions

This section defines labels for roads, street addresses and contains pointers into NOD in autorouting maps. Each road definition record is fairly complicated because it contains several

variable-length structures that must be processed sequentially. This makes it rather difficult to represent the structure in a tabular format.

A byte with value 0xf2 is occasionally used as a “padding” or “NULL record” byte in between road definition records.

Data member	*Size (bytes)*
label_info Bits 0-21: label_offset in LBL Bit 22: pointer to segmented roads section follows Bit 23: last_label ***or*** segmented_road_offset If bit 22 set in preceding *label_info* field *Note: More label info follows until last_label bit is set. Total of 4 labels, max.*	3 per label until *last_label* set
road_data? Unknown bit field Bit 0: unknown (always 0?) Bit 1: one-way? Bit 2: lock to road/show next road? Bit 3: ? Bit 4: has_street_address_info Bit 5: addr_start_right Bit 6: has_nod_info Bit 7: major highway?	1
road_length (feet) *Note: normalized by minimum map resolution*	3
rgn_index_overview Bits 0-6: segments_per_level Bit 7: last_record *Note: More node info follows until last_index bit is set.*	1 per record until *last_record* set
road_indices Bits 0-7: road_index (in subdivision) Bits 8-23: subdivision_number	3 per segment count in *rgn_index_overview* (as above)
house_number_blocks *Only if* has_street_address_info *bit is set in* road_data	1
Street address info block	*varies*

Only if has_street_address_info *is set*	
nod_info	1 byte
Bit 0: two_byte_pointer	
Bit 1: three_byte_pointer	
Bits 2-7: unknown	
Note: Bits 0 and 1 are mutually exclusive	
nod_offset	2 or 3 bytes as per *nod_info*
Unkown_data_stream	*varies, algorithm unknown*

The *lbl_info* parameter is basically a chain of 3-byte pointers into the LBL file. The last label pointer in the chain has the high bit (bit 23) set, indicating the last label in the chain. This is used, among other cases, for highway segments that carry more than one name, such as an interstate highway that doubles as a state highway, or major highways that have a "familiar" name in addition to their number.

The *road_data* block is a set of flags indicating either further road information, or the presence of additional data blocks in the record. Bit 1 indicates that this road is one-way, and changing this bit will affect routing. If bit 3 is set, then the road record also contains a street address block. If bit 5 is set, the address numbering starts on the right side of the road (odd address numbers) instead of the left. Bit 6 indicates the presence of a pointer into the NOD subfile, and indicates a routable road.

Following these two bytes is a chain of one-byte records that terminate when the high bit (bit 7) is set. This *rgn_index_overview* section defines the number of indices present for the road in each level of the map, starting with level 0. Roads that appear in multiple map levels will have multiple indices, one per level where that element is found.

Road_indices is a chain of three-byte records with one record per index in the *rgn_index_overview* chain. Bits 0-7 to contain the index of the road in the polylines section of its subdivision, and bits 8-23 define the subdivision number that contains the polyline element.

If the flag *has_street_address_info* is set, then *num_street_address_blocks* and *street_address_info* are present.

If the flag *has_nod_info* is set, then a *nod_info* byte follows. This is either a two or three byte pointer (as per the flags in *had_nod_info*) into section 2 of the NOD subfile, and indicates a routable road.

Street Address Information

If street address blocks are present in the road definition then the follow variable-length data structure is present:

Data member	*Size (bytes)*
addressing_flags	1
bits 0-1: unknown	

bits 2-3: first field 00=number field 10=zip index (from LBL) 11=no field bits 4-5: second field 00=number field 10=city index (from LBL) 11=no field bits 6-7 third field 00=number field 11=no field	
field1	*varies*
Absent if bits 2 and 3 are set	
field2	*varies*
Absent if bits 4 and 5 are set	
field3	*varies*
Absent if bits 6 and 7 are set	

The length of each of the three address field varies as follows:

- If the field is a number field, then the first byte gives the length of the data stream that follows
- If the field is a zip/postal code or city index, then the length is either 1 or 2, depending on the number of zip/postal and city records defined in LBL, respectively

The number field has not been decoded.

Segmented Roads

Nothing is known about this section at the current time.

Sorted Roads

This section contains 3-byte pointers to entries in the road definition section of the NET subfile. It may represent road intersections and is likely indexed. Each record is as follows:

Data member	*Size (bytes)*
road_pointer?	3
Bits 0-21: offset in road definitions section	
Bit 22-23: label_number (0-3)	

Bits 22-23 indicate the label number for those roads that have more than one label in their definition. The possible range of values is 0 to 3, with 0 referring to the first label. This suggests

that each road may have a maximum of four labels (one primary and three alternate/supplemental).

NOD Subfile

This section contains route information for those maps that support autorouting. Much of NOD has yet to be decoded.

Table 29: NOD Header

Offset (rel)	Contents	Length (bytes)
0x15	Offset for section NOD1	4
0x19	Length of section NOD1	4
0x1D	Unknown	2
0x1F	Unknown	2
0x21	Unknown	2
0x23	Unknown	2
0x25	Road data offset, NOD2	4
0x29	Road data length	4
0x2d	0x00000000 ???	4
0x31	Boundary nodes offset, NOD3	4
0x35	Boundary nodes length	4
0x39	Boundary nodes record length	1
0x3a	0x000000	4

NOD1

Nothing is known about this section. Portions of it appear to be referenced by the road data section.

Road Data

This section contains what appear to be variable-length records that are referenced by NET. The full meaning of these data records is not known.

Data member	*Size (bytes)*
road_classification	1
Bits 0-3: speed_class	
Bits 4-7: road_type	
nod1_offset ???	3
Bitstream	*varies*

The *road_classification* defines a speed class and a road type for the road segment. Both are odd integers in the range 0x1-0xF.

The *speed_class* defines the travel speed for the road and is used to calculate the ETA at the destination. Smaller values represent lower speeds as follows:

Speed class	*Travel speed in mph*
1	5
3	15
5	25
7	35
9	45
11	58
13	67
15	80

The *road_class* identifies the type of road, with lower value representing minor roads such as alleys and residential streets and larger values representing major highways and thoroughfares. The *road_class* is used when calculating "faster time" routes, under the assumption that higher class roads provide the most efficient means of travel. ("Shortest distance" routes use the road segment lengths found in NET, but over long distances will still favor higher class roads.)

Boundary Nodes

This section lists nodes that lie on the "edge" of an IMG file's defined area. These boundary points connect a road in one IMG file to the same road in the adjacent IMG file. Each record is 9 bytes in size.

Data member	*Size (bytes)*
coord_east	3
coord_north	3
offset	3

The east and north coordinates are in Garmin units, given as 24-bit integers. The *offset* member is a pointer of some sort, but to where is not known.