institut für astrophysik
UNIVERSITÄTSSTERNWARTE WIEN

universität wien

PLATO

Data Compression User Manual

Data Compression User Manual

Reference: PLATO-UVIE-PL-UM-0001

Version: Draft 6, January. 25, 2022

Prepared by: Dominik Loidolt[1]

Checked by: Roland Ottensamer[1]

Approved by: Franz Kerschbaum[1]

[1] Department of Astrophysics, University of Vienna

Contents

Revision History

Revision	Date	Author(s)	Description
Draft 1	12.06.2019	DL	draft document created
Draft 2	12.09.2019	DL	updated chapter 1-6
Draft 3	03.02.2020	DL	updated code listings, incorporate feedback
Draft 4	24.03.2020	DL	updated to meet the FPGA Requirement Specification V 1.1
Draft 5	05.06.2021	DL	corrected minimum allowed spill value, updated compressed data header, corrected Fig 7.3, 7.4, corrected listing 5.3
Draft 6	25.01.2020	DL	change the size of the ASW Version ID from 16 to 32 bits in the generic header, add spare bits to the adaptive imagette header and the non-imagette header, so that the compressed data start address is 4 byte-aligned.

The documents in Table 2 form an integral part of the present document. The documents in Table 3 are referenced in the present document and are for information only.

Table 2: Applicable Documents

ID	Title, Reference Number, Revision Number
AD-1	Space engineering - Software, ECSS-E-ST-40C, 6th March 2009
AD-2	Space product assurance – Software product assurance, ECSS-Q-ST-80C, 15th February 2017

Table 3: Reference Documents

ID	Title, Reference Number, Revision Number
RD-1	PLATO Data Compression Concept, PLATO-UVIE-PL-TN-0001
RD-2	PLATO Router and Data Compression Unit (RDCU) Prototype User Manual, PLATO-IWF-PL-UM-0040
RD-3	PLATO RDCU Data Throughput, PLATO-IWF-PL-TN-059, 19th August 2019
RD-4	FPGA Requirement Specification, PLATO-IWF-PL-RS-0005, 10. March 2020

ID	Title, Reference Number, Revision Number
RD-5	Level0 data generation from the payload science data, PLATO-DLR-MIS-TN-0002, 14. October 2020

1. Terms, Definitions and Abbreviated Items

1.1 Acronyms

API	Application Programming Interface
CPU	Central Processing Unit
FPGA	Field-Programmable Gate Array 6, 8
HW	Hardware 6, 10, 13–15, 19, 20, 22, 23, 26, 28, 36
ICU	Instrument Control Unit 6, 8, 14, 26–28, 31, 36
ISR	Interrupt Service Routine
MMU	Memory Management Unit
PUS	Packet Utilisation Standard
RDCU	Router and Data Compression Unit 3, 6, 8–11, 14–16, 18–21, 24, 25, 28, 29, 31, 35, 36, 38
RISC	Reduced Instruction Set Computing
RMAP	Remote Memory Access Protocol 8, 10, 18, 19, 28
SRAM	Static Random Access Memory 8, 14–16, 18, 24, 25, 28, 29, 31, 36
SW	Software 6, 10, 15, 19, 22, 23, 26, 28
UVIE	University of Vienna 6, 8, 28

2. Introduction

The University of Vienna (UVIE) team provides a set of compression algorithms to support the Instrument Control Unit (ICU) in compressing the different PLATO data products. Unlike the non-imagette compression algorithms, which are only available in the form of software libraries, the imagette compression algorithms have also been implemented in hardware on the Field-Programmable Gate Array (FPGA) on the RDCU board. Furthermore, an interface has been created which abstracts the Software (SW) and Hardware (HW) imagette compression so that both compressors can be controlled with the same parameters.

2.1 Purpose of the Document

This document is about the handling of the provided imagette compression algorithms in software and hardware.

2.2 The Data Compression Algorithm

The compression algorithm consists of several stages connected in series as shown in Figure 2.1. A brief introduction to the compression algorithm follows, for more details see **[RD-1]**.

The first stage is an optional lossy compression stage. This stage can achieve a significantly higher compression ratio at the expense of data loss. This is accomplished by rounding down the least significant digits of the input values so that the output of this stage is smaller than the input. This stage is controlled by the round parameter, which determines how many bits should be rounded.

The second stage in the compression chain is the precompression or preprocessing stage. This stage uses correlations in the data to reduce the dynamics of the data set. The precompression stage has several modes to accomplish this task, which are briefly introduced here. The raw mode writes the input data into the area of the memory which is intended for the compressed data so that no data compression takes place. Therefore, the input is the same as the output. This mode is intended as a label for uncompressed data or for debugging the compressor. Another mode is the 1d-differencing mode. This mode calculates the difference to the left neighboring pixel to reduce the dynamics of the input data. The model mode is used to perform a compression of recurring data of the same object. In addition to the input data of the current object, a model of the input data is also required. The model roughly corresponds to an average of the past data of the object. In this mode, the difference between the input data and a model of this data is formed. The model is updated after a compression and is used to

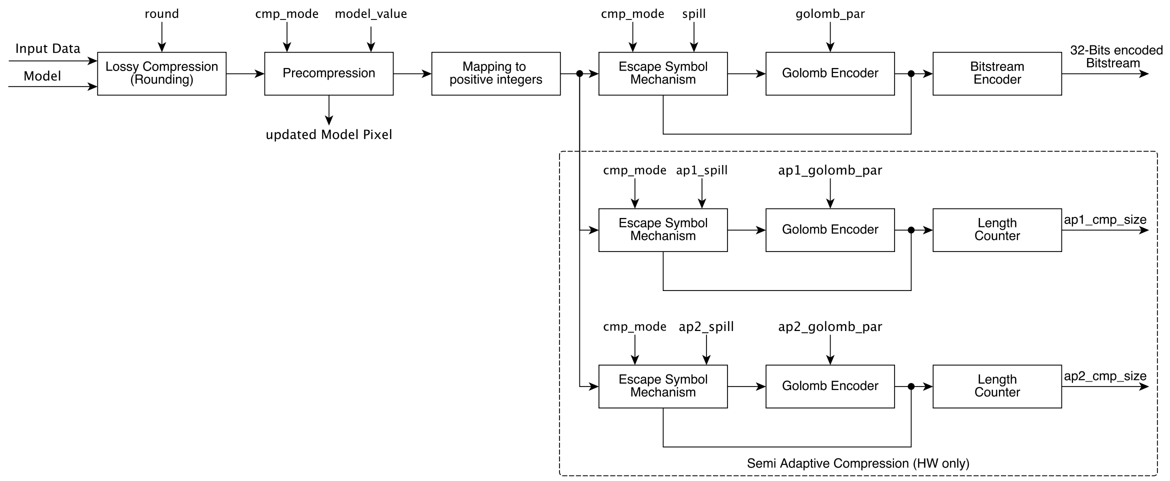

Figure 2.1: Visualization of the compression algorithm.

compress the next data of the same target object at a later point in time.

The next stage maps the output data of the precompression which are signed integers into positive integers. This is necessary because the Golomb encoder can only work with positive integers.

The escape symbol mechanism becomes active whenever statistical outliers occur. Two mechanisms are implemented to handle outliers, the zero escape symbol mechanism and the multi escape symbol mechanism. Depending on the distribution of outliers, one mechanism has slight advantages over the other.

The Golomb encoder is the heart of the compression process. The Golomb code is an algorithm that assigns an input value to a code word. The Golomb encoder assigns short code words to small values and long code words to large values.

The BitstreamEncoder generates a bitstream of code words. The Bitstream encoder has the task of stringing the generated codewords of different lengths together and dividing them into 32-bit long pieces to make it possible to write them to the memory.

It can be assumed that the structure of the data to be compressed will change due to various effects such as ageing processes. Therefore, an adaptive compression technique is needed to change the compressor settings whenever the data changes. This is needed to ensure good data compression over time. This feature is only supported by the hardware data compressor.

2.3 The ICU Software Compressor

The UVIE team provides the imagette compression algorithms which are used in the hardware compressor and also in a separate software package. The software package also included the algorithms used to compress the non-imagette data products.

The task of the software compressor should be to process small data products that do not occur frequently. Chapter 6.1 discusses the provided function for software compression in detail.

2.4 The RDCU Hardware Compressor

The data compressor is implemented in the FPGA of the RDCU. It is connected via a SpW link to the SpW router on the RDCU board, see Figure 2.2. The router is connected to the ICU via two SpW links. Therefore, the communication from the ICU to the hardware compressor always runs via the SpW router. It must be ensured that the route between the ICU and the compressor is correctly configured before communication with the hardware compressor can be started.

On the one hand, the interface of the hardware compressor consists of registers that control the compressor and provide the metadata of a compression. On the other hand, it consists of the Static Random Access Memory (SRAM) that contains the data to be compressed and, if necessary, the corresponding model, as well as the result of the data compression, the compressed bitstream. The registers, as well as the SRAM, are written and read via the Remote Memory Access Protocol (RMAP) protocol.

To compress data with the hardware compressor, the data to be compressed are written into the SRAM. Before or after the data transfer the data compressor registers are set with the parameters necessary for the data compression. Once these two steps have been completed, the compression can be started by setting the data *compressor start bit* in the *compressor control register*. While the compression is running the SRAM is not accessible via RMAP only the *compressor status register* is readable. The completion of the data compression is signaled to the ICU by an interrupt signal or by setting the data *compressor ready bit* in the *compressor status register*. Before the data can be read, it must be checked if an error occurred during compression. This is ensured by checking that the compressor *data valid bit* is set in the *compressor status register* and that no error bit is set in the *compression error register*. If this is the case, everything worked fine during compression and the remaining metadata and compressed data can be read out.

