LAZ Specification 1.4

Revision R1, Revision date October 27th, 2024

Published by:
LAStools LiDAR Processing - rapidlasso GmbH
Friedrichshafener Straße 1
82205 Gilching - Germany
https://rapidlasso.de
info@rapidlasso.de

Developer of LASzip: Dr. Martin Isenburg

Author of the LAZ Specification 1.4: Björn Eickenberg

Contents

Content by file structure

Table 1 — Overview Content by file structure

<table>
<tr><td colspan="4">Public Header Block (Clause 6.3)</td></tr>
<tr><td colspan="4">Variable Length Records (VLRs) (Clause 6.4) including the LAZ Special VLR</td></tr>
<tr><td rowspan="6">Compressed Data Block (Clause 11)</td><td colspan="3">Field Chunk table start position (Clause 11.5)</td></tr>
<tr><td rowspan="5">Chunks (Clause 11.7)</td><td rowspan="2">Chunk 1</td><td>1st record, uncompressed, split into Items (Clause 7.2)</td></tr>
<tr><td>Compressed records, optionally split into layers (Clause 11.7)</td></tr>
<tr><td rowspan="2">Chunk 2</td><td>1st record, uncompressed, split into Items (Clause 7.2)</td></tr>
<tr><td>Compressed records, optionally split into layers (Clause 11.7)</td></tr>
<tr><td colspan="2">Chunks n …</td></tr>
<tr><td colspan="3">Chunk table (Clause 11.6)</td></tr>
<tr><td colspan="4">Extended Variable Length Records (EVLRs) (Clause 6.5)</td></tr>
<tr><td colspan="4">Field Chunk table start position (EOF) (Clause 11.5)</td></tr>
</table>

LAZ Specification

1. Scope

This document specifies the LAZ 1.4 standard. This standard is provided by the open-source implementation of the LASzip library at https://github.com/LASzip/LASzip. The relevant version of the LASzip library is Release 3.4.3, released on Nov 11, 2019, and Release 3.4.4, released on April 17, 2024, which are identical in terms of specification.

LAZ 1.4 is based on the LAS 1.4 format, and only the LAZ 1.4 format using the LAS 1.4 format is specified.

Legacy support for LAS 1.0 to 1.3 formats exists (and can be read and written by the LASzip software), but is only covered as a short overview in this document, and not specified.

Additionally, only the current compressor formats are specified (although the LASzip software still supports the deprecated legacy compression formats.)

NOTE: For completeness, the LAS 1.4 fields and LAS structures, which are by design also part of the LAZ format, are described in this document too, as a shortened version of the LAS 1.4 standard as specified by The American Society for Photogrammetry & Remote Sensing, but this doesn't intend to (re-)define them.

2. Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

The American Society for Photogrammetry & Remote Sensing, LAS Specification 1.4 — R15, https://www.asprs.org/wp-content/uploads/2019/07/LAS_1_4_r15.pdf

3. Terms and definitions

The word "shall" is used to indicate a requirement to be strictly followed to conform to this document.

Additionally, the following additional terms and definitions apply.

3.1. Data types

The following data types are used in the LAZ format definition. They are conformant to the 1999 ANSI C Language Specification (ANSI/ISO/IEC 9899:1999 ("C99")).

— char (1 byte)
— unsigned char (1 byte)
— short (2 bytes)
— unsigned short (2 bytes)
— long (4 bytes)
— unsigned long (4 bytes)
— long long (8 bytes)
— unsigned long long (8 bytes)
— float (4 byte IEEE floating point format)

— double (8 byte IEEE floating point format)
— string (null-terminated variable series of 1 byte characters, ASCII encoded)

Note: Fixed-length char arrays will not be null-terminated if all bytes are utilized.

4. Conventions

Most algorithms are given as pseudocode, using the following conventions:

— assignments are done using :=
— the equality operator is =
— terms are evaluated from left to right, which might be relevant, for example, in case that there is more than one function call in an assignment
— blocks, for example the statements that follow an if-then-statement, are indented
— return exits a function
— arrays are, when explicitly indexed, starting with index 0
— // marks a comment

For use in, for example, integer divisions, “rounding towards 0” is defined as:

$$\text{round\_towards\_0}(x) := \begin{cases} \lfloor x \rfloor \text{ for } x \geq 0 \\ \lceil x \rceil \text{ for } x < 0 \end{cases}$$

5. LAZ Description

5.1. Overview

LAZ uses a lossless compression to compress a LAS file. The LAZ 1.4 format is based on LAS 1.4, as specified in the LAS Specification 1.4 — R15 by The American Society for Photogrammetry Remote Sensing.

LAZ keeps the general structure of the LAS format, but compresses the point data block. The format of all other LAS 1.4 sections, for example the header format, are not modified, although some header fields got slightly changed.

The specific meaning of the (uncompressed) LAS point data fields is not changed by the LAZ format.

Point data is stored in chunks, where each chunk can be decompressed independently from other chunks, which allows random-access in granularity of the chunk size.

LAZ 1.4 added support for LAS 1.4 point formats 6 to 10, and for those, LAZ subdivides chunks further into layers, each containing a subset of fields, allowing random access to specific fields without having to decompress all other fields.

The default file extension for a LAZ file is “laz”, as compared to “las” for a LAS file.

5.2. LAZ Developer

LASzip was developed between 2007 and 2010 by rapidlasso founder and lidar pioneer Dr. Martin Isenburg. Martin initially developed LASzip to support his lidar processing software LAStools. In 2011, he converted the implementation from an academic prototype to industry-grade production code and made LASzip open source. In November 2011, Martin introduced LASzip to the audience at ELMF in Salzburg, Austria (paper and video). The innovation quickly became very popular. LASzip became winner of the “2012 Geospatial World Forum Technology Innovation Award in LiDAR Processing” and runner-up for the “most innovative product at INTERGEO 2012”. Over the years, LASzip has become the industrial de-facto standard for LiDAR compression and is supported by virtually all existing point-cloud processing tools.

Beyond being a gifted software developer, Martin was very aware of our impact on the world. He had great respect for nature and the environment. He cared not only about technology, but also about the potential of technology to improve our planet and the human condition. Martin was talking about reducing the carbon footprint of computing before nearly anyone else. The carbon footprint he saved by giving away LASzip will be a lasting impact. Martin made outstanding contributions to the LiDAR community, and he was a source of inspiration to many. This will not be forgotten. Martin passed away on September 7, 2021. May he rest in peace.

6. LAZ file structure

6.1. Overview

Just as the LAS format, the LAZ format contains binary data consisting of a public header block (Clause 6.3), any number of (optional) Variable Length Records (VLRs) (Clause 6.4), the (compressed) Point Data Records (Clause 11), and any number of (optional) Extended Variable Length Records (EVLRs) (Clause 6.5). All data is in little-endian format.

The public header block contains generic data such as point numbers and point data bounds, and is mostly unchanged compared to the LAS format, including the file identifier. Note that only the LAS 1.4 header format is allowed for the LAZ 1.4 format. Legacy information for older LAS header formats is summarized in Clause 15.

The Variable Length Records (VLRs) contain variable types of data including projection information, metadata, waveform packet information, and user application data. They are limited to a data payload of 65,535 bytes.

LAZ adds a special VLR that contains details about the compression. The presence of this special VLR identifies the file as a LAZ file.

The central element of a LAZ file is the compressed Point Data Block, which contains the compressed data (e.g. the compressed point records). This block differs significantly from the uncompressed data points that LAS stores at this location.

The Extended Variable Length Records (EVLRs) are again a standard LAS structure, and not modified. They allow a higher payload than VLRs and can be appended to the end of a LAS file, which for example allows adding projection information to a LAS file without having to rewrite the entire file.

A LAS file that contains point record types 4, 5, 9, or 10 could potentially contain one block of waveform data packets that is stored as the payload of an Extended Variable Length Record (EVLR). This is both deprecated in LAS 1.4 and not supported by LAZ 1.4, so any Waveform Data Packets have to be stored externally (e.g. according to the LAS standard in a .WDP-file with the same name).

Optionally, the last 8 bytes of a LAZ file can contain an 8-byte Pointer to the LAZ Chunk table, which can be used if the data is written to a non-seekable medium.

Table 2 — LAZ File Format Definition

Public Header Block
Variable Length Records (VLRs)
Compressed Data Block
Extended Variable Length Records (EVLRs), cannot contain internal waveform data
Chunk table start position (EOF) (optional)

6.2. Differences between LAS and LAZ files

As an overview, the main differences between a LAZ file and a LAS file are:

— a value of 128 is added to the point record format (field **Point Data Record Format** in the LAZ header), i.e. the values 128 to 138 in that field correspond to the LAS record formats 0 to 10. This is to prevent a program that reads LAS data (but is unaware of LAZ data) to accidently read compressed data as uncompressed data, as by design, the file format is otherwise compatible.
— in a LAZ file, internally stored waveform data (in an EVLR) is not supported, and has to use an auxiliary file (e.g. a .WDP file as defined in the LAS specification). Note: internally stored waveform data is also deprecated in LAS 1.4.
— an additional LAZ specific VLR with information about the compression is added.
— the compressed data block itself, replacing the original LAS point data. It starts at the same position in the file as the LAS point data, namely the field **Offset to point data** as given in the LAZ header.
— optionally a field **Chunk table start position (EOF)** at the end of the file.

6.3. LAZ Header

The LAZ 1.4 header format is identical to the LAS 1.4 header format. Most fields are unchanged compared to the LAS 1.4 header. Fields that are modified or have a different meaning are marked in the table as “Modified from LAS”.

NOTE: The specific meaning of the fields is identical to the LAS 1.4 specification, unless modifications are specified. LAZ 1.4 doesn’t intend to redefine these fields. Their description is here mostly a slightly shortened description of the LAS 1.4 specification.

Table 3 — LAZ Header Format Definition

Field Name	Format	Size	Required	Modified from LAS
File Signature (“LASF”)	char[4]	4 bytes	*	
File Source ID	unsigned short	2 bytes	*	
Global Encoding	unsigned short	2 bytes	*	*
Project ID — GUID Data 1	unsigned long	4 bytes		
Project ID — GUID Data 2	unsigned short	2 bytes		
Project ID — GUID Data 3	unsigned short	2 bytes		
Project ID — GUID Data 4	unsigned char[8]	8 bytes		
Version Major	unsigned char	1 byte	*	
Version Minor	unsigned char	1 byte	*	
System Identifier	char[32]	32 bytes	*	
Generating Software	char[32]	32 bytes	*	
File Creation Day of Year	unsigned short	2 bytes	*	
File Creation Year	unsigned short	2 bytes	*	
Header Size	unsigned short	2 bytes	*	
Offset to Point Data	unsigned long	4 bytes	*	*
Number of Variable Length Records	unsigned long	4 bytes	*	
Point Data Record Format	unsigned char	1 byte	*	*
Point Data Record Length	unsigned short	2 bytes	*	
Legacy Number of Point Records	unsigned long	4 bytes	*	
Legacy Number of Point by Return	unsigned long[5]	20 bytes	*	
X Scale Factor	double	8 bytes	*	
Y Scale Factor	double	8 bytes	*	
Z Scale Factor	double	8 bytes	*	

Field Name	Format	Size	Required	Modified from LAS
X Offset	double	8 bytes	*	
Y Offset	double	8 bytes	*	
Z Offset	double	8 bytes	*	
Max X	double	8 bytes	*	
Min X	double	8 bytes	*	
Max Y	double	8 bytes	*	
Min Y	double	8 bytes	*	
Max Z	double	8 bytes	*	
Min Z	double	8 bytes	*	
Start of Waveform Data Packet Record	unsigned long long	8 bytes	*	*
Start of First Extended Variable Length Record	unsigned long long	8 bytes	*	
Number of Extended Variable Length Records	unsigned long	4 bytes	*	
Number of Point Records	unsigned long long	8 bytes	*	
Number of Points by Return	unsigned long long[15]	120 bytes	*	

Any field in the Public Header Block that is not required and is not used must be zero filled.

File Signature: The four characters “LASF”, identical to the LAS specification. These four characters can be checked by user software as a quick look initial determination of file type.

File Source ID: This field should be set to a value ranging from 0 to 65,535. If this file was derived from an original flight line, this is often the flight line number. A value of zero is interpreted to mean that an ID has not been assigned, which is the norm for a LAS file resulting from an aggregation of multiple independent sources (e.g., a tile merged from multiple swaths). Note that this scheme allows a LIDAR project to contain up to 65,535 unique sources. Example sources can be an original flight line or a setup identifier for static systems.

Global Encoding: This is a bit field used to indicate certain global properties about the file, defined as:

Table 4 — Encoding of bit-field “Global Encoding”

Bits	Field Name	Description
0	GPS Time Type	The meaning of GPS Time in the point records. If this bit is not set, the GPS time in the point record fields is GPS Week Time (the same as versions 1.0 through 1.2 of LAS). Otherwise, if this bit is set, the GPS Time is standard GPS Time (satellite GPS Time) minus 1 x 10^9 (Adjusted Standard GPS Time). The offset moves the time back to near zero to improve floating point resolution. The origin of standard GPS Time is defined as midnight of the morning of January 6, 1980.
1	Waveform Data Packets Internal	Not supported by LAZ, shall be unset. If this bit is set in a LAS file, the waveform data packets are located within the file. (Note that this bit is mutually exclusive with bit 2.) Note: this is also deprecated in LAS 1.4.
2	Waveform Data Packets External	If this bit is set, the waveform data packets are located externally in an auxiliary file with the same base name as this file but the extension *.wdp. (Note that this bit is mutually exclusive with bit 1)

Bits	Field Name	Description
3	Return numbers have been synthetically generated	If this bit is set, the point return numbers in the point data records have been synthetically generated. This could be the case, for example, when a composite file is created by combining a First Return File and a Last Return File, or when simulating return numbers for a system not directly supporting multiple returns.
4	WKT	If set, the Coordinate Reference System (CRS) is WKT. If not set, the CRS is GeoTIFF. It should not be set if the file writer wishes to ensure legacy compatibility (which means the CRS must be GeoTIFF). Refer to the LAS specification for details about these formats.
5:15	Reserved	Shall be set to zero
NOTE: The option “Waveform Data Packets Internal” (bit 1) is not supported by LAZ 1.4, so the field differs slightly from the LAS 1.4 header.		

Project ID (GUID data): The four fields that comprise a complete Globally Unique Identifier (GUID) are now reserved for use as a Project Identifier (Project ID). The field remains optional. The time of assignment of the Project ID is at the discretion of processing software. The Project ID should be the same for all files that are associated with a unique project. By assigning a Project ID and using a File Source ID (defined above), every file within a project and every point within a file can be uniquely identified, globally.

Version Number: The version number consists of a major and minor field of the LAS specification used, i.e. for LAZ 1.4 (and LAS 1.4), the major field shall be 1 and the minor field shall be 4.

System Identifier: Identifies the hardware type or software operation that generated the data, as defined by the LAS specification:

Table 5 — System Identifier

Generating Agent	System ID
Hardware system	String identifying hardware (e.g. “ALTM 1210”, “ALS50”, “LMS-Q680i” etc.)
Merge of one or more files	“MERGE”
Modification of a single file	“MODIFICATION”
Extraction from one or more files	“EXTRACTION”
Reprojection, rescaling, warping, etc.	“TRANSFORMATION”
Some other operation	“OTHER” or a string up to 32 characters identifying the operation

Generating Software: This information is ASCII data describing the generating software itself. If the character data is less than 32 characters, the remaining data must be null.

File Creation Day of Year: Day, expressed as an unsigned short, on which this file was created. Day is computed as the Greenwich Mean Time (GMT) day. January 1 is considered day 1.

File Creation Year: The year, expressed as a four-digit number, in which the file was created.

Header Size: The size, in bytes, of the Public Header Block itself. For LAZ 1.4 (and LAS 1.4) in the current revision, this size is 375 bytes. The Public Header Block may not be extended by users.

Offset to point data: Actual number of bytes from the beginning of the file to the start of the compressed data block. Must be updated if any software adds or removes data to or from the Variable Length Records. Note: this differs from a LAS file, where it is the actual number of bytes from the beginning of the file to the first field of the first point record data field.

Number of Variable Length Records: The number of VLRs.

Point Data Record Format: The type of point data records that are contained in the file. LAS 1.4 defines types 0 through 10, LAZ adds the value 128 to it (i.e. sets bit 7), so uses types 128 to 138 to mean LAS types 0 through 10. LAZ additionally stores details about the point type in the LAZ Special VLR. The information in that VLR has to match the **Point Data Record Format**.

Point Data Record Length: The size, in bytes, of the uncompressed Point Data Record. All Point Data Records within a single file must be the same type and hence the same length. If the specified size is larger than implied by the point format type (e.g. 32 bytes instead of 28 bytes for type 1) the remaining bytes are user-specific "extra bytes". The format and meaning of such "extra bytes" can (optionally) be described with an Extra Bytes VLR (as specified in the LAS specification). LAZ compresses such extra bytes as single, independent bytes, i.e. does not change compression based on the description in the Extra Bytes VLR.

Legacy Number of point records: This field contains the total number of point records within the file if the file is maintaining legacy compatibility, and the number of points is no greater than UINT32_MAX ($2^{32} - 1$), and the Point Data Record Format is less than 6. It must be zero otherwise.

Legacy Number of points by return: These fields contain an array of the total point records per return if the file is maintaining legacy compatibility, the number of points is no greater than UINT32_MAX ($2^{32} - 1$), and the Point Data Record Format is less than 6. Otherwise, each member of the array must be set to zero. The first value will be the total number of records from the first return, the second contains the total number for return two, and so on up to five returns.

X, Y, and Z scale factors: The scale factor fields contain a double floating point value that is used to scale the corresponding X, Y, and Z long values within the point records. The corresponding X, Y, and Z scale factor must be multiplied by the X, Y, or Z point record value to get the actual X, Y, or Z coordinate. For example, if the X, Y, and Z coordinates are intended to have two decimal digits, then each scale factor will contain the number 0.01.

X, Y, and Z offset: The offset fields should be used to set the overall offset for the point records. In general these numbers will be zero, but for certain cases the resolution of the point data may not be large enough for a given projection system. However, it should always be assumed that these numbers are used.

For example, to compute a given X from the point record, the point record X is multiplied by the X scale factor, and then the X offset is added, and so on. I.e. $\mathrm{X}_{\mathrm{coordinate}} = (\mathrm{X}_{\mathrm{record}} \cdot \mathrm{X}_{\mathrm{scale}}) + \mathrm{X}_{\mathrm{offset}}$, $\mathrm{Y}_{\mathrm{coordinate}} = (\mathrm{Y}_{\mathrm{record}} \cdot \mathrm{Y}_{\mathrm{scale}}) + \mathrm{Y}_{\mathrm{offset}}$ and $\mathrm{Z}_{\mathrm{coordinate}} = (\mathrm{Z}_{\mathrm{record}} \cdot \mathrm{Z}_{\mathrm{scale}}) + \mathrm{Z}_{\mathrm{offset}}$.

Max and Min X, Y, Z: The max and min data fields are the actual unscaled extents of the LAS point file data, specified in the coordinate system of the LAS data.

Start of Waveform Data Packet Record: LAZ 1.4 does not support internally stored Waveform Data Packet Records, this value shall be 0 (which means no internally stored Waveform Data). Note that in a LAS file, this value provides the offset, in bytes, from the beginning of the LAS file to the first byte of the Waveform Data Package Record.

Start of First Extended Variable Length Record: This value provides the offset, in bytes, from the beginning of the file to the first byte of the first EVLR.

Number of Extended Variable Length Records: This field contains the current number of EVLRs. If there are no EVLRs this value is zero.

Number of point records: This field contains the total number of point records in the file.

Number of points by return: These fields contain an array of the total point records per return. The first value will be the total number of records from the first return, the second contains the total number for return two, and so on up to fifteen returns.

6.4. Variable Length Records (VLRs)

The Public Header Block can be followed by any number of Variable Length Records (VLRs) so long as the total size does not make the start of the Compressed Data Block inaccessible by an unsigned long ("Offset to Point Data" in the Public Header Block). The number of VLRs is specified in the "Number of Variable Length Records" field in the Public Header Block.

The format of the VLR is identical to the LAS format.

The Variable Length Records must be accessed sequentially since the size of each variable length record is contained in the Variable Length Record Header. Each Variable Length Record Header is 54 bytes in length, and optionally followed by Data of length given by "Record Length After Header".

Exactly one of the VLRs (not necessarily the first or last) is the special LAZ VLR, which contains additional information about the compression. The presence of this VLR identifies the file as a LAZ file.

LAS 1.4 specifies several mandatory (and optional) VLRs. These requirements and the meaning of those VLRs transfer to a LAZ file (as a valid LAZ file is based on a valid LAS file), but are not repeated in this LAZ specification. Refer to the LAS specification for details.

Table 6 — Variable Length Record

Field Name	Format	Size	Required
Reserved	unsigned short	2 bytes	
User ID	char[16]	16 bytes	*
Record ID	unsigned short	2 bytes	*
Record Length After Header	unsigned short	2 bytes	*
Description	char[32]	32 bytes	
Data		as given by "Record Length After Header"	

Reserved: This value must be set to zero for LAS standard records, and shall be zero or 43,707 (0xAABB) for the special LAZ VLR.

User ID: The User ID field is ASCII character data that identifies the user that created the variable length record. For a standard LAS file, the User ID must be registered with the LAS specification managing body, refer to the LAS specification. A LAZ file includes a special LAZ VLR, for which that value shall be "laszip encoded".

Record ID: The Record ID is dependent upon the User ID. There can be 0 to 65,535 Record IDs for every User ID. The LAS specification manages its own Record IDs (User IDs owned by the specification), otherwise Record IDs will be managed by the owner of the given User ID. Refer to the LAS specification for details. For the LAZ specific special LAZ VLR, the value shall be 22,204.

Record Length After Header: The record length is the number of bytes for the record after the end of the standard part of the header.

Description: Optional text description of the data. Any remaining unused characters must be null.

Data: Optional content with length given by **Record Length After Header**.

6.5. Extended Variable Length Records (EVLRs)

The Compressed Data Block can be followed by any number of EVLRs, which are identical to the LAS 1.4 specification. The EVLR is similar to a VLR, but can carry a larger payload, as the **Record Length After Header** field is 8 bytes instead of 2 bytes. The number of EVLRs is specified in the **Number of Extended Variable Length Records** field in the Public Header Block. The start of the first EVLR is at the file offset indicated by the **Start of first Extended Variable Length Record** in the Public Header Block.

The Extended Variable Length Records must be accessed sequentially, since the size of each variable length record is contained in the Extended Variable Length Record Header. Each Extended Variable Length Record Header (i.e. without the optional payload data) is 60 bytes in length.

Table 7 — Extended Variable Length Record

Field	Format	Size	Required
Reserved	unsigned short	2 bytes	
User ID	char[16]	16 bytes	*
Record ID	unsigned short	2 bytes	*
Record Length After Header	unsigned long long	8 bytes	*
Description	char[32]	32 bytes	
Data		as given by “Record Length After Header”	

The fields are specified identically to those of the VLR, with the exception that **Record Length After Header** is a 64-bit value.

As for the VLRs, any mandatory or optional EVLR and their content, as specified in the LAS 1.4 specification, transfer to a LAZ file, but are not repeated in this LAZ specification. Refer to the LAS specification for details.

7. The LAZ Special VLR

7.1. The LAZ Special VLR format

For a LAZ file, exactly one of the VLR records has to be the LAZ Special VLR, which contains information about the compression in the **Data** field, and identifies the file as a LAZ file. The LAZ VLR shall contain the string “laszip encoded” in the field **User Id** and the value 22204 in the field **Record Id**, both of which are required to identify the VLR as the LAZ Special VLR. The field **Reserved** shall contain either the value 43707 (0xAABB) or 0.

The special VLR shall be removed in a decompressed LAS file.

The VLR contains a record of the following format in the **Data** field:

Table 8 — LAZ Special VLR

Field Name	Format	Size	Required
Compressor	unsigned short	2 bytes	*
Coder	unsigned short	2 bytes	*
Version Major	unsigned char	1 byte	*
Version Minor	unsigned char	1 byte	*
Version Revision	unsigned short	2 bytes	*
Options	unsigned long	4 bytes	*
Chunk Size	unsigned long	4 bytes	*
Number of special EVLRs	signed long long	8 bytes	*
Offset of special EVLRs	signed long long	8 bytes	*
Number of Items	unsigned short	2 bytes	*
Item records	Array of “Item record”	6 bytes * Number of Items	*

Compressor: Defines the compressor and format of the LAZ Compressed Data Block and the Chunk Table, as specified in Clause 11.

One of the values

Table 9 — Field “Compressor”

Value	Description	Restrictions
0	No Compression	Uncompressed Standard LAS file
1	Pointwise compression	only for point types 0 to 5
2	Pointwise and chunked compression	only for point types 0 to 5
3	Layered and chunked compression	only for point types 6 to 10

No Compression: Indicates an uncompressed LAZ file, i.e. a LAS file with a LAZ VLR.

Pointwise compression: The data is stored in a single chunk, and no chunk table is used. Only for LAS Point Data Record Formats 0 through 5.

Pointwise and chunked compression: The data is stored using chunks, and a chunk table is used. Only for LAS Point Data Record Formats 0 through 5.

Layered and chunked compression: The data is stored using chunks and layers, and a chunk table and layer tables are used. Only for LAS Point Data Record Formats 6 through 10.

Coder: Identifies the coder. Shall be 0, for “Arithmetic coder”.

Version: The version number consists of a major, a minor and a revision field. They combine to form the number that indicates the format number of the current specification.

Options: Bit field used to indicate certain options. It is defined as:

Table 10 — Field “Options”

Bits	Field Name	Description
0	LAS 1.4 compatibility mode	1: LAS Point Data Record Formats 6 to 10 have been stored as for LAS Point Data Record Formats 0 to 5 plus extra bytes, 0 otherwise. Shall be 0 for LAZ 1.4
1:31	Reserved	Shall be set to zero

LAS 1.4 compatibility mode: Shall be 0 for LAZ 1.4. Set to 1 if LAS Point Data Record Formats 6 to 10 have been stored as LAS Point Data Record Formats 0 to 5 plus extra bytes in legacy LAZ 1.2 or 1.3 files (but not LAZ 1.4 files). See Clause 15.2 for legacy information about this mode.

