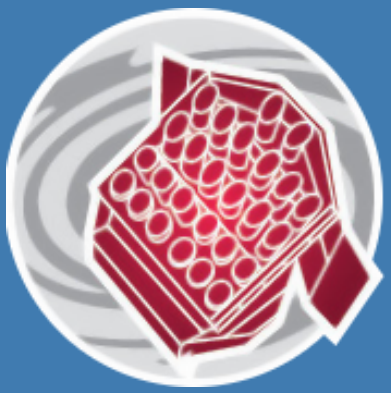
institut für astrophysik
UNIVERSITÄTSSTERNWARTE WIEN

universität wien

PLATO

Data Compression User Manual



Data Compression User Manual

Reference: PLATO-UVIE-PL-UM-0001

Version: Issue 1 Revision 1, 29. January 2025

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 4.6
Draft 6	25.01.2022	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.
Issue 1	01.07.2022	DL	major restructuring of the chapters, add HW configuration functions, add SW compression for non-imagette data (compression parameters, configuartion function), add max bit used section, add max bits used version in compression entity
Issue 1r1	30.01.2025	DL	This revision replaces the old SW compression with the new SW chunk compression; Update Introduction (Chapter 2); Update SW compression parameters (Section 3.5); major update of the Software Compression (Chapter 3); Update CE Header Chapter (Chapter 6); add internal chunk data structure (Section 6.2); add references to example code files and therefore remove code examples from the appendix

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

ID	Title, Reference Number, Revision Number
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 ICU RDCU User Manual, PLATO-IWF-PL-UM-0076 Issue 1.2, 7. October 2021
RD-3	PLATO RDCU Data Throughput, PLATO-IWF-PL-TN-059, 19. August 2019
RD-4	FPGA Requirement Specification, PLATO-IWF-PL-RS-0005 Issue 1.2, 05. March 2021
RD-5	Level0 data generation from the payload science data, PLATO-DLR-MIS-TN-0002, 14. October 2020
RD-6	PLATO N-DPU ASW Data Rate and Memory Budget (B Phase) Issue 2.9, PLATO-LESIA-PL-RP-0031, 14. October 2020

1. Terms, Definitions and Abbreviated Items

1.1 Acronyms

API Application Programming Interface
CE Compression Entity 35, 40, 42, 47
CPU Central Processing Unit
FPGA Field-Programmable Gate Array 7, 9
HW Hardware 14–16, 19, 20, 22–25, 27, 47
ICU Instrument Control Unit 7, 9, 19, 28, 30, 42, 47
ISR Interrupt Service Routine
MMU Memory Management Unit
PUS Packet Utilisation Standard
RDCU Router and Data Compression Unit 7, 9–11, 14, 17, 19–24, 26, 27, 35, 37, 38, 42, 46, 47
RISC Reduced Instruction Set Computing
RMAP Remote Memory Access Protocol 9, 11, 19, 20, 22
SRAM Static Random Access Memory 9, 14, 15, 17, 19, 20, 26, 27, 42, 47
SW Software 35
UVIE University of Vienna 7, 9, 19

2. Introduction

The University of Vienna (UVIE) team provides the data compression to support the Instrument Control Unit (ICU) in compressing the different science PLATO data products. The imagette compression algorithms have been implemented in hardware on a Field-Programmable Gate Array (FPGA) on the Router and Data Compression Unit (RDCU) board. Non-imagette data and also the imagette data can be compressed with the provided software library.

2.1 Purpose of the Document

This document is about the handling of the provided 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 lossy/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. In this mode, the difference to the previous data value of the same type is calculated in order 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