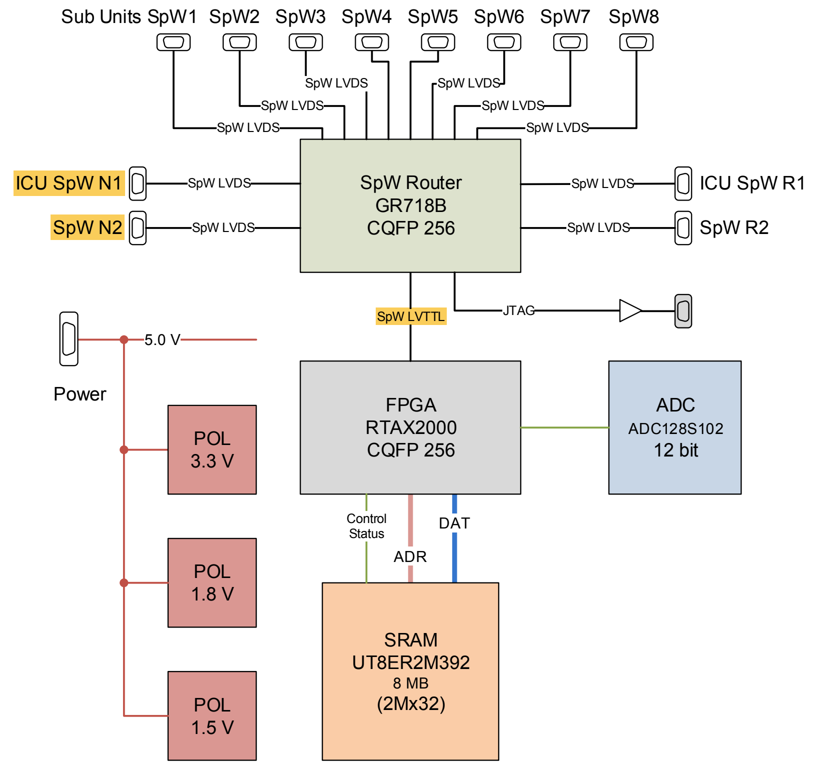

Figure 2.2: RDCU Electrical Concept.

3. Controlling the Compression

The compression is controlled by the compression parameters. For software compression, a structure is provided that contains all compression parameters, see Section 3.1. This structure is passed to the software compression function. For hardware compression, this structure can also be used to generate the necessary RMAP packets that set the corresponding hardware compressor registers. Alternatively, you can build the required RMAP package "by yourself", the required information can be found in the RDCU user manual **[RD-2]**.

3.1 The Compression Parameters Structure

The compression parameters control the SW as well as the HW compressor. The structure cmp_cfg as seen in listing 3.1, contains all parameters for the HW as well as for the SW compression. In the following, the individual parameters will be briefly discussed and their function and valid value range will be explained in more detail.

```
/**
 * @brief The cmp_cfg structure can contain the complete configuration of the HW as
 *       well as the SW compressor.
 * @note when using the 1d-differentiating mode or the raw mode (cmp_error =
 *   0,2,4), the model parameters (model_value, model_buf, rdcu_model_adr,
 *   rdcu_new_model_adr) are ignored
 * @note the icu_output_buf will not be used for HW compression
 * @note the rdcu_***_adr parameters are ignored for SW compression
 * @note semi adaptive compression not supported for SW compression;
 *   configuration parameters ap1\_golomb\_par, ap2\_golomb\_par, ap1\_spill,
 *   ap2\_spill will be ignored;
 */

struct cmp_cfg {
  uint32_t cmp_mode;            /* 0: raw mode
                                   * 1: model mode with zero escape symbol mechanism
                                 * 2: 1d differencing mode without input
                                 *    model with zero escape symbol mechanism
                                 * 3: model mode with multi escape symbol mechanism
                                 * 4: 1d differencing mode without input
                                 *    model multi escape symbol mechanism */
  uint32_t golomb_par;          /* Golomb parameter for dictionary selection */
  uint32_t spill;               /* Spillover threshold for encoding outliers */
  uint32_t model_value;         /* Model weighting parameter */
  uint32_t round;               /* Number of noise bits to be rounded */
  uint32_t ap1_golomb_par;      /* Adaptive 1 spillover threshold; HW only */
  uint32_t ap1_spill;           /* Adaptive 1 Golomb parameter; HW only */
  uint32_t ap2_golomb_par;      /* Adaptive 2 spillover threshold; HW only */
  uint32_t ap2_spill;           /* Adaptive 2 Golomb parameter; HW only */
  uint16_t *input_buf;          /* Pointer to the data to be compressed  */
  uint32_t rdcu_data_adr;       /* RDCU data to be compressed start address,
                                 * the first data address in the RDCU SRAM; HW only */
  uint16_t *model_buf;          /* Pointer to the model buffer */
  uint32_t rdcu_model_adr;      /* RDCU model start address, the first model
```

```
                                * address in the RDCU SRAM */
  uint16_t *icu_new_model_buf;/* Pointer to the updated model buffer */
  uint32_t rdcu_new_model_adr;/* RDCU updated model start address, the
                                * address in the RDCU SRAM where the
                                * updated model is stored */
  uint32_t samples;            /* Number of samples (16 bit value) to
                                * compress, length of the data and
                                * (updated) model buffer */
  uint32_t *icu_output_buf;    /* Pointer to the compressed data buffer
                                * (not used for RDCU compression) */
  uint32_t rdcu_buffer_adr;    /* RDCU compressed data start address, the
                                * first output data address in the RDCU SRAM */
  uint32_t buffer_length;      /* Length of the compressed data buffer in
                                * number of samples (16 bit values)*/
};
```

Listing 3.1: C-Implementation of the compressor configuration structure.

3.2 Compression Parameters

In the following section, the compression parameters and their effect on the compression are briefly introduced. To get more detailed information about the parameters you can read **[RD-1]**.

3.2.1 Compression Mode (cmp_mode)

The compression mode parameter controls the precompression/preprocessing as well as the escape symbol mechanism stage of the compressor. The current implementation of the compressor supports five different compression modes. The cmp_mode parameter controls which mode is used. The cmp_mode parameter can be 0 for raw mode, 1 or 3 for model mode and 2 or 4 for the 1d-differencing mode.

cmp_mode=0: raw mode

The raw mode is intended for testing and debugging operations. In this mode, the input data are read in and written back unchanged to the memory area provided for the compressed data. No compression takes place in this mode. It has to be ensured that the data buffer length for the compressed data is at least as large as the size of the input data.

Note: Including RDCU FPGA version 0.7 there is an error in the raw mode which triggers a small_buffer_err if the samples parameter is equal to the buffer_length parameter. The workaround is to choose a larger buffer_length parameter than the samples parameter.

cmp_mode=1,3: model mode

The model mode is the default mode of the compressor. In addition to the data to be compressed, a model of the input data is required for this mode. In the model mode, the compressor forms the difference between input data and their models. It also updates the models according to the method described in Section 7.2.2. In this compression mode, not only the compressed data must be read out, but also the updated model. The updated model is required again if the data for the same target object is to be compressed at a later point in time. When using the hardware compressor, the upload of the model is not necessary if the next data to be processed are from the same object as the last compression.

The difference between cmp_mode 1 and 3 is the different handling of outliers. cmp_mode = 1 uses the zero escape symbol mechanism, while cmp_mode = 3 uses the multi escape symbol mechanism. Depending on the distribution of outliers, one mechanism has slight advantages over the other. For more information about the exact function of the different escape symbol mechanisms, see **[RD-1]**.

cmp_mode=2,4: 1d-differencing without input model mode

As the name suggests, the 1d-differencing without input model mode does not require a model. With this method, the difference between neighbouring pixels is formed. This method usually has a poorer compression ratio than the model mode. It is used to compress the first image of a series of images because no model exists for that data. This mode can also be used to compress data that does not occur repeatedly.

The difference between cmp_mode 2 and 4 is the different handling of outliers. cmp_mode = 2 uses the zero escape symbol mechanism, while cmp_mode = 4 uses the multi escape symbol mechanism. Depending on the distribution of outliers, one mechanism has slight advantages over the other. For more information about the exact function of the different escape symbol mechanisms, see **[RD-1]**.

3.2.2 Golomb Parameter (golomb_par)

Based on the Golomb parameter (golomb_par) and the input value of the Golomb encoder stage the code words are formed. As shown in document **[RD-1]**, a larger Golomb parameter causes the code word length to grow slower, but code words for smaller values are longer. The input data of the Golomb encoder follow approximately a geometric distribution. The Golomb parameter should be adapted to this distribution so that the length of all code words is minimal. In the current implementation, a Golomb parameter in the range between 1 and 63 is supported. 0 is not a valid value for the Golomb parameter.

3.2.3 Spillover Threshold Parameter (spill)

The escape symbol mechanism is controlled by the spillover threshold parameter (spill). The spill parameter controls if a value is considered to be an outlier. If an outlier is recognized, the 16-bit raw value is encoded with a prefixed escape symbol. The maximum value of the spill parameter depends on the Golomb parameter selected. Because the HW Golomb encoder can only generate code words with a maximum length of 16 bits, the spill must be set to become active before a 17-bit long or longer code word would be generated. As you can see in Table 3.2 the maximum spill value is smaller for lower golomb_par values because the codeword length increases rapidly with low golomb_par values. For more information see **[RD-1]**.

3.2.4 Model Weighting Parameter (model_value)

The model weighting parameter or model_value controls the model update process in the pre-compression/preprocessing stage. The weighting parameter only affects the compression process if the compressor is in the model mode. As the name indicates the weighting parameters weight the ratio between the model and the current imagette in the model update equation. The weighting parameter is a natural number in the range between [0,16]. From the model update equation 7.5 in Section 7.2.2, you can see that the larger the weighting parameter is, the slower the updated model changes compared to the current model. The largest value is 16, which means that the updated model is the same as the current model. The lowest value is zero, which means that the updated model always corresponds to the current data to be compressed.

