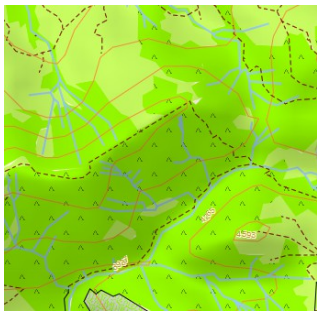
Aufbau und Codierung der Garmin-DEM-Daten



Für den Aufbau eigener Karten sind DEM-Daten nützlich, weil damit z.B. eine „Schummerung“ (Schattierung, hillshading) des Geländes möglich ist. Damit ergibt sich zusätzlich zu den Höhenlinien eine bessere Vorstellung vom Geländeprofil.
Die Garmin-Programme *BaseCamp* und *Mapsource* benötigen DEM-Daten, um ein Höhenprofil eines Tracks darstellen zu können. Die Höhenlinien reichen dafür leider nicht aus.
Es wäre deshalb nützlich, z.B. aus den frei verfügbaren Höhendaten der NASA Garmin-DEM-Daten erzeugen zu können. Einige wenige grundlegende Informationen über die Struktur dieser DEM-Daten ist bekannt (siehe https://wiki.openstreetmap.org/wiki/OSM_Map_On_Garmin/DEM_Subfile_Format).
Für eine weitergehende Untersuchungen wurde zunächst die „TOPO Deutschland v3“ genauer analysiert und anschließend viele „Trial and Error“-Untersuchungen durchgeführt.
Es ist bekannt, dass Garmin-Karten in einzelnen, unterschiedlich großen „Kacheln“ organisiert sind. Zu jeder Kachel gehören die TRE-, LBL-, RGN- usw. Dateien, die zu IMG-Dateien mit einem eigenen Dateisystem zusammengefasst sein können. Analog kann auch eine DEM-Datei zur Kachel gehören. Diese DEM-Datei organisiert die Daten wiederum in kleinen Kacheln.
Für die „Trial and Error“-Untersuchungen benötigt man einen hinreichend großen Datenbereich zum „spielen“. Entweder man sucht sich einen passenden Bereich oder man schafft ihn sich selber. Da ich zunächst einen sehr kleinen Datenbereich zu analysieren versuchte, habe ich diesen später vergrößert.

Aufbau der DEM-Datei

Die Beispieldaten sind aus I0FCC1A3.DEM.
Eine DEM-Datei hat zunächst den typischen Garmin-Header:

Adr.	*Format / Länge*	*Inhalt*	*Beispiel*
0x00	UInt16	Länge des **gesamten** Headers (i.A. 0x29)	29 00
0x02	10	„GARMIN DEM“	47 41 52 4D 49 4E 20 44 45 4D
0x0C	1	unbekannt; immer 0x01 (?)	01
0x0D	1	Sperrflag (0x00 oder 0x80)	00
0x0E	7	Datum (2 Byte Jahr, je 1 Byte Monat, Tag, Stunde, Minute, Sekunde)	D9 07 02 13 0F 12 13

Danach folgt der DEM-spezifische Teil des Headers:

Adr.	*Format / Länge*	*Inhalt*	*Beispiel*
0x15	UInt32	Flags; Bit 0 definiert, ob die Zahlenangaben Meter (0) oder Fuß (1) bezeichnen	01 00 00 00
0x19	UInt16	Anzahl der Zoomlevel	02 00
0x1B	4	unbekannt; i. A. 0	00 00 00 00
0x1F	UInt16	Datensatzgröße für die Zoomlevel (immer 0x3C ?)	3C 00 (= 60)
0x21	UInt32	Pointer auf den 1 Zoomlevel-Datensatz	30 9D 0C 00
0x25	4	unbekannt; auch 0x0	01 00 00 00

Die Datensätze der Zoomlevel stehen direkt hintereinander i. A. am Ende der Datei, im Beispiel ab 0xC9D30
Datensatz für einen Zoomlevel (im Beispiel 1. und 2. Datensatz):

Adr.	*Format / Länge*	*Inhalt*	*Beispiel*
0x00	UInt16	Satznummer (0, 1, …)	00 00 (bzw. 01 00)
0x02	Int32	waagerechte Pixelanzahl je Kachel (i.A. 0x40)	40 00 00 00 (= 64)

0x06	Int32	senkrechte Pixelanzahl je Kachel (i.A. 0x40)	40 00 00 00 (= 64)
0x0A	UInt32	unbekannt;	25 00 00 00
0x0E	UInt32	unbekannt;	23 00 00 00
0x12	UInt16	unbekannt; i.A. 0x00, aber auch 0x100, 0x200, 0x400	00 00
0x14	UInt32	Index der größten Kachel-Spalte (0, 1, ...)	1E 00 00 00 (= 30)
0x18	UInt32	Index der größten Kachel-Zeile (0, 1, ...)	1C 00 00 00 (= 28)
0x1C	UInt16	Struktur der Kacheldatensätze: Bit 0 und 1: Byteanzahl für den Offset (00 → 1 Byte, 01 → 2 Byte, 10 → 3 Byte) Bit 2: Byteanzahl der Basis-Höhe (0 → 1 Byte, 1 → 2 Byte) Bit 3: Byteanzahl der Höhendifferenz (0 → 1 Byte, 1 → 2 Byte) Bit 4: bei 1 ex. eine Extrabyte	1E 00 (= 11110) bzw. 06 00 (= 00110)
0x1E	UInt16	Kacheldatensatzgröße	08 00 (bzw. 06 00)
0x20	UInt32	Pointer auf den 1. Kacheldatensatz	29 00 00 00 (bzw. 0E 0D 04 00)
0x24	UInt32	Pointer auf den gesamten Speicherbereich der Höhendaten	41 1C 00 00 (bzw. 20 22 04 00)
0x28	Int32	westliche Grenze der DEM-Datei in Units (360° / 2^{32})	30 3F 7B 09 (= 13,3332°)
0x2C	Int32	nördliche Grenze der DEM-Datei in Units (360° / 2^{32})	50 32 E7 26 (= 54,7075°)
0x30	Int32	Punktabstand senkrecht in Units (360° / 2^{32})	F0 0C 00 00 (= 0,00028°)
0x34	Int32	Punktabstand waagerecht in Units (360° / 2^{32})	F0 0C 00 00 (= 0,00028°)
0x38	UInt16	min. Basis-Höhe	00 00 (bzw. 34 00)
0x3A	UInt16	max. Höhe (Basis-Höhe + max. Höhendifferenz)	00 02 (bzw. E1 00)

Zur Umrechnung der Units in Grad kann mit 45 multipliziert und mit 2^{29} = 536870912 dividiert bzw. 0,000000083819031715393066406 25 multipliziert werden.
Jeder dieser Datensätze zeigt also auf eine Tabelle von Kacheldatensätzen und auf den Bereich der eigentlichen Höhendaten.

Kacheldatensatz:

Länge	Inhalt
1 … 3	Offset auf die Höhendaten der Kachel (bzgl. des Pointers auf 0x24 des Zoomlevels!); 0, wenn ohne Daten
1 … 2	Basis-Höhe
1 … 2	max. Höhendifferenz zur Basis-Höhe
0 … 1	Höhendifferenz-Flags: größter Wert eingeschlossen (0x00) oder als „undefiniert“ (?) (0x02) z. B. treten bei der Höhendifferenz 3 mit 0x00 die Werte 0, 1, 2, und 3 auf, bei 0x02 nur die Werte 0, 1 und 2 (3 wird nicht angezeigt)

Für die Beispieldatei ergeben sich folgende Datenbereiche:

0x00	Standard-Header
0x15	DEM-Header
0x29	Tabelle der Kacheldatensätze zum 1. Zoomlevel ((0x1E+1) · (0x1C+1) = 0x383 = 31 * 29 = 899 Datensätze je 8 Byte) z. B. 00 00 00 00 00 00 00 xx → ohne Daten BD 0B 00 27 00 1B 00 00 → Daten auf 0xBBD (+ 0x1C41→ 0x27FE), Basis-Höhe 0x27 (= 39), max. Höhendifferenz 0x1B (= 27), also 39 ...66 50 F0 03 00 00 01 00 02 → Daten auf 0x3F050 (+ 0x1C41→ 0x40C91), Basis-Höhe 0x0, max.

	Höhendifferenz 0x1
0x1C41	Höhendaten des 1. Zoomlevels
0x40D0E	Tabelle der Kacheldatensätze zum 2. Zoomlevel ((0x1E+1) · (0x1C+1) = 0x383 = 31 * 29 = 899 Datensätze je 6 Byte) z. B. 00 00 00 E1 00 00 → Daten auf 0x0, 0xE1 (= 225), 0x0 → ohne Daten 7A 00 00 5B 00 86 → Daten auf 0x7A (leer), 0x5B (= 91), 0x86 (= 134)
0x42220	Höhendaten des 2. Zoomlevels
0xC9D30	1. Zoomlevel-Datensatz
0xC9D6C	2. Zoomlevel-Datensatz (0xC9D30 + 0x3C)

Die Höhendaten sind als Bit-Stream organisiert, der eine Zahlenmatrix entsprechend der Kachelgröße, also i.A. 64 x 64 beschreibt. Achtung: Bit 7 ist jeweils das 1. Bit, Bit 0 das letzte Bit eines Bytes. Die 12 Bits des Bit-Streams „1 1 0 0 0 0 0 0 1 1 1 1“ ergeben z. B. Die Bytefolge 0x0C 0xF0.