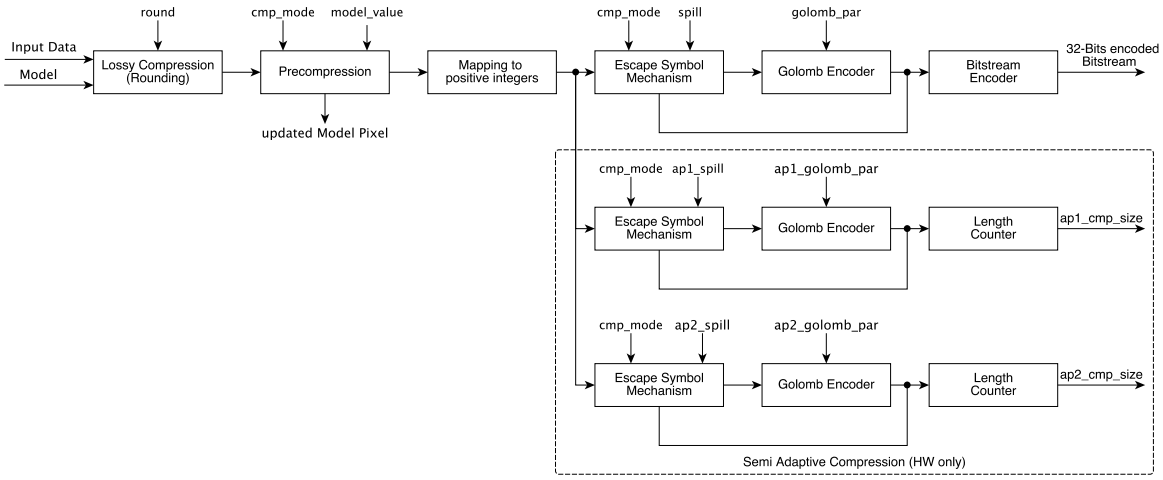


Figure 2.1: Visualization of the compression algorithm.

difference between the input data and a model of this data is formed. The model is updated after a compression and is used to 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 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 bitstream encoder 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 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 *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 signalled to the ICU by an interrupt signal or by setting the *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.

2.4 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 includes 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 5 discusses the provided function for software compression in detail.

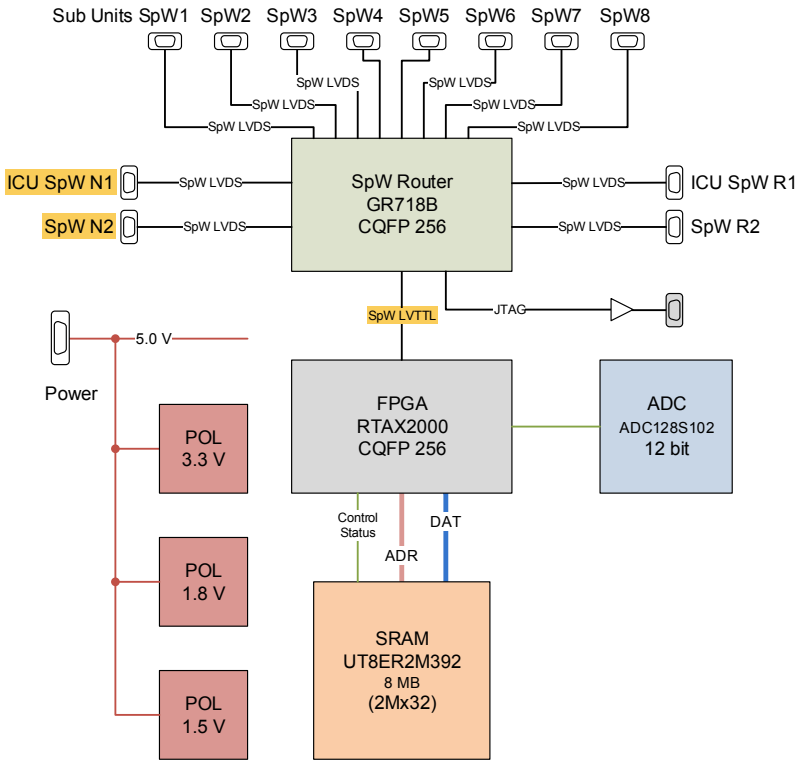


Figure 2.2: RDCU Electrical Concept.

3. Hardware & Software Compression Parameters

The compression is controlled by several compression parameters.

For the hardware compression, the provided library can 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]**.

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]**.

3.1 Generic Compression Parameters

The following generic compression parameters are required for the compression of any data product.

3.1.1 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.

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 or values 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]**.

3.1.2 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 weigh 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.

3.1.3 Lossy Rounding Parameter (lossy_par or round)

The lossy 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 RDCU imagette compression also treats the header of the imagette collection like normal data, it must be ensured that this header is not corrupted by rounding the last bits.

The SW compression does not support lossy compression and ignores this parameter.

3.2 RDCU Data Buffers Parameters

The following section describes the parameters related to the different data buffers needed for an RDCU compression.

3.2.1 Data to be Compressed Buffer (data_to_compress)

The data to be compressed buffer contains the input data for the compression located on the ICU.

3.2.2 Data Samples (data_samples)

The data samples parameter describes the length of the data to be compressed. When compressing data with the RDCU, the length of the entire data to be compressed, including the collection header, is measured in 16-bit samples. The RDCU compressor makes no distinction between header and imagette data. Therefore, the data_samples parameter is the number of imagette pixels plus the length of the collection header, measured in 16-bit units. The compression of multiple joined collections is possible.

3.2.3 Model of Data Buffer (model_of_data)

If a model compression mode is used, a model of the data to be compressed is required. The model of the data buffer contains this model data. The length of the model buffer is the same as that of the buffer for the data to be compressed.