3.2.5 Rounding Parameter (round)

The rounding parameter controls the lossy compression stage. The value specifies how many bits of the input value in the lossy compression stage are shifted to the right. The larger the rounding parameter, the higher the compression ratio, at the expense of data loss. A rounding parameter equal to zero means lossless data compression. Since the imagette collection header is also treated like normal data during compression, it must be ensured that this header is not corrupted by rounding the last bits.

3.2.6 Adaptive Golomb Parameter 1/2 (ap1_golomb_par, ap2_golomb_par), Adaptive Spillover Threshold 1/2 (ap1_spill, ap2_spill)

Semi-adaptive compression is controlled by the ap1_golomb_par, ap2_golomb_par, ap1_spill and ap2_spill parameters. This feature is only supported by the HW compressor. The semi-adaptive compression is a mechanism that allows, in addition to the compression parameters (golomb_par, spill pair) actually used for the compression, to use two additional golomb_par, spill pairs. At the end of the compression process, it is possible to read out how long the respective bitstream would have been if the additional two pairs had been used. This information can then be used to choose a better golomb_par, spill pair for the next compression. Note that ap1_spill or ap2_spill cannot be selected independently of ap1_golomb_par, ap2_golomb_par. As explained in more detail in Section 3.4.1, an ap_spill parameter can be selected up to a specific value depending on the set ap_golomb_par parameter.

3.2.7 ICU Buffers (input_buf, model_buf, icu_new_model_buf, icu_output_buf)

There are four different buffers for the ICU. The input_buf is intended for the data to be compressed.

The model_buf is intended for the model of the data. This buffer is not needed for 1d-differencing and raw mode.

The icu_new_model_buf is only intended for compression with the ICU compressor. If the icu_new_model_buf is set to NULL or equal to model_buf, the compressor simply overwrites the model_buf with the calculated updated model. If this is not desired, the icu_new_model_buf buffer can be used to specify where the updated model should be written. The size of the input, model_buf and icu_new_model_buf is described by the parameter samples in 16-bit values.

The icu_output_buf buffer is intended for the compressed data. The size of the icu_output_buf is described by the buffer_length parameter in units of 16-bit values. The icu_output_buffer is not needed to set up the HW compression.

3.2.8 RDCU Addresses (rdcu_data_adr, rdcu_model_adr, rdcu_new_model_adr, rdcu_buffer_adr)

The different RDCU address parameters are only used for the HW compression. These parameters determine the memory address of the SRAM where the uncompressed data, the model data, the updated model and the buffer for the compressed bitstream begin. The rdcu addresses have to be 4-byte aligned. The user of the HW compressor must take care that the different memory areas do not overlap.

If rdcu_new_model_adr is set equal to rdcu_model_adr, the compressor simply overwrites the old model with the new updated one. This setting also has a small speed advantage, because if parts of the updated model did not change, some expensive write access can be skipped. If rdcu_new_model_adr and rdcu_model_adr are different, the rdcu_new_model_adr can be used to specify where in the SRAM the updated model should be written. The old model will not be overwritten.

3.2.9 Number of Samples (samples)

The number of samples parameter determines the length of the data to be compressed in 16-bit words. The uncompressed data also corresponds to the length of the (updated) model data because there is a model for each data record.

3.2.10 Compressed Data Buffer Length (buffer_length)

The compressed data buffer length specifies the length of the reserved area in the unit as the samples parameter. For SW compression this parameter specifies the length of the output_buf. When using HW compression, this parameter specifies the length of the reserved area for compressed data after the rdcu_buffer_adr in the RDCU SRAM. The HW compressor does not check the overlapping of different memory spaces.

3.3 Differences between SW and HW Compression Setup

Since the SW and the HW compression are conceptually slightly different but are set up with the same structure, there are a few things to keep in mind. The parameters that control the compression itself (cmp_mode, golomb_par, spill, model_value, round) are the same for SW and HW.

The SW compressor takes the data from the input_buf and the model_buf and writes the result, which is the compressed bitstream, into the output_buf. The semi-adaptive compression parameters (ap1_***, ap2_***) and the rdcu_***_adr parameters are not used. The HW compressor transfers the data from the input_buf and the model_buf to the designated area in the RDCU SRAM, defined by the rdcu_data_adr and rdcu_model_adr addresses. Then the registers used for the compression are set. Finally, the compression is started. The icu_output_buf and the icu_new_model_buf of the cmp_cfg structure are not used for the HW compression.

Another difference between the SW and HW compressors is the starting of the compression process. With the SW compressor, the start is simply done by calling the compression function. In contrast, the HW compressor starts by setting the data compressor start bit in the compressor control register. The compressor control register can also be used to enable or disable the RDCU interrupt and to interrupt the compression process by setting the data compressor interrupt bit.

3.4 Compression Parameter Errors

A large number of compression parameters only accepts values within a specified range. If a compression parameter has an invalid value outside its range, this will cause errors in the compression process. Therefore the compressor detects possible errors and informs the user about them. It checks the input parameters for their correctness and blocks the start of the compressor in the event of an error to prevent a possible unpredictable behaviour.

The hardware compressor indicates an error in the compression error register. The software compressor displays an error in the compression information structure. Table 3.1 lists the valid value ranges of the different parameters. The maximum spill parameter is slightly more complex to determine which is described in detail in the next section.

Table 3.1: Valid value ranges for the different parameters of the compressor.

Parameter Name	Abbreviation	Valid Value Range
Compression Mode	cmp_mode	[0,4]
Weighting Parameter	model_value	[0,16]
Rounding Parameter	round	[0,2]
Golomb Parameter	golomb_par	[1,63]
Spillover Threshold Parameter	spill	[2, see Section 3.4.1]
Adaptive Golomb Parameter 1/2	ap1/2_golomb_par	[1,63]
Adaptive Spillover Threshold 1/2	ap1/2_spill	[2, see Section 3.4.1]
RDCU SRAM Addresses	rdcu_***_adr	[0x000000, 0x7FFFFF]

3.4.1 Spill Golomb Error

The choice of the spill parameter is closely related to the Golomb parameter. This connection exists because the Golomb encoder can only generate code words with a maximum length of 16 bits. The spill parameter must be set in a way that too large input values do not reach the Golomb encoder. A too high input value would result in a codeword longer than 16 bits being generated. The limitation of the spill parameter ensures that the escape symbol mechanism becomes active before the encoder produces a code word which is too long. Table 3.2 shows the maximum allowed spill parameter depending on the selected golomb_par. Since the code word length increases rapidly with smaller Golomb parameters, it is not surprising that the allowed spill parameter is smaller with small golomb_par than with large ones.

The validity ranges for the spill parameter from Table 3.2 are the same for ap1_spill and ap2_spill parameters.

3.5 Default Configuration

In order to set up the compressor quickly and with the appropriate configuration, we provide a default configuration for various applications. Currently, two default configurations are available for the compression of imagette data in 1d-differencing mode and in model mode, see Listing 3.2. The standard configurations are not yet finished and can still be adapted. It is planned

Table 3.2: Valid spillover threshold parameters (spill) range in relation to the used Golomb parameter (golomb_par).

golomb_par	spill≤	golomb_par	spill≤	golomb_par	spill≤	golomb_par	spill≤
1	8	17	194	33	353	49	497
2	22	18	204	34	362	50	506
3	35	19	214	35	371	51	515
4	48	20	224	36	380	52	524
5	60	21	234	37	389	53	533
6	72	22	244	38	398	54	542
7	84	23	254	39	407	55	551
8	96	24	264	40	416	56	560
9	107	25	274	41	425	57	569
10	118	26	284	42	434	58	578
11	129	27	294	43	443	59	587
12	140	28	304	44	452	60	596
13	151	29	314	45	461	61	605
14	162	30	324	46	470	62	614
15	173	31	334	47	479	63	623
16	184	32	344	48	488		

to provide default configuration for other data products as well.

```
/**
 * @brief Default configuration of the Compressor in model mode.
 */

const struct cmp_cfg DEFAULT_CFG_MODEL = {
  MODE_MODEL_MULTI, /* cmp_mode */
  4, /* golomb_par */
  48, /* spill */
  8, /* model_value */
  0, /* round */
  3, /* ap1_golomb_par */
  35, /* ap1_spill */
  5, /* ap2_golomb_par */
  60, /* ap2_spill */
  NULL, /* *input_buf */
  0x000000, /* rdcu_data_adr */
  NULL, /* *model_buf */
  0x2AAAAC, /* rdcu_model_adr */
  NULL, /* *up_model_buf */
  0x2AAAAC, /* rdcu_up_model_adr */
  0, /* samples */
  NULL, /* *icu_output_buf */
  0x555554, /* rdcu_buffer_adr */
  0x155556 /* buffer_length */
};


/**
 * @brief Default configuration of the Compressor in 1d-differencing mode.
 */
```

```

const struct cmp_cfg DEFAULT_CFG_DIFF = {
  MODE_DIFF_ZERO, /* cmp_mode */
  7, /* golomb_par */
  60, /* spill */
  8, /* model_value */
  0, /* round */
  6, /* ap1_golomb_par */
  48, /* ap1_spill */
  8, /* ap2_golomb_par */
  72, /* ap2_spill */
  NULL, /* *input_buf */
  0x000000, /* rdcu_data_adr */
  NULL, /* *model_buf */
  0x2AAAAC, /* rdcu_model_adr */
  NULL, /* *up_model_buf */
  0x2AAAAC, /* rdcu_up_model_adr */
  0, /* samples */
  NULL, /* *icu_output_buf */
  0x555554, /* rdcu_buffer_adr */
  0x155556 /* buffer_length */
};
```

Listing 3.2: C-implementation of various default compressor configurations.

3.6 Setup the Hardware Compressor

The provided function rdcu_compress_data, see Listing 3.3, checks the given configuration for validity and generates the RMAP packets to set the compressor registers with the parameters defined in the cmp_cfg configuration structure. The data to be compressed is then also packed into RMAP packets to be sent to the RDCU SRAM.