Chunk Size: Number of points per chunk. At the beginning of each chunk, the compression and entropy data resets, so each chunk can be decompressed independently from other chunks, allowing random access in granularity of this chunk size. If $2^{32} - 1$, adaptive chunking is used: the compressor can choose a different chunk size for each chunk (with a minimum size of 1), and for that mode, the actual chunk sizes are stored in the Chunk table, Clause 11.6. Adaptive chunking is only supported for LAS Point Data Record Formats 6 through 10 and deprecated LAS Point Data Record Formats 0 through 5.

Number of special EVLRs: Reserved. Shall be -1, which means unused.

Offset of special EVLRs: Reserved. Shall be -1, which means unused.

Number of Items: Number of item records (that follow directly afterwards).

7.2. Item records

A LAZ file contains compressed LAS Point Data Records, and a LAS Point Data Record is considered to be built out of subpart, “items”. Each item is compressed separately and potentially with a different coder version. The Item records describe a list of Item types. The combined items shall match the **Point Data Record Format** and **Point Data Record Length** as declared in the LAZ Header.

Table 11 — Item record

Field Name	Format	Size	Required
Item Type	unsigned short	2 bytes	*
Item Size	unsigned short	2 bytes	*
Item Version	unsigned short	2 bytes	*

Item Type: Type of the item, one of the following:

Table 12 — Field “Item Type”

Value	Name	Description
0	Byte	extra bytes that are appended to a LAS Point Data Record Format 0 to 5
1	Short	reserved, unsupported
2	Integer	reserved, unsupported
3	Long	reserved, unsupported

Value	Name	Description
4	Floating point	reserved, unsupported
5	Double	reserved, unsupported
6	Point10	LAS Point Data Record Format 0, containing the core fields that are shared between LAS Point Data Record Formats 0 to 5
7	GPSTime11	the GPS Time field that is added for LAS Point Data Record Formats 1, 3, 4 and 5
8	RGB12	the R, G and B fields that are added for LAS Point Data Record Formats 2, 3 and 5
9	Wavepacket13	the 7 fields for the Waveform packet that are added for LAS Point Data Record Formats 4 and 5
10	Point14	LAS Point Data Record Format 6, containing the core fields that are shared between LAS Point Data Record Formats 6 to 10
11	RGB14	the R, G and B fields that are added for LAS Point Data Record Format 7
12	RGBNIR14	the R, G, B and NIR (near infrared) fields that are added for LAS Point Data Record Formats 8 and 10
13	Wavepacket14	the 7 fields for the Waveform packet that are added for LAS Point Data Record Formast 9 and 10
14	Byte14	extra bytes that are appended to a LAS Point Data Record Format 6 to 10
NOTE: The number in the name, for example in “Point10”, refers to the LAS and LAZ version where that type got added.		

Item types 1 through 5 (Short, Integer, Long, Floating Point and Double) are reserved for future use and not supported yet.

Item Size: Size in bytes of the item. The size for item types 6 to 13 is derived from the LAS standard, i.e. the size of the (uncompressed) fields those items cover. Item Types 0 to 5 and 14 shall be multiples of their type size. This means that the allowed values are:

Table 13 — Field “Item Size”

Item Type	Allowed Values
Byte	any
Short	multiple of 2
Integer	multiple of 4
Long	multiple of 8
Floating point	multiple of 8
Double	multiple of 8
Point10	20
GPSTime11	8
RGB12	6
Wavepacket13	29
Point14	30
RGB14	6
RGBNIR14	8
Wavepacket14	29
Byte14	any

The sum of the item size of all items shall match the **Point Data Record Length** in the LAZ Header. The combined item types together build the **Point Data Record Format** as specified in the LAZ Header. Any extra bytes are stored as item types “Byte” (for LAS Point Data Record Format 0 to 5) or “Byte14” (for LAS Point Data Record Format 6 to 10).

Valid item record combinations are therefore:

Table 14 — Valid item combinations (to form a LAS Point Data Record Format)

LAS Point Data Record Format	Items
0	Point10 (+ Byte)
1	Point10 + GPSTime11 (+ Byte)
2	Point10 + RGB12 (+ Byte)
3	Point10 + GPSTime11 + RGB12 (+ Byte)
4	Point10 + GPSTime11 + Wavepacket13 (+ Byte)
5	Point10 + GPSTime11 + RGB12 + Wavepacket13 (+ Byte)
6	Point14 (+ Byte14)
7	Point14 + RBG14 (+ Byte14)
8	Point14 + RBGNIR14 (+ Byte14)
9	Point14 + Wavepacket14 (+ Byte14)
10	Point14 + RBGNIR14 + Wavepacket14 (+ Byte14)

They shall be in that order, and Byte and Byte14 are optional.

Item Version: Version for the item:

— 0 for no compression (if field “Compressor” in the header is set to “No Compression”).
— For Point10, GPSTime11, RGB12, Byte: shall be 2. (Version 1 is outdated and not covered by this specification.)
— For Wavepacket13: shall be 1.
— For Point14, RGB14, RGBNIR14, Wavepacket14, Byte14: shall be 3. (Version 1 and 2 were never supported.)

8. Arithmetic Coding

8.1. Introduction

To compress the data, LAZ uses arithmetic coding as described and implemented in Said Amir, Introduction to Arithmetic Coding.

Arithmetic coding is a lossless entropy encoding. The basic idea is, based on the probabilities of the occurrence of symbols in the alphabet, to store the data as one rational number. This number is chosen to represent which probability interval all symbols lie in. Given unlimited precision, an unlimited amount of data could be encoded in one rational number.

As an example, we encode the sequence ABACA. Their probabilities are $p_A = \frac{3}{5}$, $p_B = \frac{1}{5}$ and $p_C = \frac{1}{5}$, so we set the probability intervals in this order from 0 to $\frac{3}{5}$ for A, $\frac{3}{5}$ to $\frac{4}{5}$ for B and $\frac{4}{5}$ to 1 for C.

To encode the first A, we can pick any number in the interval from 0 to $\frac{3}{5}$. E.g., if the encoded value (the number we pick) is 0.2, we know the first symbol (letter) is A, as 0.2 lies in the interval from 0 to 0.6.

To encode the next letter B, we divide the chosen interval from 0 to $\frac{3}{5} = 0.6$, which has a length of 0.6, according to the probabilities. E.g., the intervals are now 0 to $0.6 \cdot \left(\frac{3}{5}\right) = 0.36$ for A, $0.6 \cdot \left(\frac{3}{5}\right) = 0.36$ to $0.6 \cdot \left(\frac{4}{5}\right) = 0.48$ for B and $0.6 \cdot \left(\frac{4}{5}\right) = 0.48$ to 0.6 for C. This is a rescaling of the intervals. The chosen number now has to lie within the interval 0.36 to 0.48 to encode B.

To encode the next letter A, we again divide the chosen interval from 0.36 to 0.48, which has the length 0.12, according to the probabilities. E.g., the first $\frac{3}{5}$ of that interval represents A (0.36 to 0.432), the next $\frac{1}{5}$ of that interval represents B (0.432 to 0.456) and the last $\frac{1}{5}$ of the interval stands for C (0.456 to 0.48). So for A, the number has now to lie within the interval 0.36 to 0.432.

This procedure is repeated for the next two letters: to encode C, the number has to lie in the last $\frac{1}{5}$ of the interval 0.36 to 0.432, i.e. in the interval 0.4176 to 0.432, and to encode the letter A, the number has to be picked from within the first $\frac{3}{5}$ of that interval, i.e. from 0.4176 to 0.42624 in Figure 1.

The encoded rational number can be any number from within the last interval, for example 0.42.

The decoding process works similarly. It requires the probabilities, the length of the original string (5 letters in this example), and the rational number, i.e. 0.42 for this example. The decoding process will go through the same intervals, and the rational number determines which interval and thus letter to pick.

For example, 0.42 lies in the interval 0 to 0.6, so the first letter is A. 0.42 lies in the interval 0.36 to 0.48, so the 2nd letter is B. The 3rd iteration of intervals were 0.36 to 0.432 for A, 0.432 to 0.456 for B and 0.456 to 0.48 for C, so 0.42 means A. For the 4th letter, the interval 0.4176 to 0.432 stands for C, and for the last letter, 0.42 lies within 0.4176 to 0.42624, so A.

The algorithm has to know to stop here (for example by knowing the number of letters), otherwise, the decoder would go on.

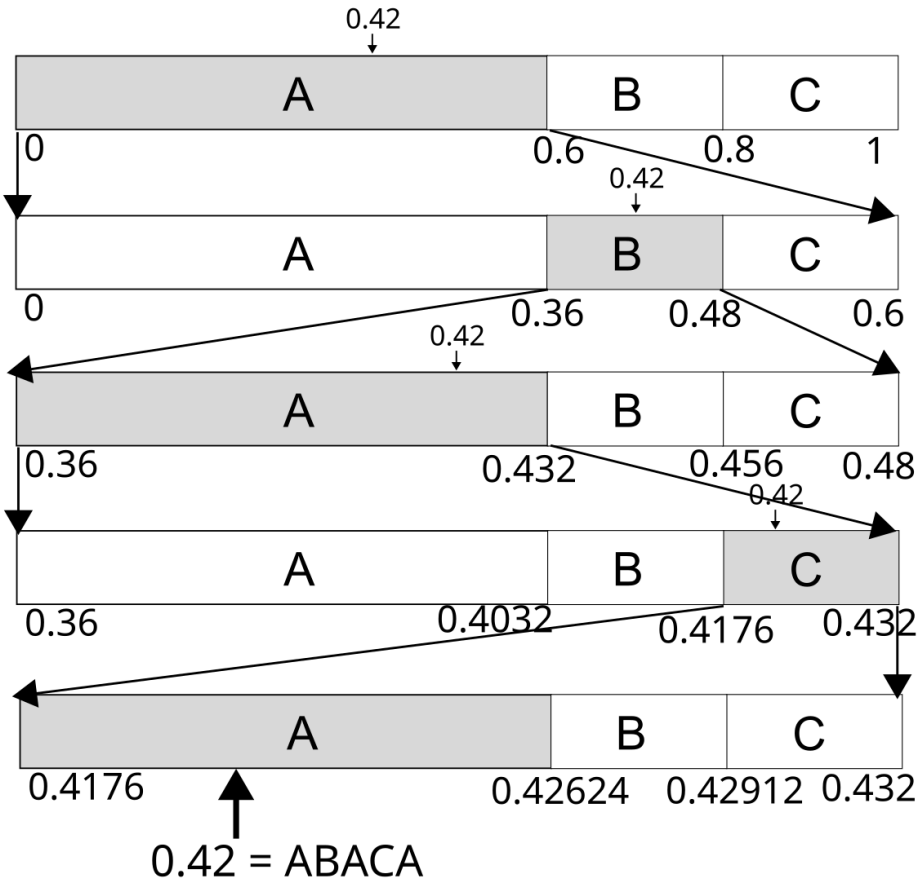

Figure 1 — Example: Arithmetic Coding for text "ABACA"

While not directly obvious that this method compresses data, it is a nearly optimal entropy encoder if the probabilities are correct.

8.2. Implementation

Practically, the implementation has to consider rounding imprecisions, for example potential differences of floating point operations on 32 bit and 64 bit hardware or similar.

While the LAZ algorithms are defined independently from an implementation, they are designed with 4 byte unsigned integer values in mind, and without the use of any floating point operations.

For example, modulo operations with 2^{32} are used because of the range limit of a 4 byte unsigned integer value. Similarly, floor in divisions by 2^x, for example, $\left\lfloor \frac{\text{some value}}{2^x} \right\rfloor$, can (for an integer variable) oftentimes be implemented with a right shift by x bits.

The algorithms use integers instead of floating point variables, and for that, the range of an integer variable is mapped to the rational values between 0 and 1. For e.g. a 4 byte unsigned long variable, 0x0 is 0 and (the non-storable) 0x100000000 would be 1 (i.e. all usable numbers are between 0 and 0.999999.., matching the required range for both the encoded rational number and the probabilities). All operations are then done using (exact) integer operations. This indirectly implies a precision of $\frac{1}{2^{32}}$.

The compressed data stream represents a very long rational number between 0 and 1 (exclusive), but only a small subset of 4 bytes is used at a time.

Additionally, to improve the compression, LAZ implements the following methods:

— the probabilities are initialized as a uniform distribution (all symbols have the same probability). During de- and encoding, the occurrence of symbols are counted, and the distribution is regularly updated after every n symbols (depending on the situation). To avoid zero probabilities (and thus zero-length intervals), each count is initialized with 1. This allows the compression to adjust to the actual symbol distribution in the data. Also, there is no need to include the probabilities, which are required for decoding, in the compressed data (which would require space), as both the decoder and encoder can calculate those values.
— each field may use several instances of distribution tables, depending on the situation. For example, there might be a correlation between the classification byte and the classification bits, so LAZ uses 256 different distribution table instances for the classification bits, selected by the classification byte. This can improve compression if the entropy is actually different. Similarly, the “classification”-field in the LAS point data is assumed to be dependent on the “return bits”-field.
— for most fields, only the difference to an offset value is stored, instead of the actual value. For example, the difference to the previous value, or the difference to the median of several previous values. For example, if the x coordinate changes only slowly to the next point, the probability for low values increases, which improves compression.

Which probability table to use is specified in the field-specific descriptions.

9. LAZ Compression

9.1. The compressed data stream

An encoded data stream represents one long rational number (with arbitrary precision) between 0 and 1, in the example in Figure 1 e.g. the value 0.42 .

The symbol distribution tables (which describe the intervals) of the encoders (specified in the following clauses) are initialized with equal distribution, and adjusted from time to time to the actual distribution, i.e. the decoded and encoded symbols. They are kept in sync during encoding and decoding.

To achieve arbitrary precision, the values are rescaled after each symbol, i.e. the intervals are always considered to go from 0 to 1. The actual length of the interval is kept, and if it gets too low, is rescaled too.

Only 4 bytes of the data stream are used at a time.

9.2. Data stream variables

For a specific encoded data stream, the following variables (to describe the current intervals) are defined and considered persistent for that one data stream. The compression algorithms operate on those variables:

Table 15 — Data stream variables

Variable	Range	Description
base	$0 \ldots 2^{32} - 1$	Used during encoding, represents the most significant, processed bits of the rational number. Is the sum of the lower interval boundaries. When the precision changes, data is written to the data stream.
value	$0 \ldots 2^{32} - 1$	Used during decoding, represents the most significant, unprocessed bits of the rational number. The lower interval boundaries get subtracted from this value, i.e. it will be the difference from the lower bound of the current interval (and indirectly from **base**), representing the remaining, not yet decoded data. When the precision changes, more data is read from the data stream. During decoding, **value** will always lie between 0 and **length**.
length	$0 \ldots 2^{32} - 1$	The length of the current interval (in integer arithmetic), i.e. the difference between lower and higher bound. When the length gets to small (note that the interval length gets smaller in each step), it means the precision changes, and the length gets rescaled. Used for both encoding and decoding.
NOTE: The variables need to only store integer values (i.e. no decimal places).		

9.3. Initialization of the stream

9.3.1. Decoding

The decoder is initialized by:

— reading the first 4 bytes from the data stream into the variable **value**, where the first byte in the data stream being the highest byte in the variable, the next one the second highest byte and so on:

value := read_one_byte_from_data_stream() · 2^{24} + read_one_byte_from_data_stream() · 2^{16} + read_one_byte_from_data_stream() · 2^{8} + read_one_byte_from_data_stream()

— **length** shall be initialized with $2^{32} - 1$ (i.e. 0xFFFFFFFF, a length of 1)

9.3.2. Encoding

— **base** shall be initialized with 0
— **length** shall be initialized with $2^{32} - 1$ (i.e. 0xFFFFFFFF, a length of 1)

9.4. Writing to and reading from a data stream

9.4.1. Decoding

Data is read from the encoded data stream whenever the interval size (**length**) gets lower than a threshold (note that **value** is always smaller than **length**), specifically, if **length** gets smaller than 4 bytes (which means it is smaller than 0x01000000). The bytes in **value** get shifted to the left, replacing the highest byte (which is 0 by design), and a byte is appended from the data stream.

Additionally, the **length** is scaled. This has the effect of increasing the precision (which means including more decimal places from the rational number).

function renorm_dec_interval():

— left shift "value" by 1 byte, keep at most 4 bytes
  value := (value · 2^{8}) mod 2^{32}
— append one byte from the data stream (as the new lowest byte):
  value := value + read_1_byte_from_data_stream()
— adjust interval length (precision has been adjusted):
  length := (length · 2^{8}) mod 2^{32}
— do this again until length is larger than 3 bytes:
  if length < 2^{24}
  then run algorithm renorm_dec_interval()

Figure 2 — Algorithm: renorm_dec_interval(), Pseudocode

9.4.2. Encoding

Similarly, data is written to the encoded data stream whenever the interval (**length**) gets lower than a threshold. The highest byte gets written to the data stream, the remaining bytes get shifted to the left. Additionally, **length** is adjusted.

function renorm_enc_interval():

— write the highest (of 4) bytes from "base" to the data stream:
  append_one_byte_to_data_stream($\left\lfloor \frac{\text{base}}{2^{24}} \right\rfloor$ mod 2^{8})
— left shift "base" by 1 byte (keep at most 4 bytes):
  base := (base · 2^{8}) mod 2^{32}
— adjust interval length (precision has been adjusted):
  length := (length · 2^{8}) mod 2^{32}
— do this again until length is larger than 3 bytes:
  if length < 2^{24}
  then run algorithm renorm_enc_interval()

Figure 3 — Algorithm renorm_enc_interval(), Pseudocode

Additionally, **base** (the sum of the lower interval bounds) can get to a value larger than $2^{32} - 1$ (4 bytes). In that case, 1 gets added to the previous byte (previously written to the data stream) in the following algorithm propagate_carry(). This can happen repeatedly, if the previous byte was already 255.

Note that, by design, the encoded data stream represents a rational number between 0 and 1. A situation to have to add 1 to the byte before the first byte of the data stream cannot occur. Also, any actual implementation will of course depend on the environment.

function propagate_carry():

— get the end of the data stream, i.e. the previous byte written to the stream:
  current_position := position_of_end_of_data_stream()
— if the previous byte is 255, set it to 0 and repeat:
  while byte_at_position(current_position) = 255
  do
    byte_at_position(current_position) := 0
    current_position := current_position - 1
— finally, add 1 to the byte before that:
    byte_at_position(current_position) := byte_at_position(current_position) + 1

Figure 4 — Algorithm propagate_carry(), Pseudocode

9.5. Finalization of the stream

9.5.1. Decoding

The decoder does not need a final step. It should be noted though that the data stream does not have an end marker. To know when the data stream is finished, it is required to know how much data has to be read (e.g. the number of points in a chunk), and to just stop then.

9.5.2. Encoding

This function has to be called to finish a data stream, i.e. after all data has been written.

When the encoding is done, **base** still contains a value that has to be written to the encoded data stream. To be in sync with the decoder (i.e. to always be able to fill **value** to 4 bytes), some zero-bytes are added.

function finalize_encoding_stream():

— adjust the base and length so data is written in renorm_enc_interval()
  if length > 2^{25}
  then
    b_{tmp} := base + 2^{24} // can be more than 4 bytes
    base := b_{tmp} mod 2^{32} // base uses only 4 bytes
    length := 2^{23} // writes 1 byte in renorm_enc_interval
    write2bytes := false
  else
    b_{tmp} := base + 2^{23} // can be more than 4 bytes
    base := b_{tmp} mod 2^{32} // base uses only 4 bytes
    length := 2^{15} // writes 2 bytes in renorm_enc_interval
    write2bytes := true

```
    if b_tmp ≥ 2^32 // was more than 4 bytes
    then run algorithm propagate_carry() // adds 1 to previous data
— write 1 or 2 bytes from "base" (length has been set accordingly)
    run algorithm renorm_enc_interval()
— append 2, and optionally a 3rd, zero-bytes (to be in sync with decoder)
    append_one_byte_to_data_stream(0)
    append_one_byte_to_data_stream(0)
    if write2bytes = false
    then append_one_byte_to_data_stream(0)
```

Figure 5 — Algorithm finalize_encoding_stream(), Pseudocode

9.6. Processing a data stream

9.6.1. Overview

A data stream is fed to encoders, which are specified in Clause 10. These operate on the **value**, **base** and **length**-variables (and use the functions defined above). A data stream can be shared among several encoders.

9.6.2. Decode

The decoding process for a data stream will look as follows:

- Initialize the data stream (e.g. initialize the values, and read the first 4 bytes)
- initialize all decoders (e.g. reset the distribution tables and previous items for all fields that read from this data stream)
- repeat until all expected data is processed:
  - use the next decoder in line and process and modify **value** and **length** of the data stream, read more data from the stream if needed. Which decoder to use is defined in the field description later in this specification. Update the distribution table, as specified in the encoder description.

Specifically for a data stream with compressed point data, the last 2 bullet points mean:

- repeat until all expected data is processed:
  - Read the next LAS point:
    - for that, read all LAZ items (i.e. subparts of a LAS point)
      - for that, read all fields of that item
        - for each field, use a specific decoder process and modify **value** and **length** of the data stream it belongs to. Update the distribution table, as specified in the encoder description.
        - the stored value may be the uncompressed value directly, or is used to calculate it, e.g. it might be the difference to a predicted value, for example the value of that field of the previous point, or an average over some points.
  - Stop when all points are read (which is known from the chunk size). There is no special “End of stream”-marker.

9.6.3. Encode

Symmetrically, the encoding process for a data stream will look as follows:

- Initialize the data stream
- initialize all encoders (e.g. reset the distribution tables and previous items)

— repeat until all expected data is processed:
  — use the next encoder in line and process and modify **base** and **length** of the data stream, write data to the stream if needed. Which encoder to use is defined in the field description later in this specification. Update the distribution table, as specified in the encoder description.
— Finalize the stream

9.7. Data streams in LAZ

A LAZ file can have several encoded data streams.

Each data stream has its own set of variables **value**, **base** and **length**, and has to be initialized separately. Each data stream can be read independently from all others.

Data streams used are:

— the chunk table
— the compressed item data in the chunks
  — For LAS formats 0 through 5, each chunk contains one compressed data stream
  — For LAS formats 6 through 10, each chunk contains several layers (covering a subset of fields), each of which is a separate data stream

10. Encoders

10.1. Overview

LAZ defines 4 types of encoders, which will operate on the **value**, **base** and **length**, and encode or decode one value:

— Symbol encoder: This is used to de- and encode data based on the symbol distributions. A symbol encoder has its own (or several) symbol distribution tables and parameters (i.e. number of symbols).
— Bit Symbol encoder (single bit): Special version of the symbol encoder with just 2 symbols (i.e. 1 bit), but slightly different internal behaviour.
— Raw encoder: stores the data as is, without distribution tables and without a compression effect. This could also be regarded as a symbol coder where all probabilities are the same
— Integer compressor: a compressor which itself uses all other three encoders to store an integer value.

A data stream can (and usually will) be shared by several coders. Each step will change the **value**, **base**, and **length** for the next coder in line. Those coders are then used to, for example, store the actual LAS point data items. Each field can be stored using a different encoder, as described in the following chapters.

10.2. Symbol encoder (generic)

10.2.1. Overview

The symbol encoder, or algorithmic encoder, translates the rational number (the compressed data stream) into a **symbol** (or vice versa), i.e. picks the letter from an alphabet according to the distribution. The distribution is adjusted based on the actual data read by this decoder.

The symbol decoder is defined with the following parameters:

— **symbols**: number of symbols, shall be an integer value between 1 and 1023
— a reference to the data stream object it uses (e.g. the values **base**, **value** and **length**, and the functions renorm_dec_interval(), renorm_enc_interval() and propagate_carry()), optionally shared by several decoders

and the following internal, encoder-instance specific variables:

Table 16 — Symbol Encoder internal variables

Variable	Range	Description
symbol_count[symbols]	$0 \dots 2^{16} - 1$	array that counts the occurrence of each symbols
distribution[symbols]	$0 \dots 2^{15} - 1$	array for the lower interval bounds of each symbol. In the example in Figure 1, it would store the information that the interval for A starts at probability 0 , B starts at 0.6 and C start at 0.8 (while “C ends at 1” is implied). The values are stored as integer values, where 0 represents 0 and 32768 represents 1 . Note that this has a specific rounding effect on the precision of the interval bounds. Also note that the value 32768 (an upper bound) never actually has to be stored (as these are the lower bounds, and each interval has a size > 0).
update_cycle	$0 \dots 2^{15} - 1$	number of symbols at which the next probability update should happen

Variable	Range	Description
symbols_until_update	$0 \ldots 2^{15} - 1$	number of symbols to the next update (counts down)
NOTE: All variables only need to store integer values (i.e. no decimal places). The arrays are assumed to be 0-indexed in the algorithms.		

and the following algorithms:

— decodeSymbol(): gets the next decompressed symbol
— encodeSymbol(): stores a symbol into the compressed data stream
— update_distribution(): updates the probability table after **update_cycle** symbols were read/written

10.2.2. Initialization

At the start of the de- and encoding process, the variables are initialized. Note: **symbols** is the number of symbols of the encoder.

— for all "symbol" in 0 … symbols - 1:
    symbol_count[symbol] := 1
  Note: not set to 0, to prevent intervals with length 0, which cannot be distinguished from each other with **value**
— run update_distribution() once, which initializes the distribution tables from the symbol counts.
— update_cycle := $\left\lfloor \frac{\text{symbols}+6}{2} \right\rfloor$
  Note: set after the call to update_distribution()
— symbols_until_update := update_cycle
  Note: set after the call to update_distribution()

Figure 6 — Algorithm Initialization of the generic symbol encoder, Pseudocode

10.2.3. Algorithm: update_distribution()

Updates the interval bounds based on the occurrence of the symbols. The values are stored as integer values, as described above, which has a specific rounding effect. If a maximum total value is reached, the counts are halved. The probabilities are then scaled to add up to 100%, and the intervals are calculated. Note that this has a specific rounding effect.

Used for both the encoding and decoding functions.