3.2.4 Updated/New Model Buffer (updated_model)

The updated or new model buffer is needed for model compression for the next data set for an object. The buffer specifies where this data is stored. It can be the same buffer as the model data buffer for an in-place update of the model. This buffer is used only for the model compression mode and has the same size as the data buffer to be compressed.

3.2.5 Compressed Data Buffer (compressed_data)

The result of the compression, the compressed data bitstream, is transferred from the RDCU to the compressed data buffer on the ICU.

3.2.6 Compressed Data Buffer Length (compressed_data_len_samples)

The compressed data buffer length parameter specifies the length of the reserved buffer for the compressed data in the same unit as the samples parameter. When using the Hardware (HW) compression, this parameter specifies the length of the reserved area for compressed data after the rdcu_buffer_adr in the RDCU SRAM. If the compressed data buffer is too small to store all the compressed data, the small_buffer_err error is returned.

Note: If the compression parameters are not set correctly, it is possible that the "compressed data" will be larger than the original 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.

3.2.7 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 RDCU SRAM where the uncompressed data, the model data, the new 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.

3.3 RDCU Imagette Specific Compression Parameters

The following compression parameters are needed to compress imagette data on the RDCU.

3.3.1 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.

3.3.2 Spillover Threshold Parameter (spillover_par)

The escape symbol mechanism is controlled by the spillover threshold parameter (spill or spillover_par). The spill parameter controls if a value is considered to be an outlier. If an outlier is recognized, the 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 3.1 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]**.

3.3.3 Spillover Golomb Parameter Error

The choice of the spill parameter is closely related to the Golomb parameter. This connection exists because the RDCU 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 3.1 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 3.1 are the same for ap1_spill and ap2_spill parameters.

Table 3.1: Valid spillover threshold parameter (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		

3.3.4 Adaptive Golomb Parameter 1/2 (ap1_golomb_par, ap2_golomb_par), Adaptive Spillover Threshold 1/2 (ap1_spillover_par, ap2_spillover_par)

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 and ap2_golomb_par. As explained in more detail in Section 3.3.3, an ap_spill parameter can be selected up to a specific value depending on the set ap_golomb_par parameter.

3.4 Compression Parameter Errors

A large number of compression parameters only accept 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 possible unpredictable behaviour.

The hardware compressor indicates an error in the compression error register. The software compressor displays an error in the return value of the compression function call. All hardware or software configuration setup functions return a non-zero value if a value is not in the correct range. Table 3.2 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 3.2: 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 3.3.3]
Adaptive Golomb Parameter 1/2	ap1/2_golomb_par	[1,63]
Adaptive Spillover Threshold 1/2	ap1/2_spill	[2, see Section 3.3.3]
RDCU SRAM Addresses	rdcu_***_adr	[0x000000, 0x7FFFFF]

3.5 Specific Software Chunk Compression Parameters

The following section describes the data type specific parameters required for the software chunk compression. A chunk in this context consists of multiple joined data collections, this data format is described in detail in **[RD-5]**.

The SW compression is controlled with the set parameters defined in the `struct cmp_par`. It contains the cmp_mode model_value and lossy_par as described in Section 3.1. Besides that the structure contains data type specific compression parameters. These parameters allow adapting the compression to the properties of the different science data. Depending on the type of a chunk, a different number of data type specific compression parameters are needed, see Table 3.3. Choosing a Spillover parameter like for an RDCU compression is not needed, the SW compressor chooses that internally.

Collection Type	**Used Compression Parameters**
N-CAM Imagette	nc_imagette
S_FX, S_FX_EFX, S_FX_NCOB, S_FX_EFX_NCOB_ECOB	s_exp_flags, s_fx, s_ncob, s_efx, s_ecob
L_FX, L_FX_EFX, L_FX_NCOB, L_FX_EFX_NCOB_ECOB	l_exp_flags, l_fx, l_ncob, l_efx, l_ecob, l_fx_cob_variance
Saturated Imagette	saturated_imagette
N-CAM Offset, Background	nc_offset_mean, nc_offset_variance, nc_background_mean, nc_background_variance, nc_background_outlier_pixels
Smearing	smearing_mean, smearing_variance_mean, smearing_outlier_pixels
F-CAM Imagette, Offset, Background	fc_imagette, fc_offset_mean, fc_offset_variance, fc_background_mean, fc_background_variance, fc_background_outlier_pixels

Table 3.3: Required specific chunk compression parameters for a given collection type.

4. Hardware/RDCU Compression

The UVIE team provides for the ICU a set of software functions to simplify the setup of an RDCU hardware compression. The HW compressor can only compress imagette data.