When the model mode is used, the model is also uploaded to the RDCU. If another compressor mode is used the model will be ignored. Finally, the RMAP package is created to start the compression. The rdcu_compress_data function only sets up and starts compression. Reading the compressed data is done by another function so that the icu_output_buf buffer_adr parameter in the configuration is not used for the hardware compression.

```
/**
 * @brief compressing data with the help of the RDCU hardware compressor
 *
 * @param cfg  configuration contains all parameters required for compression
 *
 * @note when using the 1d-differencing mode or the raw mode (cmp_mode = 0,2,4),
 *      the model parameters (model_value, model_buf, rdcu_model_adr) are ignored
 * @note the icu_output_buf will not be used for the RDCU compression
 * @note the validity of the cfg structure is checked before the compression is
 *       started
 *
 * @returns 0 on success, error otherwise
 */

int rdcu_compress_data(const struct cmp_cfg *cfg)
```

Listing 3.3: Declaration of the RDCU compression function.

4. The Status of a Compression

During a HW compression, only the compressor status register is readable via RMAP.

4.1 Compression Status Structure

The compression status structure reflects the contents of the RDCU compressor status register. Unlike the other registers, this register can also be queried during compression. The structure can also be used for SW compression if needed. However, the SW compressor does not use the cmp_interrupted and the rdcu_interrupt_en flag in the structure.

```c
/**
 * @brief The cmp_status structure can contain the information of the
 *        compressor status register from the RDCU, see RDCU-FRS-FN-0632,
 *        but can also be used for the SW compression.
 */

struct cmp_status {
  uint8_t cmp_ready; /* Data Compressor Ready; 0: Compressor is busy 1: Compressor is
    ready */
  uint8_t cmp_active; /* Data Compressor Active; 0: Compressor is on hold; 1: Compressor
    is active */
  uint8_t data_valid; /* Compressor Data Valid; 0: Data is invalid; 1: Data is valid */
  uint8_t cmp_interrupted; /* Data Compressor Interrupted; HW only; 0: No compressor
    interruption; 1: Compressor was interrupted */
  uint8_t rdcu_interrupt_en; /* RDCU Interrupt Enable; HW only; 0: Interrupt is disabled;
     1: Interrupt is enabled */
};
```

Listing 4.1: C-Implementation of the compressor status structure.

4.1.1 Data Compressor Ready (cmp_ready)

The data compressor ready value indicates whether compression is complete and the compressor is ready to start a new compression. When a data compression is running, the value of the bit is 0, when compression is finished cmp_ready is set to 1.

4.1.2 Data Compressor Active (cmp_active)

In the current implementation, the active compressor bit is the inverted compressor ready bit. This means that while a compression is running it is 1. If the compression is completed, cmp_active is 0.

4.1.3 Data Compressor Data Valid (data_valid)

The data valid value indicates whether the compressed data (and, in model mode, the updated model) is valid or not. If an error occurs during compression or if compression is interrupted, the value of this bit remains 0 after compression. If compression worked and everything went well, the bit changes to 1 after compression is complete. The value remains 1 until a new compression is started.

4.1.4 Data Compressor Interrupted (cmp_interrupted)

The data compressor interrupted bit is set when the hardware compressor is interrupted by setting the data compressor interrupt bit in the compression control register. This bit is reset when a new compression is started. It is a status value that is only used by the hardware compressor.

4.1.5 RDCU Interrupt Enable (rdcu_interrupt_en)

The RDCU interrupt enable bit is mirroring the RDCU interrupt enable value in the compression control register. This bit is therefore only used by the HW compression.

4.2 Read the Status of the Hardware Compressor

You can use the rdcu_read_cmp_status function to request the content of the compressor status register of the RDCU HW compressor. The cmp_status structure represents the contents of the compressor status register. This register is the only register that can be read out during a compression process. The function can be used to poll the status of a compression to find out when the compression is finished.

The time a compression takes depends on the size of the data to be compressed, the compression mode and the compression rate (CR) reached can be estimated as follows:
Model Mode:

$$\mathcal{O}(t_{\text{mdl}}) = \text{samples} \cdot (20 + 6/\text{CR}) \cdot 20\,\text{ns} \tag{4.1}$$

1-D Differencing mode:

$$\mathcal{O}(t_{\text{dif}}) = \text{samples} \cdot (8 + 6/\text{CR}) \cdot 20\,\text{ns} \tag{4.2}$$

```
/**
 * @brief read out the status register of the RDCU compressor
 *
 * @param status  compressor status contains the stats of the HW compressor
 *
 * @note access to the status registers is also possible during compression
 *
 * @returns 0 on success, error otherwise
 */

int rdcu_read_cmp_status(struct cmp_status *status)
```
Listing 4.2: Declaration of the RDCU read status function.

5. The Metadata of a Compression

Once the compression is complete, we need more information than the compressed bitstream to process the data further. This metadata can be stored in the provided compressor information structure (cmp_info).

5.1 Compression Information Structure

The cmp_info structure shown in Listing 5.1 contains all readable information registers of the HW compressor. These registers are only readable when the compressor is not active. They are also used by the SW compressor to return the metadata of a compression. Before the compressed data from the compressor can continue to be used, it must be checked that there is no compression error. Only if cmp_err = 0 the data of the compressor are valid. The meanings of the error codes are explained in Section 3.4.

```
/**
 * @brief The cmp_info structure can contain the information and metadata of an
 *      executed compression of the HW as well as the SW compressor.
 *
 * @note if SW compression is used the parameters rdcu_new_model_adr_used,
    rdcu_cmp_adr_used,
 *      ap1_cmp_size, ap2_cmp_size are not used and are therefore set to zero
 */

struct cmp_info {
  uint8_t  cmp_mode_used;       /* Compression mode used */
  uint8_t  model_value_used;    /* Model weighting parameter used */
  uint8_t  round_used;          /* Number of noise bits to be rounded used */
  uint16_t spill_used;          /* Spillover threshold used */
  uint8_t  golomb_par_used;     /* Golomb parameter used */
  uint32_t samples_used;        /* Number of samples (16 bit value) to be stored */
  uint32_t cmp_size;            /* Compressed data size; measured in bits */
  uint32_t ap1_cmp_size;        /* Adaptive compressed data size 1; measured in bits */
  uint32_t ap2_cmp_size;        /* Adaptive compressed data size 2; measured in bits */
  uint32_t rdcu_new_model_adr_used; /* Updated model start  address used */
  uint32_t rdcu_cmp_adr_used;   /* Compressed data start address */
  uint16_t cmp_err;             /* Compressor errors
                 * [bit 0] small_buffer_err; The length for the compressed data buffer is
    too small
                 * [bit 1] cmp_mode_err; The cmp_mode parameter is not set correctly
                 * [bit 2] model_value_err; The model_value parameter is not set correctly
                 * [bit 3] cmp_par_err; The spill, golomb_par combination is not set
    correctly
                 * [bit 4] ap1_cmp_par_err; The ap1_spill, ap1_golomb_par combination is
    not set correctly (only HW compression)
                 * [bit 5] ap2_cmp_par_err; The ap2_spill, ap2_golomb_par combination is
    not set correctly (only HW compression)
                 * [bit 6] mb_err; Multi bit error detected by the memory controller (only
   HW compression)
```

```
	     * [bit 7] slave_busy_err; The bus master has received the "slave busy" status (only HW compression)
	     * [bit 8] slave_blocked_err; The bus master has received the "slave "blocked status (only HW compression)
	     * [bit 9] invalid address_err; The bus master has received the "invalid "address status (only HW compression) */
};
```

Listing 5.1: C-Implementation of the compressor information structure.

5.1.1 Used Compression Parameters (*_used)

The compression parameters used are a copy of the respective parameters from the configuration. They are required for decompression, which must be performed with the same parameters as the compression.

5.1.2 Compressed Data Size (cmp_size)

The cmp_size parameter describes the length of the compressed bitstream located at the cmp_adr address. The compression rate (CR) can be easily calculated by:

$$\mathrm{CR} = \frac{\mathrm{model\_length\_used} \cdot 16\,\mathrm{Bit}}{\mathrm{cmp\_size}} \tag{5.1}$$

5.1.3 Adaptive Compressed Data Size 1/2 (ap1_cmp_size, ap2_cmp_size)

ap1_cmp_size shows the length of the bitstream if ap1_golomb_par and ap1_spill were used instead of the used compression parameters. This information can be used to select better compression parameters for the next compression operation. This also applies to the parameter ap2_cmp_size. This feature of semi-adaptive compression is only provided by the HW compressor. When using the SW compressor ap1_cmp_size and ap2_cmp_size are always set to 0.

5.1.4 Compressor errors (cmp_err)

The compression error register consists of eight error bits. Each bit indicates a different error. If one or more bits are set, an error occurred during compression. If this is the case, the compressed bitstream (and, in model mode, the updated model) is invalid and can no longer be used.

Small Buffer Error

If the compressed bitstream is larger than the space defined by the buffer_length parameter in the configuration, there is not enough space to write the entire bitstream to memory. The compressor, therefore, stops compression and sets the small_buffer_err bit. Note that when using the compression method with wrong parameters or unfavorably distributed data, the "compressed" bitstream may be larger than the input data.

Compression Parameter Errors

The error bits 1 to 5 deal with incorrectly set compression parameters and have already been discussed in detail in Section 3.4.

Multi-Bit Error (mb_err)

Due to the design of the RDCU SRAM, it is checked at each read access whether a multi-bit error has occurred. If this is the case, this is indicated by setting the mb_err bit. The compression will be stopped. This error can only occur when using the hardware compressor.

Compressor Bus Access Error (slave_busy_err, slave_blocked_err)

If the hardware compressor does not get access to the SRAM via the internal bus, this is signaled by setting the slave_busy_err respectively the slave_blocked_err bit. The compression will be stopped. Also, this error can only occur when using the hardware compressor.

Invalid Address Error (invalid_address_err)

If the hardware compressor accesses an address that is outside the valid SRAM range, it receives an error on the internal bus and stops the compression. This behavior is indicated by setting the error bit invalid_address_err.