Algorithm update_distribution():

— halve counts when a threshold is reached:
  if $\left(\sum_0^{\text{symbols}-1}(\text{symbol\_count[i]})\right) > 2^{15}$
  then
    for all n in 0 … symbols - 1:
      symbol_count[n] := $\left\lfloor \frac{\text{symbol\_count[n]}+1}{2} \right\rfloor$
— recalculate the probabilities/interval boundaries:
  sum := 0
  total_count := $\sum_0^{\text{symbols}-1}(\text{symbol\_count[i]})$
  scale := $\left\lfloor \frac{2^{31}}{\text{total\_count}} \right\rfloor$
  for all n in 0 … symbols - 1:

distribution[n] := $\left\lfloor \frac{\text{scale} \cdot \text{sum}}{2^{16}} \right\rfloor$
sum := sum + symbol_count[n]
— set new update intervals:
update_cycle := $\left\lfloor \frac{5 \cdot \text{update\_cycle}}{4} \right\rfloor$
update_cycle := $\min(8 \cdot (\text{symbols} + 6), \text{update\_cycle})$
symbols_until_update := update_cycle

Figure 7 — Algorithm update_distribution(), Pseudocode

10.2.4. Decoding

10.2.4.1. Algorithm: decodeSymbol()

Decodes the next symbol: checks in which interval (i.e. **distribution[ ]**) the current **value** lies in, and returns it — this is the basic step described in Arithmetic Coding. Note that distribution[s] is the lower bound of the interval, distribution[s+1]-1 the upper bound. Then rescales the interval (the new interval is lower and upper bound of the chosen interval). The previous lower bound is subtracted from **value** (so **value** is only the difference from that lower bound, as an offset), **length** becomes the length of that interval. If **length** gets too low (i.e. the precision is too small), additional data is read (by appending more digits to **value**), and the interval adjusted.

Algorithm decodeSymbol(), returns an integer value "symbol":

— calculate length scaling (rounded down):
$l_{tmp} = \left\lfloor \frac{\text{length}}{2^{15}} \right\rfloor$
— select the interval in which "value" lies (scaled by the interval length), which determines the decoded "symbol":
for all s in 0 … symbols - 1:
if distribution[s] · $l_{tmp} \leq$ value
then
symbol := s
— remove the offset from "value" (the new value is within the new interval):
value := value - (distribution[symbol] · l_{tmp})
— the interval size is the bounds of that chosen interval. Special handling for last symbol (upper bound for this symbol is "length"):
if symbol < symbols - 1
then
length := (distribution[symbol+1] · l_{tmp}) - (distribution[symbol] · l_{tmp})
else
length := length - (distribution[symbol] · l_{tmp})
— if interval size gets too low (< 4 byte), add more data by calling renorm_dec_interval():
if length < 2^{24}
then run algorithm renorm_dec_interval()
— update the symbol counts:
symbol_count[symbol] := symbol_count[symbol] + 1
— check if distribution table update is needed:
symbols_until_update := symbols_until_update - 1
if symbols_until_update = 0
then run algorithm update_distribution()
— return the decoded symbol (an integer value) as the function result:
return symbol

Figure 8 — Algorithm decodeSymbol(), Pseudocode

10.2.5. Encoding

10.2.5.1. Algorithm: encodeSymbol()

Encoding is exactly symmetrical to decoding. It operates on **base** and **length** instead of **value** and **length**. **base** adds up the lower bound of the interval, i.e. it is possible to calculate **value** by subtracting the lower bounds (the distribution[symbol]) from **base**, and vice versa.

The scaling is done exactly as in the decoding process. As soon as the interval gets too small, data is written and the interval is adjusted.

function encodeSymbol(symbol):

— function is called with an integer parameter "symbol", values shall be 0 … symbols - 1
— calculate length scaling (rounded down):
$l_{tmp} := \left\lfloor \frac{length}{2^{15}} \right\rfloor$
— get the lower bound of the interval that symbol belongs in:
$d_{lower} := distribution[symbol] \cdot l_{tmp}$
— Check if sum is larger than 4 bytes, then add 1 to previously written data stream:
if $base + d_{lower} \geq 2^{32}$
then run algorithm propagate_carry()
— adjust the global variable "base", only use lower 4 bytes (i.e. & 0xFFFFFFFF):
$base := (base + d_{lower}) \bmod 2^{32}$
— new length is upper bound - lower bound. For last symbol, upper bound is "1":
if symbol = symbols - 1
then
length := length - d_{lower}
else
length := distribution[symbol+1] $\cdot\, l_{tmp} - d_{lower}$
— if interval size (precision) gets too small (less than 4 bytes), write data to data stream:
if length < 2^{24}
then run algorithm renorm_enc_interval()
— update the symbol counts:
symbol_count[symbol] := symbol_count[symbol] + 1
— check if distribution table update is needed:
symbols_until_update := symbols_until_update - 1
if symbols_until_update = 0
then run algorithm update_distribution()

Figure 9 — Algorithm encodeSymbol(symbol), Pseudocode

10.3. Bit Symbol encoder (for a single bit)

10.3.1. Overview

This is a simpler version of the generic symbol encoder for a single bit (i.e., a 2 symbol encoder). LAZ uses it only in the Integer Compressor.

NOTE: Compared to a generic symbol encoder with 2 symbols, this special 1 bit encoder uses slightly different thresholds and initial values (e.g. the right shift is by 13 bits, and the maximum count for halving the probabilities is lower). The generic symbol encoder with 2 symbols is also used by LAZ (also for the Integer Compressor), so there are 2 different ways to encode 2 symbols.

The bit symbol encoder operates on a data stream, i.e. its values **base**, **value** and **length** and the functions renorm_dec_interval(), renorm_enc_interval() and propagate_carry().

The following internal, encoder-instance specific variables are defined:

Table 17 — Bit Symbol Encoder internal variables

Variable	Range	Description
bit_0_count	0… $2^{16}-1$	the number of occurrence of the value 0
bit_count	0… $2^{16}-1$	total number of bits
bit_0_prob	0… $2^{13}-1$	upper interval bound for bit 0, stored as an integer, where 0 represent 0 and 8192 represents 1 (a value which itself is never stored in this variable). Note that this has a specific rounding effect on the precision of the interval bound.
update_cycle	0… $2^{16}-1$	number of symbols at which the next probability update should happen
bits_until_update	0… $2^{16}-1$	number of bits to the next update (counts down)
NOTE: All variables only need to store integer values (i.e. no decimal places).		

and the following algorithms:

— decodeBit(): gets the next decompressed symbol (i.e. the next bit)
— encodeBit(): stores a symbol (bit) into the compressed data stream
— update_bit_distribution(): updates the probabilities after **update_cycle** bits were read/written

10.3.2. Initialization

At the start of the de- and encoding process, the variables are initialized with:

— bit_0_count := 1
— bit_count := 2 (i.e. bit 0 and 1 start with equal distribution)
— bit_0_prob := 4096 (i.e. a probability of $\frac{4,096}{8,192} = 0.5$)
— update_cycle := 4
— bits_until_update := 4

Figure 10 — Algorithm Initialization of the bit symbol encoder, Pseudocode

10.3.3. Algorithm: update_bit_distribution()

Similar to update_distribution() in the generic symbol encoder, update_bit_distribution() updates the interval bounds. If a maximum total value is reached, the counts are halved.

Used for both the encoding and decoding functions.

function update_bit_distribution():

— halve counts when the threshold is reached:
  bit_count := bit_count + update_cycle
  if bit_count > 8192
  then

```
        bit_count := ⌊(bit_count+1)/2⌋
        bit_0_count := ⌊(bit_0_count+1)/2⌋
        // prevent interval sizes of 0
        if bit_0_count = bit_count
        then bit_count := bit_count + 1
— recalculate the probabilities:
    bit_0_prob := ⌊(⌊2^31/bit_count⌋ · bit_0_count)/2^18⌋
— set new update frequencies:
    update_cycle := ⌊(5·update_cycle)/4⌋
    if update_cycle > 64
    then update_cycle := 64
    bits_until_update := update_cycle
```

Figure 11 — Algorithm update_bit_distribution(), Pseudocode

10.3.4. Decoding (Bit Symbol Encoder)

Just as for decodeSymbol(), it decodes (and returns) a single bit based on the (regularly updated) bit distribution, and adjusts the interval. Note that it uses slightly different thresholds.

```
function decodeBit():
— decode bit by checking which interval "value" is in:
    if value ≥ bit_0_prob · ⌊length/2^13⌋
    then
        bit := 1
    else
        bit := 0
— set the new interval:
    if bit = 0
    then
        length := bit_0_prob · ⌊length/2^13⌋
    else
        value := value - bit_0_prob · ⌊length/2^13⌋
        length := length - bit_0_prob · ⌊length/2^13⌋
— if interval size gets too small (less than 4 bytes), read more data:
    if length < 2^24
    then run algorithm renorm_dec_interval()
— count the bit occurrence:
    if (bit = 0)
    then
        bit_0_count := bit_0_count + 1
— check for symbol update interval:
    bits_until_update := bits_until_update - 1
    if bits_until_update = 0
    then run algorithm update_bit_distribution()
```

— return the bit
  return bit

Figure 12 — Algorithm decodeBit(), Pseudocode

10.3.5. Encoding (Bit Symbol Encoder)

The encoding for a single bit works symmetrical to the decoding of a single bit. Just as the bit decoder, it uses slightly different thresholds compared to the generic symbol encoder.

function encodeBit(bit):

— one parameter "bit", values 0 or 1
— adjust base (if bit is 0, base doesn't change, as lower bound is 0):
  if bit = 1
  then
    if base + $\text{bit\_0\_prob} \cdot \left\lfloor \frac{\text{length}}{2^{13}} \right\rfloor \geq 2^{32}$
    then run algorithm propagate_carry()
    base := $\left(\text{base} + \text{bit\_0\_prob} \cdot \left\lfloor \frac{\text{length}}{2^{13}} \right\rfloor\right) \bmod 2^{32}$
— adjust the new interval length:
  if bit = 0
  then
    length := $\text{bit\_0\_prob} \cdot \left\lfloor \frac{\text{length}}{2^{13}} \right\rfloor$
  else
    length := length - $\text{bit\_0\_prob} \cdot \left\lfloor \frac{\text{length}}{2^{13}} \right\rfloor$
— if interval size (precision) gets too small (less than 4 bytes), write data to data stream:
  if length < 2^{24}
  then run algorithm renorm_enc_interval()
— count the bit occurrence
  if (bit = 0)
  then
    bit_0_count := bit_0_count + 1
— check for symbol update interval:
  bits_until_update := bits_until_update - 1
  if bits_until_update = 0
  then run algorithm update_bit_distribution()

Figure 13 — Algorithm encodeBit(bit), Pseudocode

10.4. Raw encoder

10.4.1. Overview

The Raw encoder stores bits "as is" in the data stream.

It does not use any distributions and doesn't require an initialization or internal state variables. It works directly with the **value**, **base** and **length** variables of a given data stream.

For data with a size of more than 19 bits, the lowest 16 bits are stored first in the data stream, recursively.

The LAZ specification uses the raw encoder for at most 64 bits.

10.4.2. Decoding (Raw encoder)

The logic is similar to the Algorithmic Decoder and the decodeSymbol()-Algorithm, but doesn't require a symbol lookup, but takes the data as is.

Note that "overwriting" the value doesn't change the previously encoded symbol (as the value can be chosen freely within the interval, which includes the scaled raw value). Since all bits of a raw value are required though, **length** will be adjusted by the complete size of the encoded value (e.g. by 16 bits for a 16 bit integer), and there is no compression effect.

readBits(bit_count) reads bit_count raw bits from the data stream. For more than 19 bits, the lowest 16 bits are read first (by calling readBits() recursively). Afterwards, the remaining bits are read recursively.

The function renorm_dec_interval() has been defined in Clause 9.

function readBits(bit_count):

— parameter with the number of bits to read (1 .. 64)
— if more than 19 bits, read 16 bits first, and then the rest, recursively:
  if bit_count > 19
  then
    return (readBits(16) + readBits(bit_count - 16) · 2^{16})
    exit function
— otherwise, read them as is:
  raw := $\left\lfloor \frac{\text{value}}{\left\lfloor \frac{\text{length}}{2^{\text{bit\_count}}} \right\rfloor} \right\rfloor$
— adjust interval:
  length := $\left\lfloor \frac{\text{length}}{2^{\text{bit\_count}}} \right\rfloor$
  value := $\text{value} - \text{length} \cdot \text{raw}$
— if interval size gets too small (less than 4 bytes), read more data:
  if $\text{length} < 2^{24}$
  then run algorithm renorm_dec_interval()
— return it:
  return raw

Figure 14 — Algorithm readBits(bit_count), Pseudocode

10.4.3. Encoding (Raw encoder)

Encoding a value **sym** is completely symmetrical, operating on **base** and **length**.

The function renorm_enc_interval() and propagate_carry() have been defined in Clause 9.

Analogously to readBits(bit_count), writeBits(bit_count, sym) writes bit_count raw bits to the data stream. For more than 19 bits, the lowest 16 bits are written first, and then the remaining bits, recursively.

function writeBits(bit_count, sym):

— called with "bit_count" (the number of bits of "sym" to store), and sym, an unsigned integer value with at most the lowest "bit_count" bits used.
— if more than 19 bits, write lowest 16 bits first, and then the rest, recursively:
  if bit_count > 19
  then

run writeBits(16, $\text{sym} \bmod 2^{16}$)
run writeBits(bit_count - 16, $\left\lfloor \frac{\text{sym}}{2^{16}} \right\rfloor$)
exit function

— adjust interval:
length := $\left\lfloor \frac{\text{length}}{2^{\text{bit\_count}}} \right\rfloor$
— sum larger than 4 bytes? Then add 1 to the previous byte in the data stream
if $\text{base} + \text{sym} \cdot \text{length} \geq 2^{32}$
then run algorithm propagate_carry()
— encode the raw value as is:
base := $(\text{base} + \text{sym} \cdot \text{length}) \bmod 2^{32}$
— if interval size gets too small (less than 4 bytes), write some data:
if $\text{length} < 2^{24}$
then run algorithm renorm_enc_interval()

Figure 15 — Algorithm writeBits(bit_count, sym), Pseudocode

10.5. Integer Compressor

10.5.1. Overview

The Integer Compressor is a special encoder that uses a combination all 3 of the previously specified encoders, i.e. the generic symbol encoder, the single bit symbol encoder and the raw encoder, to store a signed integer number (4 bytes).

LAZ uses this compressor to store the difference between a number and a predecessor (or some other predicted value). It is designed to store values close to 0 more efficiently. That means that, for example, monotonically increasing values (like x coordinates when flying over an area) oftentimes have a good compression rate, while for randomly distributed values, the integer compression can even require more bits than the original number.

LAZ uses it to store the difference between 8-, 16-, and 32-bit integer values. As a convention, for 32-bit values, LAZ uses signed integers for the number and the predecessor, while for 8-bit and 16-bit values, LAZ uses unsigned integers as the number and the predecessor — while their difference, i.e. the actually stored value using the compressor, can be negative. To enforce these conventions and get the correct target ranges, the calculation of the difference (i.e. the value that is actually stored) is specified in IDiff8(), IDiff16() and IDiff32() for encoding, and in ISum8(), ISum16() and ISum32() for the reverse operation during decoding (i.e. calculating the final value after reading the stored difference from the data stream), see Clause 10.5.4.

To actually encode the integer value (i.e. the difference), LAZ first encodes the number k that describe the tightest interval $[-(2^k - 1), +(2^k)]$ that the value falls into, entropy codes up to 8 of its highest bits as one symbol, and stores any remaining lower bits using the raw encoder.

An n-bit integer compressor consists of

— a symbol encoder with n+1 symbols, which encodes k as defined above
— a raw encoder, to store the lowest k-8 bits (if needed)
— n+1 symbol encoders, one for each k, to store up to 8 highest bits. For k = 0, the special single bit Symbol encoder is used. For $1 \leq k \leq 8$, the (generic) symbol encoders with 2^k symbols are used. For $k > 8$, the (generic) symbol encoders with 256 symbols are used. E.g., for a 32-bit integer encoder, the 33 encoders have 2 (single bit), 2 (generic), 4, 8, 16, 32, 64, 128, 256, 256, … 256 symbols. Note that (only) for k = 32, the last symbol encoder is not actually used, as it is mapped to a constant value, see below.

In contrast to other encoder types, these n+1 encoders are shared among instances (of the same context) for one field. E.g., a field which uses 10 instances of a 32-bit Integer Encoder uses 10 separate symbol encoders (each with their own distributions) to store k, and these 33 encoders (with 33 distributions), shared between the 10 instances, to store up to 8 highest bits, and finally, a raw encoder.

Some fields use the **k**-value of the previous record in their calculation. The k-value is also shared among different instances (of the same context) of the same field.

10.5.2. Encoding format

Encoded are three things:

— the number k of low-order bits and
— the k-bit number that corrects the misprediction, usually broken into two chunks:
  — the highest 8 bits are compressed using an arithmetic coder
  — the lower bits, if any remain, are stored raw

For a given k, only the values that do not lie in the intervals with smaller k need to be considered (as those would have used a smaller k by definition), and to utilize the full range, for $1 \leq k \leq 31$, the interval $[-(2^k - 1), +(2^k)]$ is mapped to the interval $[0,\ 2^k - 1]$.

For an n-bit Integer Compressor, the data is stored as:

Table 18 — Format of the encoded and compressed n-bit integer value (Integer Compressor)

Field Name	Type	Size	Encoder	No of instances	Required
k	byte	1 byte	n+1 Symbols	1	*
Corrector	byte	1 byte	Bit Symbol; 2..256 Symbols	n+1, shared	
lower bits	unsigned int	k-8 bits	Raw		

The notation used in this table is described in Clause 10.6.

k: The number k as defined above, encoded using a symbol encoder with n+1 symbols.

Corrector: Only stored if $k < 32$. Encoded using the symbol encoder number **k**. For k = 0, the bit symbol encoder is used (to store a single bit), for $1 \leq k < 8$, the (generic) symbol encoder with 2^k symbols is used, and for $k \geq 8$, symbol encoders with 256 symbols each are used.

The **Corrector** encodes up to the 8 most significant bits using the process described below (decoding and encoding).

lower bits: Only stored if k > 8 (i.e. if the **Corrector** couldn't store all bits). The k-8 bits are stored using a raw encoder for k-8 bits.

10.5.3. Decoding (Integer compressor)

The signed 32-bit integer value **c** is calculated as follows:

— if **k** is 0, set **c** := **Corrector** (which is either 0 or 1), using the bit symbol encoder.
— if **k** is 32, set **c** := -2^{31} (i.e. the minimum for a signed 4 byte integer value)
— if **k** ≥ 1 and **k** ≤ 31 (using the **k**-th symbol encoder for **Corrector**):
  — if **k** ≤ 8, set **c** := **Corrector**
  — if **k** > 8, set **c** := **Corrector** · 2^{k-8} + **lower bits**
  — move the value back to the interval $[-(2^k - 1), +(2^k)]$:

— if **c** ≥ 2^{k-1} , set **c** := **c** + 1
— else set **c** := **c** – $(2^k - 1)$

10.5.4. Calculating the difference

LAZ stores only the difference **c** between a predecessor (or predicted value) **pred** and the signed 32-bit integer, actual value **intvalue** (even if **pred** is 0), i.e. **intvalue** = **pred** + **c**. This calculation is done using 32-bit signed variables, and specifically wraps around.

Additionally, for 8- and 16-bit integer decoders, due to the convention declared above, this difference is then mapped to the unsigned integer range, as specified in ISum8() and ISum16().

For the 32 bit Integer compressor, this is specified for decoding (i.e. reading from the stream) as follows:

function ISum32(pred, c):

— parameters are: the predecessor "pred" and the decompressed stored value "c" (both in the range -2^{31} ... $+ 2^{31} - 1$)
— calculate the sum "intvalue = pred + c" with wrap around:

```
if (pred + c) > 2^31 – 1
then
   return pred + c - 2^32
else if (pred + c) < –2^31
then
   return pred + c + 2^32
else
   return pred + c
```

Figure 16 — Algorithm ISum32(pred, c), Pseudocode

Note that this is, written as C-Code, the same as (I32)intvalue = (I32)pred + (I32)c, and just wants to clarify how to use the stored value of the integer compressor.

For 8- and 16-bit integer decoders, the calculation is also done using the 32-bit signed variables, with an additional mapping afterwards, specified in ISum8() and ISum16(). Note that this mapping has to be done even if the predecessor **pred** is 0 (so even if there is no actual difference to calculate), to ensure the result is not a negative value.

function ISum16(pred, c):

— parameters are: the predecessor "pred" ($0 \ldots 2^{16} - 1$) and the decompressed stored value "c" (-2^{15} ... $+ 2^{15} - 1$)
— get the sum using 32 bit integers:
  intvalue := ISum32(pred, c)
— apply an additional mapping into the correct range:

```
  if intvalue < 0
  then
     intvalue := intvalue + 2^16
  else if intvalue ≥ 2^16
  then
     intvalue := intvalue –2^16
```

— and return that value
  return intvalue

Figure 17 — Algorithm ISum16(pred, c), Pseudocode

function ISum8(pred, c):

— parameters are: the predecessor "pred" ($0 \ldots 2^8 - 1$) and the decompressed stored value "c" ($-2^7 \ldots + 2^7 - 1$)
— get the sum using 32 bit integers:
  intvalue := ISum32(pred, c)
— apply an additional mapping into the correct range:
  if intvalue < 0
  then
    intvalue := intvalue + 2^8
  else if intvalue $\geq 2^8$
  then
    intvalue := intvalue -2^8
— and return that value
  return intvalue

Figure 18 — Algorithm ISum8(pred, c), Pseudocode

For encoding, first, the difference to the predecessor “pred”, and (for 8 and 16 bit encoders) the mapping from unsigned integers, is calculated, and only this resulting signed integer value **c** is stored in the data stream. The difference and mapping is specified analogously to decoding:

function IDiff32(pred, intvalue):

— parameters are: the predecessor "pred" and the to-be-stored integer value "intvalue" (both in the range $-2^{31} \ldots + 2^{31} - 1$)
— calculate the difference "c = intvalue - pred" with wrap around:

```
if (intvalue - pred) > 2^31 − 1
then
   return intvalue - pred - 2^32
else if (intvalue - pred) < −2^31
then
   return intvalue - pred + 2^32
else
   return intvalue - pred
```

Figure 19 — Algorithm IDiff32(pred, intvalue), Pseudocode

Just as for decoding, note that this is, written as C-Code, just (I32)c = (I32)intvalue — (I32)pred.

For 8- and 16-bit integer encoders, the additional mapping is required as specified in IDiff8() and IDiff16():

function IDiff16(pred, intvalue):

— parameters are: the predecessor "pred" and the to-be-stored integer value "intvalue" (both in the range $0 \ldots 2^{16} - 1$)

— get the sum using 32 bit integers:
  c := IDiff32(pred, intvalue)
— apply an additional mapping into the correct range:
  if c $< -2^{15}$
  then
    c := c + 2^{16}
  else if c $> 2^{15} - 1$
  then
    c := c -2^{16}
— and return that value
  return c

Figure 20 — Algorithm IDiff16(pred, intvalue), Pseudocode

function IDiff8(pred, intvalue):

— parameters are: the predecessor "pred" and the to-be-stored integer value "intvalue" (both in the range $0 \ldots 2^8 - 1$)
— get the sum using 32 bit integers:
  c := IDiff32(pred, intvalue)
— apply an additional mapping into the correct range:
  if c $< -2^7$
  then
    c := c + 2^8
  else if c $> 2^7 - 1$
  then
    c := c -2^8
— and return that value
  return c

Figure 21 — Algorithm ISum8(pred, intvalue), Pseudocode

10.5.5. Encoding (Integer compressor)

Compressing works completely symmetrically to encode a value **c**, that has been calculated as the difference between an integer value and the predecessor using IDiff32, IDiff16 or IDiff8.

NOTE: Analogous to the decoder, for 8-bit and 16-bit integers, the mapping described in Clause 10.5.4, i.e. one of IDiff32, IDiff16 or IDiff8, has to be used first, even if the predicted value is “0”.

With the value **c**:

— find the value for **k**, so that the interval $[-(2^k - 1), +(2^k)]$ is the tightest interval that contains **c**
— encode **k** using the symbol encoder with n+1 symbols
— if **k** is 0 (which means **c** is 0 or 1), encode that bit as the field **Compressor** using the bit symbol encoder. No **lower bits** field is stored
— else if **k** is 32 (i.e. **c** is -2^{31}), it’s done, no **Compressor** or **lower bits** fields are stored
— else
  — map the value from $[-(2^k - 1), +(2^k)]$ to the interval $[0, 2^k - 1]$:
    — if **c** < 0, set **c** := **c** + $2^k - 1$
    — else set **c** := **c** -1

— if k ≤ 8, encode **c** as the field **Compressor** := **c** using the **k**-th symbol encoder (i.e. the symbol encoder with 2^k symbols). No **lower bits** field is stored.
— if k > 8, encode $\left\lfloor \frac{c}{2^{k-8}} \right\rfloor$ as the field **Compressor** using the **k**-th symbol encoder (which has 256 symbols). The field **lower bits** is set to **lower bits** := **c** mod $\left(2^{k-8}\right)$ and stored using a raw encoder with k-8 bits.

10.6. Encoder notation and encoding overview

The following chapters describe how these encoders are used to compress and decompress the data.

In general, a compressed LAZ item field is stored using a specific encoder, e.g. a symbol encoder with 256 bytes.

There can be several instances of that symbol encoder that are used for the same field. Each instance has its own distribution table and symbol count and is initialized independently. Which instance is used to decode or encode the field is part of the field description. It can e.g. be chosen based of the value of the previous or current point or some other calculation.

E.g., in an object oriented programming language, to encode a field with 256 instances of a symbol encoder, one could instantiate 256 symbol-encoder-objects, and pick the correct instance based on the situation.

In case of an Integer Compressor, the (up to 33) symbol encoders for the k-value are shared among those instances, as described in Clause 10.5.

Only the selected instance is used and it is fed the compressed data value, returning a decompressed value, and only adjusting the internal variables of that instance.

This value will then be used to calculate the decompressed LAS field. In many cases, it will be added to the value of that field of the previous (decompressed) point, so the compressed field will only store the difference between two consecutive points. For some fields, a different logic can be used, for example the difference to the average value of several previous points, or the difference to a previous point with a specific characteristic.

Which calculation and which previous point(s) are used is specified in the field descriptions.

As an example, the following notation would describe two fields of an uncompressed LAS data format,

Table 19 — Notation example for an uncompressed LAS format

Field Name	Format	Size	Required
Classification	unsigned char	1 byte	*
Scan Angle Rank	char	1 byte	*

which would for example be stored based on the following notation:

Table 20 — Notation example for a compressed LAZ item for the example LAS data

Field Name	Type	Size	Encoder	No of instances	Required
Classification	unsigned char	1 byte	256 Symbols	256	
dScan Angle Rank	unsigned short	1 byte	256 Symbols	2	

Those can have for example the following field specifications:

Classification: Stored using a symbol encoder with 256 symbols and 256 instances. The instance is selected by the value of **Classification** of the previous item. The uncompressed value is stored as is, i.e. it is the value of the uncompressed LAS field **Classification**.

dScan Angle Rank: Stored using a symbol encoder with 256 symbols and 2 instances. If **Scan Angle Rank** of the previous item was ≤ 128 , the first instance is used, if **Scan Angle Rank** of the previous item was > 128 , the second instance is used. The uncompressed value is stored as the difference to the previous point, i.e. the uncompressed value **Scan Angle Rank** := (**Scan Angle Rank (previous point)** + **dScan Angle Rank**) MODULO 256.