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 transferred to the appropriate registers. Then the hardware compressor is started. These setup steps are done with the `rdcu_compress_data()` function. The parameters necessary for this step are stored in the compression configuration structure. This is discussed in detail in Section 4.1

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. In this case, you can query the status register after the interrupt to check the data_valid bit. More information on this topic can be found in Section 4.2. If the compression takes too long it can be interrupted with the `rdcu_interrupt_compression()` function. After interrupting the compression, the data in the SRAM is invalid and cannot be processed any further.

When the compression is finished, the required metadata of the compression can be read from the RDCU. This is done with 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 compression error parameter is zero (cmp_err = 0), no error occurred and the data can be read, see Section 4.3.

In the next 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, see Section 4.4. After these steps, the compression is finished and a new one can be started. Please refer to the example for the use of the hardware compression library in `examples/example_cmp_rdcu.c` and the function declarations in `lib/cmp_rdcu.h` within the project directory.

In the end, the compressed data have to be prefixed with a compression entity header that contains the information necessary to decompress the compressed data. The compression

entity header is described in Chapter 6.

4.1 Configure and Start the Hardware/RDCU Compressor

Listing 4.1 shows the functions required to configure and set up a HW compression. The first step to start a hardware compression is to create a compression configuration with the `rdcu_cfg_create()` function. The parameters of the function are the generic compression parameters, refer to Section 3.1 for more details.

The returned configuration can now be used as input to the `rdcu_cfg_buffers()` function for configuring the buffer-related parameters described in Section 3.2. The configuration structure is also an input for the `rdcu_cfg_imagette()` function for configuring the RDCU imagette compression parameters, see Section 3.3.

The provided function `rdcu_compress_data()` 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 function also packs the data to be compressed into RMAP packets which are sent to the RDCU SRAM. If model mode is used, the model is also uploaded to the RDCU SRAM. If non-model compression mode is used, the model is ignored. Finally, an RMAP packet is created to start the compression.

The `rdcu_compress_data()` function only sets up and starts the compression, the download of the compressed data is done by another function, see Section 4.4. Note: Before the `rdcu_compress_data()` function can be used, an initialisation of the RMAP library is required. This is achieved with the functions `rdcu_ctrl_init()` and texttttrdcu_rmap_init().

```
/**
 * @brief create an RDCU compression configuration
 *
 * @param data_type compression data product type
 * @param cmp_mode  compression mode
 * @param model_value model weighting parameter (only needed for model compression mode)
 * @param lossy_par lossy rounding parameter (use CMP_LOSSLESS for lossless compression)
 *
 * @returns a compression configuration containing the chosen parameters;
 *  on error the data_type record is set to DATA_TYPE_UNKOWN
 */

struct cmp_cfg rdcu_cfg_create(enum cmp_data_type data_type, enum cmp_mode cmp_mode,
            uint32_t model_value, uint32_t lossy_par)


/**
 * @brief setup of the different data buffers for an RDCU compression
 *
 * @param cfg      pointer to a compression configuration (created
 *         with the rdcu_cfg_create() function)
 * @param data_to_compress  pointer to the data to be compressed (if NULL no
 *         data transfer to the RDCU)
 * @param data_samples    length of the data to be compressed measured in
 *         16-bit data samples (ignoring the collection header)
 * @param model_of_data    pointer to the model data buffer (only needed for
```

```
27  *         model compression mode, if NULL no model data is
28  *         transferred to the RDCU)
29  * @param rdcu_data_adr   RDCU SRAM data to compress start address
30  * @param rdcu_model_adr  RDCU SRAM model start address (only needed for
31  *         model compression mode)
32  * @param rdcu_new_model_adr  RDCU SRAM new/updated model start address (can be
33  *         the same as rdcu_model_adr for in-place model update)
34  * @param rdcu_buffer_adr RDCU SRAM compressed data start address
35  * @param rdcu_buffer_lenght  length of the RDCU compressed data SRAM buffer
36  *         measured in 16-bit units (same as data_samples)
37  *
38  * @returns 0 if parameters are valid, non-zero if parameters are invalid
39  */
40
41 int rdcu_cfg_buffers(struct cmp_cfg *cfg, uint16_t *data_to_compress,
42          uint32_t data_samples, uint16_t *model_of_data,
43          uint32_t rdcu_data_adr, uint32_t rdcu_model_adr,
44          uint32_t rdcu_new_model_adr, uint32_t rdcu_buffer_adr,
45
46
47 /**
48  * @brief set up the configuration parameters for an RDCU imagette compression
49  *
50  * @param cfg      pointer to a compression configuration (created
51  *         with the rdcu_cfg_create() function)
52  * @param golomb_par     imagette compression parameter
53  * @param spillover_par   imagette spillover threshold parameter
54  * @param ap1_golomb_par  adaptive 1 imagette compression parameter
55  * @param ap1_spillover_par adaptive 1 imagette spillover threshold parameter
56  * @param ap2_golomb_par  adaptive 2 imagette compression parameter
57  * @param ap2_spillover_par adaptive 2 imagette spillover threshold parameter
58  *
59  * @returns 0 if parameters are valid, non-zero if parameters are invalid
60  */
61
62 int rdcu_cfg_imagette(struct cmp_cfg *cfg,
63          uint32_t golomb_par, uint32_t spillover_par,
64          uint32_t ap1_golomb_par, uint32_t ap1_spillover_par,
65          uint32_t ap2_golomb_par, uint32_t ap2_spillover_par)
66
67
68 /**
69  * @brief compressing data with the help of the RDCU hardware compressor
70  *
71  * @param cfg  configuration contains all parameters required for compression
72  *
73  * @note Before the rdcu_compress function can be used, an initialisation of
74  *  the RMAP library is required. This is achieved with the functions
75  *  rdcu_ctrl_init() and rdcu_rmap_init().
76  * @note The validity of the cfg structure is checked before the compression is
77  *   started.
78  *
79  * @returns 0 on success, error otherwise
80  */
81
82 int rdcu_compress_data(const struct cmp_cfg *cfg)
```