Testkarte

Bei der „TOPO Deutschland v3“ sind die Kacheldaten maximal 3909 Bytes lang. Die größte maximale Differenz für eine Kachel beträgt 0x1972, die größte Basishöhe 0x2119 (außer 0xFFxx bei Basemap), die größte absolute Höhe 9614 bei Basis 0x1EA3 mit maximaler Differenz 0x06EB und Datenlänge 2329 Byte.
93,5% der Kacheln haben als Höhendifferenz-Flag 0x0, der Rest 0x2.
Es gibt 157810 Tiles. Davon enthalten 146938 Daten. Die Summe der Datenlängen beträgt für den Datenbereich 1 165330982 Byte = 157,7MB.
Für die „TOPO Deutschland v3“ sind in der Datei 16564643\I0FCC1A3.DEM die Höhen in Fuß angegeben (1 ft = 30,48 cm). Bei den Untersuchungen sollte deshalb natürlich auch in *Mapsource* die Anzeige in ft erfolgen. Die einzelnen Kacheln sind im Raster 31 x 29 angegeben, also insgesamt 899 Kacheln. Nicht für jede dieser Kacheln liegen auch Daten vor, da einige vollständig über der Ostsee liegen. Jede Kachel ist 64 x 64 Höhendatenpunkte groß und umfasst etwa den Bereich 0,017767° x 0,017767°. Damit wird ein Bereich von 1,1km x 1,5km abgedeckt, die Punktabstände sind also etwa 17,5m bzw. 23,8m.
Die Satzlänge für die Beschreibung der 899 Kacheln beträgt im 1. Datenbereich 8 Byte im 2. 6 Byte. Die Position des jeweiligen Satzes ergibt sich aus

Block1Start + idx · Block1RecordSize

Die Kachel in der 5. Spalte der 10. Zeile hat die Nummer 9 · 31 + 5 = 284. Wegen der Zählung ab 0 ist es die Nummer 283 oder hexadezimal 0x11B. Der Satz für diese Kachel und den 1. Datenbereich steht deshalb auf Position

0x29 + 8 · 0x11B = 0x901

für den 2. Datenbereich auf

0x40d0e + 6 · 0x11B = 0x413B0

Der Satz für den 1. Datenbereich hat die Bytes 58 B6 00 00 00 03 00 00 (rot markiert). Da die eigentlichen Höhendaten den Offset 0x1c41 haben, ergibt sich für deren Position 0x1c41 + 0xB658 = 0xD299. Die Basishöhe ist 0, die darauf bezogene Maximalhöhe ist 3. Die folgenden 4 Datensätze (blau markiert) zeigen auf den Datenbereich 0. Sie stehen für die Kacheln in der 6., 7., 8. und 9. Spalte der 9. Zeile.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000900	00	58	B6	00	00	00	03	00	00	00	00	00	00	00	00	00
00000910	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000920	00	00	00	00	00	00	00	00	00	64	B6	00	00	00	1A	00
00000930	00	74	B7	00	00	00	AB	00	00	7E	BC	00	37	00	7A	00
00000940	00	B3	C1	00	4D	00	75	00	00	B7	C6	00	73	00	AB	00
00000950	00	E0	CB	00	A1	00	1A	01	00	64	D1	00	21	01	9F	00
00000960	00	F3	D6	00	45	01	80	00	00	B8	DA	00	55	01	87	00

Füllt man z. B. die folgenden 4 Datensätze (grün markiert) auch noch mit 0, kann deren Datenbereich für Testzwecke mit verwendet werden. Die Daten des 1. Datensatzes nach den „grünen“ beginnen z. B. bei 0xC6B7 + 0x1c41 = 0xE2F8. Damit hätte man also einen Datenbereich von 0xD299 bis 0xE2F8 (0x105F = 4191 Byte) zur freien Verfügung.

Hinweise zum Untersuchungsverfahren

Da es um sehr viele Versuche geht, muss man die Untersuchung soweit wie möglich automatisieren. Das betrifft auf jeden Fall die Erzeugung der Testdaten und das Ermitteln der Auswirkung dieser Daten. Für das „Lesen“ der Höhendaten wurde *Mapsource* verwendet. Damit ist naheliegend, dass die gesamte Arbeit zweckmäßigerweise unter Windows erfolgt. Einzelne Schritte werden in Kommandodateien (*.CMD) zusammengefasst.

Es wurden einige „maßgeschneiderte“ Hilfsprogramme in C# geschrieben. Damit ist jederzeit eine leichte Anpassung an neue Erkenntnisse möglich. Außerdem wird das Programm *gmt* (Kommandozeilenversion des GmapTool) verwendet, um die Testkarte aus den Einzeldateien zu erzeugen. Mit dem aus der Unix-Welt bekannten sed werden einfache Filteraufgaben erledigt.
I.W. hat man folgenden Ablauf:
Mit dem C#-Programm *Input1* (ja, der Name ist nicht sehr einfallsreich) wird die zu testende Bitfolge in die DEM-Datei geschrieben. Danach wird eine Kommandodatei aufgerufen.

Beispiel:

```
REM funktioniert nur, wenn Mapsource nicht läuft
del 16564643.img
gmt -j -x -o 16564643.img 16564643\I0FCC1A3.tre 16564643\I0FCC1A3.rgn 16564643\I0FCC1A3.net
                          16564643\I0FCC1A3.lbl 16564643\I0FCC1A3.dem

del /Q "%APPDATA%\GARMIN\MapSource\TileCache\*.*"

set MAUSPOS=86 166 1701 166
```

```
SimpleProgControl %MAUSPOS% 0 0 64 1 7 tmp 1500 "" "%ProgramFiles(x86)%/Garmin/MapSource.exe"

REM gewünschte Zeilennummern herausfiltern und neu formatieren
sed -n "3,3p" < tmp | sed -e "s/\([0-9]\+\)\t/ : \1 : /" -e s/\t/,/g >> protokoll1.txt
del tmp
```

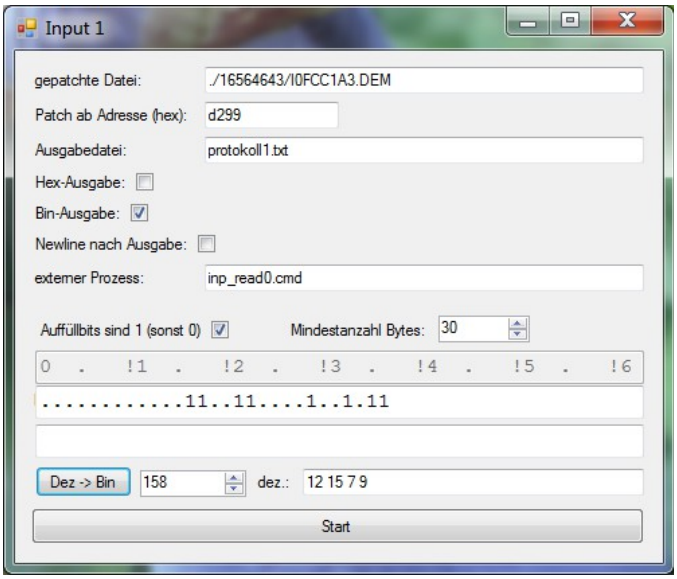
Die alte 16564643.img wird gelöscht. Mit gmt werden die originalen TRE-, RGN-, NET-, LBL- sowie die gepatchten DEM-Daten zusammengefügt. Der Tile-Kache von *Mapsource* sollte sicherheitshalber immer gelöscht werden. Mit dem C#-Programm *SimpleProgControl* wird *Mapsource* gestartet und die Daten werden in die Datei tmp geschrieben. Mit *sed* kann noch eine Filterung erfolgen. Das Ergebnis wird an die Datei protokoll1.txt angehängt.
Das Auslesen der Daten mit *Mapsource* ist etwas schwierig. Es gibt leider nur einen Bereich in der Statuszeile, in der die Höhe des Punktes auf den die Maus zeigt, angezeigt wird.

> Mit spyxx.exe aus „Microsoft Visual Studio“ kann man aber leicht die GUID {7A96B96B-E756-4e42-8274-54CBF24F7944} und den notwendigen Klassennamen „msctls_statusbar32“ ermitteln, um den Text auszulesen. Der Text, z.B. „N54.53305 E13.41344, 17 ft“ gibt dann die Höhe mit einem entsprechenden regulären Ausdruck preis. Nun muss nur noch die Maus schrittweise auf die gewünschten Positionen gesetzt werden.

Problematisch ist, dass immer genau der gewünschte Bereich angezeigt werden muss.
SimpleProgControl startet Mapsource. Man kann eine Tastenfolge zur Initialisierung mitschicken z.B. „%AK77777{ENTER}%AzN54.53305 E13.41344{ENTER}“. Damit wird über Alt+A, K der Zoom auf 70m eingestellt und Alt+A, z die Position der Karte festgelegt. Das das Programm im Vollbildmodus startet und wie z.B. die Symbolleisten angeordnet sind muss man aber „per Hand“ vorher festlegen.
Mit dem C#-Programm *ShowMousePos* muss man außerdem vorher die Lesepositionen in Bildschirmkoordinaten ermittelt haben. In der Beispiel-Kommandodatei hat die linke obere Mausposition die Koordinaten [86, 166] und die rechte untere [1701, 166]. Aus den folgenden Angaben geht hervor, das die Höhenpunkte von Spalte 0 und Zeile 0 bis Spalte 64 und Zeile 1 zu diesen Mauskoordinaten passen und ausgelesen werden sollen.
Für eine möglichst hohe Genauigkeit des Auslesens sollte *Mapsource* im Vollbildmodus betrieben werden. Für die Kartenanzeige sollte soviel Platz wie möglich vorhanden sein. Die Bildschirmauflösung sollte möglichst hoch sein.

> Bei den Daten der Beispiel-Kommandodatei liegen in O-W-Richtung etwa 25,651 Pixel Abstand zwischen 2 Punkten. Die „Messung“ kann jedoch nur für ganzzahlige Pixelabstände erfolgen. Bei korrekter Rundung liegt der 2. Messpunkt also bei Pixel 26 und damit etwa beim 1,039-fachen der eigentlich gewünschten Entfernung.
> Prinzipiell beträgt der Positions-Fehler ±0,5 Pixel, also etwa ±2%. Bei starken Höhenunterschieden zwischen 2 Messpunkten muss dieser Fehler einkalkuliert werden. Z.B. bei einer Änderung von 50 auf 150 kann der 2. "Messwert" im Bereich 148..152 liegen.

Mit dem Programm *Input1* wird i.W. die DEM-Datei mit einer Bitfolge gepatched, d.h. es werden neue Testdaten eingetragen. Nach dem patchen wird ein weiterer Prozess gestartet, der für das Auslesen der Daten zuständig sein sollte.
Da die Bitfolge i.A. nicht vollständig zum Ausfüllen des letzten Bytes ausreicht, muss die Art der Auffüllbits angegeben werden. Außerdem wird immer eine Mindestanzahl an Bytes geschrieben. Es hat sich bewährt, als Auffüllbit 1 zu verwenden. Die Zusatzbytes sind also 0xFF.
Wegen der besseren Unterscheidbarkeit werden 0-Bits als „.“ geschrieben.
Im Programm ist ein Encoder enthalten, der das aktuelle Wissen über die Codier-Regeln enthält. Deshalb kann man auch direkt Höhenwerte als Dezimalwerte eintragen und codieren lassen. Wichtig für die Codierung ist die Angabe der maximalen Höhendifferenz.