This means:

The fields are (de-)compressed with the specified coder of the given size, i.e. here in both cases a symbol encoder with 256 symbols to store a byte value. Other compressors are the Integer compressor (with a specific byte size) and the RAW encoder (also with a specific byte size).

The **Classification**-field would have 256 instances of that symbol encoder, each with its own distribution tables. The field description specifies that the instance to use is chosen by the **Classification** value of the previous point, which means that if the previous point had, for example, the **Classification** value of 142, the 142nd instance of the compressor shall be used.

To get to the uncompressed LAS data, the LAZ field may have to be processed further. E.g., in case of **Scan Angle Rank**, the uncompressed LAS field is calculated as the sum modulo 256 of the value from the previous point and the value of **dScan Angle Rank**, as only the difference is stored.

The column **Required** indicates that this field might not actually be stored. It can depend on the content of another field, e.g., some bit fields are used to store the information if specific fields are unchanged, and if they are not changed, they shall not be stored (and thus also not read). This also applies to fields or values which are noted as skipped or unused, usually depending on conditions in the description of the fields, they are also neither stored while encoding nor shall they be read during decoding.

Figure 22 shows a graphical overview of this situation.

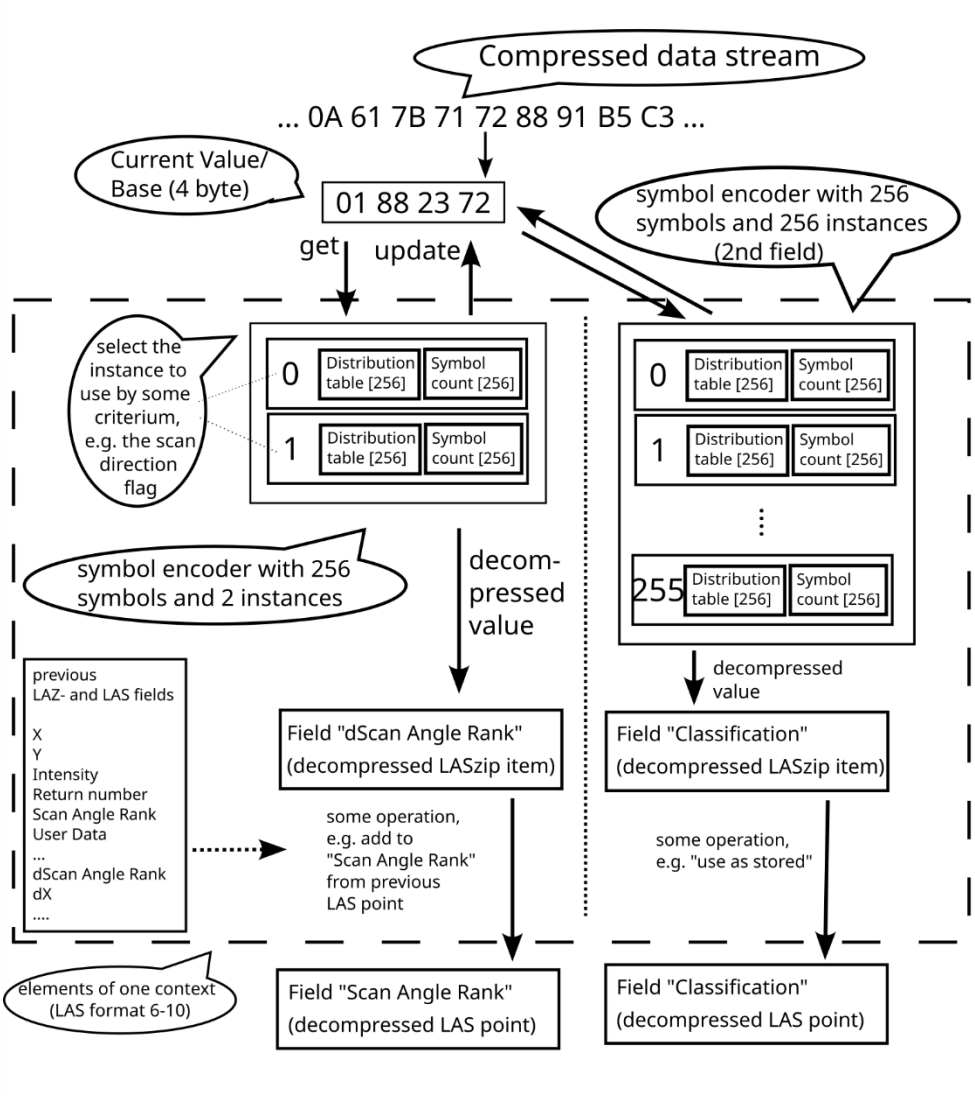

Figure 22 — Example process for decoding a compressed data stream for 2 sample fields

For LAZ items Point14, RGB14, RGBNIR14, Byte14 and Wavepacket14 (corresponding to LAS record format types 6 to 10), 4 “contexts” are used. A context includes their own copy of all the instances of all encoders for all fields, the previous items (i.e. previous items are context dependent) and all intermediate values (for example, the calculation of mean values). The context to use for the next point is based on some criterion. They share the data stream though, as specified in Clause 12.2.

10.7. Compressing and decompressing a field

Compressing and decompressing a specific field means to run the encoding or decoding algorithms for that field, as declared in the previous chapters.

The encoder to use is specified in the field specification as the “encoder”.

Depending on the encoder, the following algorithms shall be used to compress or decompress a field:

Table 21 — Algorithms to use to (de-)compress a field, depending on encoder type

Encoder	Decoding Algorithm	Encoding Algorithm
Symbol	decodeSymbol()	encodeSymbol()
Bit Symbol	decodeBit()	encodeBit()
Raw	decodeBits()	encodeBits()
Integer Compressor, 32-bit	Integer Decoder and ISum32()	IDiff32() and Integer Encoder
Integer Compressor, 16-bit	Integer Decoder and ISum16()	IDiff16() and Integer Encoder
Integer Compressor, 8-bit	Integer Decoder and ISum8()	IDiff8() and Integer Encoder

11. LAZ Compressed Data Block and Chunk Table

11.1. Overview

The compressed data block is located at the position **Offset to point data** (as declared in the LAS/LAZ header) and replaces the uncompressed point data of the LAS format. It is the central element of the LAZ format.

Depending on the field **Compressor** in the LAZ Special VLR, the block is structured differently:

— for “Pointwise compression” (LAS record format types 0-5 only), only a single chunk is used and no chunk table is included
— for “Pointwise and chunked compression” (LAS record format types 0-5 only), the points are divided into chucks, and a chunk table is included
— for “Layered and chunked compression” (LAS record format types 6-10 only), the points are divided into chucks, a chunk table is included, and each chunk is divided again into layers (which contain a subset of fields, but for all points), and each chunk includes a layer table

11.2. Pointwise compression

For pointwise compression, the compressed data block for LAS record points format 0-5 uses a single chunk and no chunk table and is structured the following way:

Table 22 — Overview LAZ Compressed Data Block for LAS record points 0-5, Pointwise compression

Chunk 1 (only 1 chunk exists)	1st record, uncompressed	Item 1 (Point10)
		Item 2 (e.g. RGB12)
		… other items
	Compressed item data	

11.3. Pointwise and chunked compression

The compressed data block for LAS record points format 0-5 with chunks (Pointwise and chunked compression) is structured the following way:

Table 23 — Overview LAZ Compressed Data Block for LAS record points 0-5, Pointwise and chunked compression

Chunk table start position		
Chunk 1	1st record, uncompressed	Item 1 (Point10)
		Item 2 (e.g. RGB12)
		… other items
	Compressed item data	
Chunk 2	1st record, uncompressed	Item 1 (Point10)
		Item 2 (e.g. RGB12)
		… other items
	Compressed item data	
Chunk n	…	
Chunk table	Version	
	Number of chunks	
	Compressed data with chunk sizes in bytes and record counts	

11.4. Layered and chunked compression

The compressed data block for points 6-10 uses layers and is structured the following way:

Table 24 — Overview LAZ Compressed Data Block for LAS record points 6-10, layered compression

<table>
<tr><td colspan="3">Chunk table start position</td></tr>
<tr><td rowspan="8">Chunk 1</td><td rowspan="3">1st record, uncompressed</td><td>Item 1 (Point14)</td></tr>
<tr><td>Item 2 (e.g. RGB14)</td></tr>
<tr><td>… other items</td></tr>
<tr><td colspan="2">Chunk size (record count)</td></tr>
<tr><td rowspan="3">Layer table (layer sizes in bytes)</td><td>Size of Layer 1</td></tr>
<tr><td>Size of Layer 2</td></tr>
<tr><td>…</td></tr>
<tr><td colspan="2">Layer 1: compressed item data (with subset of fields)</td></tr>
<tr><td colspan="2">Layer 2: compressed item data (with subset of fields)</td></tr>
<tr><td colspan="2">Layer 3: compressed item data (with subset of fields)</td></tr>
<tr><td colspan="2">… remaining layers</td></tr>
<tr><td rowspan="8">Chunk 2</td><td rowspan="3">1st record, uncompressed</td><td>Item 1 (Point14)</td></tr>
<tr><td>Item 2 (e.g. RGB14)</td></tr>
<tr><td>… other items</td></tr>
<tr><td colspan="2">Chunk size (record count)</td></tr>
<tr><td rowspan="3">Layer table (layer sizes in bytes)</td><td>Size of Layer 1</td></tr>
<tr><td>Size of Layer 2</td></tr>
<tr><td>…</td></tr>
<tr><td colspan="2">Layer 1: compressed item data (with subset of fields)</td></tr>
<tr><td colspan="2">Layer 2: compressed item data (with subset of fields)</td></tr>
<tr><td colspan="2">Layer 3: compressed item data (with subset of fields)</td></tr>
<tr><td colspan="2">… remaining layers</td></tr>
<tr><td>Chunk n</td><td colspan="2">…</td></tr>
<tr><td rowspan="3">Chunk table</td><td colspan="2">Version</td></tr>
<tr><td colspan="2">Number of chunks</td></tr>
<tr><td colspan="2">Compressed data with chunk sizes in bytes and record counts</td></tr>
</table>

11.5. LAZ Compressed Data Block Specification

The point data is stored in chunks. After each chunk, the compressors, distribution tables and all chunk specific settings (e.g. the “last item”) are reset and reinitialized.

The chunk table allows to start reading at a later chunk and to skip records. The chunks are stored sequentially and shall not have gaps between them.

In case of adaptive chunk sizes (only supported for LAS record type formats 6 through 10), the sizes can vary, and the actual chunk size is stored in the chunk table; otherwise, each chunk contains the same number of points (**Chunk size** as specified in the LAZ VLR, Clause 7) and the chunk table does not store this (constant) value.

For **Pointwise and chunked compression** and **Layered and chunked compression**, a chunk table is included, and the compressed data block is defined as follows:

Table 25 — LAZ compressed data block format, chunked compression

Field Name	Format	Size	Required
Chunk table start position	long long	8 bytes	*
Chunks	Array of Chunk		*
Chunk table	Chunk table format		*

Chunk table start position: Only stored if **Pointwise and chunked compression** or **Layered and chunked compression** is used. Absolute position in the file where the chunk table is stored. If -1, the actual value of the Chunk table start position is stored in the last 8 bytes of the file, the optional field **Chunk table start position (EOF)** (This can be used, for example, if the file isn't seekable, so the field **Chunk table start position** cannot be written after the point data has been written).

Chunk table start position (EOF): Value for **Chunk table start position**, stored at the end of the file (for example, if the file is not seekable while writing). Only stored if **Chunk table start position** is -1.

Chunks: An array of chunks which store the compressed data points. There are **Number of chunks** chunks (as declared in the chunk table) or 1 chunk for **Pointwise compression**. Can contain 0 chunks (i.e., it can be empty), if no points are contained in the file.

The chunks are stored in consecutive order, without gaps, and the contained compressed data stream can be read independently from each other, as the compressors reset for each new chunk. The position of each chunk is specified in the Chunk Table, Clause 11.6. Its format is specified in Clause 11.7.

Pointwise compression only has a single chunk and no chunk table, and the compressed data block is defined as follows:

Table 26 — LAZ compressed data block format, pointwise compression

Field Name	Format	Size	Required
Chunks	Chunk		*

11.6. Chunk table

Chunk table: List of information about the chunks:

Table 27 — Chunk table format

Field Name	Format	Size	Required
Version	unsigned long	4 bytes	*
Number of chunks	unsigned long	4 bytes	*
Compressed Chunk Table Data			

The chunk table follows directly after the chunks, and the absolute position in the file is given by the **Chunk table start position** or, if this is -1, by **Chunk table start position (EOF)**.

Version: Version of chunk table. Shall be "0".

Number of chunks: Number of chunks. Is allowed to be 0 (for a file without any points), in which case no compressed chunk table data is stored.

Compressed Chunk Table Data: Compressed list of unsigned long integer values. The data is compressed using one 32-bit Integer Compressor, with 2 instances. Only stored if **Number of chunks** is not 0.

Table 28 — Chunk Table Data Entry compression format

Field Name	Type	Size	Encoder	No of instances	Required
Chunk Table Data	unsigned long[]	4 bytes	Integer Compressor, 32 bits	2	*

Chunk Table Data: The data is an array of unsigned integer values, either 1 or 2 values per **number of chunks**.

If adaptive chunk sizes are not used, i.e. **Chunk size** (as specified in the LAZ VLR in Clause 7) is not equal to ($2^{32} - 1$), the integer values store the difference of the chunk size in bytes to the chunk size in bytes of the previous chunk (or 0 for the first chunk). The same compressor is used for all values (i.e. one instance is unused):

Table 29 — Chunk Table Data Entry, without adaptive chunk sizes

Field Name	Format	Size	Required
dChunk byte size	long	4 bytes	*

If adaptive chunk sizes are used (i.e. **Chunk size** = $2^{32} - 1$), the integer values store the number of points in the first chunk, then the size of the first chunk in bytes, then the difference of number of points in the 2nd chunk to the number of points in the first chunk, then the difference of chunk size in bytes of the 2nd chunk to the chunk size in bytes of the first chunk, and so on. For the number of points and for the chunk in bytes, a different compressor is used:

Table 30 — Chunk Table Data Entry, with adaptive chunk sizes

Field Name	Format	Size	Required
dChunk count	long	4 bytes	*
dChunk byte size	long	4 bytes	*

dChunk count: Only stored if adaptive chunk sizes are used. Difference of the number of points in this chunk to the number of points in the previous chunk, i.e. **Chunk count** := ISum32(**Chunk count (previous chunk)**, **dChunk count**). Compressor 1 is used for this field. For non-adaptive chunks, each chunk (apart from the last one) has the same number of points, declared by the global **Chunk size**, and **dChunk count** is not stored. The sum **Chunk count** has to be larger than 0, i.e. each chunk has to contain at least 1 point.

dChunk byte count: Difference of the size of the chunk in bytes to the size of the chunk in bytes of the previous chunk (or 0 for the first chunk), i.e. **Chunk byte count** := ISum32(**Chunk byte count (previous chunk)**, **dChunk byte count**). Compressor 2 is used for this field.

NOTE: The absolute chunk positions in the file are thus the position of the field **Chunk table start position** (given by the **Offset to point data** from the LAZ header) + 8 for the first chunk, and for the next chunk incremented by **Chunk byte count** of the previous chunk.

11.7. Chunk format

The chunks are stored consecutively, in order, and without gaps. For each chunk, the compression resets: all compressors and distribution tables are reinitialized, and the previous items are cleared.

Each chunk can be decoded independently, the position of each chunk is given by the chunk table. For **Pointwise Compression**, all data is stored in a single chunk, and no chunk table is stored.

Each chunk starts with one uncompressed item record for each occurring item. The uncompressed item form is specified in Clause 12.

For LAS points 0 to 5, the remaining data is one compressed data stream shared by all items and encoders that read from it.

For LAS points 6 to 10, layered compression is used (mandatory). Each item is stored separately in "layers". Additionally, some items are subdivided further into more layers which contain a subset of fields of the item. For this, each chunk contains an additional layer table with the byte sizes of each layer.

Each layer is their own data stream, and can be decoded independently from the other layers (which e.g. allows to skip decoding some of the fields, if they are not of interest to the reading application).

A layer can be empty, i.e. can have a size of 0. This denotes: all fields of that layer are equal to that field of the first item (which has been stored uncompressed at the beginning of the chunk).

Table 31 — Chunk format definition

Field Name	Format	Size	Required
Point10	bytes	20	
GPSTime11	bytes	8	
RGB12	bytes	6	
Wavepacket13	bytes	29	
Byte	bytes	as declared in the LAZ VLR	
Point14	bytes	30	
RGB14	bytes	6	
RGBNIR14	bytes	8	
Wavepacket14	bytes	29	
Byte14	bytes	as declared in the LAZ VLR	
Layer table (6 to 10 only)			
Compressed Item Data			*

Only the items specified in the LAZ VLR are stored, and the fields are stored in the order specified there:

Point10: Uncompressed Point10-item, only stored if declared in the LAZ VLR.

GPSTime11: Uncompressed GPSTime11-item, only stored if declared in the LAZ VLR.

RGB12: Uncompressed RGB12-item, only stored if declared in the LAZ VLR.

Wavepacket13: Uncompressed Wavepacket13-item, only stored if declared in the LAZ VLR.

Byte: Uncompressed Byte-item, only stored if declared in the LAZ VLR.

Point14: Uncompressed Point14-item, only stored if declared in the LAZ VLR.

RGB14: Uncompressed RGB14-item, only stored if declared in the LAZ VLR.

RGBNIR14: Uncompressed RGBNIR14-item, only stored if declared in the LAZ VLR.

Wavepacket14: Uncompressed Wavepacket14-item, only stored if declared in the LAZ VLR.

Byte14: Uncompressed Byte14-item, only stored if declared in the LAZ VLR.

Layer table (6 to 10 only): Only stored for LAS point data record formats 6 to 10. The layer table contains the sizes of each layer of the occurring items. Each layer contains a subset of the fields, and uses their own data stream (so it is possible to just read a specific layer, and skip others, saving decoding time).

The first entry of the table is the chunk size (which is also redundantly stored in the chunk table).

After that, the layer size in bytes is stored as unsigned long values (4 byte). The layers for the Point14-item are always stored (as they are part of all point data record formats), the remaining layers are only stored if the item is specified in the LAZ VLR. They are stored in the specified order.

A layer can have a size of 0. This means that the value has the same value as the first (uncompressed) item, and the layer itself is not stored.

The actual layer data starts directly after the layer table. The file positions of the layers can be calculated from the sizes in bytes.

The table has the following format:

Table 32 — Layer table

Layer Number	Field Name	Format	Size	Required
-	Chunk size	unsigned short	4	*
1	Point14: channel, returns, XY layer size	unsigned short	4	*
2	Point14: Z layer size	unsigned short	4	*
3	Point14: classification layer size	unsigned short	4	*
4	Point14: flags layer size	unsigned short	4	*
5	Point14: intensity layer size	unsigned short	4	*
6	Point14: scan angle layer size	unsigned short	4	*
7	Point14: user data layer size	unsigned short	4	*
8	Point14: point source layer size	unsigned short	4	*
9	Point14: gpstime layer size	unsigned short	4	*
10	RGB14: rgb layer size	unsigned short	4	
11	RGBNIR14: rgb layer size	unsigned short	4	
12	RGBNIR14: nir layer size	unsigned short	4	
13	Wavepacket14: wavepacket layer size	unsigned short	4	
14	Byte14: bytes layer size	unsigned short [n]	4 * n	
NOTE: The layer number is just used to refer to it later within this specification (the number itself is not stored, nor do all 14 layer table entries have to be stored). The layers, if they exist, have to be stored in this order.				

Chunk size: Number of points in this chunk. For adaptive chunks, shall be the same as **Chunk size** calculated from the in the Chunk Table, specified in Table 30. For non-adaptive chunks, shall be the same as **Chunk size** specified in the LAZ VLR in Clause 7.

Point14: channel, returns, XY layer size: Layer size in bytes, contains the compressed fields **Changed Values**, **Scanner Channel**, **Return Number**, **Number of returns**, **dX** and **dY** of the Point14-item.

Point14: Z layer size: Layer size in bytes, contains the compressed field **dZ** of the Point14-item.

Point14: classification layer size: Layer size in bytes, contains the compressed field **Classification** of the Point14-item.

Point14: flags layer size: Layer size in bytes, contains the compressed field **Flags** (which encodes the LAS fields **Classification Flags**, **Scan Direction Flag** and **Edge of Flight Line**) of the Point14-item.

Point14: intensity layer size: Layer size in bytes, contains the compressed field **dIntensity** of the Point14-item.

Point14: scan angle layer size: Layer size in bytes, contains the compressed field **Scan Angle** of the Point14-item.

Point14: user data layer size: Layer size in bytes, contains the compressed field **User Data** of the Point14-item.

Point14: point source layer size: Layer size in bytes, contains the compressed field **dPoint Source ID** of the Point14-item.

Point14: gpstime layer size: Layer size in bytes, contains the compressed field **gpstime** of the Point14-item.

RGB14: rgb layer size: Layer size in bytes, contains the compressed fields **Changed values, dRed**, **dGreen** and **dBlue** (low and high each) of the RGB14-item.

RGBNIR14: rgb layer size: Layer size in bytes, contains the compressed fields **Changed values**, **dRed**, **dGreen** and **dBlue** (low and high each) of the RGBNIR14-item.

RGBNIR14: nir layer size: Layer size in bytes, contains the compressed fields **Changed values NIR** and **dNIR** (low and high) of the RGBNIR14-item.

Wavepacket 14: wavepacket layer size: Layer size in bytes, contains all the compressed fields of the Wavepacket14-item.

Byte14: bytes layer size: Array of layer sizes in bytes. Each layer contains one of the n bytes of the Byte14-item. Note: as with all layers, the size can be 0, which means that the bytes of that layer are equal to that byte in the first item (that had been stored uncompressed).

12. Accessing LAZ Items

12.1. Overview

The items are read from and written to the data stream in order.

After a chunk is completed, the instances and contexts are reset and reinitialized.

The items for LAS Points 6 through 10 (items Point14, RGB14, RGBNIR14, BYTE14 and Wavepacket14) are stored using layers.

12.2. Contexts

Additionally, the items for LAS Points 6 through 10 use 4 contexts: 4 copies of all instances, and each context with their own previous items, including medians.

Which context to use is decided by the POINT14-item. The remaining items (RGB14, RGBNIR14, BYTE14 and Wavepacket14) use a context based on the context of the POINT14-item.

The context is selected by the value of **scanner channel** (values 0 to 3). The proceeding is as follows:

- the first item in a chunk is stored uncompressed. Pick the **scanner channel** of that item as the initial context, and this item as the “previous item” for this context.
- de-/encode the field **Changed values** of the next (compressed) Point14-item, using the current context
  - if bit 6 of that field has been set, the scanner channel has been changed, so de-/encode the field **dScanner channel** to calculate **Scanner Channel** using the current context, and make a context switch for the remaining fields of the Point14-item:
    - if the context has not been used yet (in this chunk), copy the “previous item” from the current context as the “previous item” for the new context, and use it as the initial item for that context.
    - set the context to the value of **scanner channel**. This context is used for all remaining fields and all other items
  - after the item is done, set the “previous item” of the current/new context to this item
- for all other items (RGB14, RGBNIR14, BYTE14 and Wavepacket14), due to an implementation problem — which nevertheless is part of the standard — the context switch doesn’t work in an obvious manner:
  - if the new context (as given by the last read Point14-item) is different from the old context:
    - if the new context (as set by the Point14-item) has not been used for this item, copy the “previous item” from the current context as the “previous item” for the new context. After processing the item, it is stored as the “previous item” for this new context, and used as the initial item for that context.
    - if the new context has already been used by this item, the “previous item” is taken from the old context! All calculations that involve the previous item (for example, the difference to the previous item) are using this (non-obvious) “previous item”. Additionally, after processing the new item, it is used as the previous item of the old context!
    - The instances and other context-specific elements (for example, average values) are taken from the “correct”, new context.
  - if the new context is the same as the old context, the “previous item” of that context is used and changed.

References to the “previous item of this context” are used as described above, and for RGB14, RGBNIR14, BYTE14 and Wavepacket14 optionally referring to the previous item of the old context.

13. LAZ Items, LAS formats 0 to 5

13.1. Point10 (version 2)

The Point10-item is the common part for all LAS Point Data Record Formats 0 through 5. The uncompressed fields are the following:

Table 33 — Point10-item format, uncompressed

Field Name	Format	Size	Required
X	long	4 bytes	*
Y	long	4 bytes	*
Z	long	4 bytes	*
Intensity	unsigned short	2 bytes	
Return Number 3 bits (bits 0 — 2)	3 bits	3 bits	*
Number of Returns (given pulse) 3 bits (bits 3 — 5)	3 bits	3 bits	*
Scan Direction Flag 1 bit (bit 6)	1 bit	1 bit	*
Edge of Flight Line 1 bit (bit 7)	1 bit	1 bit	*
Classification	unsigned char	1 byte	*
Scan Angle Rank (-90 to +90), Left side	char	1 byte	*
User Data	unsigned char	1 byte	
Point Source ID	unsigned short	2 bytes	*

The first entry per chunk is this uncompressed item.

The first uncompressed item is considered the previous item of the first compressed item.

Details about the fields are described in Annex A.

The compressed item is stored as follows:

Table 34 — Point10-item format, compressed

Field Name	Type	Size	Encoder	No of instances	Required
Changed values	bits	6 bits	64 Symbols	1	*
Bit-Byte	8 bits	1 byte	256 Symbols	256	
dIntensity	short	2 bytes	Integer Compressor, 16 bits	4	
Classification	unsigned char	1 byte	256 Symbols	256	
dScan Angle Rank	unsigned short	1 byte	256 Symbols	2	
User Data	unsigned char	1 byte	256 Symbols	256	
dPoint Source ID	short	2 bytes	Integer Compressor, 16 bits	1	
dX	long	4 bytes	Integer Compressor, 32 bits	2	*
dY	long	4 bytes	Integer Compressor, 32 bits	22	*
dZ	long	4 bytes	Integer Compressor, 32 bits	20	*

Changed Values: Bit-mask of 6 bits that specifies whether any of the next 6 fields have changed in comparison to a previously processed point, specified in the field (it doesn't have to be the direct

predecessor for all fields). If an attribute has not changed, the bit is not set, and the value of that field is not stored in the data stream, i.e. not en-/decoded, and has to be skipped during the decoding, i.e., no data is read for such a field, as no data has been stored for this field. Stored using the symbol coder with 64 symbols (6 bits).