Listing 4.1: Declaration of the RDCU configuration and compression function.

4.2 Reading the RDCU Status Register

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

4.2.1 HW 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.

```
/**
 * @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 4.2: C-Implementation of the compressor status structure.

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.

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.

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 is set to 1 after compression is complete. The value remains 1 until a new compression is started.

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. To interrupt a compression use the `rdcu_interrupt_compression()` function.

RDCU Interrupt Signal Enable (rdcu_interrupt_en)

The RDCU interrupt signal enable bit is mirroring the RDCU interrupt signal enable value in the compression control register. To enable or disable the interrupt signal use the `rdcu_enable_interrput_signal()` respectively `rdcu_disable_interrput_signal()` function before starting a RDCU compression.

4.2.2 RDCU Status Register Read Function

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 HW compression process. The function can be used to poll the status of a compression to find out when the compression is finished.

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

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

1-D Differencing mode:

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

```
/**
 * @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 4.3: Declaration of the RDCU read status function.

4.3 RDCU Compression Information Register Read Function

Once the HW compression is complete, we need more information than the compressed bit-stream to process the data further. This metadata can be stored in the provided compressor information structure cmp_info.

4.3.1 Compression Information Structure

The cmp_info structure shown in Listing 4.4 contains all readable information registers of the HW compressor. These registers are only readable when the compressor is not active. Before the compressed data from the compressor can 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.

```
/* The cmp_info structure can contain the information and metadata of an
 * executed RDCU compression.
 */

struct cmp_info {
  uint32_t cmp_mode_used;        /* Compression mode used */
  uint32_t spill_used;           /* Spillover threshold used */
  uint32_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 */
  uint8_t  model_value_used;     /* Model weighting parameter used */
  uint8_t  round_used;           /* Number of noise bits to be rounded used */
  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)
```

```
27                 * [bit 9] invalid address_err; The bus master has received the "invalid
       "address status (only HW compression)
28                 */
29 };
```

Listing 4.4: C-Implementation of the compressor information structure.

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.

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:

$$\mathrm{CR} = \frac{\mathrm{model\_length\_used} \cdot 16\,\mathrm{Bit}}{\mathrm{cmp\_size}} \tag{4.3}$$

Note: This calculation assumes imagette compression, where one sample has 16 bits.

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.

4.3.2 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 unfavourably 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 signalled 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 behaviour is indicated by setting the error bit invalid_address_err.

4.3.3 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 an RDCU compression
 *
 * @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
 */
```

```
10
11 int rdcu_read_cmp_info(struct cmp_info *info)
```

Listing 4.5: Declaration of the read metadata from the RDCU function.

4.4 RDCU SRAM Read Function

After verifying that the RDCU HW compression is complete (by reading the status register and checking the data compressor ready flag) and verifying that no compression error occurred (by reading the metadata and checking the cmp_err register is zero), we can read the results from the RDCU SRAM. The `rdcu_read_cmp_bitstream()` and `rdcu_read_model()` functions can be used to get either the bitstream or respectively the updated model from the RDCU SRAM.