Für weitere Versuche gibt es noch das C#-Programm *SimplePatcher*, mit dem das patchen auf der Kommandozeile erfolgen kann. Mit *Bin2Hex* kann die Umwandlung zwischen Hexadezimalzahlen und Bitfolgen erfolgen.

Details der Codierung

Komprimierung

Mit einer Speicherung der Höhendaten als 2-Byte-Zahl (16 Bit, also Zahlenbereich 0 ... 65536) könnte man alle Höhen auf der Erde auch in Fuß speichern. Selbst 15 Bit würden ausreichen: 32768 / 3,28084 = 9987m. Bei 64 x 64 Punkten je Kachel wären das $64 \cdot 64 \cdot 16 = 2^{16}$ Bit = 2^{13} Byte = 2^3 kByte = 8 kB (bzw. 4 kB bei 15-Bit-Zahlen) je Kachel.
Die „TOPO Deutschland v3“ hat 157810 Kacheln. Davon enthalten 146938 Daten. Die Summe der Datenlängen beträgt für den Datenbereich 1 165330982 Byte = 157,7 MB. Würde man die oben beschriebene 15-Bit-Speicherung verwenden käme man auf 574 MB. Garmin gelingt also eine Kompression auf etwa 28%.

> Es ist allerdings fraglich, ob es nicht insgesamt effektiver wäre, wenn bei dem heute zur Verfügung stehenden Speichervermögen der SD-Karten eine nur geringe aber einfache Komprimierung verwendet wird, gleichzeitig dadurch aber die Decodierung einfacher und damit die Anzeige schneller wird.

Die anfängliche Vermutung, dass Garmin eines der üblichen und bekannten Kompressionsverfahren verwendet, also Huffman, RLE, Arithmetische Kodierung, LZW usw. usf., hat sich leider nicht bestätigt.

> Liegt das daran, dass diese Verfahren nicht stark genug komprimieren oder wollte man lieber ein „geheimes“ Verfahren?

Das verwendete Verfahren ist auf jeden Fall sehr effektiv. Eigentlich sind es mehrere Verfahren. Eines ist besser für größere Zahlen geeignet, ein anderes besser für kleinere. Das Codierverfahren ändert sich nach bestimmten Regeln. Dabei werden jedoch nur die bereits codierten Höhen einbezogen, d.h. es findet keine „Vorausschau“ statt.

> Es wäre vermutlich noch effektiver, wenn es optimal auf **alle** aktuell vorhandenen Daten zugeschnitten wäre. Da der Decoder die „zukünftigen“ Daten aber noch nicht kennen kann, hätte man zusätzliche Informationen in die codierten Daten aufnehmen müssen, wenn das aktuelle Codierverfahren auf ein anderes „umgeschaltet“ wird. Diesen Speicherplatz hat man sich gespart. Das resultierende Verfahren dürfte deshalb weniger effektiv sein, spart aber die „Umschaltinformationen“ ein.

Encoder und Decoder müssen natürlich beide nach den gleichen Regeln arbeiten. Um einen Encoder zu schreiben, muss man genau diese Regeln kennen.

Datenberechnung

Eine Kachel (64 x 64) enthält einfach 4096 Höhenwerte. Deshalb hat man zunächst je Kachel eine Zahlen-Matrix von 64 x 64 Zahlen, die die Höhen punktuell angeben.
Diese Höhen werden aber nicht direkt gespeichert. Wenn man möglichst kleine Zahlen mit genau angepassten Codierverfahren speichert, kann man sehr viel Speicherplatz sparen. „Kleine Zahlen“ bedeutet eigentlich „wenige Bits zur Speicherung der Zahlen“, aber eine Voraussetzung ist trotzdem, dass die Zahlen selbst möglich klein sind.
Deshalb werden die realen Höhen zunächst auf eine Basis-Höhe bezogen (gespeichert im Kacheldatensatz). Diese Basis-Höhe ist einfach die kleinste vorhandene Höhe. In Beispiel sei das die Höhe 90. Dann ergibt sich eine neue, normierte Zahlen-Matrix.
Aber auch diese Werte werden nicht gespeichert. Es werden aus diesen Werten normalerweise noch gewisse Differenzen gebildet. In einigen Fällen werden nach bestimmten Regeln auch Bereiche konstanter Höhe verkürzt gespeichert mit der Länge des Bereiches und der nachfolgenden Höhe gespeichert. In der endgültigen Zahlen-Matrix erkennt man z. B., dass in der 1. Zeile einfach nur die Differenzen zum Vorgänger enthalten sind.

95	110	115	110	107	…
110	112	116	119	111	…
115	117	115	113	119	…
120	122	118	116	113	…
125	124	121	119	111	…
…	…	…	…	…	…

Originalhöhen

5	20	25	20	17	…
20	22	26	29	21	…
25	27	25	23	29	…
30	32	28	26	23	…
35	34	31	29	21	…
…	…	…	…	…	…

normierte Höhen

5	15	5	-5	-3	…
15	2	1	8	-5	…
5	0	-6	5	14	…
5	0	-2	0	9	…
5	-3	1	0	-5	…
…	…	…	…	…	…

zu speichernde Datenwerte

Die Höhenwerte werden im Weiteren durch die Angabe der Spalten- und Zeilennummer, beide 0-basiert, angegeben. $h(0, 0)$ ist also die Höhe in der linken oberen Ecke, $h(1, 0)$ die Höhe rechts daneben und $h(0, 1)$ die Höhe darunter in der nächsten Zeile. Analog werden die dazu gehörenden Datenwerte mit $d(\text{col}, \text{line})$ bezeichnet.

Außerdem fügen wir noch eine virtuelle Zeile über der Zeile 0 und eine virtuelle Spalte vor der Spalte 0 ein. Die virtuelle Zeile soll nur die Höhen 0 enthalten, also $h(i, -1) = 0$. Die virtuelle Spalte soll die gleichen Werte wie die erste Spalte, aber um 1 Zeile nach unten verschoben, enthalten, also $h(-1, i) = h(0, i-1)$.
An einigen Stellen werden Differenzen zweier benachbarter Werte verwendet. Es gilt dann:
für die „horizontale“ Differenz $hdiff(n,m) = \boldsymbol{h(n,m)} - h(n-1,m)$,
für die „vertikale“ Differenz $vdiff(n,m) = \boldsymbol{h(n,m)} - h(n,m-1)$
und für die „diagonale“ Differenz $ddiff(n,m) = h(n,m-1) - h(n-1,m)$ (immer „aufwärts diagonal“).

	h(n-1, m-1)	h(n, m-1)
	h(n-1, m)	**h(n, m)**

	0	**1**	**2**	**3**	**4**	
	0	0	0	0	0	…
0	0	5	15	5	-5	-3	…
1	5	15	2	1	8	-5	…
2	15	5	0	-6	5	14	…
3	5	5	0	-2	0	9	…
4	5	5	-3	1	0	-5	…
	…	…	…	…	…	…	…

Matrix mit virtueller Spalte und Zeile

Standardwerte

Für Standardwerte muss gelten: $ddiff(n, m) \neq 0$, also Vorgängerhöhe ≠ Höhe „darüber“. Andernfalls beginnt ein „Plateau“ (s.u.).

> Daraus folgt übrigens, dass die 1. Spalte niemals einen Standardwert enthalten kann und 0-Höhen in der 1. Zeile auch keine Standardwerte sind.

Für Standardwerte gilt:

$$d(n,m) = -sgn(ddiff(n,m)) \cdot (hdiff(n,m) - hdiff(n,m-1))$$

Gilt zusätzlich $h(n\text{-}1,m) < -hdiff(n, m\text{-}1)$, gilt:

$$d(n,m) = -sgn(ddiff(n,m)) \cdot h(n,m)$$

> Das ist tatsächlich effektiver. Mit der 1. Gleichung wäre $d(n, m)$ auf keinen Fall kleiner:
>
> $$|d(n,m)| = |hdiff(n,m) - hdiff(n,m-1)|$$
> $$|d(n,m)| = |h(n,m) - h(n-1,m) - hdiff(n,m-1)|$$
> $$|d(n,m)| = |h(n,m) - (-D - hdiff(n,m)) - hdiff(n,m-1)| \quad \text{und } D > 0$$
> $$|d(n,m)| = |h(n,m) + D|$$
> $$|d(n,m)| \geq |h(n,m)|$$
>
> Hinweis:
> Für die 1. Zeile gilt wegen der virtuellen Zeile vereinfacht:
>
> $$d(n,0) = -sgn(ddiff(n,0)) \cdot (hdiff(n,0) - 0)$$
> $$d(n,0) = hdiff(n,0)$$

Plateaus

Ein Plateau besteht aus mehreren nebeneinanderliegenden konstanten Höhenwerten. Es liegt auf der Hand, dass in einem solchen Fall nicht jede Höhe einzeln angegeben werden muss. Es genügt, wenn die Länge eines Plateaus angegeben wird. Leider kann nicht willkürlich ein Plateau „gestartet“ werden, sondern für d_p muss die spezielle Bedingung $ddiff = 0$ gelten. Daraus folgt auch, dass immer entweder ein Standardwert oder ein Plateau vorliegt.
Wenn $hdiff = 0$ und $vdiff = 0$, also h_p mit seinem Vorgänger und der Höhe „darüber“ identisch ist, ist die aktuelle Position der Startpunkt eines echten Plateaus mit der aktuellen Höhe (des Vorgängers) und mindestens der Länge 1. Andernfalls handelt es sich um ein unechtes Plateau mit der Länge 0. Dieser Fall kommt vermutlich selten genug vor, so dass sich dieses Verfahren zum Einsparen von Speicherplatz trotzdem lohnt. Auf jeden Fall wird statt eines Datenwertes d_i (für h_i) nur die Länge des Plateaus gespeichert, dass mit h_i beginnt.

> Im folgenden Beispiel beginnt nach der 4 ein Plateau. Dessen Länge ist 4 (nicht 5!). der Nachfolgewert ist 5.

…	…	…	…	…	…	…	…	…	…
…	…	…	…	4	…	…	…	…	…
…	2	3	4	4	4	4	4	5	…
…	…	…	…	…	…	…	…	…	…

Achtung: Plateaus können auch über mehrere Zeilen reichen.

Da in der 1. Spalte wegen der virtuellen Spalte immer *ddiff* = 0 gilt, beginnt jede Zeile mit einem Plateau, wenn sich dort nicht schon ein „überlanges“ Plateau aus der vorherigen Zeile befindet.
In der 1. Zeile beginnt wegen der virtuellen Zeile auch nach jeder Höhe 0 ein Plateau.