The bitmask is defined as follows:

Table 35 — Bit mask field “Changed values”

Bit	Field
5	Bit-Byte
4	Intensity
3	Classification
2	Scan Angle Rank
1	User Data
0	Point Source ID

Bit-Byte: Only stored if changed to the previous item (i.e. bit 5 of **Changed values** is set). The 3 + 3 + 1 + 1 bits of **Bit-Byte** are the uncompressed values:

— **return number** (bits 0 to 2 of **Bit-Byte**)
— **number of returns of given pulse** (bits 3 to 5 of **Bit-Byte**)
— **scan direction flag** (bit 6 of **Bit-Byte**)
— **edge of flight line flag** (bit 7 of **Bit-Byte**).

The field **Bit-Byte** is encoded using a symbol coder with 256 symbols. There are 256 instance for that field, it is chosen by the **Bit-Byte**-value of the previous item.

From r := **return number** and n := **number of returns of the given pulse**, for use in the next calculations two values are defined: **m**, which serializes the combinations for **r** and **n** (valid and invalid ones), and **l**, the return level (how many returns there have already been prior to this return), again for valid and invalid combinations, with the following mapping:

```
m := number_return_map[n][r]
l := number_return_level[n][r]

return_map_m[8][8] :=
{
  { 15, 14, 13, 12, 11, 10,  9,  8 },
  { 14,  0,  1,  3,  6, 10, 10,  9 },
  { 13,  1,  2,  4,  7, 11, 11, 10 },
  { 12,  3,  4,  5,  8, 12, 12, 11 },
  { 11,  6,  7,  8,  9, 13, 13, 12 },
  { 10, 10, 11, 12, 13, 14, 14, 13 },
  {  9, 10, 11, 12, 13, 14, 15, 14 },
  {  8,  9, 10, 11, 12, 13, 14, 15 }
}

number_return_level[8][8] :=
{
  {  0,  1,  2,  3,  4,  5,  6,  7 },
  {  1,  0,  1,  2,  3,  4,  5,  6 },
  {  2,  1,  0,  1,  2,  3,  4,  5 },
  {  3,  2,  1,  0,  1,  2,  3,  4 },
  {  4,  3,  2,  1,  0,  1,  2,  3 },
```

```
  { 5, 4, 3, 2, 1, 0, 1, 2 },
  { 6, 5, 4, 3, 2, 1, 0, 1 },
  { 7, 6, 5, 4, 3, 2, 1, 0 }
}
```

Figure 23 — Mapping for values m and l, Point10 item

dIntensity: For this field, the uncompressed first item of the chunk shall be treated by all compressed items as if it had had an intensity of 0, i.e. ignoring the first item.

Only stored if changed compared to the previous item that had the same value for **m** (**m** as defined in the field specification of **Bit-Byte**). If no such item has been processed yet, **Intensity (previous item with same m)** is considered to be 0.

If not stored, bit 4 of **Changed values** is not set, otherwise it has to be set. Difference encoded with a 16-bit Integer Compressor: **Intensity** := ISum16(**Intensity (previous item with same m)**, **dIntensity**), as specified in Clause 10.5.4. There are 4 different instances of the compressor, depending on the value for **m**: for m = 0 through 2, pick instance m. For m ≥ 3, pick instance 3.

Classification: Only stored if changed compared to the last point that has been processed (i.e. bit 3 of **Changed values** is set). Encoded using a symbol coder with 256 symbols. The **Classification** value of the previous item is used to select from 256 instances.

dScan Angle Rank: Only stored if changed compared to the last point that has been processed (i.e. bit 2 of **Changed values** is set). The difference to the scan angle rank of the last processed point is stored. The symbol encoder only returns positive values (a byte value), and to store negative differences, the value wraps around, i.e. **Scan Angle Rank** := (**Scan Angle Rank(previous item)** + **dScan Angle Rank**) MOD 256. Encoded using a symbol coder with 256 symbols. There are 2 instances, selected by the **scan direction flag** of the current item.

User Data: Only stored if changed compared to the last point that has been processed (i.e. bit 1 of **Changed values** is set). Encoded using a symbol coder with 256 symbols. 256 instances are used, selected by the value for **User Data** of the previous item.

dPoint Source ID: Only stored if changed compared to the last point that has been processed (i.e. bit 0 of **Changed values** is set). The difference to the **Point Source ID** of the last processed point is stored with a 16-bit Integer Compressor, i.e. **Point Source ID** := ISum16(**Point Source ID (previous item)**, **dPoint Source ID**). It has only 1 instance.

dX: The x coordinate is stored as the difference **dX** from an expected coordinate. The expected coordinate is the x coordinate of the previous item, plus the streaming median difference for the x coordinate of the previous items that had the same value m (i.e. there are up to 16 streaming medians). Encoded with a 32-bit Integer Compressor. There are 2 instances: one for $n = 1$ and one for $n \neq 1$ (where n is the **number of returns of given pulse** as defined in the description of the field **Bit-Byte**). The coordinate **X** is then calculated as **X** := **X (previous item)** + ISum32(streaming median[m], **dX**). The value **streaming median[m]** + **dX** is then inserted to the streaming median list for m.

The streaming median is defined as:

— keep a list of 5 values, ordered by size, initialized with 0
— the streaming median is the 3rd out of the 5 ordered valued
— when inserting a new value to the list, either the largest or the lowest value will be removed from the list, the new value is added, and the list is then reordered by size
— the first insertion will always remove the largest value
— if the next insertion will remove the lowest or the largest value depends on the inserted new value:

* if the added value is lower than the current streaming median value (i.e. the 3rd out of 5 values, before removing a value from the list), the next insertion will remove the largest value

* if the added value is larger than the current median value, the next insertion will remove the lowest value

* if the added value is equal to the current median value, the next insertion removes the opposite of the current insertion

NOTE: The first, uncompressed item of the chunk is not used for/inserted into the list of the streaming median. The streaming medians get reset / reinitialized at the beginning of each chunk.

Figure 24 — Definition Streaming median

Example — Example for streaming median

— the list may be “1, 5, 5, 8, 9”, and the largest value shall be removed next
— inserting “5” results in “1, 5, 5, 5, 8”, and the lowest value will be removed next (5 is equal to the median value “5”, so the direction changes)
— inserting “9” results in “5, 5, 5, 8, 9”, and the lowest value will be removed next (9 is larger than the median value “5”, so the lowest value will be removed next)

dY: The y coordinate is similarly to **dX** stored as the difference **dY** from an expected coordinate. The expected coordinate is the y-coordinate of the previous item, plus the streaming median difference for the y coordinate of the previous items that had the same value m (with 16 possible values for m). Note: the fields **dX** and **dY** have their own streaming medians.

Encoded with a 32-bit Integer Compressor. The field has 22 instances. The instance is chosen by using the **k** value used while (de)compressing **dX** (the **k** value is a number of significant bits, used in the Integer (De-)Compressor). For each of the following 22 values for **instance**, a different instance has to be used:

— if **k** < 20, set **instance** := $2 \cdot \left\lfloor \frac{k}{2} \right\rfloor$ (this unsets bit 0)
— else set **instance** := 20
— if **n** = 1, set **instance** := **instance** + 1 (where **n** is the **number of returns of given pulse** as defined in the description of the field **Bit-Byte**)

The coordinate **Y** is then calculated as **Y** := **Y (previous item)** + ISum32(streaming median[m], **dY**). The value **streaming median[m]** + **dY** is then inserted into the streaming median list for **m**.

dZ: The z coordinate is stored as the difference **dZ** from the previous z-coordinate of the point with the same value **l** (as defined in the description for the field **Bit-Byte**), i.e. there can be 8 relevant previous z coordinates. If there has not been such a previous item (in the same chunk), it is

considered to has been 0. Also, the first, uncompressed item of the chunk is ignored for this, i.e., it is treated as having had a value **Z** of 0.

Encoded with a 32-bit Integer Compressor, the field has 20 instances. The instance is chosen by using both **k** values used while (de-)compressing **dX** and **dY**. For each of the following 20 values for **instance**, a different instance has to be used:

— set **kXY** := $\left\lfloor \frac{\text{k (used in dX)}+\text{k (used in dY)}}{2} \right\rfloor$
— if **kXY** < 18, set **instance** := $2 \cdot \left\lfloor \frac{\text{kXY}}{2} \right\rfloor$ (this unsets bit 0)
— else set **instance** := 18
— if **n** = 1, set **instance** := **instance** + 1 (where **n** is the **number of returns of given pulse** as defined in the description of the field **Bit-Byte**)

The coordinate **Z** is then calculated as **Z** := ISum32(**Z (for last point with same I)**, **dZ**). (As described above, ignoring the value of the first, uncompressed item of the chunk.)

13.2. GPSTime11 (version 2)

The GPSTime11-item compresses one LAS field, GPS Time. It is part of LAS Point Data Record Formats 1, 3, 4 and 5. The format of the uncompressed item is as follows:

Table 36 — GPSTime11-item format, uncompressed

Field Name	Format	Size	Required
GPS Time	double	8 bytes	*

The first entry per chunk is this uncompressed item.

Details about the field are described in Annex A.

LAZ treats double-precision floating-point GPS times as if they were signed 64 bit integers and predicts the deltas between them.

NOTE: The compression is independent of the **GPS Time Type** setting in the **Global Encoding** field in the LAS header, i.e. whether an offset is subtracted or not.

It remembers up to four previously compressed GPS times with corresponding deltas. Keeping multiple prediction reference frames can account for repeated jumps in GPS time that arise when multiple flight paths are merged with fine spatial granularity.

The compression works best if the GPS Times of a reference frame are monotonically increasing values with a more or less constant spacing in time. For random values, the compression will be inefficient and will usually require more than 8 bytes.

For each reference frame (numbered 0 to 3), the following 3 variables are defined:

— **delta**
— **counter**: keeps track of how often the difference between GPS times is too large (or too low) and couldn't be stored using low multiples of **delta**
— **previous GPS Time**

The initial reference frame shall be "0", and the "next" reference frames shall be 1, 2, 3, 0 in that order.

All variables are initialized with 0 at the start of each chunk.

The **previous GPS time[0]** for reference frame 0 is initialized with the **GPS time** of the first, uncompressed item.

The current reference frame can be switched depending on the content of the fields.

All reference frames share the same encoder instances.

The compressed item is stored as follows:

Table 37 — GPSTime11-item format, compressed

Field Name	Type	Size	Encoder	No of instances	Required
cases	array of 1 or 2 “case” or “case_0delta-fields”				*
dGPS Time (low)	long	4 bytes	Integer Compressor, 32 bits	9	
GPS Time (extra)	unsigned long	4 bytes	Raw, 32 bit		

cases: An array that stores up to 2 **case** and/or **case_0delta**-fields. If a **case** or **case_0delta**-field comes next, depends on the **delta[reference frame]**: if it is 0, the next entry is a **case_0delta**, otherwise, a **case**-field.

If the first entry is **case** = 513..515 or **case_0delta** = 3..5, the array contains two fields, otherwise just one.

case_0delta is essentially a shorter version of the **case**-field, without all the **case**-options that use a multiple of **delta**, which are not relevant when **delta** is 0.

Both can occur in arbitrary order. E.g., a GPS Time might be compressed using the fields “case, dGPSTime(low)”, another value might be compressed using “case, case_0delta, dGPSTime(low), dGPSTime(high)”. Which fields are needed will be decided based on the values in the **cases**-array.

Table 38 — “Case” Array, GPSTime11

Field Name	Type	Size	Encoder	No of instances	Required
case	unsigned short	2 bytes	516 symbols		
case_0delta	bits	3 bits	6 Symbols		

case:

Table 39 — Values for field “case”, GPSTime11

Value	Description
0	predicted with a delta of zero
1–500	predicted using the current delta times 1 to 500
501–510	predicted using the current delta times -1 to -10
511	identical to the last gps time (of same reference frame)
512	starting a new reference frame (using both **dGPS Time (low)** and **GPS Time (extra)**)
513–515	predicted with one of the other three reference frames

For each of the cases, a different encoder instance (of the 32-bit Integer Compressor) for the field **dGPS Time (low)** (with their own distribution table) has to be used (if the field is used at all). Their given numbers are arbitrary and just for convenience (and to align them with those of the

case_0delta-field). After processing a case, except **513..515**, the **previous GPS Time[reference frame]** is set to the calculated **GPS time**.

The cases are handled as follows:

0: **dGPS Time (low)** is read with instance 7. The uncompressed **GPS Time** is **previous GPS Time[reference frame]** + ISum32(0, **dGPS Time (low)**).

counter[reference frame] is increased by 1. If **counter[reference frame]** is > 3, **delta[reference frame]** is set to ISum32(0, **dGPS Time (low)**), and **counter[reference frame]** is set to 0.

The GPS Time is complete after this, **GPS Time (extra)** is unused/skipped.

1: **dGPS Time (low)** is read with the instance 1. It contains the difference from **delta** of the current reference frame, i.e. **GPS time** := **previous GPS Time[reference frame]** + ISum32(**delta[reference frame]**, **dGPS Time (low)**).

counter[reference frame] is reset to 0. The GPS Time is complete after this, **GPS Time (extra)** is unused/skipped.

2..499: **dGPS Time (low)** is read with the instance 2 for **case** = 2..9 and instance 3 for **case** = 10..499. It contains the difference from **case** * **delta[reference frame]**, i.e. **GPS time** := **previous GPS Time[reference frame]** + ISum32(**case** * **delta[reference frame]**, **dGPS Time (low)**). The GPS Time is complete after this, **GPS Time (extra)** is unused/skipped.

500: **dGPS Time (low)** is read with the instance 4. It contains the difference from 500 * **delta[reference frame]**, i.e. **GPS time** := **previous GPS Time[reference frame]** + ISum32(500 * **delta[reference frame]**, **dGPS Time (low)**).

counter[reference frame] is increased by 1. If **counter[reference frame]** is > 3, **delta[reference frame]** is set to ISum32(500 * **delta[reference frame]**, **dGPS Time (low)**), and **counter[reference frame]** is set to 0.

The GPS Time is complete after this, **GPS Time (extra)** is unused/skipped.

501..509: This is used for negative differences. **dGPS Time (low)** is read with the instance 5. **GPS time** is calculated as **previous GPS Time[reference frame]** + ISum32(- (**case** — 500) * **delta[reference frame]**, **dGPS Time (low)**). The GPS Time is complete after this.

510: This is also used for negative differences. **dGPS Time (low)** is read with the instance 6. **GPS time** is calculated as **previous GPS Time[reference frame]** + ISum32(- 10 * **delta[reference frame]**, **dGPS Time (low)**).

counter[reference frame] is increased by 1. If **counter[reference frame]** is > 3, **delta[reference frame]** is set to ISum32(- 10 * **delta[reference frame]**, **dGPS Time (low)**), and **counter[reference frame]** is set to 0.

The GPS Time is complete after this, **GPS Time (extra)** is unused/skipped.

511: The GPS time has not changed, **GPS time** is set to **previous GPS Time[reference frame]**. Both **dGPS Time (low)** and **GPS Time (extra)** are unused/skipped.

512: The gps time is stored using both time fields. (Due to the ambiguity in interpreting the numbers, the data types are specified, as the integer compressor returns 32 bit signed integer values, and the gps time is a 64 bit floating point value treated as a 64 bit unsigned integer). First, the instance 8 (of the 32-bit Integer Compressor) for **dGPS Time (low)** and the RAW encoder (32 bit) for **GPS Time**

(extra) are used to calculate (unsigned long long)**tmp** := (unsigned long long) ISum32((signed integer)((unsigned long long)**previous GPS Time[reference frame]** RIGHT SHIFT BY 32 BITS),**dGPS Time (low)**) LEFT SHIFT BY 32 BITS + **GPS Time (extra)**.

Then the current reference frame is changed to the next reference frame in line (in the order 0, 1, 2, 3, 0). **GPS Time** is then set to **tmp**. The **previous GPS Time[reference frame]** for the new/changed reference frame is then set to **GPS Time** (while **previous GPS Time** for the previous reference frame is unchanged). **delta[reference frame]** and **delta[reference frame]** (for the new reference frame) are set to 0.

The GPS Time is complete after this. **512** is only allowed to be the first entry in the **cases**-array.

513..515: The current reference frame is changed to the 1st (513), 2nd (514) or 3rd (515) successor, in the order 0, 1, 2, 3, 0. Then, the next entry in the **cases** array is evaluated. **513..515** are only allowed to be the first entry in the **cases**-array. **previous GPS Time[reference frame]** is not changed here.

case_0delta:

Table 40 — Values for field “case_0delta”, GPSTime11

Value	Description
0	identical to the last gps time (of same reference frame)
1	stored using just **dGPS Time (low)**
2	starting a new reference frame (using both **dGPS Time (low)** and **GPS Time (extra)**)
3-5	predicted with one of the other three reference frames

The field **case_0delta** is a shorter version of the **case**-field, for cases where **delta[reference frame]** = 0 (so the multiplications for **case** 0 to 510 are not relevant). The instance numbers are the same as used for the **case** field. After processing a case, except **3..5**, the **previous GPS Time[reference frame]** is set to the calculated **GPS time**. The cases are:

0: Identical to **case** = 511 (no change).

1: **dGPS Time (low)** is read with the instance 0, and **GPS time** := **previous GPS Time[reference frame]** + ISum32(0, **dGPS Time (low)**). **delta[reference frame]** is set to ISum32(0, **dGPS Time (low)**). (Note that this, and the instance number, differ from **case** = 1).

counter[reference frame] is reset to 0. The GPS Time is complete after this, **GPS Time (extra)** is unused/skipped.

2: Identical to **case** = 512 (stored using both **dGPS Time (low)** and **GPS Time (extra)**). **2** is only allowed to be the first entry in the **cases**-array.

3..5: Analogous to **case** = **513..515**: The current reference frame is changed to the 1st (3), 2nd (4) or 3rd (5) successor, in the order 0, 1, 2, 3, 0. Then, the next entry in the **cases** array is evaluated. **3..5** are only allowed to be the first entry in the **cases**-array. **previous GPS Time[reference frame]** is not changed here.

dGPS Time (low): Stored using a 32-bit Integer Compressor with 9 different instances. Usage is described above. May not be stored, depending on the **case**/**case_0delta** values.

GPS Time (extra): Stored using a Raw Encoder with 32 bits. Only stored if the full gps time has to be stored (i.e. for **case** = 512 and **case_0delta** = 2). Usage is described above.

13.3. RGB12 (version 2)

The RGB12-item compresses the **Red**, **Green** and **Blue**-fields that are part of LAS Point Data Record Formats 2, 3 and 5. The format of the uncompressed item is as follows:

Table 41 — RGB12-item format, uncompressed

Field Name	Format	Size	Required
Red	unsigned short	2 bytes	*
Green	unsigned short	2 bytes	*
Blue	unsigned short	2 bytes	*

The first entry per chunk is this uncompressed item.

This first item is considered the previous item for the first compressed item.

Details about the fields are described in Annex A.

The compressed item first stores a field that indicates which bytes are changed. The high and low bytes of the red, green and blue channel are treated separately. If they changed, only the difference to the corresponding value of the previous item is stored.

As there is often a correlation between changes to the red, green, and blue channels (e.g. if the intensity for a gray color changes), the delta for the red channel is added to the green channel first, and only the difference to that is stored. For the blue channel, the average delta of the red and green channel is used.

For use in the following fields, the algorithm “clamp255”, which sets values above 255 to 255 and values below 0 to 0, is defined as:

```
function clamp255(n):
* one parameter, a number
  if n < 0
  then
     return 0
  else if n > 255
  then
     return 255
  else
     return n
```

Figure 25 — Algorithm clamp255(), Pseudocode

Note: the lower byte of a short value is value & 0x00FF, the higher byte of a short value is (value & 0xFF00) >> 8.

The compressed item is stored as follows:

Table 42 — RGB12-item format, compressed

Field Name	Type	Size	Encoder	No of instances	Required
Changed values	bits	7 bits	128 Symbols	1	*
dRed (low)	byte	1 byte	256 Symbols	1	
dRed (high)	byte	1 byte	256 Symbols	1	
dGreen (low)	byte	1 byte	256 Symbols	1	

Field Name	Type	Size	Encoder	No of instances	Required
dGreen (high)	byte	1 byte	256 Symbols	1	
dBlue (low)	byte	1 byte	256 Symbols	1	
dBlue (high)	byte	1 byte	256 Symbols	1	

Changed values: Indicates, which of the fields have been changed. Only changed fields are stored.

Table 43 — Bit values for field “Changed values”, RGB12

Bit	Description
0	Field dRed (low) is stored, otherwise dRed (low) is 0
1	Field dRed (high) is present, otherwise dRed (high) is 0
2	Field dGreen (low) is present, otherwise dGreen (low) is 0
3	Field dGreen (high) is present, otherwise dGreen (high) is 0
4	Field dBlue (low) is present, otherwise dBlue (low) is 0
5	Field dBlue (high) is present, otherwise dBlue (high) is 0
6	(Uncompressed) **Green** and **Blue** are identical to **Red**, bits 2-5 are ignored

If bit 6 is set, **Green** and **Blue** are equal to the uncompressed value of **Red** (i.e. after processing **dRed (low)** and **dRed (high)**). In that case, bits 2 to 5 are ignored, the fields **dGreen (low)**, **dGreen (high)**, **dBlue (low)** and **dBlue (high)** are not stored, and the calculation described below for Green and Blue based on those fields is not used.

dRed (low): Only stored if bit 0 of **Changed values** is set, otherwise considered 0. Stores the delta to the lower bit of **Red** of the previous item. The sum is then mapped to 0..255, using modulo 256 and adding 256 if negative. I.e. **lower byte of Red** := (**lower byte of Red (previous item)** + **dRed (low)** + 256) MOD 256.

dRed (high): Only stored if bit 1 of **Changed values** is set, otherwise considered 0. Stores the delta to the higher bit of **Red** of the previous item. The sum is then mapped to 0..255. I.e. **higher byte of Red** := (**higher byte of Red (previous item)** + **dRed (high)** + 256) MOD 256.

Note: The remaining four fields are only used if bit 6 of **Changed values** is not set.

dGreen (low): Only stored if bit 2 of **Changed values** is set (and bit 6 is not set), otherwise the lower byte of **Green** is the same as the lower byte of **Green** of the previous item. If stored, the difference for the **Red** value is added first, and **dGreen (low)** stores only the difference to that.

Calculate diff := **lower byte of Red** — **lower byte of Red (previous item)**. This value is added to the previous value of the lower byte of **Green**, clamping the sum to 255 (as defined above). Then, the total sum is mapped to 0..255. I.e. **lower byte of Green** := (**dGreen (low)** + clamp255(**lower byte of Green (previous item)** + diff) + 256) MOD 256.

dGreen (high): Only stored if bit 3 of **Changed values** is set (and bit 6 is not set), otherwise the higher byte of **Green** is the same as the higher byte of **Green** of the previous item. If stored, similar to **dGreen (low)**, the difference for the **Red** value is calculated, clamped and added:

Calculate diff := **higher byte of Red** — **higher byte of Red (previous item)**. This value is added to the previous value of the higher byte of **Green**, clamping the sum to 255. Then, the total sum is mapped to 0..255. I.e. **higher byte of Green** := (**dGreen (low)** + clamp255(**higher byte of Green (previous item)** + diff) + 256) MOD 256.

dBlue (low): Only stored if bit 4 of **Changed values** is set (and bit 6 is not set), otherwise the lower byte of **Blue** is the same as the lower byte of **Blue** of the previous item. If stored, the average difference for the **Red** and **Green** value is added first:

Calculate diff := round_towards_0((**lower byte of Red** — **lower byte of Red (previous item)** + **lower byte of Green** — **lower byte of Green (previous item)**) / 2), with round_towards_0() as defined in Conventions (i.e. rounding down above 0, and rounding up below 0), and then clamping it to 255. Then, the total sum is mapped to 0..255. I.e. **lower byte of Blue** := (**dBlue (low)** + clamp255(**lower byte of Blue (previous item)** + diff) + 256) MOD 256.

dBlue (high): Only stored if bit 5 of **Changed values** is set (and bit 6 is not set), otherwise the higher byte of **Blue** is the same as the higher byte of **Blue** of the previous item. As for **dBlue (low)**, the average difference for the **Red** and **Green** value is added first:

Calculate diff := round_towards_0((**higher byte of Red** — **higher byte of Red (previous item)** + **higher byte of Green** — **higher byte of Green (previous item)**) / 2), rounding towards 0, and then clamping it to 255. Then, the total sum is mapped to 0..255. I.e. **lower byte of Blue** := (**dBlue (low)** + clamp255(**higher byte of Blue (previous item)** + diff) + 256) MOD 256.

13.4. BYTE (version 2)

The BYTE-item compresses any additional bytes that are optionally appended to LAS Point Data Record Formats 0 through 5. The number of bytes, **n**, is declared in in the LAZ VLR, specified in Clause 7. The uncompressed item looks as:

Table 44 — BYTE-item format, uncompressed

Field Name	Format	Size	Required
Bytes	byte[n]	n bytes	*

The first entry per chunk is this uncompressed item.

This uncompressed item is considered the previous item for the first compressed item.

LAZ compresses each byte separately, and stores only the difference to the corresponding byte in the previous item.

The compressed item is stored as follows:

Table 45 — BYTE-item format, compressed

Field Name	Type	Size	Encoder	No of instances	Required
dBytes	byte[n]	n byte	256 Symbols	n	*

dBytes: Array of n bytes, using a separate symbol encoder for each of the n bytes (each with 256 symbols). Stores the difference to the corresponding byte of the previous item. The sum is then mapped to 0..255, using modulo 256 and adding 256 if negative. I.e. the uncompressed **Bytes[i]** := (**Bytes[i] (previous item)** + **dBytes[i]** + 256) mod 256.

13.5. Wavepacket13 (version 1)

The Wavepacket13-item compresses the Waveform data that are part of LAS Point Data Record Formats 4 and 5. The format of the uncompressed item is as follows:

Table 46 — Wavepacket13-item, uncompressed

Field Name	Format	Size	Required
Wave Packet Descriptor Index	unsigned char	1 byte	*
Byte Offset to Waveform Data	unsigned long long	8 bytes	*
Waveform Packet Size in Bytes	unsigned long	4 bytes	*

Field Name	Format	Size	Required
Return Point Waveform Location	float	4 bytes	*
Parametric dx	float	4 bytes	*
Parametric dz	float	4 bytes	*
Parametric dz	float	4 bytes	*

The first entry per chunk is this uncompressed item.