```
1 /**
2  * @brief read the compressed bitstream from the RDCU SRAM
3  *
4  * @param info  compression information contains the metadata of a compression
5  * @param compressed_data the buffer to store the bitstream (if NULL, the
6  *        required size is returned)
7  *
8  * @returns the number of bytes read, < 0 on error
9  */
10
11 int rdcu_read_cmp_bitstream(const struct cmp_info *info, void *compressed_data)
12
13
14
15 /**
16  * @brief read the updated model from the RDCU SRAM
17  *
18  * @param info    compression information contains the metadata of a compression
19  *
20  * @param updated_model the buffer to store the updated model (if NULL, the required size
21  *       is returned)
22  *
23  * @returns the number of bytes read, < 0 on error
24  */
25
26 int rdcu_read_model(const struct cmp_info *info, void *updated_model)
```

Listing 4.6: Declaration of the RDCU read bitstream and model functions.

5. Software Compression

This chapter describes the software chunk compression for the ICU.

The way the PLATO science data are organised is that one or many parameters, like the flux and the center of brightness, are combined in one entry. Many entries together with a collection header build a collection. For the definition of science packets see **[RD-6]**. Multiple collections are then further combined into so-called chunks, as described in **[RD-5]**.

Please refer to the example for the use of the compression library in `examples/example_compress_chunk.c` and the function declarations in `lib/cmp_chunk.h` within the project directory.

5.1 Initialisation

In contrast to the hardware compression, the compression entity header is created by the chunk compression itself. The user does not need to add the header. In order for the software chunk compression to build the compression entity header, it needs to be initialised with a function returning the current timestamp and the version identifier of the application software. This is done with the function `compress_chunk_init()` once before the first chunk compression is performed.

```
/**
 * @brief initialise the compress_chunk() function
 *
 * If not initialised the compress_chunk() function sets the timestamps and
 * version_id in the compression entity header to zero
 *
 * @param return_timestamp  pointer to a function returning a current 48-bit timestamp
 * @param version_id    application software version identifier
 */

void compress_chunk_init(uint64_t (*return_timestamp)(void), uint32_t version_id);
```

Listing 5.1: Function prototype of the ICU chunk software compression initialisation function.

5.2 Compression Process

After the compression is initialized, we are ready to compress data. This is done with the function `compress_chunk()`, see Listing 5.2. The function needs several parameters to perform a compression.

First, it requires a pointer to the chunk to be compressed (`chunk`). The size of the chunk

in bytes is specified by the `chunk_size` argument. When we perform a model mode compression, a model of the chunk (`chunk_model`) has to be provided. The model has the same size as the chunk. More information about the model mode can be found in Section 7.2.2. The `chunk_model` pointer can be `NULL` if no model compression mode is used.

A pointer to store the updated chunk model (`updated_chunk_model`) can also be supplied, which will be needed for the next round of model mode compression for the same objects. This buffer must have at least the same size as the chunk. The `updated_chunk_model` pointer can be the same as the `chunk_model` pointer for an in-place update of the model or `NULL` if the updated model is not needed.

Next, a destination pointer (`dst`) to the compressed data buffer is provided. This pointer address must be 4-byte-aligned (a multiple of 4). The `dst` pointer can be `NULL` if only the size of the compressed data is of interest, without the compressed data actually being saved.

The capacity measured in bytes of this destination buffer is given by `dst_capacity` argument. It's recommended to provide a `dst_capacity` that is equal to or larger than the maximum compression size returned by `compress_chunk_cmp_size_bound(chunk, chunk_size)` function. This recommendation helps to avoid failure scenarios where there is not enough space in the destination buffer to write the compressed data into it. Note that the size is internally rounded down to a multiple of 4.

The `compress_chunk()` function also requires a pointer to a compression parameters structure (`cmp_par`). This structure contains various parameters that control the compression process, which are described in Section 3.5. We propose to use two compression structures, one for model compression and one for compression without model mode because the ideal data type specific compression parameters differ in the two modes.

```
/**
 * @brief returns the maximum compressed size in a worst-case scenario
 * In case the input data is not compressible. This function is primarily useful for
    memory allocation purposes (destination buffer size).
 * @note if the number of collections is known you can use the COMPRESS_CHUNK_BOUND macro
     for compilation-time evaluation (stack memory allocation for example)
 *
 * @param chunk        pointer to the chunk you want to compress
 * @param chunk_size   size of the chunk in bytes
 *
 * @returns maximum compressed size for a chunk compression on success or an error code
    if it fails (which can be tested with cmp_is_error())
 */

uint32_t compress_chunk_cmp_size_bound(const void *chunk, size_t chunk_size);