5.2 Read out the RDCU Hardware Information Registers

To read all metadata of the hardware compressor we provide the rdcu_read_cmp_info function. This function queries all compressor information registers and writes them into the cmp_info structure.

```
/**
 * @brief read out the metadata of a RDCU compression
```

```c
 *
 * @param info  compression information contains the metadata of a compression
 *
 * @note the compression information registers cannot be accessed during a compression
 *
 * @returns 0 on success, error otherwise
 */

int rdcu_read_cmp_info(struct cmp_info *info)
```

Listing 5.2: Declaration of the read metadata from the RDCU function.

5.3 Read out the RDCU SRAM

The function rdcu_read_cmp_bitstream and rdcu_read_model can be used to read the bitstream as well as the updated model from the RDCU SRAM.

```c
/**
 * @brief read the compressed bitstream from the RDCU SRAM
 *
 * @param info  compression information contains the metadata of a compression
 *
 * @param output_buf  the buffer to store the bitstream (if NULL, the required
 *          size is returned)
 *
 * @returns the number of bytes read, < 0 on error
 */

int rdcu_read_cmp_bitstream(const struct cmp_info *info, void *output_buf)


/**
 * @brief read the model from the RDCU SRAM
 *
 * @param info  compression information contains the metadata of a compression
 *
 * @param model_buf  the buffer to store the model (if NULL, the required size
 *          is returned)
 *
 * @returns the number of bytes read, < 0 on error
 */

int rdcu_read_model(const struct cmp_info *info, void *model_buf)
```

Listing 5.3: Declaration of the RDCU read bitstream and model functions.

6. Compressing Data

Finally, we get to the core of the whole thing — compressing data.

6.1 How to compress data with the Software Compressor on the ICU

We provide the function icu_compress_data for compression in software, see Listing 6.1. The function has three pointers to different structures as function parameters. The first structure is the compressor configuration structure cmp_cfg. It contains all parameters that control the compression. The structure and its parameters are explained in more detail in Chapter 3.

The second structure passed to the icu_compress_data function is the status structure cmp_status. This structure can be used to check the status of the compression. More details are explained in Chapter 4.

The cmp_info structure, which is also passed to the ICU compression function, contains all metadata such as the length of the compressed data of a compression. The content of the structure is the subject of Chapter 5.

The function takes the data and model (if necessary) from the buffers referenced in cmp_cfg and compresses them. The compressed bitstream is written to the icu_output_buf buffer. The metadata of the compression including the length of the compressed bitstream is written into the compressor information structure (cmp_info). This structure also contains the error codes to indicate that an error occurred during the compression. After each compression, it must be checked that no errors have occurred to verify that the compression data and the updated model are valid. The SW imagette compression works with the same algorithms as the HW compression. It is therefore controlled with the same configuration structure as the compressor. Since the parameters rdcu_***_adr are not needed to set up the SW compression, they are ignored. When compressing in 1d-differencing mode, no model is needed to compress data, so model-specific parameters (model_value, model_adr, rdcu_new_model_adr) are ignored.

The feature of semi adaptive compression is not supported by the SW compressor. As a result, the SW compressor will ignore the configuration parameters with the prefix ap1 or ap2. For this reason, the result of the adaptive compression the status parameters ap1_cmp_size and ap2_cmp_size will always be zero.

```
/**
 * @brief compress data on the ICU
 *
 * @param cfg   compressor configuration contains all parameters required for compression
 * @param info  compressor information contains information of an executed compression
 *
 * @note the validity of the cfg structure is checked before the compression is started
 * @note when using the 1d-differencing  mode or the raw mode (cmp_error 0-2), the model
    is ignored and not used
 * @note the rdcu_***_adr configuration parameters are ignored for SW compression
 * @note the rdcu_***_used info parameters are always set to zero
 * @note semi adaptive compression not supported; configuration parameters ap1\_golomb\
    _par,
 *          ap2\_golomb\_par, ap1\_spill ap2\_spill  will be ignored;
 *          information parameters ap1_cmp_size, ap2_cmp_size will always be zero
 *
 * @returns 0 on success, error otherwise
 */

int icu_compress_data(struct cmp_cfg *cfg, struct cmp_status *status, struct cmp_info *
    info)
```

Listing 6.1: Declaration of the ICU compress function.

6.1.1 Software Compression Example

Listing 6.2 shows an example of using the software compressor.

```
#define DATA_LEN 6 /* number of 16 bit samples to compress */

size_t i;
/* declare configuration and information structure */
struct cmp_cfg example_cfg;
struct cmp_status example_status;
struct cmp_info example_info;
/* declare data buffers */
uint16_t example_data[DATA_LEN] = {42, 23, 1, 13, 20, 1000};
uint16_t example_model[DATA_LEN] = {0, 22, 3, 42, 23, 16};
uint32_t *example_output_buf;

/* we make the buffer for the compress data as long as the input data buffer */
example_output_buf = (uint32_t *)malloc(sizeof(uint16_t[DATA_LEN]));

if (example_output_buf == NULL) {
  printf("Error allocating memory for the output buffer\n");
  return -1;
}
/* set up compressor configuration */
example_cfg = DEFAULT_CFG_MODEL;
example_cfg.input_buf = example_data;
example_cfg.model_buf = example_model;
example_cfg.samples = DATA_LEN;

example_cfg.icu_output_buf = example_output_buf;
example_cfg.buffer_length = DATA_LEN; /* we allocated the same buffer length as for the
    input_buf */

/* start SW compression */
icu_compress_data(&example_cfg, &example_status, &example_info);

/* check if data are valid */
if (example_status.data_valid == 0) {
  printf("Data are not valid. cmp_err = %d\n", example_info.cmp_err);
  free(example_output_buf);
  return -1;
}
```

```

printf("\nHere's the compressed data:\n"
       "================================\n");
for (i = 0; i < (example_info.cmp_size + 31)/32; i++) {
  printf("%08X ", example_output_buf[i]);
}

printf("\n\ncompressed data: %d bits\n", example_info.cmp_size);

printf("\n\nHere's the updated model:\n"
       "================================\n");
         for (i = 0; i < example_info.samples_used; i++) {
  printf("%04X ", example_model[i]);
}
printf("\n");

free(example_output_buf);
return 0;
```

Listing 6.2: Example of a software compression.

6.2 How to compress data with the Hardware Compressor on the RDCU

The UVIE team provides for the ICU a set of software to simplify the set up of an RDCU hardware compression. The functionality of the hardware compressor is very similar to that of the SW compressor.

By not compressing the data itself, but controlling the HW compressor that compresses the data, the HW compression is more complicated than just calling a function. First, the data and if needed the model must be written into the SRAM of the RDCU and the compressor configuration must be set into the appropriate registers. Then the hardware compressor is started. This is done with the rdcu_compress_data function. The parameters necessary for this step are stored in the cmp_cfg structure. For more information see Chapter 3

If the compression is running it is not possible to access the SRAM via RMAP. Only the compression status register is accessible. With the provided function rdcu_read_cmp_status these registers can be read. The function reads these registers and writes the content into the cmp_status structure. The cmp_ready bit in the structure can now be used to find out if a compression is still running (cmp_ready = 0) or the compression is finished and the compressor is ready to start a new one (cmp_ready = 1). If this is the case, the data_valid bit can also be checked to indicate that the compressed data is valid. Alternatively, you can wait for an interrupt from the RDCU, which tells you when the compressor is ready. It is also possible to query the status register afterwards to control the data_valid bit. More information on this topic can be found in Chapter 4. If the compression takes too long it can be interrupted with the rdcu_interrupt_compression function. After interrupting compression, the data in SRAM is invalid and cannot be processed any further.

When the compression is finished, the required metadata of the compression can be read by the RDCU. This job takes over the rdcu_read_cmp_info function. It reads the corresponding registers and writes the content into the passed cmp_info structure. Before the data of SRAM can be read, it must be checked that no error occurred during compression. If the parameter cmp_err = 0 no error occurred and the data can be read, see Chapter 5.

In the last step, the data can finally be read from the SRAM. The rdcu_read_cmp_bitstream function can be used to read the compressed data. To read the updated model from the SRAM the rdcu_read_model function can be used. After these steps, the compression is finished and a new one can be started.

6.2.1 Hardware Compression Example

Listing 6.3 shows a sample compression of the RDCU hardware compressor, it demonstrates how the different functions play together to achieve a compression of the data.

```
#define DATA_LEN 6 /* number of 16 bit samples to compress */

int error;
int cnt = 0;
/* declare configuration and information structure */
struct cmp_cfg example_cfg;
struct cmp_status example_status;
struct cmp_info example_info;
/* declare buffers */
uint16_t example_data[DATA_LEN] = {42, 23, 1, 13, 20, 1000};
uint16_t example_model[DATA_LEN] = {0, 22, 3, 42, 23, 16};

/* set up compressor configuration */
example_cfg = DEFAULT_CFG_MODEL;
example_cfg.input_buf = example_data;
example_cfg.model_buf = example_model;
example_cfg.samples = DATA_LEN;

/* start HW compression */
error = rdcu_compress_data(&example_cfg);
if (error != 0)
  printf("An error occurred during rdcu_compress_data\n");

/* start polling the compression status */
error = rdcu_read_cmp_status(&example_status);
if (error != 0)
  printf("An error occurred during rdcu_read_cmp_status\n");

cnt = 0;
while (example_status.cmp_ready != 0) {
  /* check compression status */
  error = rdcu_read_cmp_status(&example_status);
  if (error != 0)
    printf("An error occurred during rdcu_read_cmp_status\n");

  cnt++;
  if (cnt < 5)  /* wait for 5 polls */
    continue;


  printf("Not waiting for compressor to become ready, will check status and abort\n");