<u>Bestimmung des Nachfolgewertes</u>
Eine Besonderheit der Plateaucodierung ist die damit verbundene spezielle Bestimmung des 1. nach dem Plateau folgenden Wertes $h(f, m)$. Dafür spielt die Differenz *ddiff* des Nachfolgewertes eine wichtige Rolle.

Für $ddiff(n, m) \neq 0$ gilt: $\boxed{d(n,m) = sgn(ddiff(n,m)) \cdot vdiff(n,m)}$

Wenn *ddiff* und *vdiff* das gleiche Vorzeichen haben ist *d* positiv, sonst negativ bzw. 0 bei *vdiff* = 0.

Andernfalls gilt: $\boxed{d(n,m) = -\frac{sgn(vdiff(n,m))-1}{2} + vdiff(n,m)}$

Das ist eigentlich die Bedingung für den Start eines neuen Plateaus. Natürlich muss dann v*diff*$(n, m) \neq 0$ gelten, da sonst die Plateaulänge einfach zu kurz wäre.

Berechnung z.B.:

```
if 0 < ddiff
        d(f, m) = vdiff
else if ddiff = 0
        if 0 <  vdiff
                d(f, m) = vdiff
        else if vdiff = 0
                FEHLER (Plateaulänge sollte größer sein)
        else (vdiff < 0)
                d(f, m) = vdiff +1
else (ddiff < 0)
        d(f, m) = -vdiff
```

Zahlencodierungen

Nachdem nun klar ist, welche Werte zu speichern sind, kommen wir zur eigentlichen Codierung. Es gibt verschiedene Verfahren deren Anwendung allerdings nicht frei gewählt werden kann. Sie unterscheiden sich in ihrer Effektivität bei der Speicherung kleiner oder großer Zahlen.
Dadurch entsteht ein Bitstream der als Byte-Folge, immer beginnend mit dem höchstwertigen Bit, gespeichert wird.

Längencodierung

Die Länge einer 0-Bitfolge, abgeschlossen durch ein 1-Bit (wer hätte das erwartet) gibt die Zahl an. Die Länge der 0-Bitfolge für Daten im Bereich -5 bis +5 stehen in der folgende Tabelle.
Wie man sieht, gibt es 2 verschiedene Längencodierungen L0 und L1. Man kann sich aber auch leicht weitere Codierungen „ausdenken“. Nach L0 steht die z. B. Bitfolge „`...1 ..1 1 1 1 ....1`“ für die Datenwerte +2, -1, 0, 0, 0 und -2. L0 ist also offensichtlich für konstante Höhen besonders effektiv. Das trifft z.B. für einen See zu. L1 scheint eher für leicht ansteigendes Gelände geeignet.

In Integer-Arithmetik ergibt sich der Datenwert *d* aus der 0-Bitfolgenlänge *l* für L0

$$d = (l \bmod 2)\cdot(l+1)/2 - ((l+1) \bmod 2)\cdot(l/2)$$

und für L1

$$d = -((l \bmod 2)\cdot(l+1)/2 - ((l+1) \bmod 2)\cdot(l/2) - 1)$$

Die Bitlänge *l* der 0-Bitfolge zum Datenwert *d* für L0 ist

$$l = 2\cdot|d| - (sgn(d)+1)/2$$

und für L1

$$l = 2\cdot|d-1| + (sgn(d-1)-1)/2$$

Die Umrechnung der Datenwerte beider Längencodierungen ineinander ist übrigens einfach $L0 + L1 = 1$.
Bei geradzahliger 0-Bitanzahl für L0 kommt 1 Bit beim Übergang zu L1 dazu, bei ungeradzahliger Anzahl fällt eins weg.
Da die zulässige Länge der 0-Bit-Folge in Abhängigkeit von der maximalen Höhendifferenz der Kachel begrenzt ist (siehe Tabelle) ergibt sich mit der maximalen Anzahl 0-Bits *b0* der gültige Zahlenbereich für L0
$-b0/2 \le d \le (b0+1)/2$ und für L1
$-(b0+1)/2+1 \le d \le b0/2+1$.

Wert	0-Bits	
	L0	L1
...		
+5	9	8
+4	7	6
+3	5	4
+2	3	2
+1	1	0
0	0	1
-1	2	3
-2	4	5
-3	6	7
-4	8	9
-5	10	11
...		

Wert	L0	L1	H1	H2	H4	H8	H16
20	-	-	21	12	8	7	7
19	-	-	20	12	8	7	7
18	-	-	19	11	8	7	7
17	-	-	18	11	8	7	7
16	-	-	17	10	7	6	6
15	-	-	16	10	7	6	6
14	-	-	15	9	7	6	6
13	-	-	14	9	7	6	6
12	-	23	13	8	6	6	6
11	22	21	12	8	6	6	6
10	20	19	11	7	6	6	6
9	18	17	10	7	6	6	6
8	16	15	9	6	5	5	6
7	14	13	8	6	5	5	6
6	12	11	7	5	5	5	6
5	10	9	6	5	5	5	6
4	8	7	5	4	4	5	6
3	6	5	4	4	4	5	6
2	4	3	3	3	4	5	6
1	2	1	2	3	4	5	6
0	1	2	2	3	4	5	6
-1	3	4	3	3	4	5	6
-2	5	6	4	4	4	5	6
-3	7	8	5	4	4	5	6
-4	9	10	6	5	5	5	6
-5	11	12	7	5	5	5	6
-6	13	14	8	6	5	5	6
-7	15	16	9	6	5	5	6
-8	17	18	10	7	6	6	6
-9	19	20	11	7	6	6	6
-10	21	22	12	8	6	6	6
-11	23	-	13	8	6	6	6
-12	-	-	14	9	7	6	6
-13	-	-	15	9	7	6	6
-14	-	-	16	10	7	6	6
-15	-	-	17	10	7	6	6
-16	-	-	18	11	8	7	7
-17	-	-	19	11	8	7	7
-18	-	-	20	12	8	7	7
-19	-	-	21	12	8	7	7
-20	-	-	22	13	9	7	7

Bitanzahl je Wert und Codierungsart

Hybride Codierung

Das ist eine Kombination einer Längencodierung und einer Binärcodierung.
Die Längencodierung verwendet eine spezielle Einheit („Heightunit“) *hunit*. Diese ist immer eine 2er Potenz 2^i und wird nach bestimmten Regeln automatisch angepasst. *hunit* ist auch die größte Zahl, die mit dem Binärteil dargestellt werden kann.
Um große Zahlen darzustellen, ist die Längencodierung allein ungünstig. Deshalb wird *hunit* dann vergrößert. Allerdings können dann die „Zwischenwerte“ nicht mehr dargestellt werden. Dafür ist aber jetzt der Binärteil zuständig.

Die Bitfolge $0_{s-1} \ldots 0_0 1 \, b_{ld(hunit)-1} \ldots b_0 \, v$

steht für den Wert: $d = (2 \cdot v - 1) \cdot \left(s \cdot hunit + \left(\sum_{i=0}^{ld(hunit)-1} b_i \cdot 2^i \right) + v \right)$

Dabei ist *v* das Vorzeichenbit (1 für positive, 0 für negative Werte), s die Länge der 0-Bitfolge und b_i sind die Binärbits.

Beispiele:
hunit=4, „`..11.1`“
3x0-Bits Längencodierung → 3 * 4
1x1-Bit Trennzeichen
1x1-Bit für 2^1
1x0-Bit für 2^0
1x1-Bit Vorzeichen→ positiv
→ (2 * 1 - 1) * (3 * 4 + 1 * 2^1 + 0 * 2^0 + 1) = 14

hunit=1, „`11`“
0x0-Bits Längencodierung → 0
1x1-Bit Trennzeichen
1x1-Bit Vorzeichen→ positiv
→ (2 * 1 - 1) * (0 * 1 + 1) = 1

hunit=1, „`.1.`“
1x0-Bits Längencodierung → 1 * 1
1x1-Bit Trennzeichen
1x0-Bit Vorzeichen → negativ
→ (2 * 0 - 1) * (1 * 1 + 0) = -1

hunit=1, „`1.`“
0x0-Bits Längencodierung → 0
1x1-Bit Trennzeichen
1x0-Bit Vorzeichen → negativ
→ (2 * 0 - 1) * (0 * 1 + 0) = 0

Bis auf 2 Sonderfälle für die Werte 0 und 1 ist die Hybridcodierung immer besser als die einfache L0- bzw. L1-Längencodierung, d.h. es werden weniger Bits für die Codierung benötigt. Vermutlich treten aber gerade diese Sonderfälle in der Praxis häufig genug auf, so dass sich die einfache Längencodierung eben doch lohnt.

> Warum wird aber eine Hybridcodierung statt einer reinen Binärcodierung verwendet? Die darin enthaltene Längencodierung ist ja vergleichsweise uneffektiv.
> Der Wertebereich bei der Binärcodierung ist immer begrenzt. Z.B. können mit 3 Bit grundsätzlich nur 8 verschiedene Werte codiert werden, mit 8 Bit 256 Werte. Um sicher zu gehen, dass auch eine sehr hohe, steile Klippe codiert werden kann, müssten vermutlich Werte bis ±1000 (Fuß) verfügbar sein. In Binärcodierung müßten für jeden Wert also 11 Bit (±1024) verwendet werden. Am weitaus häufigsten kommen jedoch sehr kleine Werte vor. Die meisten Bits der Binärcodierung wären also verschwendet. Selbst bei *hunit* = 1 können in Hybridcodierung mit 11 Bit schon Werte von -11 bis +12 codiert werden. Bei kleineren Werten ist die Hybridcodierung dann sogar effektiver als die Binärcodierung. Würde man eine variable Bitanzahl für die Binärcodierung verwenden, müsste vor jedem Binärwert noch eine Festlegung der Bitanzahl erfolgen. Dafür wären aber auch 4 Bit (1 bis 16) nötig. Insgesamt wären dann je Wert 4+2=6 bis 4+11=15 Bit nötig. Mit 8 Bit erreicht man z.B. einen Wertebereich von 2^4=16, bei Hybridcodierung 12, bei 7 Bit 2^3=8 bzw. 10. Bei Werten im Bereich ±5 ist die Hybridcodierung also auch effektiver.
> → Bei häufig relativ kleinen Werten ist die Hybridcodierung effektiver als die Binärcodierung.