/**
 * @brief compress a data chunk consisting of put together data collections
 *
 * @param chunk        pointer to the chunk to be compressed
 * @param chunk_size   byte size of the chunk
 * @param chunk_model  pointer to a model of a chunk; has the same size as the chunk (can
    be NULL if no model compression mode is used)
 * @param updated_chunk_model pointer to store the updated model for the next model mode
```

```
    compression; has the same size as the chunk (can be the same as the model_of_data
    buffer for in-place update or NULL if updated model is not needed)
 * @param dst          destination pointer to the compressed data buffer; has to be 4-byte
    aligned; can be NULL to only get the compressed data size
 * @param dst_capacity  capacity of the dst buffer; it's recommended to provide a
    dst_capacity >= compress_chunk_cmp_size_bound(chunk, chunk_size) as it eliminates one
    potential failure scenario: not enough space in the dst buffer to write the
    compressed data; size is internally rounded down to a multiple of 4
 * @param cmp_par      pointer to a compression parameters struct
 *
 * @returns the byte size of the compressed data or an error code if it fails (which can
    be tested with cmp_is_error())
 */

uint32_t compress_chunk(const void *chunk, uint32_t chunk_size,
                        const void *chunk_model, void *updated_chunk_model,
                        uint32_t *dst, uint32_t dst_capacity,
                        const struct cmp_par *cmp_par);
```

Listing 5.2: Function prototype of the ICU compression size bound and chunk software compression function.

5.3 Error Handling

The return value of the compression function (`ret = compress_chunk()`) has to be tested with the `cmp_is_error(ret)` function. If this check returns false, the compression was successful and the return value of the `compress_chunk()` function is the byte size of the compressed data. If the `cmp_is_error(ret)` check returns true, the compression has failed. The compression data and the updated model are invalid.

A detailed error code can be created with the `cmp_get_error_code(ret)` function. The meaning of the return code is described in the next section.

Note also the return value of the `compress_chunk_cmp_size_bound()` and `compress_chunk_set_model_id_and_counter()` have to be handled in the same way.

```
/**
 * @brief tells if a result is an error code
 *
 * @param code  return value to check
 *
 * @returns non-zero if the code is an error
 */

unsigned int cmp_is_error(uint32_t code);


/**
 * @brief convert a function result into a cmp_error enum
 *
 * @param code  compression return value to get the error code
 *
 * @returns error code
 */

```

```
20 enum cmp_error cmp_get_error_code(uint32_t code);
```

Listing 5.3: Function prototype of the chunk software compression error handling function.

5.3.1 Chunk Compression Error codes

Table 5.1 lists the possible error codes returned by the `cmp_get_error_code()` function.

Table 5.1: Compression error codes.

Error Code Name	**Error Value**	**Description**
CMP_ERROR_NO_ERROR	0	No error detected
CMP_ERROR_GENERIC	1	Error (generic)
CMP_ERROR_SMALL_BUFFER	2	Destination buffer is too small to hold the whole compressed data
CMP_ERROR_DATA_VALUE_TOO_LARGE	3	Data value is larger than expected
CMP_ERROR_PAR_GENERIC	20	Compression mode or model value or lossy rounding parameter is unsupported
CMP_ERROR_PAR_SPECIFIC	21	Specific compression parameters or combination is unsupported
CMP_ERROR_PAR_BUFFERS	22	Buffer related parameter is not valid
CMP_ERROR_PAR_NULL	24	Pointer to the compression parameters structure is NULL
CMP_ERROR_PAR_NO_MODEL	25	Model needed for model mode compression
CMP_ERROR_CHUNK_NULL	40	Pointer to the chunk is NULL. No data, no compression
CMP_ERROR_CHUNK_TOO_LARGE	41	Chunk size too large
CMP_ERROR_CHUNK_TOO_SMALL	42	Chunk size too small. Minimum size is the size of a collection header
CMP_ERROR_CHUNK_SIZE_INCONSISTENT	43	Chunk size is not consistent with the sum of the sizes in the compression headers. Chunk size may be wrong?
CMP_ERROR_CHUNK_SUBSERVICE_INCONSISTENT	44	The chunk contains collections with an incompatible combination of subservices

Table 5.1: Compression error codes.

Error Code Name	Error Value	Description
CMP_ERROR_COL_SUBSERVICE_UNSUPPORTED	60	Unsupported collection subservice
CMP_ERROR_COL_SIZE_INCONSISTENT	61	Inconsistency detected between the collection subservice and data length
CMP_ERROR_ENTITY_NULL	80	Compression entity pointer is NULL
CMP_ERROR_ENTITY_TOO_SMALL	81	Compression entity size is too small
CMP_ERROR_ENTITY_HEADER	82	An error occurred while generating the compression entity header
CMP_ERROR_ENTITY_TIMESTAMP	83	Timestamp too large for the compression entity header
CMP_ERROR_INT_DECODER	100	Internal decoder error occurred
CMP_ERROR_INT_DATA_TYPE_UNSUPPORTED	101	Internal error: Data type not supported
CMP_ERROR_INT_CMP_COL_TOO_LARGE	102	Internal error: compressed collection too large

5.4 Model Management

The compression function does not track how frequently the model mode was used with the respective data set. Consequently, after compression, the prefixed compression entity header lacks the correct model ID and model counter field values. The meaning of these values is explained in detail in Section 7.2.2.

These values can be set in the header using the `compress_chunk_set_model_id_and_counter()` function, which is shown in Listing 5.4. The other input parameters for the function, besides the model ID (`model_id`) and counter (`model_counter`), is a pointer to the destination buffer (`dst`) pointing to the compressed data from the chunk compression as well as the size (`dst_size`) of the compressed data.

Handling the return value is done in the same manner as for the `compress_chunk()` function. After that, the compression process is finished and the compressed data are ready for the decompression.