```

```
    /* interrupt the data compression */
    rdcu_interrupt_compression();

    /* now we may read the error code */
    error = rdcu_read_cmp_info(&example_info);
    if (error != 0){
      printf("An error occurred during rdcu_read_cmp_info\n");
      return;
    }
    printf("Compressor error code: 0x%02X\n", example_info.cmp_err);
    return;
  }

  printf("Compression took %d polling cycles\n\n", cnt);

  printf("Compressor status: ACT: %d, RDY: %d, DATA VALID: %d, INT: %d, INT_EN: %d\n",
         example_status.cmp_active,
         example_status.cmp_ready,
         example_status.data_valid,
         example_status.cmp_interrupted,
         example_status.rdcu_interrupt_en);

  /* now we may read the error code */
  error = rdcu_read_cmp_info(&example_info);
  if (error != 0)
    printf("An error occurred during rdcu_read_cmp_info\n");

  printf("Compressor error code: 0x%02X\n",
         example_info.cmp_err);

  printf("Compressed data size: %u bits\n", example_info.cmp_size);

  /* issue sync back of compressed data */
  /* read compressed data to some buffer and print */
  if (1) {
    uint32_t i;
    uint32_t s = (((example_info.samples_used >> 3) + 3) & ~0x3UL);
    uint8_t *myresult = malloc(s);

    error = rdcu_read_cmp_bitstream(&example_info, myresult);
    if (error != 0)
      printf("Error occurred by reading in the compressed data\n");

    printf("\n\nHere's the compressed data:\n"
           "================================\n");

    for (i = 0; i < s; i++) {
      printf("%02X ", myresult[i]);
      if (i && !((i+1) % 40))
        printf("\n");
    }
    printf("\n");

    free(myresult);
  }
  return;
```

Listing 6.3: Example of a hardware compression.

7. Frame Processing

There's a problem with compressing imagettes: the memory of the RDCU SRAM is much smaller than the sum of all imagettes of a readout cycle. For this reason, all imagettes must be divided into several chunks and each chunk must be individually compressed. Note that the imagettes in a chunk must be in the same order over time. The sum of all imagettes generated during a readout cycle of all cameras (every 25 seconds) is called a frame. As shown in Figure 7.1, depending on the processing strategy of the chunks, it may be necessary to wait until enough data is available for compression. If enough data is available, it can be divided into chunks. These data chunks are compressed individually by the RDCU, a detailed description of the process can be found in Section 7.1. After successful compression, a header is added to the compressed data. This header contains the necessary information to decompress the data again. The header is described in Section 8.

Once a chunk is compressed, the next one can be compressed. With an optimized processing order of the chunks, the throughput performance can be increased significantly, which is discussed in detail in Section 7.3.1.

7.1 Chunk Processing

The necessary data and configuration are transferred to the RDCU with the rdcu_compress_data() function. Once these steps have been taken, the function also starts the compression of the chunk. When the compression has finished the metadata of the compression can be read out from the compressor registers. A part of the metadata is the error register, which has to be checked. If an error occurs for example if the buffer for the compressed data was too small (small_buffer_err) or there was a multi-bit error (mb_err) when reading the SRAM, we suggest to let the data uncompressed because there is no time for further compression. In this case, the uncompressed data flag in the compression entity header should be set to "uncompressed" as well as the *used Compression Mode* should be set to raw mode.

If no error occurred, the compressed data can be read from the RDCU SRAM with the rdcu_read_cmp_bitstream() function. The compressed data must then be prefixed by a header that allows the data to be decompressed later.

If the updated model is still needed it can be transferred from the RDCU to the ICU with the function rdcu_read_model(). After that, the chunk is finished processing.

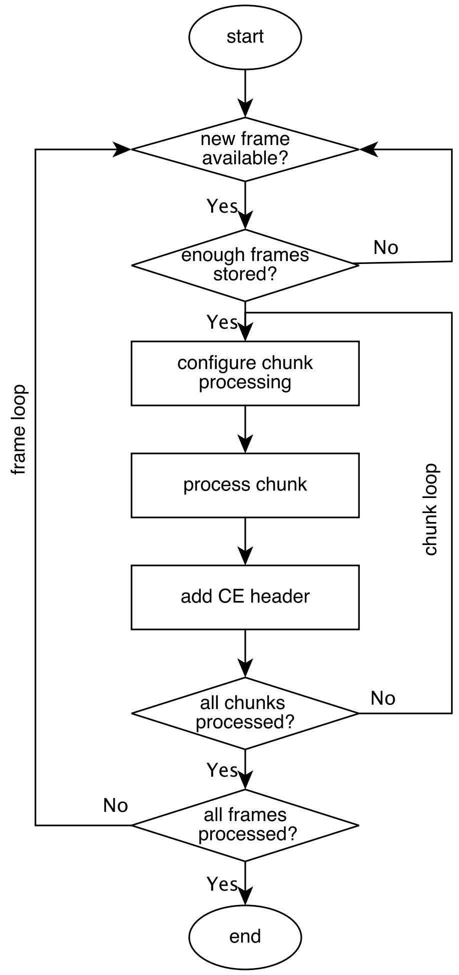

Figure 7.1: Frame processing workflow.

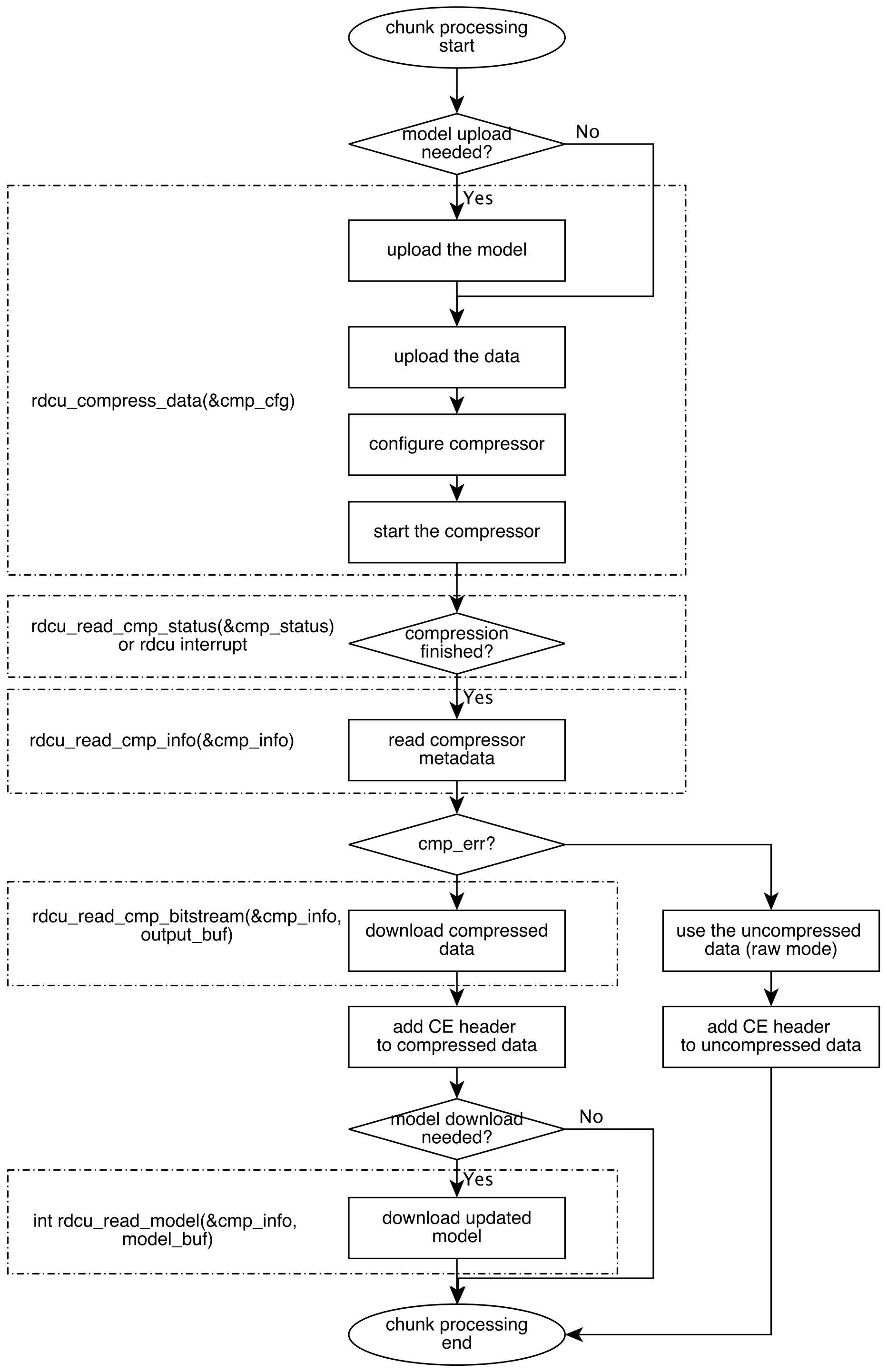
chunk processing start
model upload needed?
No
Yes
upload the model
upload the data
rdcu_compress_data(&cmp_cfg)
configure compressor
start the compressor
rdcu_read_cmp_status(&cmp_status) or rdcu interrupt
compression finished?
Yes
rdcu_read_cmp_info(&cmp_info)
read compressor metadata
cmp_err?
rdcu_read_cmp_bitstream(&cmp_info, output_buf)
download compressed data
use the uncompressed data (raw mode)
add CE header to compressed data
add CE header to uncompressed data
model download needed?
No
Yes
int rdcu_read_model(&cmp_info, model_buf)
download updated model
chunk processing end

Figure 7.2: Chunk processing workflow.

7.2 1d-Differencing Mode and Model Mode

The provided algorithms basically distinguish between repeating data and uniquely occurring data without "prehistory". It is important to understand how these modes work and how to use them to achieve good data compression. For single or first time data the 1d-differencing mode is used, for repeating data the model mode is used.