Für sehr große Zahlen ist die Hybridcodierung natürlich trotzdem nicht sehr gut geeignet. Bei einem kleinen *hunit* wird die 0-Bit-Folge dann sehr lang. Deshalb ist die zulässige Länge der 0-Bit-Folge in Abhängigkeit von der maximalen Höhendifferenz der Kachel begrenzt (siehe Tabelle).
Mit der maximalen Anzahl 0-Bits *b0* ergibt sich der gültige Zahlenbereich

$$-(b0+1) \cdot hunit + 1 \;\leq\; d \;\leq\; (b0+1) \cdot hunit$$

Spezielle Binärcodierung für große Zahlen

Benötigt man in sehr seltenen Fällen größere Zahlen, als sich mit den 0-Bitfolgen in der Längen- oder in der Hybridcodierung erzeugen lassen, muss eine spezielle Binärcodierung verwenden werden. Sie kann direkt hinter einer Längen- oder einer Hybridcodierung erfolgen. Dafür wird zunächst eine 0-Bit-Folge verwendet, die den erlaubten Längenbereich (siehe Tabelle) um mindestens 1 Bit überschreitet. Danach folgt ein Trennbit ('1') und danach eine Reihe von Binärbits, deren Anzahl ebenfalls von der maximalen Höhendifferenz abhängig ist (siehe Tabelle). Die „Längenbits“ stellen jetzt nur noch eine Kennung für diesen Sonderfall dar. Der Zahlenwert ist ausschließlich durch die Binärbitfolge $b_n \ldots b_0\, b_v$ angegeben. Wäre jetzt ein Wert in Hybridcodierung oder Längencodierung L0 an der Reihe, ist die Codierung $d_{H,L0} = -(2 \cdot b_v - 1) \cdot \left(1 + \sum_{i=0}^{n} b_i \cdot 2^i\right)$. Durch b_v ist hier tatsächlich nur das Vorzeichen definiert, 1 für negative und 0 für positive Werte. Der Wert 0 kann nicht angegeben werden.

Wäre jetzt ein Wert in Längencodierung L1 an der Reihe, ist die Codierung $d_{L1} = (2 \cdot b_v - 1) \cdot \left(1 + \sum_{i=0}^{n} b_i \cdot 2^i\right) + 1$. Das Vorzeichen 1 steht jetzt für positive, 0 für negative Werte. Der Wert 1 kann nicht angegeben werde. Dazu wäre aber natürlich sowieso die normale Codierung ausreichend.

Um die gleiche Codierungsfunktion für beide Varianten verwenden zu können, könnte man z.B. einen d_{L1}-Wert zunächst in $d_{L1'}$ konvertieren und dann die Funktion für $d_{H,L0}$ verwenden:

$d_{H,L0} = 1 - d_{L1} \quad \rightarrow \quad d_{L1'} = 1 - d_{L1}$

Die Anzahl der Binärbits (noch ohne Vorzeichen !) ist jeweils so groß, dass, notfalls mit „wraparound“, der gesamte Zahlenbereich bis zur max. Höhendifferenz abgedeckt wird. Sie ergibt sich aus der maximalen Höhendifferenz mit $b = integer(ld(max))$. Der Wertebereich (außer 0) ergibt sich aus $-2^{integer(ld(max))} \le d \le 2^{integer(ld(max))}$.

maximale Höhendifferenz	maximale Anzahl 0-Bits	Binär-bits	Wertebereich (außer 0)
8 .. 15	18	4	-8 .. +8
16 .. 31	19	5	-16 .. +16
32 .. 63	20	6	-32 .. +32
64 .. 127	21	7	-64 .. +64
128 .. 255	22	8	-128 .. +128
256 .. 511	23	9	-256 .. +256
512 .. 1023	28	10	-512 .. +512
1024 .. 2047	31	11	-1024 .. +1024
2048 .. 4095	34	12	-2048 .. +2048
4096 .. 8191	37	13	-4096 .. +4096
8192 .. 16383	40	14	-8192 .. +8192
16384 .. 32767	43	15	-16384 .. +16384

Für eine Maximalhöhe 158 ergeben sich $integer(ld(158)) = integer(7,3) = 7$ Bits. Der Wertebereich ist also $-2^7 \le d \le 2^7$, also $-128 \le d \le 128$

Bei einer Maximalhöhe 158 erfolgten Versuche mit 23, 24 und 25 0-Bits. Die zusätzlichen 1 bzw. 2 0-Bits hatten keinen erkennbaren Einfluss. Nach dem 1-Bit folgte immer die Binärbitfolge mit insgesamt 8 Bit. Insofern ist klar, dass das 1-Bit als Endekennung der 0-Bitfolge nötig ist. Unklar ist jedoch, welche Auswirkungen die scheinbar unnötigen zusätzlichen 0-Bits haben. Es ist kaum zu erwarten, dass sich GARMIN hier nicht etwas gedacht hat.

Vermutlich gibt es auch für die kleineren Höhendifferenzen diese Codier-Möglichkeit. Sie ist aber unnötig, weil dann die jeweils maximale 0-Bit-Anzahl schon zum Erreichen jedes Wertes ausreicht. Selbst für die Höhendifferenz 31 reicht die 0-Bit-Anzahl aus, wenn man den „wraparound“ berücksichtigt.

Bei der maximale Höhendifferenz von 1000 dürfen maximal 28 0-Bits für die normale Hybridcodierung verwendet werden. Werden jedoch 29 0-Bits gefolgt von einem 1-Bit verwendet, folgen danach immer 10 Binärbits. Davon ist das letzte Bit ein Vorzeichenbit, 1 für positiv, 0 für negative Werte. Damit kann ein Wertebereich von -512 bis +512 (außer 0) codiert werden, d.h. es kann jede beliebige Zielhöhe erreicht werden. Die insgesamt 29+1+10=40 Bit sind zwar ausgesprochen viel, aber diese Fälle treten vermutlich sehr selten auf. In diesem Beispiel müsste die *hunit* kleiner als 32 sein, damit diese Codierung überhaupt nötig wird.

Für Plateau-Nachfolger wird 1 0-Bit weniger in der Kennung verwendet als für Standardwerte. (?)

Codierung der Plateaulänge

Wie schon in der Überschrift gesagt, wird diese Codierung ausschließlich für die Längenangabe eines Plateaus verwendet.

Die Längenangabe besteht aus einer Folge von 1-Bits, die durch ein 0-Bit abgeschlossen und eventuell von Binärbits gefolgt wird: $1_s \ldots 1_0\, 0\, b_k \ldots b_0$. Während die Binärbits wie üblich den Wert $\sum_{i=0}^{k} b_i \cdot 2^i$ ergeben, haben die 1-Bits, abhängig von ihrer Position, unterschiedliche Werte. Diese Werte und die Anzahl der Binärbits ergeben sich aus folgender Tabelle:

Pos.	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***	***19***	***20***	***21***	***22***	***23***
Bitwert	1	1	1	1	2	2	2	2	4	4	4	4	8	8	8	8	16	16	16	32	32	32	64	64
Binärbits	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	5	5	6	6	6	7

Bitwert	1	1	1	1	2	2	2	2	4	4	4	4	8	8	8	8	8	16	16	32	32	32	64	64
Binärbits	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3	3	4	4	5	5	6	6	6	7

Die Plateaulänge ergibt sich aus der der Summe der 1-Bit-Werte und dem Binärwert. Es kann damit jeder beliebige Wert codiert werden.

Beispiele:
Tabellenposition = 0 (1. Plateaulänge)

`.`	0 + 0 = 0
`11.`	2 * 1 = 2
`1111..`	4 * 1 + 0 = 4
`11111.1`	4 * 1 + 1 * 2 + 1 = 7
`1111111111111.1.1`	4 * 1 + 4 * 2 + 4 * 4 + 1 * 8 + 5 = 41
`1111111..`	4 * 1 + 3 * 2 + 0 = 10

Die Codierung erfolgt immer soweit wie möglich mit den 1-Bits. Erst wenn die Länge damit nicht codiert werden kann, werden die Binärbits entsprechend gesetzt.

Eine Plateaulänge 1 die mit der Codierung an Tabellenposition 3 startet, wird nicht als „`.1`“ codiert, sondern als „`1..`“.

Die Anzahl der Binärbits vergrößert sich immer um 1, wenn das nächste 1-Bit einen höheren Wert als das bisherige hat.

Da die Verwendung der 1-Bits Vorrang vor den Binärbits hat, gibt es ein Problem z. B. beim Codieren der Länge 5. 4x 1-Bit ergibt 4, 5x 1-Bit jedoch schon 6. Deshalb muss beim 4. 1-Bit schon 1 Binärbit vorhanden sein. Analog gilt das für die Positionen 7, 11, 15 usw..

Bevor die nächste Plateaulänge codiert wird, wird die Startposition in der Tabelle zunächst dekrementiert (aber nicht kleiner als 0). Die Anzahl der Binärbits ergibt sich vorläufig aus dieser Position. Für jedes verwendete 1-Bit in der 1-Bit-Folge wird die Position inkrementiert (aber nicht größer als 23). Die Bedeutung der 1-Bits ergibt sich aus ihrer Position. Die Anzahl der tatsächlich zu verwendenden Binärbits ergibt sich aus der letzten Position.
Hat das letzte 1-Bit eines Plateaus die Position n in der Tabelle, geht man für das folgende Plateau von der Position $n - 1$ aus. Hat dieses folgende Plateau k 1-Bits, ergibt sich die Anzahl der Binärbits aus der Position $n - 1 + k$. Daraus ergibt sich, dass bei jeder Plateaulänge 0 die Position effektiv um 1 geringer wird.

Beispiele:
Tabellenposition = 0 (1. Plateaulänge)

`1111111..`	$4 \cdot 1 + 3 \cdot 2 + 0 = 10$

dann 2. Plateaulänge mit Tabellenposition = 6

`1111..1`	$2 \cdot 2 + 2 \cdot 4 + 1 = 13$

Tendenziell können bei größer werdender Startposition größere Längen mit weniger Bits codiert werden, da die Binärcodierung für größere Zahlen besser geeignet ist.
Wird das Zeilenende mit einer 1-Bit-Folge erreicht oder überschritten, zählt das nächste 1-Bit ab Beginn der folgenden Zeile. Die bis dahin noch nicht „verbrauchten“ 1-Bits und die Binärbits bilden die Plateaulänge auf der neuen Zeile. Allerdings ist die Höhe in der neuen Zeile identisch mit der Höhe der 1. Spalte der Vorgängerzeile, also $h(0, n\text{-}1)$. Das ist vielleicht nicht unbedingt das, was man erreichen wollte!