It is considered the previous item for the first compressed item.

Details about the fields are described in Annex A.

All LAS floating point values are treated as signed 32-bit integer values (i.e. uses the byte representation of the floating point values as is).

All fields except the **Byte Offset to Waveform Data** are stored in a straight forward way using the specified encoder. That offset-field however uses an additional 2-bit field that determines 4 cases: (0) the value is the same as the offset of the previous item, (1) it just differs by the packet size of the previous item, (2) the difference is small enough to be stored with just 4 bytes, and (3) the full 8 byte value is stored.

The compressed item is stored as follows:

Table 47 — Wavepacket13-item, compressed

Field Name	Type	Size	Encoder	No of instances	Required
Wave Packet Descriptor Index	byte	1 byte	256 Symbols	1	*
Offset Diff Type	bits	2 bits	4 Symbols	4	*
dOffset Diff (low)	long	4 bytes	Integer Compressor, 32 bits	1	
Offset (full)	long long	8 bytes	Raw, 64 bit	1	
dPacket Size	long	4 bytes	Integer Compressor, 32 bits	1	*
dReturn Point	long	4 bytes	Integer Compressor, 32 bits	1	*
dPdXYZ	long[3]	12 bytes	Integer Compressor, 32 bits	3	*

Wave Packet Descriptor Index: Stored using a symbol encoder with 256 symbols.

Offset Diff Type: Stored using a symbol encoder with 4 symbols, and 4 different instances. The instance is chosen by the value of **Offset Diff Type** of the previous item, or 0 for the first compressed item (as the uncompressed first item does not have this field). Encodes how **Byte offset to waveform data** is compressed:

0: **Byte Offset to Waveform Data** is the same as the value from the previous item. **dOffset Diff (low)** and **Offset (full)** are not stored or used.

1: The offset is increased by the packet size of the previous item, **Byte Offset to Waveform Data** := **Byte Offset to Waveform Data (previous item)** + **Waveform Packet Size in Bytes (previous item)**. **dOffset Diff (low)** and **Offset (full)** are not stored.

2: **Byte Offset to Waveform Data** can be encoded with just a small difference, called **Offset Diff**, to the previous item. Stored in **dOffset Diff (low)** is only the difference to the last value of **Offset Diff** used in this chunk (i.e. by the last item that had an **Offset Diff Type** of 2) or to 0, if it has not been used yet. I.e. **Byte Offset to Waveform Data** := **Byte Offset to Waveform Data (previous item)** + ISum32(**Offset Diff (previous item with Offset Diff Type of 2)**, **dOffset Diff (low)**). Note that **Offset Diff** is only used when **Offset Diff Type** is 2. **Offset (full)** is not stored for this **Offset Diff Type**.

3: **Byte Offset to Waveform Data** is stored as the full 8 byte value in **Offset (full)**, i.e. **Byte offset to waveform data** := **Offset (full)**. **Offset Diff (low)** is not stored.

dOffset Diff (low): Only stored if **Offset Diff Type** is 2, using a 32-bit Integer Compressor. Used as described above.

Offset (full): Only stored if **Offset Diff Type** is 3. Using a Raw integer encoder with 64 bits, contains the full **Byte offset to waveform data** value.

dPacket Size: Using a 32-bit Integer Compressor, stores the difference to the previous value of **Waveform Packet Size in Bytes**, i.e. **Waveform Packet Size in Bytes**:= ISum32(**Waveform Packet Size in Bytes (previous item)**, **dPacket Size**).

dReturn Point: Using a 32-bit Integer Compressor, stores the difference to the previous value of **Return Point Waveform Location**, i.e. **Return Point Waveform Location** := ISum32(**Return Point Waveform Location (previous item)**, **dReturn Point**). Note that the LAS floating point value **Return Point Waveform Location** is treated as a signed 32-bit integer value in this calculation.

dPdXYZ: An array of 3 values, namely **dPdx**, **dPdy** and **dPdz**. Those store the difference to the **Parametric dx**, **Parametric dx** and **Parametric dz** values to those values of the previous item, where the floating point values are treated as signed long integer values, i.e. the difference is taken from and added to the 4-byte representation of the floating point value: **Parametric dx (as signed 32 bit integer)** := ISum32(**Parametric dx (as signed 32 bit integer) (previous item)**, **dPdx**), and the same for **Parametric dy** and **Parametric dz**. Using a 32-bit Integer Compressor with 3 instances, one for each of **dPdx**, **dPdy** and **dPdz**. Note that this is not the same as storing the 3 fields using a separate 32-bit Integer Compressor for each, as they share the symbol encoders for **k**, as described in Clause 10.5.

14. LAZ Items, LAS formats 6 to 10

14.1. Point14 (version 3)

The Point14-item is the common part for all LAS point types 6 to 10. The uncompressed fields are the following:

Table 48 — Point14-item format (LAS Point Data Record Format 6), uncompressed

Field Name	Format	Size	Required
X	long	4 bytes	*
Y	long	4 bytes	*
Z	long	4 bytes	*
Intensity	unsigned short	2 bytes	
Return Number	4 bits (bits 0 — 3)	4 bits	*
Number of Returns (given pulse)	4 bits (bits 4 — 7)	4 bits	*
Classification Flags	4 bits (bits 0 — 3)	4 bits	
Scanner Channel	2 bits (bits 4 — 5)	2 bits	*
Scan Direction Flag	1 bit (bit 6)	1 bit	*
Edge of Flight Line	1 bit (bit 7)	1 bit	*
Classification	unsigned char	1 byte	*
User Data	unsigned char	1 byte	
Scan Angle	short	2 bytes	*
Point Source ID	unsigned short	2 bytes	*
GPS Time	double	8 bytes	*

The first entry per chunk is this uncompressed item.

Details about the fields are described in Annex A.

The item has 4 contexts, which means 4 sets of all encoder instances (all initialized independently). The previous item used is always the previous item from the same context (so there are 4 previous items).

The initial previous item of the context is selected when a context is first used, as specified in Clause 12.2: the uncompressed item for context 0, and for the other contexts a later item that is determined by when the context has been switch.

This initial item of a context is considered the previous item for the first compressed item in this context. Additionally, it is used as a fallback item in calculations for the fields **dZ**, **dIntensity** and to initialize **previous GPS Time[0]** for the GPS-calculation. This is described in the field specifications.

The context 0 to 3 is selected by the value of the field **Scanner Channel** (values 0 to 3). For that, first, the field **Changed values** has to be read (or written), and, if it indicates that the **Scanner Channel** has been changed, also the field **Scanner Channel**. For those 2 fields, the context of the previous item is used. The context is then switched to number **Scanner Channel** (that has just been processed), and this new context is used for the remaining fields of the item.

The compressed item is stored using 9 layers and is stored the following way:

Table 49 — Point14-item format, compressed

Field Name	Type	Size	Encoder	No of instances	Required	Layer
Changed values	bits	7 bits	128 Symbols	8	*	1
dScanner Channel	bits	2 bits	3 Symbols	1		1
Number of returns (given pulse)	4 bits	4 bits	16 Symbols	16		1
dReturn Number (same time)	bits	4 bits	13 Symbols	1		1
Return Number (different time)	bits	4 bits	16 Symbols	16		1
dX	long	4 bytes	Integer Compressor, 32 bits	2		1
dY	long	4 bytes	Integer Compressor, 32 bits	22		1
dZ	long	4 bytes	Integer Compressor, 32 bits	20		2
Classification	unsigned char	1 byte	256 Symbols	64		3
Flags	bits	6 bits	64 Symbols	64		4
dIntensity	short	2 bytes	Integer Compressor, 16 bits	4		5
dScan Angle	short	2 bytes	Integer Compressor, 16 bits	2		6
User Data	unsigned char	1 byte	256 Symbols	64		7
dPoint Source ID	short	2 bytes	Integer Compressor, 16 bits	1		8
GPS cases	array of 1 or 2 “gps case” or “gps case_0delta-fields”					9
dGPS Time (low)	long	4 bytes	Integer Compressor, 32 bits	9		9
GPS Time (extra)	unsigned long	4 bytes	Raw, 32 bit			9

The first layer and the field **Changed values** is always present, all others are optional. For the first layer, the field **Changed values** specifies which fields are stored. The fields of the other layers can be missing as the layers can have size 0. Additionally, the GPS-fields (layer 9) **dGPS Time (low)** and **GPS Time (extra)** are optional, specified by the **GPS cases**-field.

Changed Values: Bit-mask of 7 bits that specifies whether any of the next 6 fields have changed in comparison to the previously processed item (of any context, and where the uncompressed item is considered the first processed item). If an attribute has not changed, the bit is not set, and the value of that field is not stored in the data stream (i.e. not en-/decoded, and has to be skipped in the decoding). Bits 1 and 0 are used to mark if the return number has changed by 0, -1 or +1, or more.

Only if it has changed by more than 1, it is stored in the data stream. Stored using the symbol coder with 128 symbols (7 bits).

For later use, the value **cpr** is defined as

— **cpr** := 3 (single return) if **return number** = 1 and **number of returns** < 2
— **cpr** := 2 (first return) if **return number** = 1 and **number of returns** > 1
— **cpr** := 1 (last return) if **return number** ≠ 1 and **return number** ≥ **number of returns**
— **cpr** := 0 (intermediate return) if **return number** ≠ 1 and **return number** < **number of returns**

Additionally, define **cprgps** := **cpr** * 2 + {1 if bit 4 of **Changed Values** is set}. For the first item (i.e. the item stored uncompressed in the beginning of the chunk), that 4th bit is defined as unset (although it did not have that field, as it was not compressed).

The field **Changed Values** uses 8 different instances, which is selected by the value of **cprgps** of the previously processed item.

The bitmask is defined as follows:

Table 50 — Bit mask for field “Changed value”, Point14

Bit	Field
6	1: Field “dScanner channel” changed, 0: unchanged
5	1: Field “Point source” changed, 0: unchanged
4	1: Field(s) “gps time” changed, 0: unchanged
3	1: Field “scan angle” changed, 0: unchanged
2	1: Field “number of returns” changed, 0: unchanged
0-1	0: Return number unchanged, 1: Return number is +1 from previous item, 2: Return number -1 from previous item, 3: stored in the data stream

dScanner Channel: Only stored if bit 6 of **Changed Values** is set, i.e. if changed to the previous item processed (with the uncompressed item considered to be the first processed item), otherwise **Scanner Channel** is the same value as the value from the previous item. Stored using a symbol encoder with 3 Symbols as a difference: **Scanner Channel** := (**Scanner Channel (previous item)** + **dScanner Channel** + 1) MOD 4.

NOTE 1: As described in contexts, the **Scanner Channel** defines which of the 4 context to use for all the remaining fields and the fields of the other items, so a context switch may occur after processing this field. The **previous item of this context** (which many of the following fields used to compare with or to calculate the difference from) is specific to the context, i.e. each context has a different previous item. This also applies to averages or the running median, they are also context specific.

Number of returns (given pulse): Only stored if bit 2 of **Changed Values** is set, otherwise the value from the previous item is used (of the same context; note that this field is the first field using the context selected by the value of **Scanner Channel**). Stored using a symbol encoder with 16 symbols and 16 instances. The instance is chosen by the value **Number of returns (given pulse)** of the previous item of the same context.

dReturn Number (same time): Only stored if bit 0 and 1 of **Changed Values** are set and bit 4 (“gps changed”) is not set. Stored using a symbol encoder with 13 symbols and 1 instance. The uncompressed **Return Number** is then calculated as **Return Number** := (**Return Number (previous item of same context)** + **dReturn Number (same time)** + 2) MOD 16. The next field specifies the other cases.

Return Number (different time): Only stored if bit 0 and 1 and 4 of **Changed Values** are set. Stored using a symbol encoder with 16 symbols and 16 instance. The instance is chosen by the value **Return Number** of the previous item of the same context. The uncompressed **Return Number** is then the value **Return Number (different time)**.

In the other cases, the **Return Number** is calculated from the information in the bits 0 and 1 of **Changed Values**:

— Bit 0 is 0 and bit 1 is 0: **Return Number** := **Return Number (previous item of the same context)**
— Bit 0 is 1 and bit 1 is 0: **Return Number** := (**Return Number (previous item of the same context)** + 1) MOD 16
— Bit 0 is 0 and bit 1 is 1: **Return Number** := (**Return Number (previous item of the same context)** + 15) MOD 16
— Bit 0 is 1 and bit 1 is 1: **Return Number** is calculated from **dReturn Number (same time)** or **Return Number (different time)** depending on bit 4, see above.

From the uncompressed values **r** := **Return number** ($0 \leq r \leq 15$) and **n** := **Number of returns (given pulse)** ($0 \leq n \leq 15$), for use in the next calculations two values are defined: **m**, which serializes the combinations for **r** and **n** (valid and invalid ones, and reduced to 6 different cases), and **l**, the return level (how many returns there have already been prior to this return, including valid and invalid combinations, and reduced to 8 different cases):

```
m := number_return_map_6ctx[n][r]
l := number_return_level_8ctx[n][r]

number_return_map_6ctx[16][16] :=
{
  { 0, 1, 2, 3, 4, 5, 3, 4, 4, 5, 5, 5, 5, 5, 5, 5 },
  { 1, 0, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 },
  { 2, 1, 2, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3 },
  { 3, 3, 4, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 },
  { 4, 3, 4, 4, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 },
  { 5, 3, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 },
  { 3, 3, 4, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4 },
  { 4, 3, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4, 4, 4 },
  { 4, 3, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4, 4 },
  { 5, 3, 4, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4 },
  { 5, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4, 4 },
  { 5, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 4, 4, 4 },
  { 5, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 4, 4 },
  { 5, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 4, 4 },
  { 5, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 4 },
  { 5, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5 },
  { 5, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5 }
}

number_return_level_8ctx[16][16] :=
{
  { 0, 1, 2, 3, 4, 5, 6, 7, 7, 7, 7, 7, 7, 7, 7, 7 },
  { 1, 0, 1, 2, 3, 4, 5, 6, 7, 7, 7, 7, 7, 7, 7, 7 },
  { 2, 1, 0, 1, 2, 3, 4, 5, 6, 7, 7, 7, 7, 7, 7, 7 },
  { 3, 2, 1, 0, 1, 2, 3, 4, 5, 6, 7, 7, 7, 7, 7, 7 },
  { 4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 6, 7, 7, 7, 7, 7 },
  { 5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 6, 7, 7, 7, 7 },
  { 6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 6, 7, 7, 7 },
```

```
  { 7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 6, 7, 7 },
  { 7, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 6, 7 },
  { 7, 7, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 6 },
  { 7, 7, 7, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5 },
  { 7, 7, 7, 7, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4 },
  { 7, 7, 7, 7, 7, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3 },
  { 7, 7, 7, 7, 7, 7, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2 },
  { 7, 7, 7, 7, 7, 7, 7, 7, 6, 5, 4, 3, 2, 1, 0, 1 },
  { 7, 7, 7, 7, 7, 7, 7, 7, 7, 6, 5, 4, 3, 2, 1, 0 }
}
```

Figure 26 — Mapping for values m and l, Point14-item

NOTE 2: These constants differ from the corresponding constants for the Point10 item. E.g., m and l only have 6 and 10 different values now, i.e. the map has been simplified, as higher combinations tend to not have significant entropy differences.

Additionally, set **mgps** := **m** * 2 + {1 if bit 4 of field **Changed value** is set}. With $0 \leq m \leq 5$, this means $0 \leq mgps \leq 11$. This value is used in **dX** and **dY**.

dX: The x coordinate is stored as the difference **dX** from an expected coordinate. The expected coordinate is the x coordinate of the previous item of the same context, plus the streaming median difference for the x coordinate of the previous items of the same context that had the same value of **mgps** (as defined above), i.e. up to 12 medians for each of the 4 contexts. Encoded with a 32-bit Integer Compressor and two instances: one for $n = 1$ and one for $n \neq 1$ (where **n** is the **Number of returns (given pulse)**). The coordinate **X** is then calculated as **X** := **X (previous item of the same context)** + ISum32(**streaming median[mgps]**, **dX**). The value ISum32(**streaming median[m]**, **dX**) is then inserted to the streaming median list for **mgps** (of this context).

The streaming median is defined in the specification for the **dX**-field of the Point10-item.

Each context has its own set of streaming medians. Also, each field (i.e. dX and dY) has their own medians. Note that the medians also get reset at the beginning of a chunk. Just as for Point10-items, the first, uncompressed item of the chunk is not used for the medians.

Note: ISum32() for decoding (and the corresponding IDiff32() for encoding) is specified with the Integer (De-)Compressor, as well as ISum16() used below.

dY: The y coordinate is similarly to **dX** stored as the difference **dY** from an expected coordinate. The expected coordinate is the y-coordinate of the previous item, plus the streaming median difference for the y coordinate of the previous items that had the same value **mgps**, i.e. up to 12 medians per context.

Encoded with a 32-bit Integer Compressor. The field has 22 instances. The instance is chosen by using the **k** value used while (de-)compressing **dX** (the **k** value is a number of significant bits, used in the Integer (De-)Compressor). For each of the following 22 values for **instance**, a different instance shall be used:

— if **k** < 20, set **instance** := $2 \cdot \left\lfloor \frac{k}{2} \right\rfloor$ (which unsets bit 0)
— else set **instance** := 20
— if **n** = 1, set **instance** := **instance** + 1 (where **n** is the **Number of returns (given pulse)** of this item)

The coordinate **Y** is then calculated as **Y** := **Y (previous item of the same context)** + ISum32(**streaming median[mgps]**, **dY**). The value ISum32(**streaming median[m]**, **dY**) is then inserted into the streaming median list for **mgps** (of this context).

NOTE 3: Each of the following fields (up to the GPS fields) is using their own layer, i.e. a different data stream each (see Clause 11.7).

dZ: The z coordinate is stored as the difference **dZ** from **Z (previous item with same l and the same context)** (with **l** as defined in the map for l and m), i.e. one of 8 previous z coordinates for each of the 4 contexts.

If no item with the same value of **l** has been processed yet in the current context, **Z (initial item of the context)** is used instead of **Z (previous item with same l and the same context)**.

Encoded with a 32-bit Integer Compressor, the field has 20 instances. The instance is chosen by using both **k** values used while (de-)compressing **dX** and **dY**. For each of the following 20 values for **instance**, a different instance has to be used:

- set **kXY** := $\left\lfloor \frac{\text{k (used in dX)} + \text{k (used in dY)}}{2} \right\rfloor$
- if **kXY** < 18, set **instance** := $2 \cdot \left\lfloor \frac{\text{kXY}}{2} \right\rfloor$ (which unsets bit 0)
- else set **instance** := 18
- if **n** = 1, set **instance** := **instance** + 1 (where **n** is the **Number of Returns (given pulse)**).

The coordinate **Z** is then calculated as **Z** := **Z (for previous item with same l and the same context)** + **dZ**.

Classification: Encoded using a symbol coder with 256 symbols with 64 instances. The instance number is chosen by calculating (**Classification (previous item of the same context**) mod 32) * 2 + { 1 if **return number** = 1 and **Number of returns (given pulse)** < 2 } (i.e. a value between 0 and 63).

Flags: 6-bit value that is composed of the (uncompressed) fields **Classification Flags** (as bits 0 to 3), **Scan Direction Flag** (as bit 4) and **Edge of Flight Line** (as bit 5). Stored using a symbol encoder with 64 symbols with 64 instances. The value of **Flags** from the previous item of the same context is used to select the instance.

NOTE 4: The uncompressed item contains the additional flag **Scanner Channel**, which is compressed using the field **dScanner Channel** and not part of **Flags**, so the bit positions are different in this **Flag**-field and the corresponding flags in the uncompressed item.

dIntensity: Difference encoded value with a 16-bit Integer Compressor with 4 instances. The instance is chosen by the value **cpr** (of the current item) as defined for the field **Changed Values**.

Only the difference to **Intensity (previous item of the same context with the same cprgps)** is stored, i.e. the uncompressed value is calculated as **Intensity** := ISum16(**Intensity (previous item of the same context with the same cprgps)**, **dIntensity**), which includes an additional mapping as specified in ISum16().

If no previous item with the same **cprgps** has been processed yet, the **Intensity (initial item of the context)** is used instead of **Intensity (previous item of the same context with the same cprgps)**.

dScan Angle: Only stored if changed compared to the last item of the same context, i.e. if bit 3 of **Changed values** is set. Stored using a 16-bit Integer Compressor with 2 instances. The instance is selected by whether the bit 4 (“gps changed”) of **Changed values** is set or not. Only the difference to the **Scan angle** of the last item (of the same context) is stored, i.e. the uncompressed value is

calculated as **Scan Angle** := ISum16(**Scan Angle (previous item of the same context)**, **dScan Angle**), which includes an additional mapping as specified in ISum16().

User Data: Encoded using a symbol coder with 256 symbols and 64 instances. The instance number is selected by calculating $\left\lfloor \frac{\text{User Data (previous item of the same context)}}{4} \right\rfloor$.

dPoint Source ID: Only stored if changed compared to the last item of the same context, i.e. if bit 5 of **Changed values** is set. Stored using a 16-bit integer encoder. Only the difference to the **Point Source ID** of the last item (of the same context) is stored, i.e. the uncompressed value is calculated as **Point Source ID** := ISum16(**Point Source ID (previous item of the same context)**, **dPoint Source ID**), which the additional mapping as specified in ISum16().

NOTE 5: The 3 following fields **GPS cases**, **dGPS Time (low)** and **GPS Time (extra)** share a layer and are, in combination, used to store the uncompressed field **GPS time**. They are only stored if bit 4 of **Changed values** is set.

NOTE 6: The GPS data is stored very similarly, but not identically, to the GPSTIME11-item; specifically, the **case** and **case_0delta**-fields both have one less value, as the situation “identical to the last gps time” (values 511 for **case** and 0 for **case_0delta**) is already covered by bit 4 of **Changed values**.

LAZ treats the LAS floating point GPS times as integer values (i.e. just stores their byte value, and does integer calculations with those values).

LAZ stores GPS times of items with respect to one of four reference frames, which the gps times are sorted into.

For each reference frame (numbered 0 to 3), the values **delta**, a **counter**, and the **previous GPS Time** are remembered. **counter** keeps track of how often the difference between GPS times is too large (or too low) and couldn’t be stored using low multiples of **delta**; if it happens too often, **delta** will be set to the current difference.

Each of the 4 contexts has its own set of reference frames and values for **delta**, **counter** and the **previous GPS Time**.

The initial reference frame will be “0”, and the “next” reference frame will be 1, 2, 3, 0 in that order.

All values are initialized with 0. The current reference frame can be switched depending on the content of the fields.

previous GPS Time[0] is initialized with the **GPS Time (initial item of the context)**.

All reference frames (of the same context) share the same encoder instances, and especially the same distribution tables.

GPS cases: Only stored if bit 4 of **Changed values** is set, otherwise, the uncompressed field **GPS time** is the same as the **GPS time** of the previous item of the same context.

An array that stores up to 2 **case** and/or **case_0delta**-fields. If a **case** or **case_0delta**-field comes next, depends on the **delta[reference frame] of the same context**: if it is 0, the next entry is a **case_0delta**, otherwise, a **case**-field.

If the first entry is **case** = 512..514 or **case_0delta** = 2..4, the array contains two entries, otherwise just one.

case_0delta is essentially a shorter version of the **case**-field, without all the **case**-options that use a multiple of **delta**, which are not relevant when **delta** is 0.

Both can occur in arbitrary order. E.g., a GPS Time might be compressed using the fields “case, dGPSTime(low)”, another value might be compressed using “case, case_0delta, dGPSTime(low), dGPSTime(high)”. Which fields are needed will be decided on-the-fly.

Table 51 — “GPS Case” Array, Point14

Field Name	Type	Size	Encoder	No of instances	Required
case	unsigned short	2 bytes	515 symbols		
case_0delta	bits	3 bits	5 Symbols		

case:

Table 52 — Values for field “case”, Point14

Value	Description
0	predicted with a delta of zero
1–500	predicted using the current delta times 1 to 500
501–510	predicted using the current delta times -1 to -10
511	starting a new reference frame (using both **dGPS Time (low)** and **GPS Time (extra)**)
512–514	predicted with one of the other three reference frames

For each of the cases, a different encoder instance (of the 32-bit Integer Compressor) for the field **dGPS Time (low)** (with their own distribution table) has to be used (if the field is used at all). Their given numbers are arbitrary and just for convenience (and to align them with those of the **case_0delta**-field). After processing a case, except **512..514**, the **previous GPS Time[reference frame]** is set to the calculated **GPS time**.

The cases are handled as follows:

0: **dGPS Time (low)** is read with instance 7. The uncompressed **GPS Time** := **previous GPS Time[current reference frame] of the same context** + ISum32(0, **dGPS Time (low)**).

counter[reference frame] of the same context is increased by 1. If **counter[reference frame] of the same context** is > 3, **delta[reference frame] of the same context** is set to ISum32(0, **dGPS Time (low)**), and **counter[reference frame] of the same context** is set to 0.

The GPS Time is complete after this, **GPS Time (extra)** is unused/skipped.

1: **dGPS Time (low)** is read with instance 1. It contains the difference from **delta** of the current reference frame, i.e. **GPS time** := **previous GPS Time[reference frame] of the same context** + ISum32(**delta[reference frame] of the same context**, **dGPS Time (low)**).

counter[reference frame] of the same context is reset to 0. The GPS Time is complete after this, **GPS Time (extra)** is unused/skipped.

2..499: **dGPS Time (low)** is read with instance 2 for **case** = 2..9 and instance 3 for **case** = 10..499. It contains the difference from **case** * **delta[reference frame] of the same context**, i.e. **GPS time** := **previous GPS Time[reference frame] of the same context** + ISum32(**case** * **delta[reference frame] of the same context**, **dGPS Time (low)**). The GPS Time is complete after this, **GPS Time (extra)** is unused/skipped.

500: **dGPS Time (low)** is read with the instance 4. It contains the difference from 500 * **delta[reference frame] of the same context**, i.e. **GPS time** := **previous GPS Time[reference frame] of the same context** + ISum32(500 * **delta[reference frame] of the same context**, **dGPS Time (low)**).

counter[reference frame] of the same context is increased by 1. If **counter[reference frame] of the same context** is > 3, **delta[reference frame] of the same context** is set to ISum32(500 * **delta[reference frame] of the same context**, **dGPS Time (low)**), and **counter[reference frame] of the same context** is set to 0.