7.2.1 1d-Differencing Mode

This procedure considers all the data as a 1-dimensional array. The 1d-differencing algorithm is straightforward. The first value is the first value of the data chunk, after that only the difference to the left value is written. Example: the value series 100, 102, 99, 99, 105 will be processed in 100, 2, -3, 0, 6. That can be mathematically expressed as:

$$\text{output}_0 = \text{input}_0 \tag{7.1}$$

$$\text{output}_i = \text{input}_i - \text{input}_{i-1} \qquad i = 1, \dots, n \tag{7.2}$$

The results are then further processed and finally encoded with the Golomb code. The compression ratio, however, is usually not as good with this method as with the model method.

7.2.2 Model Mode

The output of this preprocessing process is simply the difference between the input data and its model:

$$\text{output}_i = \text{input}_i - \text{model}_i \tag{7.3}$$

The model should be understood as an average of the input data over time, which has the same size as the input data. The model is updated after every compression for the next compression of the same object in the following way:

$$\text{model}_1 = \text{input}_0 \tag{7.4}$$

$$\text{model}_{j+1} = \left\lfloor \frac{\text{model\_value} \cdot \text{model}_j + (16 - \text{model\_value}) \cdot \text{input}_j}{16} \right\rfloor \tag{7.5}$$

The model_value determines how fast the model changes. It is an integer value in the range [0,16].

The first input data in the model mode are preprocessed differently because no model is yet available. Depending on the input data, the first frame is preprocessed as 1d-differencing or raw mode (uncompressed) and used as the model for the next time. All other frames are preprocessed in model mode, where the input data is subtracted from the model data to reduce

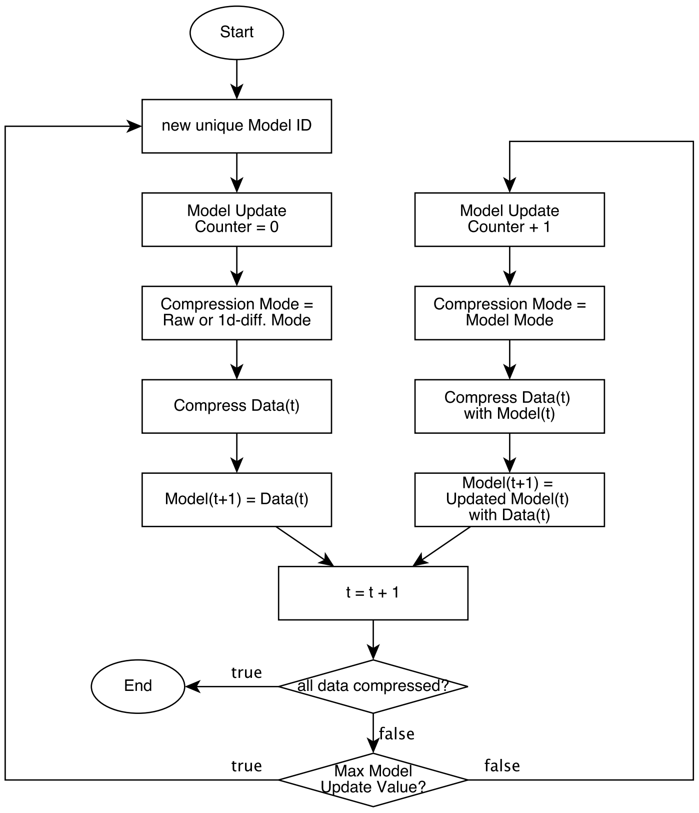

Figure 7.3: Flowchart of the 1d-differencing and model mode.

the data value to be compressed. The flowchart for using the 1d differencing and model modes together can be seen in Figure 7.3.

We recommend resetting the model after 8 model compression operations and starting again with a transfer using the 1d-differencing or raw mode. The model update counter counts how often the model is updated. It is zero if a non-model mode is used. A new unique model ID must be used for the next data sets when using a new start model (using raw or 1d-dif. mode). The model ID, together with the model update counter, can be used to determine which data set was compressed and in which order. Both parameters, the model update counter and the model ID, are part of the compression entity header (see section 8) to ensure the correct order in the decompression process.

7.3 Chunk Procedure Order

The not optimised chunk processing order works as follows. The chunk and his model are transferred to the RDCU. The data get compressed with RDCU. The compressed bitstream is (together with the metadata) downloaded from the RDCU and prefixed with a header. Also, the uploaded model is downloaded from the RDCU, which is needed to compress the same chunk of the next frame. Then the next chunk and its model will be uploaded and compressed for compression and so on.

7.3.1 Optimised Chunk Processing

Data throughput analyses of the compressor have shown that without optimisation chunk processing order it is not possible to compress the required 23,400 imagettes (assuming a compression factor of 3) in the given time. However, this problem can be solved by an optimized chunk processing order we suggested in **[RD-3]**. With this procedure, it is necessary to store 2 complete sets of imagettes. We call these sets N and N+1. First, the imagettes are divided into chunks. Then a chunk from the set N and its model is sent to the RDCU and processed. In the next step, the metadata and the compressed bitstream are downloaded from the RDCU but not the updated model. Now the same chunk but from the N+1 set is sent to the RDCU. An upload of the model is not necessary because it is already in the RDCU SRAM. Now the 2nd chunk can be compressed. After the compression the bitstream and now also the updated model will be downloaded from the RDCU. The updated model is needed to compress the same chunk from the N+2 set. Then the process starts from the beginning and the next part of the N+2 set can be compressed.

By this procedure, an upload and download of the model can be saved and the required data throughput can be achieved. A more detailed analysis of the data throughput of the HW compressor can be found in **[RD-3]**. Figure 7.4 shows a visualization of the optimised chunk processing order.

7.3.2 Chunk Size

Since the RDCU has an 8 MB SRAM of memory available we propose to divide the SRAM into three parts and use a chunk size of 2.6 MB. The first third should be used for the input data, the second third for the model data, and the last third for the compressed bitstream. It is also possible to shorten the memory area for the compressed data to create more space for the other areas. Even smaller chunk sizes are an option. The only disadvantage with small chunk sizes is that the overhead is increased by adding the header. So it is up to the ICU team to decide for larger chunks and a few compressions per frame or small chunks and more compressions.

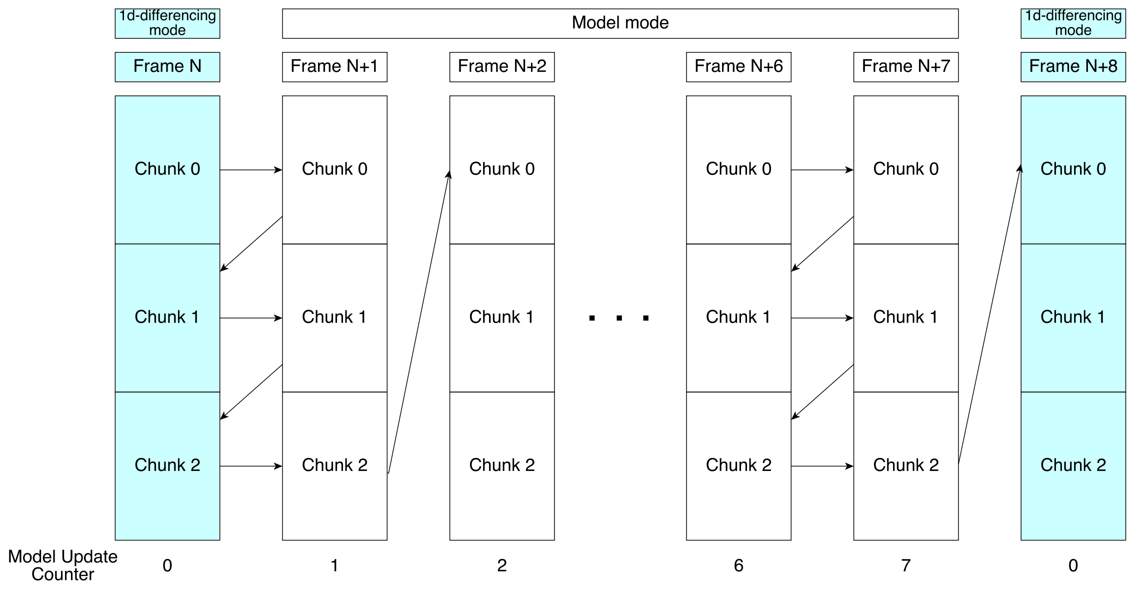

Figure 7.4: Visualization of the optimised chunk processing order.

8. Compression Entity Format

All compressed data has to be prefixed by a header. This header contains the necessary parameters for decompressing the compressed data and the required information for reconstructing the original data. We call the compressed data together with the header a *compression entity*. The compression entity header consists of two parts:

- **generic compression entity header** containing all parameters that are needed for all data product types. This header is used for all data product types.
- **specific compression entity header** containing parameters that are specific for the compressed data product type. This header is different for different data product types.

The structure of a compression entity can be seen in Figure 8.1. A detailed description of the header parameters can be found in Table 8.1. The compressed data from the RDCU does not contain the compression entity header and therefore the header must be added after downloading the compressed data from the RDCU.

Note: As described in **[RD-5]** a PLATO science packet is limited to 64 kilobytes. Therefore, the compression entity has to be split into several packets, each containing a chunk header, to restructure the compression unit and map the compressed data to a chunk ID. This chunk header is not part of the compression unit described in this document.

8.1 Specific Compression Entity Header

There are two specific compression entity headers for compressed imagette data defined. The one shown in 8.2 includes additional to the imagette specific decompression parameters also the parameters which control the semi-adaptive compression feature. If the semi-adaptive compression parameters are not needed or available the specific compression entity shown in 8.3 can be used for compressed imagette data.

For non-imagette data, the specific compression entity header shown in Figure 8.4 should be used. Table 8.2 lists which data product can be used for which data product type.

For uncompressed data (raw mode) indicated by the uncompressed data bit in the data product type field, no specific compression entity header is used.