Wenn man z.B. von der Spalte 57 ausgeht, muss die Plateaulänge für einen Zeilenwechsel mindestens 7 sein. Wenn es ein Plateau mit der Tabellenstartposition 2 ist, ergeben in der Bitfolge „`1111.1`“ (= 7) die 1-Bits aber nur 6. Man benötigt zum Zeilenwechsel also „`11111.x`“ (= 8 + x). Die restlichen Bits bilden die Längenangabe für das nächste Plateau, mit dem die neue Zeile beginnt, im Beispiel also „`.x`“. Die Bitfolge „`111111.11`“ ergibt in der neuen Zeile „`1.11`“, also 2 + 3 = 5.

„Trifft“ das letzte 1-Bit der 1-Bit-Folge hinter das Zeilenende, also ohne Berücksichtigung der Binärbits, wird die Startposition für das folgende Plateau zusätzlich dekrementiert.

Beispiel:
Tabellenposition = 3, „11.1“ → Länge 1 · 1 + 1 · 2 + 1 = 3 + 1
Plateau ab Spalte 61 → kein zusätzliches Dekrementieren
Plateau ab Spalte 62 → zusätzliches Dekrementieren

Ist außerdem die Binärbitanzahl größer als die auf der vorherigen Position, wird die Binärbitanzahl um 1 verringert.

Beispiel:
Tabellenposition = 10,
„11.x“ → Länge 2 · 4 + x = 8 + x
Plateau ab Spalte > 56:
Für Tabellenposition 11 werden normalerweise 3 Binärbits verwendet. In dieser speziellen Fall werden aber nur 2 Binärbits verwendet.
Plateau ab Spalte 56:
Dann werden 3 Binärbits verwendet.

Wird zum ersten mal der Bitwert 16 erreicht und im aktuellen Plateau ist nicht auch 4 mal der Bitwert 8 enthalten, wird dauerhaft auf die alternative Tabelle umgeschaltet.

In der folgenden Tabelle ist ablesbar, welche Maximallänge bei einer bestimmten Bit-Anzahl (einschließlich Trennbit) in Abhängigkeit von der Startposition angegeben werden kann.

Bits \ Pos.	0	1	2	3	4	5	6	7	8	9
2	1	1	1							
3	2	2		2	3	3	3			
4	3		3	4	5	5		5	7	7
5		4	5	6	7		7	9	11	11
6	5	6	7	8		9	11	13	15	
7	7	8	9		11	13	15	17		19
8	9	10		12	15	17	19		23	27
9	11		13	16	19	21		25	31	35
10		14	17	20	23		27	33	39	43
11	15	18	21	24		29	35	41	47	
12	19	22	25		31	37	43	49		59
13	23	26		32	39	45	51		63	75
14	27		33	40	47	53		65	79	91
15		34	41	48	55		67	81	95	
16	35	42	49	56		69	83	97		123
17	43	50	57		71	85	99		127	155
18	51	58		72	87	101		129	159	187
19	59		73	88	103		131	161	191	
20		74	89	104		133	163	193		251
21	75	90	105		135	165	195		255	315
22	91	106		136	167	197		257	319	
23	107		137	168	199		259	321		443
24		138	169	200		261	323		447	...
25	139	170	201		263	325		449	...	...
26	171	202		264	327		451	...	...	...
27	203		265	328		453	...	...	...	...
28		266	329		455	...	...	...	...	...
29	267	330		456	...	...	...	...	...	...
30	331		457	...	...	...	...	...	...	...
31		458	...	...	...	...	...	...	...	...
32	459	...	...	...	...	...	...	...	...	...

In der DEM-Datei wurde die folgende Bytefolge für eine Kachel mit der Maximalhöhe 3 gefunden:
FF FF FF FF FF FF FF FF FF FF C0 2E
Der Bitstream besteht also aus 82 1-Bits, danach folgt „.....1.111.“
Damit werden alle Punkte auf 0 bzw. die Basishöhe der Kachel gesetzt. Nur der Punkt in der Ecke links unten ist 3.
Erklärung:
Die ersten 17 1-Bits stellen die Plateaulänge 4*1+4*2+4*4+4*8+16 = 76 dar, d.h. die Länge der 1. Zeile wird überschritten. Die nächsten 3 1-Bits stellen die Plateaulänge 16+16+32 = 64 dar. Damit wird die 3. Zeile erreicht. Die nächsten 2 1-Bits stellen die Plateaulänge 32+32 = 64 dar. Damit wird die 4. Zeile erreicht. Die nächsten 60 1-Bits stellen einzeln jeweils 64 dar, d.h. es folgen weitere 20 Zeilenwechsel. Damit befindet man sich nach den 82 1-Bits am Anfang der 64. Zeile. Es folgen 1 Trennbit und 7 Binärbits (Länge 0).
Es bleiben jetzt noch folgende Bits: „1.111.“. Die ersten 2 Bits „1.“ stehen bei *hunit*=1 für 0, als Nachfolgewert eines Plateaus bedeutet das die Höhe -1. Wegen des Wraparound steht -1 für die max. Höhe, also 3. Man hätte natürlich auch die 3 direkt codieren können. Bei *hunit*=1 ergibt das „..11“, also 4 Bit, d.h. 2 Bit mehr.
Ähnlich sieht es für die folgenden 2 Bit aus. „11“ steht bei *hunit*=1 für 1 und führt zur Höhe 4, die aber wegen des Wraparound für 0 steht. -3 „...1.“ führt zum gleichen Ergebnis, benötigt aber 5 statt 2 Bit.
Die restlichen 2 Bits „1.“ stellen eine Plateaulänge für den Rest der Zeile mit der aktuellen Höhe 0 dar. Die Binärbits und der Nachfolger können offensichtlich entfallen.

Regeln der Codierart-Umschaltung

Da dem Decoder nicht explizit mitgeteilt wird, welche Codierart jeweils verwendet wird, muss es Regeln für die Verwendung der Codierarten geben. Diese Regeln können nur auf den bisher schon decodierten Zahlen basieren.
In der Annahme, dass eine hybride Codierung erfolgt, wird zunächst *hunit* bestimmt. Ist *hunit* kleiner als 1, wird eine

Längencodierung verwendet. Es muss dann aber noch zwischen L0 und L1 entschieden werden. Die Binärcodierung für große Zahlen wird nur verwendet, wenn die aktuelle Zahl mit den anderen Varianten nicht codiert werden kann. Hier findet der Decoder ausnahmsweise in den (zu) vielen führenden 0-Bits die notwendige Information.
Zusätzlich muss noch beachtet werden, dass nicht alle Werte zusammen berücksichtigt werden, sondern dass mehrere Gruppen gebildet werden. In jeder dieser Gruppen wird separat entschieden. Folgende Gruppen sind bekannt:

- Standardwerte
- Plateau-Nachfolger mit $ddiff(n, m) \neq 0$
- Plateau-Nachfolger mit $ddiff(n, m) = 0$

Wraparound

In jedem Kacheldatensatz ist die Größe des verwendeten Wertebereichs gespeichert. Wird der damit definierte Wertebereich nach oben oder unten überschritten, erfolgt ein „Wraparound“. Dadurch wird der Wert wieder im gültigen Wertebereich abgebildet:

$$h_w = h_i + n \cdot (max + 1) \qquad n \in G, n \text{ so dass } 0 \le h_j \le max$$

Welche Auswirkungen hat das auf die berechneten und gespeicherten Datenwerte *d*? Für Standardwerte gilt bei gegebenen $h_{n-1,m-1}$, $h_{n,m-q}$ und $h_{n-1,m}$ für $h_{n,m}$ und $d_{n,m}$:

$$\begin{aligned} d_{n,m} &= -sgn(h_{n,m-1} - h_{n-1,m}) \cdot ((h_{n,m} - h_{n-1,m}) - (h_{n,m-1} - h_{n-1,m-1})) \quad \text{bzw.} \\ d_{n,m} &= \pm h_{n,m} + konst \\ d_{n,m} &= -sgn(h_{n,m-1} - h_{n-1,m}) \cdot h_{n,m} \\ d_{n,m} &= \pm h_{n,m} \end{aligned}$$

Höhe ohne Wrapping (Max. 9)	Höhe mit Höhendifferenz-Flag	
	0x00	0x02
11	1	1
10	0	0
9	9	-
8	8	8
1	1	1
0	0	0
-1	9	-
-2	8	8

Da also *d* nur durch durch eine Addition aus der Höhe mit einem konstanten Wert entsteht, wird auch für jedes d_w mit

$$d_w = d_i + n \cdot (max + 1) \qquad n \in G$$

die gleiche Höhe erreicht, wie mit d_i. Prinzipiell ist zu erwarten, das betragsmäßig kleinere Zahlen auch weniger Bits bei der Codierung benötigen als größere Zahlen. Für eine möglichst platzsparende Codierung muss man das „kleinste“ *d* finden.

> Ist z. B. der Maximalwert 10 und die gewünschte Zielhöhe erhält man mit $d = 8$. Dann erhält man die gleiche Höhe u. a. auch mit 19, 20, 31 usw., aber auch mit -3, -14 usw.. Wahrscheinlich wird -3 effektiver codiert als 8.

Praktisch sinnvolle Werte erhält man aber bestenfalls für n = -1 bzw. n = 1. Andernfalls ist d_w schon betragsmäßig viel zu groß.
Für L0 ist jedes d_w effektiver als d_i, für das gilt:

$$\begin{aligned} d_i > 0: &\quad -d_i < d_w < d_i \\ d_i < 0: &\quad d_i < d_w < -d_i + 1 \end{aligned}$$

Für $d_i > 0$ folgt daraus, dass nur für folgende d_i ein Wraparound sinnvoll ist:

$$\begin{aligned} &-d_i < d_i + n \cdot max < d_i \\ &-2 \cdot d_i < n \cdot max < 0 \Rightarrow \boldsymbol{n = -1} \\ &\boldsymbol{d_i > max/2} \end{aligned}$$

Für $d_i < 0$ muss für d_i gelten:

$$\begin{aligned} &-d_i < d_i + n \cdot max < d_i \\ &0 < n \cdot max < -2 \cdot d_i + 1 \Rightarrow \boldsymbol{n = 1} \\ &\boldsymbol{d_i < -(max - 1)/2} \end{aligned}$$

> Ist z. B. der Maximalwert 11, ist ein Wraparound nur für Datenwerte d_i größer als 11/2 = 5,5 mit d_i - 11 effektiv (6 → -5, … 11 → 0) bzw. für d_i kleiner als -(11-1)/2 = -5 mit d_i + 11 effektiv (-6 → 5, … -11 → 0).