The GPS Time is complete after this, **GPS Time (extra)** is unused/skipped.

501..509: This is used for negative differences. **dGPS Time (low)** is read with instance 5. **GPS time** is calculated as **previous GPS Time[reference frame] of the same context** + ISum32(- (**case** - 500) * **delta[reference frame] of the same context**, **dGPS Time (low)**). The GPS Time is complete after this.

510: This is also used for negative differences. **dGPS Time (low)** is read with instance 6. **GPS time** is calculated as **previous GPS Time[reference frame] of the same context** + ISum32(- 10 * **delta[reference frame] of the same context**, **dGPS Time (low)**).

counter[reference frame] of the same context is increased by 1. If **counter[reference frame] of the same context** is > 3, **delta[reference frame] of the same context** is set to ISum32(- 10 * **delta[reference frame] of the same context**, **dGPS Time (low)**), and **counter[reference frame] of the same context** is set to 0.

The GPS Time is complete after this, **GPS Time (extra)** is unused/skipped.

511: The gps time is stored using both time fields. (Due to the ambiguity in interpreting the numbers, the data types are specified, as the integer compressor returns 32 bit signed integer values, and the gps time is a 64 bit floating point value treated as a 64 bit unsigned integer). First, the instance 8 (of the 32-bit Integer Compressor) for **dGPS Time (low)** and the RAW encoder for **GPS Time (extra)** are used to calculate (unsigned long long)**tmp** := (unsigned long long) ISum32(((signed integer)((unsigned long long)**previous GPS Time[reference frame] of the same context** RIGHT SHIFT BY 32), **dGPS Time (low)**)) LEFT SHIFT BY 32 + **GPS Time (extra)**.

Then the current reference frame is changed to the next reference frame in line (in the order 0, 1, 2, 3, 0). **GPS Time** is then set to **tmp**. The **previous GPS Time[reference frame] of the same context** of the new current reference frame is now **GPS Time**, while **previous GPS Time[]** for the previous reference frame is unchanged. **delta[reference frame] of the same context** and **delta[reference frame] of the same context** are set to 0.

The GPS Time is complete after this. **511** is only allowed to be the first entry in the **GPS cases**-array.

512..514: The current reference frame is changed to the 1st (512), 2nd (513) or 3rd (514) successor, in the order 0, 1, 2, 3, 0. Then, the next entry in the **GPS cases** array is evaluated. **512..514** are only allowed to be the first entry in the **GPS cases**-array. **previous GPS Time[reference frame]** is not changed here.

case_0delta:

Table 53 — Values for field “case_0delta”, Point14

Value	Description
0	stored using just **dGPS Time (low)**
1	starting a new reference frame (using both **dGPS Time (low)** and **GPS Time (extra)**)
2-4	predicted with one of the other three reference frames

The field **case_0delta** is a shorter version of the **case**-field, for cases where **delta[reference frame] of the same context** = 0 (so the multiplications for **case** 0 to 510 are not relevant). The instance numbers are the same as used for the **case** field. After processing a case, except **2..4**, the **previous GPS Time[reference frame]** is set to the calculated **GPS time**.

The cases are:

0: **dGPS Time (low)** is read with instance 0, and **GPS time** := **previous GPS Time[reference frame] of the same context** + ISum32(0, **dGPS Time (low)**). **delta[reference frame] of the same context** is set to ISum32(0, **dGPS Time (low)**). (Note that this, and the instance number, differ from **case** = 1).

counter[reference frame] of the same context is reset to 0. The GPS Time is complete after this, **GPS Time (extra)** is unused/skipped.

1: Identical to **case** = 511 (stored using both **dGPS Time (low)** and **GPS Time (extra)**). **1** is only allowed to be the first entry in the **GPS cases**-array.

2..4: Analogously to **case** = **512..514**: The current reference frame is changed to the 1st (**case** = 2), 2nd (3) or 3rd (4) successor, in the order 0, 1, 2, 3, 0. Then, the next entry in the **cases** array is evaluated. **2..4** are only allowed to be the first entry in the **cases**-array. **previous GPS Time[reference frame]** is not changed here.

dGPS Time (low): Stored using a 32-bit Integer Compressor with 9 different instances. Usage is described above. Optional field.

GPS Time (extra): Stored using a RAW Encoder with 32 bits. Only stored if the full gps time has to be stored (i.e. for **case** = 512 and **case_0delta** = 2). Usage is described above.

14.2. RGB14 (version 3)

The RGB14-item compresses the **Red**, **Green** and **Blue**-fields that are part of LAS Point Data Record Formats 7, 8 and 10. The format of the uncompressed item is as follows:

Table 54 — RGB14-item format, uncompressed

Field Name	Format	Size	Required
Red	unsigned short	2 bytes	*
Green	unsigned short	2 bytes	*
Blue	unsigned short	2 bytes	*

The first entry per chunk is this uncompressed item.

Details about the fields are described in Annex A.

The compressed item first stores a field that indicates which bytes are changed. The high and low bytes of the red, green and blue channel are treated separately. If they changed, only the difference to the corresponding value of the previous item is stored.

As there is often a correlation between changes to the red, green, and blue channels (e.g. if the intensity for a gray color changes), the delta for the red channel is added to the green channel first, and only the difference to that is stored. For the blue channel, the average delta of the red and green channel is used.

The item has 4 contexts, i.e. 4 sets of all instances (all initialized independently). The previous item used is always the previous item from the same context (i.e. there are 4 previous items). The initial item per context (that is set when the context is first used) is considered the previous item for the first item processed.

The context is the context of the previously processed Point14-item.

NOTE: As described in Contexts, the “previous item of the same context” for the RGB14-item has a non-obvious implementation, and specifically might refer to a different context after a context switch.

The layer for this item can be empty, in which case all values of that layer are the same as the first (uncompressed) item for all items of that chunk.

The algorithm clamp255() has been defined in Clause 13.3 for the RGB12-item.

The compressed item is stored as follows (this is identical to the RGB12-item):

Table 55 — RGB14-item format, compressed

Field Name	Type	Size	Encoder	No of instances	Required	Layer
Changed values	bits	7 bits	128 Symbols	1	*	10
dRed (low)	byte	1 byte	256 Symbols	1		10
dRed (high)	byte	1 byte	256 Symbols	1		10
dGreen (low)	byte	1 byte	256 Symbols	1		10
dGreen (high)	byte	1 byte	256 Symbols	1		10
dBlue (low)	byte	1 byte	256 Symbols	1		10
dBlue (high)	byte	1 byte	256 Symbols	1		10

Changed values: Indicates, which of the fields have been changed. Only changed fields are stored.

Table 56 — Bit-values for field “Changed values”, RGB14 and RGBNIR14

Bit	Description
0	Field dRed (low) is stored, otherwise dRed (low) is 0
1	Field dRed (high) is present, otherwise dRed (high) is 0
2	Field dGreen (low) is present, otherwise dGreen (low) is 0
3	Field dGreen (high) is present, otherwise dGreen (high) is 0
4	Field dBlue (low) is present, otherwise dBlue (low) is 0
5	Field dBlue (high) is present, otherwise dBlue (high) is 0
6	(Uncompressed) **Green** and **Blue** are identical to **Red**, bits 2-5 are ignored

If bit 6 is set, **Green** and **Blue** are equal to the uncompressed value of **Red** (i.e. after processing **dRed (low)** and **dRed (high)**). In that case, bits 2 to 5 are ignored, the fields **dGreen (low)**, **dGreen (high)**, **dBlue (low)** and **dBlue (high)** are not stored, and the calculation described below for **Green** and **Blue** based on those fields is not used.

dRed (low): Only stored if bit 0 of **Changed values** is set, otherwise considered 0. Stores the delta to the lower bit of **Red** of the previous item (of the same context). Then, the sum is mapped to 0..255 using modulo 256 and adding 256 if negative. I.e. **lower byte of Red** := (**lower byte of Red (previous item of the same context)** + **dRed (low)** + 256) MOD 256.

dRed (high): Only stored if bit 1 of **Changed values** is set, otherwise considered 0. Stores the delta to the higher bit of **Red** of the previous item (of the same context). Then, the sum is mapped to 0..255. I.e. **higher byte of Red** := (**higher byte of Red (previous item of the same context)** + **dRed (high)** + 256) MOD 256.

Note: The remaining four fields are only used if bit 6 of **Changed values** is not set.

dGreen (low): Only stored if bit 2 of **Changed values** is set (and bit 6 is not set), otherwise the lower byte of **Green** is the same as the lower byte of **Green** of the previous item (of the same context). If stored, the difference for the **Red** value is added first, and **dGreen (low)** stores only the difference to that.

Calculate diff := **lower byte of Red** — **lower byte of Red (previous item of the same context)**. This value is added to the previous value of the lower byte of **Green**, clamping to 255. Then, the total sum is mapped to 0..255. I.e. **lower byte of Green** := (**dGreen (low)** + clamp255(**lower byte of Green (previous item of the same context)** + diff) + 256) MOD 256.

dGreen (high): Only stored if bit 3 of **Changed values** is set (and bit 6 is not set), otherwise the higher byte of **Green** is the same as the higher byte of **Green** of the previous item (of the same context). If stored, similar to **dGreen (low)**, the difference for the **Red** value is calculated, clamped and added:

Calculate diff := **higher byte of Red** — **higher byte of Red (previous item of the same context)**. This value is added to the previous value of the higher byte of **Green**, clamping at 255. Then, the total sum is mapped to 0..255. I.e. **higher byte of Green** := (**dGreen (low)** + clamp255(**higher byte of Green (previous item of the same context)** + diff) + 256) MOD 256.

dBlue (low): Only stored if bit 4 of **Changed values** is set (and bit 6 is not set), otherwise the lower byte of **Blue** is the same as the lower byte of **Blue** of the previous item (of the same context). If stored, the average difference for the **Red** and **Green** value is added first:

Calculate diff := round_towards_0((**lower byte of Red** — **lower byte of Red (previous item of the same context)** + **lower byte of Green** — **lower byte of Green (previous item of the same context)**) / 2), with round_towards_0() as defined in Conventions (i.e. rounding down above 0, and rounding up below 0). This is added and clamped. Then, the total sum is mapped to 0..255. That means, **lower byte of Blue** := (**dBlue (low)** + clamp255(**lower byte of Blue (previous item of the same context)** + diff) + 256) MOD 256.

dBlue (high): Only stored if bit 5 of **Changed values** is set (and bit 6 is not set), otherwise the higher byte of **Blue** is the same as the higher byte of **Blue** of the previous item (of the same context). As for **dBlue (low)**, the average difference for the **Red** and **Green** value is added first:

Calculate diff := round_towards_0((**higher byte of Red** — **higher byte of Red (previous item of the same context)** + **higher byte of Green** — **higher byte of Green (previous item of the same context of the same context)**) / 2). This is added and clamped. Then, the total sum is mapped to 0..255. I.e. **lower byte of Blue** := (**dBlue (low)** + clamp255(**higher byte of Blue (previous item of the same context)** + diff) + 256) MOD 256.

14.3. RGBNIR14 (version 3)

The RGBNIR14-item compresses the **Red**, **Green**, **Blue** and **NIR** (near infrared)-fields that are part of LAS Point Data Record Formats 8 and 10. The format of the uncompressed item is as follows:

Table 57 — RGBNIR14-item format, uncompressed

Field Name	Format	Size	Required
Red	unsigned short	2 bytes	*
Green	unsigned short	2 bytes	*
Blue	unsigned short	2 bytes	*
NIR	unsigned short	2 bytes	*

The first entry per chunk is this uncompressed item.

Details about the fields are described in Annex A.

The 3 RGB-fields are compressed in exactly the same way as the RGB14-item, and in their own layer.

For the additional NIR-field, first, a field is stored that indicates if the high or low byte of the value has changed - the high and low bytes of the NIR-field are treated separately. If they changed, only the difference to the corresponding value of the previous item is stored.

The item has 4 contexts, i.e. 4 sets of all instances (all initialized independently). The previous item used is always the previous item from the same context (i.e. there are 4 previous items). The initial item per context (that is set when the context is first used) is considered the previous item for the first item processed.

The context is the context of the previously processed Point14-item.

NOTE: As described in Contexts, the “previous item of that context” for the RGBNIR14-item has a non-obvious implementation, and specifically might refer to a different context after a context switch.

The fields **Changed values NIR**, **dNIR (low)** and **dNIR (high)** are in a separate layer stream from the other 7 fields.

A layer can be empty, in which case all values of that layer are the same as the first (uncompressed) item for all items of that chunk.

The compressed item is stored as follows:

Table 58 — RGBNIR14-item format, compressed

Field Name	Type	Size	Encoder	No of instances	Required	Layer
Changed values	bits	7 bits	128 Symbols	1	*	11
dRed (low)	byte	1 byte	256 Symbols	1		11
dRed (high)	byte	1 byte	256 Symbols	1		11
dGreen (low)	byte	1 byte	256 Symbols	1		11
dGreen (high)	byte	1 byte	256 Symbols	1		11
dBlue (low)	byte	1 byte	256 Symbols	1		11
dBlue (high)	byte	1 byte	256 Symbols	1		11
Changed values NIR	bits	2 bits	4 Symbols	1	*	12
dNIR (low)	byte	1 byte	256 Symbols	1		12

Field Name	Type	Size	Encoder	No of instances	Required	Layer
dNIR (high)	byte	1 byte	256 Symbols	1		12

The fields **Changed values**, **dRed (low)**, **dRed (high)**, **dGreen (low)**, **dGreen (high)**, **dBlue (low)** and **dBlue (high)** are stored identically as the RGB14-item (and in their own layer), and are specified there.

Changed values NIR: Indicates, which of the NIR-fields have been changed. Only changed fields are stored.

Table 59 — Bit-values for field “Changed Values NIR”, RGBNIR14

Bit	Description
0	Field dNIR (low) is stored, otherwise dNIR (low) is 0
1	Field dNIR (high) is present, otherwise dNIR (high) is 0

dNIR (low): Only stored if bit 0 of **Changed values NIR** is set, otherwise considered 0 for the following calculation. Stores the delta to the lower bit of **NIR** of the previous item (of the same context). The sum is then mapped to 0..255, i.e. mod 256 and adding 256 if negative. I.e. **lower byte of NIR** := (**lower byte of NIR (previous item of the same context)** AND 255 + **dNIR (low)** + 256) MOD 256.

dNIR (high): Only stored if bit 1 of **Changed values NIR** is set, otherwise considered 0. Stores the delta to the higher bit of **NIR** of the previous item (of the same context). The sum is then mapped to 0..255, i.e. mod 256 and adding 256 if negative. I.e. **higher byte of NIR** := (**higher byte of Red (previous item of the same context)** + **dNIR (high)** + 256) MOD 256.

14.4. BYTE14 (version 3)

The BYTE14-item compresses any additional bytes that are optionally appended to LAS Point Data Record Formats 6 through 10. The number of bytes, **n**, is declared in the LAZ VLR, specified in Clause 7. The uncompressed item looks as:

Table 60 — BYTE14-item format, uncompressed

Field Name	Format	Size	Required
Bytes	byte[n]	n bytes	*

The first entry per chunk is this uncompressed item.

LAZ compresses each byte separately and in its own layer, as given by the **Byte14: bytes layer size** in the layer table.

The item has 4 contexts, e.g. specifically 4 sets of all instances (all initialized independently). The previous item used is always the previous item from the same context (so there are 4 previous items). The initial item per context (that is set when the context is first used) is considered the previous item for the first item processed.

The context is the context of the previously processed Point14-item.

NOTE: As described in Contexts, the “previous item of that context” for the Byte14-item has a non-obvious implementation, and specifically might refer to a different context after a context switch.

The compressed item is stored as follows:

Table 61 — BYTE14-item format, compressed

Field Name	Type	Size	Encoder	No of instances	Required	Layer
dBytes	byte[n]	n byte	256 Symbols	n		14-

dBytes: Array of n bytes, using a separate symbol encoder for each of the n bytes (each with 256 symbols). Stores the difference to the corresponding byte of the previous item. The sum is then mapped to 0..255, i.e. mod 256 and adding 256 if negative. I.e. **Bytes[i]** := (**Bytes[i] (previous item of the same context)** + **dBytes[i]** + 256) mod 256.

The layer for some byte[i] can be empty, in which case the i-th Byte-value is the same as the i-th Byte-value in the first (uncompressed) item for all items of that chunk.

14.5. Wavepacket14 (version 3)

The Wavepacket14-item compresses the Waveform data that is part of LAS Point Data Record Formats 9 and 10. The format of the uncompressed item is as follows:

Table 62 — Wavepacket14-item format, uncompressed

Field Name	Format	Size	Required
Wave Packet Descriptor Index	unsigned char	1 byte	*
Byte Offset to Waveform Data	unsigned long long	8 bytes	*
Waveform Packet Size in Bytes	unsigned long	4 bytes	*
Return Point Waveform Location	float	4 bytes	*
Parametric dx	float	4 bytes	*
Parametric dy	float	4 bytes	*
Parametric dz	float	4 bytes	*

The first entry per chunk is this uncompressed item.

Details about the fields are described in Annex A.

All LAS floating point values are treated as signed 32-bit integer values, i.e. uses the byte representation of the floating point values as is.

All fields except the **Byte Offset to Waveform Data** are stored in a straight forward way using the specified encoder. That offset-field however uses an additional 2-bit field that determines 4 cases: (0) the value is the same as the offset of the previous item (of the same context), (1) it just differs by the packet size of the previous item, (2) the difference is small enough to be stored with just 4 bytes, and (3) the full 8 byte value is stored.

The item has 4 contexts, specifically 4 sets of all instances (all initialized independently). The previous item used is always the previous item from the same context (i.e. there are 4 previous items). The initial item per context (that is set when the context is first used) is considered the previous item for the first item processed.

The context is the context of the previously processed Point14-item.

NOTE: As described in Contexts, the “previous item of that context” for the Wavepacket14-item has a non-obvious implementation, and specifically might refer to a different context after a context switch.

Except for those contexts, the format is identical to the Wavepacket13-item. Additionally, all fields are in the same layer.

The layer can be empty, in which case all values are the same as the first (uncompressed) item for all items of that chunk.

The compressed item is stored as follows:

Table 63 — Wavepacket14-item format, compressed

Field Name	Type	Size	Encoder	No of instances	Required	Layer
Wave Packet Descriptor Index	byte	1 byte	256 Symbols	1	*	13
Offset Diff Type	bits	2 bits	4 Symbols	4	*	13
dOffset Diff (low)	long	4 bytes	Integer Compressor, 32 bits	1		13
Offset (full)	long long	8 bytes	Raw, 64 bit	1		13
dPacket Size	long	4 bytes	Integer Compressor, 32 bits	1	*	13
dReturn Point	long	4 bytes	Integer Compressor, 32 bits	1	*	13
dPdXYZ	long[3]	12 bytes	Integer Compressor, 32 bits	3	*	13

Wave Packet Descriptor Index: Stored using a symbol encoder with 256 symbols.

Offset Diff Type: Stored using a symbol encoder with 4 symbols, and 4 different instances. The instance is chosen by the value of **Offset Diff Type** of the previous item (of the same context), or 0 for the first compressed item (as the uncompressed first item does not have this field). Encodes how **Byte Offset to Waveform Data** is compressed:

0: **Byte Offset to Waveform Data** is the same as the value from the previous item (of the same context). **dOffset Diff (low)** and **Offset (full)** are not stored or used.

1: The offset is increased by the packet size of the previous item, **Byte Offset to Waveform Data** := **Byte Offset to Waveform Data (previous item of the same context)** + **Waveform Packet Size in Bytes (previous item of the same context)**. **dOffset Diff (low)** and **Offset (full)** are not stored.

2: **Byte Offset to Waveform Data** can be encoded with just a small difference, called **Offset Diff**, to the previous item. Stored in **dOffset Diff (low)** is only the difference to the last value of **Offset Diff** of the same context used in this chunk (i.e. by the last item that had an **Offset Diff Type** of 2) or to 0, if it has not been used yet. I.e. **Byte Offset to Waveform Data** := **Byte Offset to Waveform Data (previous item of the same context)** + ISum32(**Offset Diff (previous item of the same context with Offset Diff Type of 2)**, **dOffset Diff (low)**). Note that **Offset Diff** is only used when **Offset Diff Type** is 2. **Offset (full)** is not stored for this **Offset Diff Type**.

3: **Byte Offset to Waveform Data** is stored as the full 8 byte value in **Offset (full)**, i.e. **Byte Offset to Waveform Data** := **Offset (full)**. **Offset Diff (low)** is not stored.

dOffset Diff (low): Only stored if **Offset Diff Type** is 2. Using a 32-bit Integer Compressor. Used as described above.

Offset (full): Only stored if **Offset Diff Type** is 3. Using a Raw encoder with 64 bits, contains the full **Byte Offset to Waveform Data** value.

dPacket Size: Using a 32-bit Integer Compressor, stores the difference to the previous value of **Waveform Packet Size in Bytes**, i.e. **Waveform Packet Size in Bytes** := ISum32(**Waveform Packet Size in Bytes (previous item of the same context)**, **dPacket Size**).

dReturn Point: Using a 32-bit Integer Compressor, stores the difference to the previous value of **Return Point**, i.e. **Return Point** := ISum32(**Return Point (previous item of the same context)**, **dReturn Point**). Note that the LAS floating point value **Return Point Waveform Location** is treated as a signed 32-bit integer value in this calculation.

dPdXYZ: An array of 3 values, namely **dPdx**, **dPdy** and **dPdz**. Those store the difference to the **Parametric dx**, **Parametric dx** and **Parametric dz** values to those values of the previous item of the same context, where the floating point values are treated as signed long integer values, i.e. the difference is taken from and added to the 4-byte representation of the floating point value: **Parametric dx (as signed 32 bit integer)** := ISum32(**Parametric dx (as signed 32 bit integer) (previous item of the same context)**, **dPdx**), and the same for **Parametric dy** and **Parametric dz**. Using a 32-bit Integer Compressor with 3 instances, one for each of **dPdx**, **dPdy** and **dPdz**. Note that this is not the same as storing the 3 fields using a separate 32-bit Integer Compressor for each, as they share the symbol encoders for **k**, as described in Clause 10.5.

15. LAZ Legacy information

15.1. LAZ 1.2 and 1.3 formats

This specification covers only the LAZ 1.4 format based on the current LAS 1.4 format.

Just as an overview however, a legacy support with LAZ 1.2 and 1.3 headers, based on the LAS 1.2 and 1.3 format, exists, that can encode older formats without converting them to LAS 1.4 first. The LAZ 1.2 and 1.3 formats are similar to LAZ 1.4, but are using the (slightly smaller) LAS 1.2 and 1.3 format headers.

All limitations of a LAS 1.2 or 1.3 format will also apply to the legacy LAZ 1.2 or 1.3 format, e.g. regarding EVLRs, the limit of 2^{32} points, and so on.

Also, LAS Point Data Record Formats 6 to 10 are not supported, so the LAZ items for those formats, as well as the features exclusive to those items (e.g. layered compression), cannot be used.

Otherwise, the LAZ-specific encodings (with item version 2 for LAS Point Data Record Formats 0 to 5) are unchanged, including the structure of the special LAZ VLR, the compressed data block and the compressors.

Item version 1, except for Wavepacket13, is deprecated, and not covered in this standard.

15.2. LAS 1.4 compatibility mode (Legacy information)

15.2.1. Overview

Setting the bit **LAS 1.4 compatibility mode** to 1 in the **Options** field of the LAZ Special VLR indicates that LAS Point Data Record Formats 6 through 10 have been stored as LAS Point Data Record Formats 0 through 5 (or rather their respective LAZ items), plus extra bytes that contain the additional data exclusive to formats 6 to 10.

This mode can only be used if the LAZ header format uses the legacy LAZ 1.2 or 1.3 header (and point formats).

These are not part of this specification, and information about this mode is only meant to provide a quick, informal overview.

This mode can only be used if the number of points do not exceed 2^{32} .

The following chapters describe the mapping used between the formats.

15.2.2. Mapping of Point14-item

A Point14-item will be stored as a Point10-item and a GPStime11-item plus additional bytes:

— The fields **X**, **Y**, **Z**, **Intensity**, **User Data** and **Point Source ID** are identical between those items, as well as the flags **Scan Direction Flag** and **Edge of Flight Line**.
— The Point14-field **GPS time** field is stored as a **GPStime11**-item, the fields are identical.
— The following 5 bytes are stored as Byte-items, behind potentially already existing extra bytes.

Table 64 — Point14-fields stored as extra bytes in LAS 1.4 compatibility mode

Field Name	Format	Size	Required
Scan Angle	short	2 bytes	*
internal	2 bits	2 bits (bits 0 — 1)	*
Scanner Channel	2 bits (bits 2 — 3)	2 bits	*
Classification Flags	4 bits (bits 4 — 7)	4 bits	
Classification	unsigned char	1 byte	*
Return Number	4 bits (bits 0 — 3)	4 bits	*
Number of Returns (given pulse)	4 bits (bits 4 — 7)	4 bits	*

The LAS 1.2 and LAS 1.3 fields are not identical to these fields, e.g., the LAS 1.4 **Return Number** has 4 bits, while LAS 1.3 **Return number** only uses 3 bits. When storing the Point10-item from Point14-item-data, any reasonable transformation can be done (for example, if the Point14-item has a return number above 7, use 7 for the corresponding compatible Point10-item).

This is, just as the compatibility mode in general, not specified in this specification.

15.2.3. Mapping of RGB14-item

The RGB14-item is stored as an RGB12-item, the fields are identical.

15.2.4. Mapping of RGBNIR14-item

The RGBNIR14-item is stored as an RGB12-item plus 2 extra Byte10-items.

— The fields **Red**, **Green**, and **Blue** of the RGBNIR14-item are identical to the fields of the RGB12-item.
— The 2 bytes for the **NIR**-field of the RGBNIR14-item are stored as 2 extra Byte10-Bytes, appended after any existing bytes, and after the Point14 extra bytes.

15.2.5. Mapping of Wavepacket14-item

The Wavepacket14-item is stored as a Wavepacket13-item, the fields are identical.

15.2.6. Mapping of Byte14-item

The Byte14-items are stored as Byte-items, the fields are identical.

15.2.7. Additional LAS Header fields

Additional header fields from LAS Header 1.4 that are not part of the LAS 1.2 or LAS 1.3 header are stored as the payload of an additional VLR identified with the string “lascompatible” in the field **User Id** and the value 22204 in the field **Record Id**.