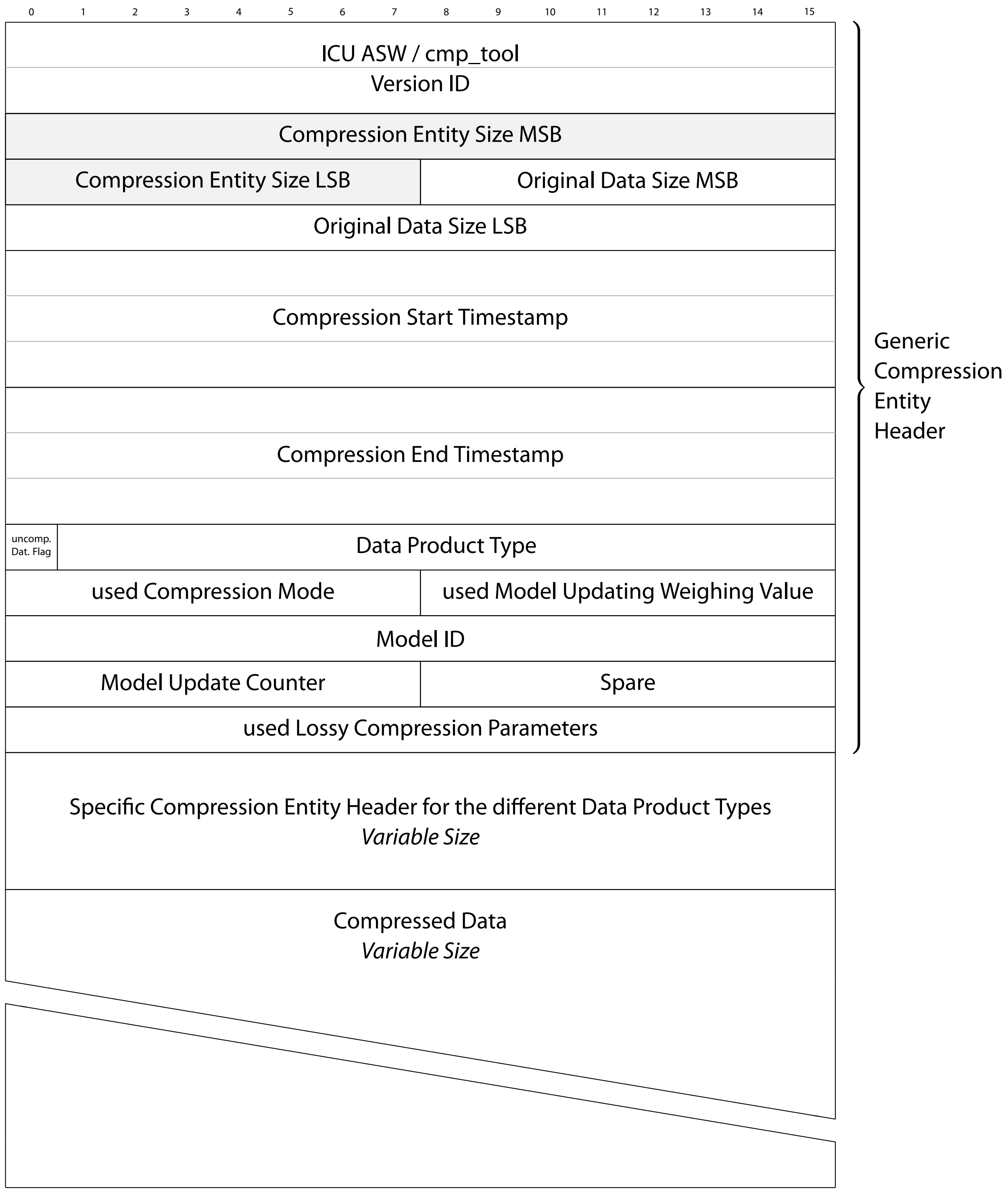
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
ICU ASW / cmp_tool
Version ID
Compression Entity Size MSB
Compression Entity Size LSB
Original Data Size MSB
Original Data Size LSB
Compression Start Timestamp
Compression End Timestamp
uncomp. Dat. Flag
Data Product Type
used Compression Mode
used Model Updating Weighing Value
Model ID
Model Update Counter
Spare
used Lossy Compression Parameters
Specific Compression Entity Header for the different Data Product Types
Variable Size
Compressed Data
Variable Size
Generic Compression Entity Header

Figure 8.1: Structure of a compression entity consisting of a generic header, a data product type specific header and the compressed data.

Length [Bit]	Parameter	Description	Value Range
32	ICU ASW Version ID	ICU application software/cmp_tool identifier. The first bit is used to distinguish betw. ICU ASW and cmp_tool.	uint32_t
24	Compression Entity Size	Describes the size of the entity (header + compressed data) in bytes	$[0..2^{24}[$
24	Original Data Size	Size of the data before compression in bytes	$[0..2^{24}[$
48	Comp. Start Timestamp	Time when the compression was started	CUC time
48	Comp. End Timestamp	Time when the compression was finished	CUC time
16	Data Product Type	To specify which data product is compressed see Table 8.2. The MSB in the data product type is set for uncompressed data.	uint16_t
8	used Compression Mode	Selected compression mode	uint8_t
8	u. Model Upd. Weigh. Val.	Used model weighting parameter	0..16
16	Model ID	Model identifier for identifying entities that originate from the same starting model.	uint16_t
8	Model Update Counter	Counts how many times the model was updated.	uint8_t
8	Spare		
16	used Lossy Comp. Par.	Parameter controlling the lossy compression	uint16_t
96, 32, 256, 0	Specific Entity Header	Data product type specific header for imagette and non-imagette data	custom see Fig. 8.2, 8.3, 8.4
var.	Compressed Data	Compressed data	custom

Table 8.1: Compression entry header parameters description.

Data Product Type	Description	specific compression header
1	NCxx_S_SCIENCE_IMAGETTE	(adaptive) imagette header
2	NCxx_S_SCIENCE_SAT_IMAGETTE	TBC (adaptive) imagette header
3	NCxx_S_SCIENCE_OFFSET	non-imagette header
4	NCxx_S_SCIENCE_BACKGROUND	non-imagette header
5	NCxx_S_SCIENCE_SMEARING	non-imagette header
6	NCxx_S_SCIENCE_S_FX	non-imagette header
7	NCxx_S_SCIENCE_S_FX_DFX	non-imagette header
8	NCxx_S_SCIENCE_S_FX_NCOB	non-imagette header
9	NCxx_S_SCIENCE_S_FX_DFX_NCOB_ECOB	non-imagette header
10	NCxx_S_SCIENCE_L_FX	non-imagette header
11	NCxx_S_SCIENCE_L_FX_DFX	non-imagette header
12	NCxx_S_SCIENCE_L_FX_NCOB	non-imagette header
13	NCxx_S_SCIENCE_L_FX_DFX_NCOB_ECOB	non-imagette header
14	NCxx_S_SCIENCE_F_FX	non-imagette header
15	NCxx_S_SCIENCE_F_FX_DFX	non-imagette header
16	NCxx_S_SCIENCE_F_FX_NCOB	non-imagette header
17	NCxx_S_SCIENCE_F_FX_DFX_NCOB_ECOB	non-imagette header
18	FCx_R_SCIENCE_IMAGETTE	TBC (adaptive) imagette header
19	FCx_R_SCIENCE_OFFSET_VALUES	TBC (adaptive) imagette header
20	FCx_R_BACKGROUND_VALUES	non-imagette header

Table 8.2: Which specific compression header can be used for which data product type.

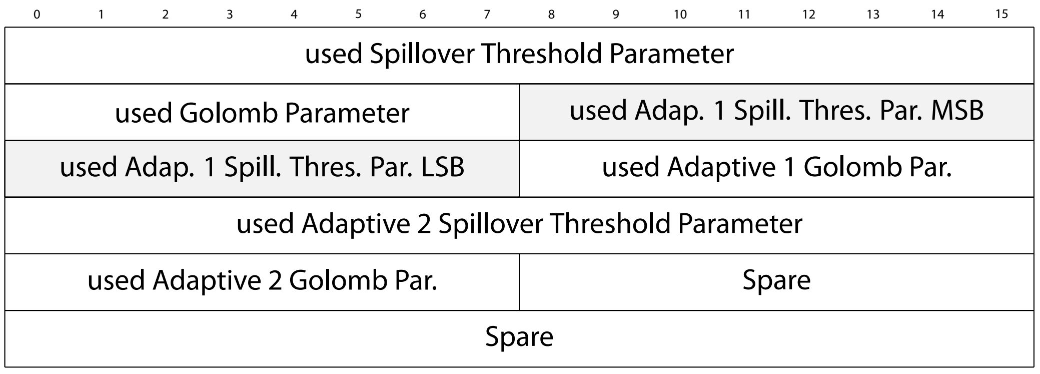
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
used Spillover Threshold Parameter
used Golomb Parameter
used Adap. 1 Spill. Thres. Par. MSB
used Adap. 1 Spill. Thres. Par. LSB
used Adaptive 1 Golomb Par.
used Adaptive 2 Spillover Threshold Parameter
used Adaptive 2 Golomb Par.
Spare
Spare

Figure 8.2: Specific compression entity header for RDCU imagette compression containing the semi-adaptive compression feature.

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
used Spillover Threshold Parameter	
used Golomb Parameter	Spare

Figure 8.3: Specific compression entity header for RDCU (or ICU) imagette compression without containing the semi-adaptive compression feature.

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
used Spillover Threshold Parameter 1 MSB	
used Spillover Threshold Par. 1 LSB	used Compression Par. 1 MSB
used Compression Parameter 1 LSB	used Spillover Threshold Par. 2 MSB
used Spillover Threshold Parameter 2 LSB	
used Compression Parameter 2	
⋮	
used Spillover Threshold Parameter 5 MSB	
used Spillover Threshold Par. 5 LSB	used Compression Par. 5 MSB
used Compression Parameter 5 LSB	used Spillover Threshold Par. 6 MSB
used Spillover Threshold Parameter 6 LSB	
used Compression Parameter 6	
Spare	

Figure 8.4: Specific compression entity header for non-imagette data compression.