Für L1 ist jedes d_w effektiver als d_i, für das gilt:

$d_i > 0$: $\quad -d_i+1 \;<\; d_w \;<\; d_i$
$d_i \le 0$: $\quad d_i \;<\; d_w \;<\; -d_i+2$

Für $d_i > 0$ folgt daraus, dass nur für folgende d_i ein Wraparound sinnvoll ist:

$$-d_i+1 \;<\; d_i+n\cdot max \;<\; d_i$$
$$-2\cdot d_i+1 \;<\; n\cdot max \;<\; 0 \;\Rightarrow\; \boldsymbol{n=-1}$$
$$\boldsymbol{d_i \;>\; (max+1)/2}$$

Für $d_i < 0$ muss für d_i gelten:

$$d_i \;<\; d_i+n\cdot max \;<\; -d_i+2$$
$$0 \;<\; n\cdot max \;<\; -2\cdot d_i+2 \;\Rightarrow\; \boldsymbol{n=1}$$
$$\boldsymbol{d_i \;<\; -(max-2)/2}$$

Ist z. B. der Maximalwert 11, ist ein Wraparound nur für Datenwerte d_i größer als (11+1)/2 = 6 mit d_i - 11 effektiv (7 → -4, … 11 → 0) bzw. für d_i kleiner als -(11-2)/2 = -4,5 mit d_i + 11 effektiv (-5 → 6, … -11 → 0).

Bei der hybriden Codierung kann man bei gegebener *hunit* ohne 0-Bits die Zahlen im Bereich $-(hunit-1) \le d_i \le hunit$ codieren. Mit jedem 0-Bit erweitern sich diese Grenzen um *hunit*. Betragsmäßig größere Zahlen haben also auf keinen Fall eine kleinere Bitanzahl. Insofern kann grob abgeschätzt werden, dass sich ein Wraparound höchstens für Zahlen größer als *max*/2 (wegen der Subtraktion von *max*+1) bzw. kleiner als -*max*/2 (wegen der Addition von *max*+1) lohnt. Andererseits ist bekannt, dass schon für Zahlen größer als *max*/2+*hunit* mindestens ein weiteres Bit benötigt wird, während die Wraparound-Zahl betragsmäßig kleiner wird, also höchstens noch weniger Bit benötig. D. h., für Zahlen größer als *max*/2+*hunit* bzw. kleiner als -(*max*/2+*hunit*) ist ein Wraparound immer sinnvoll. Da keine einfache (Un-)Gleichung für die Berechnung der Grenzwerte gefunden wurde, muss für die Zahlen in den beiden kleinen „Unsicherheitsbereichen" die Bitanzahl für diese Zahlen und die zugehörigen Wraparound-Zahlen berechnet und verglichen werden.
Um bei gegebener *hunit* die Zahl d_i zu codieren benötigt man *b* 0-Bits:

$$d_i > 0: \quad b = \frac{d_i - 1}{hunit} \qquad \text{bzw.} \quad d_i < 0: \quad b = -\frac{d_i}{hunit}$$

Prinzipiell wären die folgenden Ungleichungen zu lösen:

$$d_i > 0: \quad integer\left(\frac{d_i-1}{hunit}\right) \;>\; integer\left(\frac{-(d_i-(max+1))}{hunit}\right)$$
$$d_i < 0: \quad integer\left(\frac{-d_i}{hunit}\right) \;>\; integer\left(\frac{d_i+(max+1)-1}{hunit}\right)$$

Für den Spezialfall *hunit*=1 erhält man:

$$d_i < 0: \quad d_i \;<\; -\frac{max}{2} \qquad \text{bzw.} \quad 0 < d_i: \quad \frac{max+2}{2} \;<\; d_i$$

Am einfachsten ist die Betrachtung für die Binärcodierung für große Zahlen. Die Bitanzahl ist hier konstant. Trotzdem muss ein Wraparound angewendet werden, weil die Bitanzahl nur für den Wertebereich $-2^{integer(ld(max))} \;\le\; d \;\le\; 2^{integer(ld(max))}$ ausreicht. Für Zahlen mit $d_i < -2^{integer(ld(max))}$ bzw. $2^{integer(ld(max))} < d_i$ muss deshalb zwingend der Wraparound erfolgen.
Bei den Plateau-Nachfolgern muss man etwas vorsichtig sein. Für *ddiff* = 0 wird ein negatives *vdiff* um 1 vergrößert. Allerdings muss hier das u. U. schon gewrappte *vdiff* betrachtet werden.

Beispiel:
Bei einer Maximalhöhe 19 sind folgende Werte vorhanden:

x	19
19	2

d.h. *ddiff* = 0, *vdiff* = -17
Da *vdiff* < 0 müsste -17+1=-16 gespeichert werden. Wird jetzt ein Wraparound angewendet, ensteht -16+(19+1)=4. Beim Decodieren muss aus *d* > 0 geschlossen werden, dass *vdiff* = *d* = 4, also die Höhe 19+4=23 und mit Wrapping 3 ist. Das ist natürlich falsch.
Wird zuerst der Wraparound auf -17 angewendet, erhält man das neue *vdiff* = 3. Beim Decodieren erhält man nun die Höhe 19+3=22 und mit Wrapping 2.
Ähnlich verhält es sich in folgendem Fall:

x	2
2	19

d.h. *ddiff* = 0, *vdiff* = 17

Da *vdiff* > 0 müsste 17 gespeichert werden. Wird jetzt ein Wraparound angewendet, ensteht 17-(19+1)=-3. Beim Decodieren muss aus $d < 0$ geschlossen werden, dass *vdiff* = *d*-1 = -4, also die Höhe 2-4=-2 und mit Wrapping 18 ist. Das ist wieder falsch.
Wird zuerst der Wraparound auf 17 angewendet, erhält man das neue *vdiff* = -3. Gespeichert wird -3+1=-2. Beim Decodieren muss der negative Wert -2 zunächst um 1 verringert werden. Man erhält nun die Höhe 2-3=-1 und mit Wrapping 19.

Für ddiff ≠ 0 spielt das übrigens keine Rolle.

hunit-Bestimmung

Im Prinzip wird aus den bisherigen Daten auf eine bestimmte Art eine Summe gebildet und diese Summe in Relation zur Anzahl *vg* der bisherigen Werte gesetzt. Für den so erhaltenen Wert kann man aus einer Tabelle *hunit* ablesen. Da für den ersten Wert einer Gruppe noch keine Vorgänger existieren, wird immer die Hybridcodierung verwendet. *hunit* wird aus der maximalen Höhendifferenz der Kachel abgeleitet (siehe Tabelle).
Es wurden 2 Varianten gefunden.

maximalen Differenz zum Basiswert		**hunit für d_0**
Hex	**dez**	
… 0x9e	… 158	$2^0 = 1$
0x9f … 0x11e	159 … 286	$2^1 = 2$
0x11f … 0x21e	287 … 542	$2^2 = 4$
0x21f … 0x41e	543 … 1054	$2^3 = 8$
0x41f … 0x81e	1055 … 2078	$2^4 = 16$
0x81f … 0x101e	2079 … 4126	$2^5 = 32$
0x101f … 0x201e	4127 … 8222	$2^6 = 64$
0x201f … 0x401e	8223 … 16414	$2^7 = 128$
0x401f … 0x7fff	16415 … 32767	$2^8 = 256$

Tabelle der *hunit* für d_0

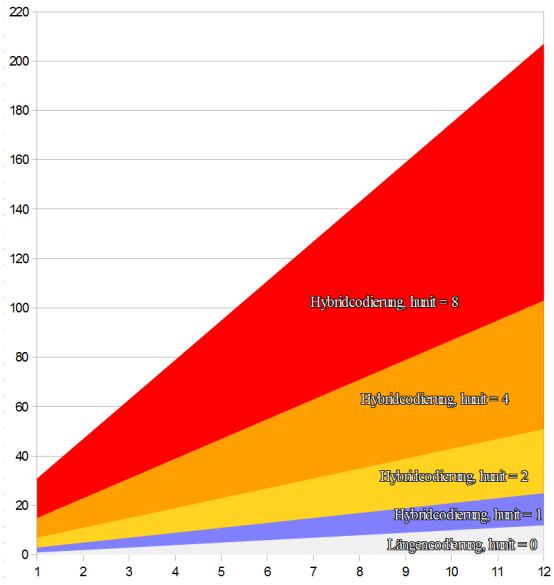
Variante HS (Hybridcodierung für Standardwerte):
Bei der Variante für die Standardwerte wird die Summe aus den Absolutwerten der Daten gebildet.
In der Basistabelle sind einige experimentell gefundene Minimalsummen für die zugehörige *hunit* und Anzahl der bisherigen Datenwerte (Vorgängeranzahl *vg*) eingetragen. Die Tabellenwerte ergeben sich aus

$$minsumHS_{hunit\ ,\ vg} = (vg+1)\cdot hunit - 1$$.

Sie gelten allerdings nur, wenn die maximale Höhendifferenz der Kachel nicht größer als 158 ist!

hunit / vg	**1**	**2**	**3**	**4**	**5**	**...**	**63**
1	1	2	3	4	5		63
2	3	5	7	9	11		127
4	7	11	15	19	23		255
8	15	23	31	39	47		511
16	31	47	63	79	95		1023
32	63	95	127	159	191		2047
64	127	191	255	319	383		4095
...							

Basistabelle HS der *hunit*-Wechsel (für Start-*hunit* = 1)



minsumHS in Abhängigkeit von *vg* und *hunit*

Reicht *minsumHS* nicht wenigstens zum Erreichen von hunit = 1 aus (also bei $minsumHS_{vg} < (vg+1)\cdot 1 - 1$ bzw. $minsumHS_{vg} < vg$), wird der aktuelle Wert mit L0 oder L1 codiert.

Beispiel:
Für die Datenwerte „1, -3, 2, 4“ ist die Summe der Absolutwerte a*sum* = 10 und *vg* = 4. Für den nächsten Datenwert ergibt sich wegen 9 ≤ 10 < 19 → *hunit* = 2.

hunit scheint auf maximal 256 begrenzt zu sein. Theoretisch ist die Größe von *hunit* aber natürlich nicht begrenzt. Denkbar wäre eine (praktisch wohl unbedeutende) Grenze bei 2 Byte, d.h. 65535. Es ist natürlich auch möglich, dass Garmin einen generellen Grenzwert von 256 definiert hat.