Table 65 — Point14-fields stored as extra bytes in LAS 1.4 compatibility mode

Field Name	Format	Size	Required
Version	unsigned short	2 bytes	*
Compatible version	unsigned short	2 bytes	*
Reserved	unsigned long	4 bytes	*
Start of Waveform Data Packet Record	unsigned long long	8 bytes	*
Start of First Extended Variable Length Record	unsigned long long	8 bytes	*
Number of Extended Variable Length Records	unsigned long	4 bytes	*
Number of Point Records	unsigned long long	8 bytes	*

Field Name	Format	Size	Required
Number of Points by Return	unsigned long long[15]	120 bytes	*

Three LAZ-specific fields are used in this VLR, the remaining fields are taken from the original LAS 1.4 header:

Version: Version information.

Compatible version: Shall be 3

Reserved: Shall be 0.

Annex A
(normative)
APPENDIX: LAS Point Data Record Format

A.1. Scope

The (uncompressed) LAS Point Data Record Formats are defined in the LAS 1.4 specification. LAZ does not change their meaning or specification. For convenience and completeness, this chapter gives a slightly shortened version of the field description in the LAS 1.4 specification.

A.2. LAS Point Data Record Format 0

Point Data Record Format 0 contains the core 20 bytes that are shared by Point Data Record Formats 0 to 5.

Table A.1 — LAS Point Data Record Format 0, compressed by LAZ item “Point10”

Field Name	Format	Size	Required
X	long	4 bytes	*
Y	long	4 bytes	*
Z	long	4 bytes	*
Intensity	unsigned short	2 bytes	
Return Number	3 bits (bits 0 — 2)	3 bits	*
Number of Returns (given pulse)	3 bits (bits 3 — 5)	3 bits	*
Scan Direction Flag	1 bit (bit 6)	1 bit	*
Edge of Flight Line	1 bit (bit 7)	1 bit	*
Classification	unsigned char	1 byte	*
Scan Angle Rank (-90 to +90), Left side	char	1 byte	*
User Data	unsigned char	1 byte	
Point Source ID	unsigned short	2 bytes	*

X, Y, Z: The X, Y and Z values are stored as long integers. The X, Y, and Z values are used in conjunction with the scale values and the offset values to determine the coordinate for each point as described in the Public Header Block section.

Intensity: The intensity value is the integer representation of the pulse return magnitude. This value is optional and system specific. However, it should always be included if available. If Intensity is not included, this value must be set to zero.

Intensity, when included, is always normalized to a 16 bit, unsigned value by multiplying the value by 65,536/(intensity dynamic range of the sensor). For example, if the dynamic range of the sensor is 10 bits, the scaling value would be (65,536/1,024). This normalization is required to ensure that data from different sensors can be correctly merged.

Please note that the following four fields (Return Number, Number of Returns, Scan Direction Flag and Edge of Flight Line) are bit fields within a single byte.

Return Number: The Return Number is the pulse return number for a given output pulse. A given output laser pulse can have many returns, and they must be marked in sequence of return. The first return will have a Return Number of one, the second a Return Number of two, and so on up to five returns. The Return Number must be between 1 and the Number of Returns, inclusive.

Number of Returns (given pulse): The Number of Returns is the total number of returns for a given pulse. For example, a laser data point may be return two (Return Number) within a total number of five returns.

Scan Direction Flag: The Scan Direction Flag denotes the direction at which the scanner mirror was traveling at the time of the output pulse. A bit value of 1 is a positive scan direction, and a bit value of 0 is a negative scan direction (where positive scan direction is a scan moving from the left side of the in-track direction to the right side and negative the opposite).

Edge of Flight Line: The Edge of Flight Line data bit has a value of 1 only when the point is at the end of a scan. It is the last point on a given scan line before it changes direction or the mirror facet changes.

Classification: This field represents the “class” attributes of a point. If a point has never been classified, this byte must be set to zero. The format for classification is a bit encoded field with the lower five bits used for the class and the three high bits used for flags. The bit definitions are listed in Table A.2 and the classification values in Table A.3.

Table A.2 — Classification Bit Field Encoding for LAS Point Data Record types 0 to 5

Bit	Field Name	Description
0:4	Classification	Standard ASPRS classification from 0 — 31 as defined in the classification table for legacy point formats (Table A.3)
5	Synthetic	If set, this point was created by a technique other than direct observation such as digitized from a photogrammetric stereo model or by traversing a waveform. Point attribute interpretation might differ from non-Synthetic points. Unused attributes must be set to the appropriate default value.
6	KeyPpoint	If set, this point is considered to be a model key-point and therefore generally should not be withheld in a thinning algorithm.
7	Withheld	If set, this point should not be included in processing (synonymous with Deleted).

Note that bits 5, 6 and 7 are treated as flags and can be set or clear in any combination. For example, a point with bits 5 and 6 both set to one and the lower five bits set to 2 would be a ground point that had been Synthetically collected and marked as a model Key-Point.

Table A.3 — SPRS Standard LIDAR Point Classes for LAS Point Data Record types 0 to 5

Classification Value (bits 0:4)	Meaning
0	Created, never classified
1	Unclassified
2	Ground
3	Low Vegetation
4	Medium Vegetation
5	High Vegetation
6	Building
7	Low Point (noise)
8	Model Key-point (mass point)
9	Water
10	Reserved for ASPRS Definition
11	Reserved for ASPRS Definition
12	Overlap Points
13-31	Reserved for ASPRS Definition

A note on Bit Fields — The LAS storage format is “Little Endian”. This means that multi-byte data fields are stored in memory from the least significant byte at the low address to the most significant byte at the high address. Bit fields are always interpreted as bit 0 set to 1 equals 1, bit 1 set to 1 equals 2, bit 2 set to 1 equals 4 and so forth.

Scan Angle Rank: The Scan Angle Rank is a signed one-byte integer with a valid range from -90 to +90. The Scan Angle Rank is the angle (rounded to the nearest integer in the absolute value sense) at which the laser point was output from the laser system including the roll of the aircraft. The scan angle is within 1 degree of accuracy from +90 to -90 degrees. The scan angle is an angle based on 0 degrees being nadir, and -90 degrees to the left side of the aircraft in the direction of flight.

User Data: This field may be used at the user’s discretion.

Point Source ID: This value indicates the source from which this point originated. A source is typically defined as a grouping of temporally consistent data, such as a flight line or sortie number for airborne systems, a route number for mobile systems, or a setup identifier for static systems. Valid values for this field are 1 to 65,535 inclusive. Zero is reserved as a convenience to system implementers.

A.3. LAS Point Data Record Format 1

Point Data Record Format 1 is the same as Point Data Record Format 0 with the addition of GPS Time.

Table A.4 — LAS Point Data Record Format 1, compressed by LAZ items “Point10” and “GPSTime11”

Field Name	Format	Size	Required
X	long	4 bytes	*
Y	long	4 bytes	*
Z	long	4 bytes	*
Intensity	unsigned short	2 bytes	
Return Number	3 bits (bits 0 — 2)	3 bits	*
Number of Returns (given pulse)	3 bits (bits 3 — 5)	3 bits	*
Scan Direction Flag	1 bit (bit 6)	1 bit	*
Edge of Flight Line	1 bit (bit 7)	1 bit	*
Classification	unsigned char	1 byte	*
Scan Angle Rank (-90 to +90), Left side	char	1 byte	*
User Data	unsigned char	1 byte	
Point Source ID	unsigned short	2 bytes	*
GPS Time	double	8 bytes	*

GPS Time: The GPS Time is the double floating point time tag value at which the point was acquired. It is GPS Week Time if the Global Encoding low bit is clear and Adjusted Standard GPS Time if the Global Encoding low bit is set (**Global Encoding** in the Public Header Block description).

A.4. LAS Point Data Record Format 2

Point Data Record Format 2 is the same as Point Data Record Format 0 with the addition of three color channels. These fields are used when “colorizing” a LIDAR point using ancillary data, typically from a camera.

Table A.5 — LAS Point Data Record Format 2, compressed by LAZ items “Point10” and “RGB12”

Field Name	Format	Size	Required
X	long	4 bytes	*
Y	long	4 bytes	*
Z	long	4 bytes	*
Intensity	unsigned short	2 bytes	
Return Number	3 bits (bits 0 — 2)	3 bits	*
Number of Returns (given pulse)	3 bits (bits 3 — 5)	3 bits	*
Scan Direction Flag	1 bit (bit 6)	1 bit	*
Edge of Flight Line	1 bit (bit 7)	1 bit	*
Classification	unsigned char	1 byte	*
Scan Angle Rank (-90 to +90), Left side	char	1 byte	*
User Data	unsigned char	1 byte	
Point Source ID	unsigned short	2 bytes	*
Red	unsigned short	2 bytes	*
Green	unsigned short	2 bytes	*
Blue	unsigned short	2 bytes	*

Red, Green, and Blue: The Red, Green, and Blue image channel values associated with this point.

The Red, Green, Blue values should always be normalized to 16 bit values. For example, when encoding an 8 bit per channel pixel, multiply each channel value by 256 prior to storage in these fields. This normalization allows color values from different camera bit depths to be accurately merged.

A.5. LAS Point Data Record Format 3

Point Data Record Format 3 is the same as Point Data Record Format 2 with the addition of GPS Time.

Table A.6 — LAS Point Data Record Format 3, compressed by LAZ items “Point10”, “GPSTime11” and “RGB12”

Field Name	Format	Size	Required
X	long	4 bytes	*
Y	long	4 bytes	*
Z	long	4 bytes	*
Intensity	unsigned short	2 bytes	
Return Number	3 bits (bits 0 — 2)	3 bits	*
Number of Returns (given pulse)	3 bits (bits 3 — 5)	3 bits	*
Scan Direction Flag	1 bit (bit 6)	1 bit	*
Edge of Flight Line	1 bit (bit 7)	1 bit	*
Classification	unsigned char	1 byte	*
Scan Angle Rank (-90 to +90), Left side	char	1 byte	*
User Data	unsigned char	1 byte	
Point Source ID	unsigned short	2 bytes	*
GPS Time	double	8 bytes	*
Red	unsigned short	2 bytes	*
Green	unsigned short	2 bytes	*
Blue	unsigned short	2 bytes	*

A.6. LAS Point Data Record Format 4

Point Data Record Format 4 adds Wave Packets to Point Data Record Format 1.

Table A.7 — LAS Point Data Record Format 4, compressed by LAZ items “Point10”, “GPSTime11” and “Wavepacket13”

Field Name	Format	Size	Required
X	long	4 bytes	*
Y	long	4 bytes	*
Z	long	4 bytes	*
Intensity	unsigned short	2 bytes	
Return Number	3 bits (bits 0 — 2)	3 bits	*
Number of Returns (given pulse)	3 bits (bits 3 — 5)	3 bits	*
Scan Direction Flag	1 bit (bit 6)	1 bit	*
Edge of Flight Line	1 bit (bit 7)	1 bit	*
Classification	unsigned char	1 byte	*
Scan Angle Rank (-90 to +90), Left side	char	1 byte	*
User Data	unsigned char	1 byte	
Point Source ID	unsigned short	2 bytes	*
GPS Time	double	8 bytes	*
Wave Packet Descriptor Index	unsigned char	1 byte	*
Byte Offset to Waveform Data	unsigned long long	8 bytes	*
Waveform Packet Size in Bytes	unsigned long	4 bytes	*
Return Point Waveform Location	float	4 bytes	*
Parametric dx	float	4 bytes	*
Parametric dy	float	4 bytes	*
Parametric dz	float	4 bytes	*

Wave Packet Descriptor Index: This value plus 99 is the Record ID of the Waveform Packet Descriptor and indicates the User Defined Record that describes the waveform packet associated with this Point Record. Up to 255 different User Defined Records which describe the waveform packet are supported. A value of zero indicates that there is no waveform data associated with this Point Record.

Byte Offset to Waveform Data: The waveform packet data are stored in the LAS file in an Extended Variable Length Record or in an auxiliary *.wdp file. The Byte Offset represents the location of the start of this Point Record’s waveform packet within the waveform data variable length record (or external file) relative to the beginning of the Waveform Data Packets header. The absolute location of the beginning of this waveform packet relative to the beginning of the file is given by **Start of Waveform Data Packet Record + Byte Offset to Waveform Data** for waveform packets stored within the LAS file and **Byte Offset to Waveform Data** for data stored in an auxiliary *.wdp file.

Waveform Packet Size in Bytes: The size, in bytes, of the waveform packet associated with this return. Note that each waveform can be of a different size (even those with the same Waveform Packet Descriptor index) due to packet compression. Also note that waveform packets can be located only via the **Byte offset to Waveform Data** value since there is no requirement that records be stored sequentially.

Return Point location: The offset in picoseconds (10^{-12}) from the arbitrary "anchor point" to the location within the waveform packet for this Point Record.

Parametric dx, dy, dz: These parameters define a parametric line equation for extrapolating points along the associated waveform. The position along the wave is given by $X = X_0 + t \cdot dx$, $Y = Y_0 + t \cdot dy$ and $Z = Z_0 + t \cdot dz$, where where (X, Y, Z) is the spatial position of a derived point, (X_0, Y_0, Z_0) is the position of the "anchor" point, and t is the time, in picoseconds, relative to the anchor point.

The anchor point is an arbitrary location at the origin of the associated waveform — i.e. $t = 0$ at the anchor point — with coordinates defined by $X_0 = X_P + L \cdot dx$, $Y_0 = Y_P + L \cdot dy$ and $Z_0 = Z_P + L \cdot dz$, where (X_P, Y_P, Z_P) is the Point Record's transformed position (as a double) and L is this Point Record's Return Point Waveform Location.

The units of X, Y and Z are the units of the coordinate systems of the LAS data. If the coordinate system is geographic, the horizontal units are decimal degrees and the vertical units are meters.

A.7. LAS Point Data Record Format 5

Point Data Record Format 5 adds Wave Packets to Point Data Record Format 3.

Table A.8 — LAS Point Data Record Format 5, compressed by LAZ Items "Point10", "GPSTime11", "RGB12" and "Wavepacket13"

Field Name	Format	Size	Required
X	long	4 bytes	*
Y	long	4 bytes	*
Z	long	4 bytes	*
Intensity	unsigned short	2 bytes	
Return Number	3 bits (bits 0 — 2)	3 bits	*
Number of Returns (given pulse)	3 bits (bits 3 — 5)	3 bits	*
Scan Direction Flag	1 bit (bit 6)	1 bit	*
Edge of Flight Line	1 bit (bit 7)	1 bit	*
Classification	unsigned char	1 byte	*
Scan Angle Rank (-90 to +90), Left side	char	1 byte	*
User Data	unsigned char	1 byte	
Point Source ID	unsigned short	2 bytes	*
GPS Time	double	8 bytes	*
Red	unsigned short	2 bytes	*
Green	unsigned short	2 bytes	*
Blue	unsigned short	2 bytes	*
Wave Packet Descriptor Index	unsigned char	1 byte	*
Byte Offset to Waveform Data	unsigned long long	8 bytes	*
Waveform Packet Size in Bytes	unsigned long	4 bytes	*
Return Point Waveform Location	float	4 bytes	*
Parametric dx	float	4 bytes	*
Parametric dy	float	4 bytes	*
Parametric dz	float	4 bytes	*

A.8. LAS Point Data Record Format 6

Point Data Record Format 6 contains the core 30 bytes that are shared by Point Data Record Formats 6 to 10. The difference to the core 20 bytes of Point Data Record Formats 0 to 5 is that there are more bits for return numbers in order to support up to 15 returns, there are more bits for point classifications to support up to 256 classes, there is a higher precision scan angle (16 bits instead of 8), and the GPS time is mandatory.

Table A.9 — LAS Point Data Record Format 6, compressed by LAZ Item “Point14”

Field Name	Format	Size	Required
X	long	4 bytes	*
Y	long	4 bytes	*
Z	long	4 bytes	*
Intensity	unsigned short	2 bytes	
Return Number	4 bits (bits 0 — 3)	4 bits	*
Number of Returns (given pulse)	4 bits (bits 4 — 7)	4 bits	*
Classification Flags	4 bits (bits 0 — 3)	4 bits	
Scanner Channel	2 bits (bits 4 — 5)	2 bits	*
Scan Direction Flag	1 bit (bit 6)	1 bit	*
Edge of Flight Line	1 bit (bit 7)	1 bit	*
Classification	unsigned char	1 byte	*
User Data	unsigned char	1 byte	
Scan Angle	short	2 bytes	*
Point Source ID	unsigned short	2 bytes	*
GPS Time	double	8 bytes	*

Note that the following five fields (Return Number, Number of Returns, Classification Flags, Scan Direction Flag and Edge of Flight Line) are bit fields, encoded into two bytes.

Return Number: The Return Number is the pulse return number for a given output pulse. A given output laser pulse can have many returns, and they must be marked in sequence of return. The first return will have a Return Number of one, the second a Return Number of two, and so on up to fifteen returns. The Return Number must be between 1 and the Number of Returns, inclusive.

Number of Returns (given pulse): The Number of Returns is the total number of returns for a given pulse. For example, a laser data point may be return two (Return Number) within a total number of up to fifteen returns.

Classification Flags: Classification flags are used to indicate special characteristics associated with the point. The bit definitions are:

Table A.10 — “Classification Flags” Bit Field Encoding for LAS Point Record types 6 to 10

Bit	Field Name	Description
0	Synthetic	If set, this point was created by a technique other than direct observation such as digitized from a photogrammetric stereo model or by traversing a waveform. Point attribute interpretation might differ from non-Synthetic points. Unused attributes must be set to the appropriate default value.
1	Key-Point	If set, this point is considered to be a model key-point and therefore generally should not be withheld in a thinning algorithm.
2	Withheld	If set, this point should not be included in processing (synonymous with Deleted).
3	Overlap	If set, this point is within the overlap region of two or more swaths or takes. Setting this bit is not mandatory (unless required by a specification other than this document) but allows Classification of overlap points to be preserved.

Note that these bits are treated as flags and can be set or cleared in any combination. For example, a point with bits 0 and 1 both set to one and the Classification field set to 2 would be a ground point that had been synthetically collected and marked as a model Key-Point.

Scanner Channel: Scanner Channel is used to indicate the channel (scanner head) of a multi-channel system. Channel 0 is used for single scanner systems. Up to four channels are supported (0-3).

Scan Direction Flag: The Scan Direction Flag denotes the direction at which the scanner mirror was traveling at the time of the output pulse. A bit value of 1 is a positive scan direction, and a bit value of 0 is a negative scan direction (where positive scan direction is a scan moving from the left side of the in-track direction to the right side and negative the opposite).

Edge of Flight Line: The Edge of Flight Line data bit has a value of 1 only when the point is at the end of a scan. It is the last point on a given scan line before it changes direction or the mirror facet changes.

Classification: Classification must adhere to the following standard:

Table A.11 — SPRS Standard LIDAR Point Classes for LAS Point Data Record Formats 6 to 10

Classification Value	Meaning
0	Created, never classified
1	Unclassified
2	Ground
3	Low Vegetation
4	Medium Vegetation
5	High Vegetation
6	Building
7	Low Point (noise)
8	Reserved
9	Water
10	Rail
11	Road Surface
12	Reserved
13	Wire — Guard (Shield)
14	Wire — Conductor (Phase)
15	Transmission Tower
16	Wire-structure Connector (e.g. Insulator)
17	Bridge Deck
18	High Noise
19	Overhead Structure (e.g., conveyors, mining equipment, traffic lights)
20	Ignored Ground (e.g., breakline proximity)
21	Snow
22	Temporal Exclusion (Features excluded due to changes over time between data sources, e.g., water levels, landslides, permafrost)
23-63	Reserved
64-255	User definable

Scan Angle: The Scan Angle is a signed short that represents the rotational position of the emitted laser pulse with respect to the vertical of the coordinate system of the data. Down in the data coordinate system is the 0.0 position. Each increment represents 0.006 degrees. Counter-Clockwise rotation, as viewed from the rear of the sensor, facing in the along-track (positive trajectory) direction, is positive. The maximum value in the positive sense is 30,000 (180 degrees which is up in the coordinate system of the data). The maximum value in the negative direction is -30,000 which is also directly up.

A.9. LAS Point Data Record Format 7

Point Data Record Format 7 is the same as Point Data Record Format 6 with the addition of three RGB color channels. These fields are used when “colorizing” a LIDAR point using ancillary data, typically from a camera.

Table A.12 — LAS Point Data Record Format 7, compressed by LAZ Items “Point14” and “RGB14”

Field Name	Format	Size	Required
X	long	4 bytes	*
Y	long	4 bytes	*
Z	long	4 bytes	*
Intensity	unsigned short	2 bytes	
Return Number	4 bits (bits 0 — 3)	4 bits	*
Number of Returns (given pulse)	4 bits (bits 4 — 7)	4 bits	*
Classification Flags	4 bits (bits 0 — 3)	4 bits	
Scanner Channel	2 bits (bits 4 — 5)	2 bits	*
Scan Direction Flag	1 bit (bit 6)	1 bit	*
Edge of Flight Line	1 bit (bit 7)	1 bit	*
Classification	unsigned char	1 byte	*
User Data	unsigned char	1 byte	
Scan Angle	short	2 bytes	*
Point Source ID	unsigned short	2 bytes	*
GPS Time	double	8 bytes	*
Red	unsigned short	2 bytes	*
Green	unsigned short	2 bytes	*
Blue	unsigned short	2 bytes	*

A.10.LAS Point Data Record Format 8

Point Data Record Format 8 is the same as Point Data Record Format 7 with the addition of a NIR (near infrared) channel.

Table A.13 — LAS Point Data Record Format 8, compressed by LAZ Items “Point14” and “RGBNIR14”

Field Name	Format	Size	Required
X	long	4 bytes	*
Y	long	4 bytes	*
Z	long	4 bytes	*
Intensity	unsigned short	2 bytes	
Return Number	4 bits (bits 0 — 3)	4 bits	*
Number of Returns (given pulse)	4 bits (bits 4 — 7)	4 bits	*
Classification Flags	4 bits (bits 0 — 3)	4 bits	
Scanner Channel	2 bits (bits 4 — 5)	2 bits	*
Scan Direction Flag	1 bit (bit 6)	1 bit	*
Edge of Flight Line	1 bit (bit 7)	1 bit	*
Classification	unsigned char	1 byte	*
User Data	unsigned char	1 byte	
Scan Angle	short	2 bytes	*
Point Source ID	unsigned short	2 bytes	*
GPS Time	double	8 bytes	*
Red	unsigned short	2 bytes	*

Field Name	Format	Size	Required
Green	unsigned short	2 bytes	*
Blue	unsigned short	2 bytes	*
NIR	unsigned short	2 bytes	*

NIR: The NIR (near infrared) channel value associated with this point.

Note that Red, Green, Blue, and NIR values should always be normalized to 16 bit values. For example, when encoding an 8 bit per channel pixel, multiply each channel value by 256 prior to storage in these fields. This normalization allows color values from different camera bit depths to be accurately merged.

A.11.LAS Point Data Record Format 9

Point Data Record Format 9 adds Wave Packets to Point Data Record Format 6.

Table A.14 — LAS Point Data Record Format 9, compressed by LAZ Items “Point14” and “Wavepacket14”

Field Name	Format	Size	Required
X	long	4 bytes	*
Y	long	4 bytes	*
Z	long	4 bytes	*
Intensity	unsigned short	2 bytes	
Return Number	4 bits (bits 0 — 3)	4 bits	*
Number of Returns (given pulse)	4 bits (bits 4 — 7)	4 bits	*
Classification Flags	4 bits (bits 0 — 3)	4 bits	
Scanner Channel	2 bits (bits 4 — 5)	2 bits	*
Scan Direction Flag	1 bit (bit 6)	1 bit	*
Edge of Flight Line	1 bit (bit 7)	1 bit	*
Classification	unsigned char	1 byte	*
User Data	unsigned char	1 byte	
Scan Angle	short	2 bytes	*
Point Source ID	unsigned short	2 bytes	*
GPS Time	double	8 bytes	*
Wave Packet Descriptor Index	unsigned char	1 byte	*
Byte Offset to Waveform Data	unsigned long long	8 bytes	*
Waveform Packet Size in Bytes	unsigned long	4 bytes	*
Return Point Waveform Location	float	4 bytes	*
Parametric dx	float	4 bytes	*
Parametric dy	float	4 bytes	*
Parametric dz	float	4 bytes	*

A.12.LAS Point Data Record Format 10

Point Data Record Format 10 adds Wave Packets to Point Data Record Format 8.

Table A.15 — LAS Point Data Record Format 10, compressed by LAZ Items “Point14”, “RGBNIR14” and “Wavepacket14”

Field Name	Format	Size	Required
X	long	4 bytes	*
Y	long	4 bytes	*

Field Name	Format	Size	Required
Z	long	4 bytes	*
Intensity	unsigned short	2 bytes	
Return Number	4 bits (bits 0 — 3)	4 bits	*
Number of Returns (given pulse)	4 bits (bits 4 — 7)	4 bits	*
Classification Flags	4 bits (bits 0 — 3)	4 bits	
Scanner Channel	2 bits (bits 4 — 5)	2 bits	*
Scan Direction Flag	1 bit (bit 6)	1 bit	*
Edge of Flight Line	1 bit (bit 7)	1 bit	*
Classification	unsigned char	1 byte	*
User Data	unsigned char	1 byte	
Scan Angle	short	2 bytes	*
Point Source ID	unsigned short	2 bytes	*
GPS Time	double	8 bytes	*
Red	unsigned short	2 bytes	*
Green	unsigned short	2 bytes	*
Blue	unsigned short	2 bytes	*
NIR	unsigned short	2 bytes	*
Wave Packet Descriptor Index	unsigned char	1 byte	*
Byte Offset to Waveform Data	unsigned long long	8 bytes	*
Waveform Packet Size in Bytes	unsigned long	4 bytes	*
Return Point Waveform Location	float	4 bytes	*
Parametric dx	float	4 bytes	*
Parametric dy	float	4 bytes	*
Parametric dz	float	4 bytes	*

Annex B
(informative)
Revision History

Table — Revision History

Date	Release	Author	Primary clauses modified	Description
2024-06-16	R0	rapidlasso GmbH	all	initial version
2024-10-27	R1	rapidlasso GmbH	10.5	Typos (whole document), clarification on usage of Integer encoder

Bibliography

[1] Said, Amir: **Introduction to Arithmetic Coding — Theory and Practice** (2004)

[2] The American Society for Photogrammetry & Remote Sensing, LAS Specification 1.4 — R15, https://www.asprs.org/wp-content/uploads/2019/07/LAS_1_4_r15.pdf