Aus der Gleichung für $minsumHS_{hunit,\ vg}$ folgt $hunit_{vg} = \frac{minsumHS_{vg}+1}{vg+1}$. Gleichzeitig ist *hunit* aber immer die größte passende 2er Potenz. Deshalb kann *hunit* folgendermaßen berechnet werden:

$$hunit = 2^{integer\left(ld\left(\frac{asum_{vg}+1}{vg+1}\right)\right)}$$ („integer“ liefert den ganzzahligen Anteil der Zahl).

Beispiele:

Datenwerte „5, 7, 3“ → $hunit = 2^{integer\left(ld\left(\frac{5+7+3+1}{3+1}\right)\right)} = 2^{integer(ld(4))} = 2^2 = 4$

Datenwerte „5, 6, 3“ → $hunit = 2^{integer\left(ld\left(\frac{5+6+3+1}{3+1}\right)\right)} = 2^{integer(ld(3{,}75))} = 2^1 = 2$

Datenwerte „1, 1, -1“ → $hunit = 2^{integer\left(ld\left(\frac{1+1+1+1}{3+1}\right)\right)} = 2^{integer(ld(1))} = 2^0 = 1$

Wie bereits erwähnt, gilt die Basistabelle nur, wenn die maximalen Höhendifferenz der Kachel nicht größer 158 ist. Die Werte der Basistabelle „verschieben“ sich abhängig von dieser Differenz, die aus dem Beschreibungsdatensatz bekannt

ist.
Für Höhendifferenzen im Bereich 0x9f bis 0x11e wird dieser in 2 gleichgroße Bereiche aufgeteilt. Für 0x9f bis 0xde werden alle Werte der Basistabelle um 1 verringert. Für den 2. Bereich von 0xdf bis 0x11e werden alle Werte der Basistabelle um 2 verringert. Das gleiche Prinzip gilt für größere *hunit*. Da mit jeder Verdopplung des *hunit* auch die Gesamtbereichsbreite verdoppelt wird, gleichzeitig aber auch die Anzahl der Teilbereiche verdoppelt wird, bleibt die Breite eines Teilbereiches immer konstant 0x40. Für jeden Teilbereich ab 0x9f werden alle Werte der Basistabelle jeweils um 1 verringert.
Ist die maximale Höhendifferenzen größer oder gleich 0x9f, muss also folgende Verringerung aller Werte der Basistabelle um *hunitdelta* vorgenommen werden:

$$hunitdelta = 1 + \text{integer}\left(\frac{maxdiff - 0\text{x9f}}{0\text{x40}}\right)$$

oder allgemein:

$$hunitdelta = \text{integer}\left(\frac{\max(0\text{x5f}, maxdiff) - 0\text{x5f}}{0\text{x40}}\right)$$

(„max“ liefert die größere Zahl → für *maxdiff* < 0x9f wird *hunitdelta* 0).

Damit gilt insgesamt

$$minsumHS_{hunit,\,vg} = hunit \cdot (vg + 1) - 1 - \text{integer}\left(\frac{\max(0\text{x5f}, maxdiff) - 0\text{x5f}}{0\text{x40}}\right)$$

bzw. dezimal

$$minsumHS_{hunit,\,vg} = hunit \cdot (vg + 1) - 1 - \text{integer}\left(\frac{\max(95, maxdiff) - 95}{64}\right)$$

Beispiel:
Bei maxdiff = 0x816 = 2070 ist *starthunit* = 16. Für d_2 gilt:

$$asum_1 = hunit_2 \cdot 2 - 1 - \text{integer}((\max(95, 2070) - 95) \div 64)$$
$$asum_1 = hunit_2 \cdot 2 - 1 - \text{integer}(30{,}86)$$
$$asum_1 = hunit_2 \cdot 2 - 31$$

$hunit_2$ = 32 gilt also ab $asum_1$ = 33 (33 = 32 * 2 - 31) und nicht erst ab huv_1 = 63 wie in der Basistabelle.
$hunit_2$ = 16 gilt ab $asum_1$ = 1 (1 = 16 * 2 - 31).
$hunit_2$ = 8 gilt ab $asum_1$ = -15. Da es sich aber um die Summe von Absolutwerten handelt, gilt das nur für 0.
Kleinere *hunit* und damit auch eine Längencodierung sind für d_2 offensichtlich nicht möglich.

Beispiel:
Bei *maxdiff* = 0x21f ist *starthunit* = 8 und *hunitdelta* = 7.
Für i < 4 kann *hunit* = 1 nicht erreicht werden.
Für i = 1 kann *hunit* = 1 und 2 nicht erreicht werden.
Die negativen Werte wurden zwar berechnet und z.T. eingetragen, können aber praktisch nicht erreicht werden. Sie müssten eigentlich durch 0 ersetzt werden.

hunit / i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	...
1				-3	-2	-1	0	1	2	3	
2		-2	0	2	4	6	8	10	12	14	
4	0	4	8	12	16	20	28	...			
8	8	16	24	32	40	48					
...											

verschobene Basistabelle:
maxdiff = 0x21f → starthunit = 8, hunitdelta = 7

Es läßt sich eine allgemeine Gleichung formulieren, mit der *hunit* aus dem $asum_{vg}$ der Vorgänger und der Anzahl *vg* der Vorgänger ermittelt wird:

$$hunit = 2^{integer\left(ld\left(\frac{asum_{vg} + 1 + \text{integer}\left(\frac{\max(95, maxdiff) - 95}{64}\right)}{vg + 1}\right)\right)}$$

Und zum Schluß noch eine kleine Besonderheit:
Immer wenn 64 Vorgänger existieren, wird die Anzahl der Vorgänger auf 32 heruntergesetzt und die Summe auf $asum_{vg} = integer(asum/2) - 1$ verringert.

<u>Variante HP0 (Hybridcodierung für Plateauwerte mit *ddiff* = 0):</u>
Diese Variante funktioniert analog zu HS. Nur die Summe wird anders gebildet und es wird eine andere Tabelle verwendet. Bei dieser Variante wird bei positiven Werten $2 \cdot (d - 1)$ summiert, bei negativen Werten $2 \cdot -d$. Außerdem ist

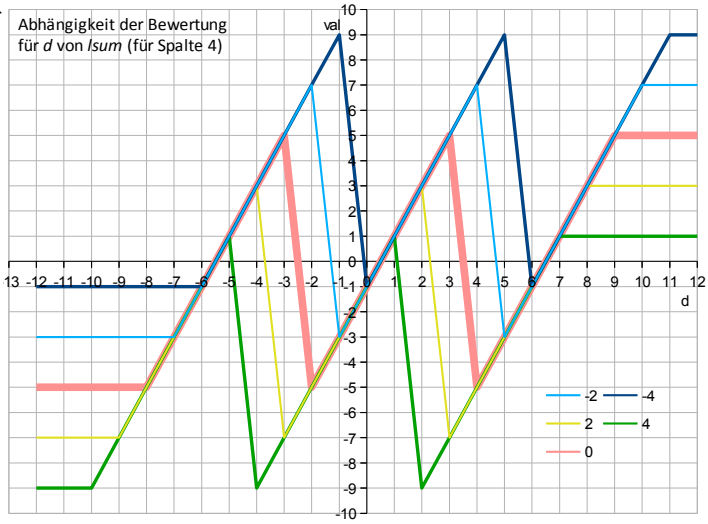
die Summe von vornherein mit 2 initialisiert.
Die Tabellenwerte ergeben sich aus $\boxed{minsumHP0_{hunit,vg} = 2\cdot(vg+1)\cdot hunit-(vg+1)}$. Bezogen auf die andere Tabelle von HS kann man die Werte auch so berechnen: $minsumHP0_{hunit,vg} = 2\cdot minsumHS_{hunit,vg}+1-vg$.

hunit / vg	**1**	**2**	**3**	**4**	**5**	**...**	**63**
1	2	3	4	5	6		64
2	6	9	12	15	18		192
4	14	21	28	35	42		448
8	30	45	60	75	90		960
16	62	93	124	155	186		1984
32	126	189	252	315	378		4032
64	254	381	508	635	762		8128
...							

Basistabelle HP0 der *hunit*-Wechsel (für Start-*hunit* = 1)

Art der Längencodierung

Wie bereits erwähnt, wird die Längencodierung immer dann verwendet wenn bei der spezifischen Summenbildung nicht der notwendige Mindestwert für *hunit* = 1 erreicht wird.
Um zu bestimmen, ob L0 oder L1 verwendet wird, wird eine Summe $valsum_{vg}$ von speziellen „Bewertungen“ aller Vorgänger gebildet. Die Datenwerte werden also zunächst neu „bewertet“ und diese „Neubewertungen“ werden dann summiert. Ist $valsum_{vg}$ größer als 0, wird die L1-Codierung verwendet, sonst die L0-Codierung.
Das Prinzip der Bewertung wird im Diagramm als Beispiel für den 5. Datenwert, also 4 Vorgänger gezeigt. Für relativ große bzw. kleine *d* ist die Bewertung konstant und hängt nur von $valsum_{vg}$ ab. Für $valsum_{vg} = 4$ ist z. B. die Bewertung für alle $d \geq 7$ konstant 1, für alle $d \leq -10$ konstant -9. Dazwischen gibt es allerdings einen „Zickzack“-Bereich.



Durch viele Versuche wurde folgender Zusammenhang ermittelt:

wenn	$d < -2 - (valsum_{vg} + 3 \cdot vg) / 2$	$val = -1 - valsum_{vg} - vg$
sonst wenn	$d < -(valsum_{vg} + vg) / 2$	$val = 2 \cdot (d + k) + 3$
sonst wenn	$d < 2 - (valsum_{vg} - vg) / 2$	$val = 2 \cdot d - 1$
sonst wenn	$d < 4 - (valsum_{vg} - 3 \cdot vg) / 2$	$val = 2 \cdot (d - k) - 5$
sonst		$val = 1 - valsum_{vg} + vg$

Immer wenn 64 Vorgänger existieren, wird die Bewertung u. U. anders berechnet. Wenn *d* positive ist und $valsum_{vg}+2\cdot d = 69$ oder wenn *d* negativ ist und $valsum_{vg}+2\cdot d = -65$, gilt $val = 2\cdot d - 1$.
Außerdem wird bei 64 Vorgängern die Anzahl der Vorgänger auf 32 heruntergesetzt und es erfolgt eine Korrektur der neu berechneten Summe. Zunächst wird $valsum_{vg}$ durch 2 dividiert. Entsteht dadurch ein ungerader Wert, wird wieder 1 addiert. Andernfalls wird, falls *d* = 1 oder *d* = -1, 2 addiert.

Achtung!
Für Plateau-Nachfolger mit $ddiff \neq 0$ wird nur L0 verwendet. (?)