```
1 /**
2  * @brief set the model id and model counter in the compression entity header
3  *
4  * @param dst pointer to the compressed data (starting with a compression entity header)
```

```
5  * @param dst_size  byte size of the dst buffer @param model_id model identifier; for
       identifying entities that originate from the same starting model
6  * @param model_counter model_counter; counts how many times the model was updated; for
       non model mode compression use 0
7  *
8  * @returns the byte size of the dst buffer (= dst_size) on success or an error code if
       it fails (which can be tested with cmp_is_error())
9  */
10
11 uint32_t compress_chunk_set_model_id_and_counter(void *dst, uint32_t dst_size,
12                                     uint16_t model_id, uint8_t model_counter);
```

Listing 5.4: Function prototype of the model id and counter setter function.

5.5 Debugging

To assist with development and troubleshooting, the compression system supports several debugging levels:

- DEBUGLEVEL=3: Provides the most detailed output, ideal for comprehensive investigation.
- DEBUGLEVEL=1: Offers essential debug prints for general troubleshooting.
- DEBUGLEVEL=0: Disables all debug output, suitable for production environments.

5.6 Building the Chunk Compression Library

For the flight configuration, the chunk compression library should be built as follows:

```
    make lib-release LIB_RDCU_COMPRESSION=0 LIB_DECOMPRESSION=0 CC=<ICU ASW compiler>
MOREFLAGS="-mcpu=v8 -DICU_ASW" DEBUGLEVEL=0
```

- `make lib-release`: Compiles the compression library in a release version with optimizations enabled (`-O2`) and without additional warning flags.
- `LIB_RDCU_COMPRESSION=0`: Excludes the RDCU hardware compression control functionality from the library.
- `LIB_DECOMPRESSION=0`: Excludes the decompression functionality from the library.
- `MOREFLAGS="-mcpu=v8 -DICU_ASW"`: Adds additional flags to the compilation process:
  - `-mcpu=v8`: Enabling the division and multiplication SPARC v8 instructions. Depending on the compiler, the syntax may vary slightly.

  - `-DICU_ASW`: Defines the preprocessor macro `ICU_ASW` to compile the compression library in ICU configuration.

- `DEBUGLEVEL=0`: Sets the debug level to 0, meaning no debug output will be included.

The built library can be found in `lib/libcmp.a`.

5.7 Differences between RDCU and SW Compression

The RDCU compression treats input data as an array of 16-bit values. It has no awareness of the data structure and compresses collection headers and data uniformly. This RDCU code word generation is configured using a specific compression parameter and a configurable spillover parameter.

In contrast, the SW compression understands the data structure (consisting of a chunk of collections). When compressing an (imagette) chunk, it does not compress the collection headers, only the collection data, see Section 6.2. The SW compression method automatically selects the spillover parameter based on the compression parameter. This spillover is slightly higher as it allows it to be configured on the RDCU.

From a technical perspective, it is not possible to generate exactly identical compressed data with the RDCU and SW compression. While minor differences exist, the same compression parameter generally works well across both methods, since the core algorithms are the same.

6. 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 (CE). The compression entity header consists of two parts:

- **generic compression entity header** containing all parameters that are needed for any data. This header is used for all compressed and uncompressed data.
- **specific compression entity header** containing parameters that are specific for the compressed data. This header is different for RDCU and Software (SW) compression.

The structure of a compression entity can be seen in Figure 6.1. A detailed description of the header parameters can be found in Table 6.1. 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. In contrast, the compressed data from the SW compression have already the compression header prefixed.

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 entity described in this document.

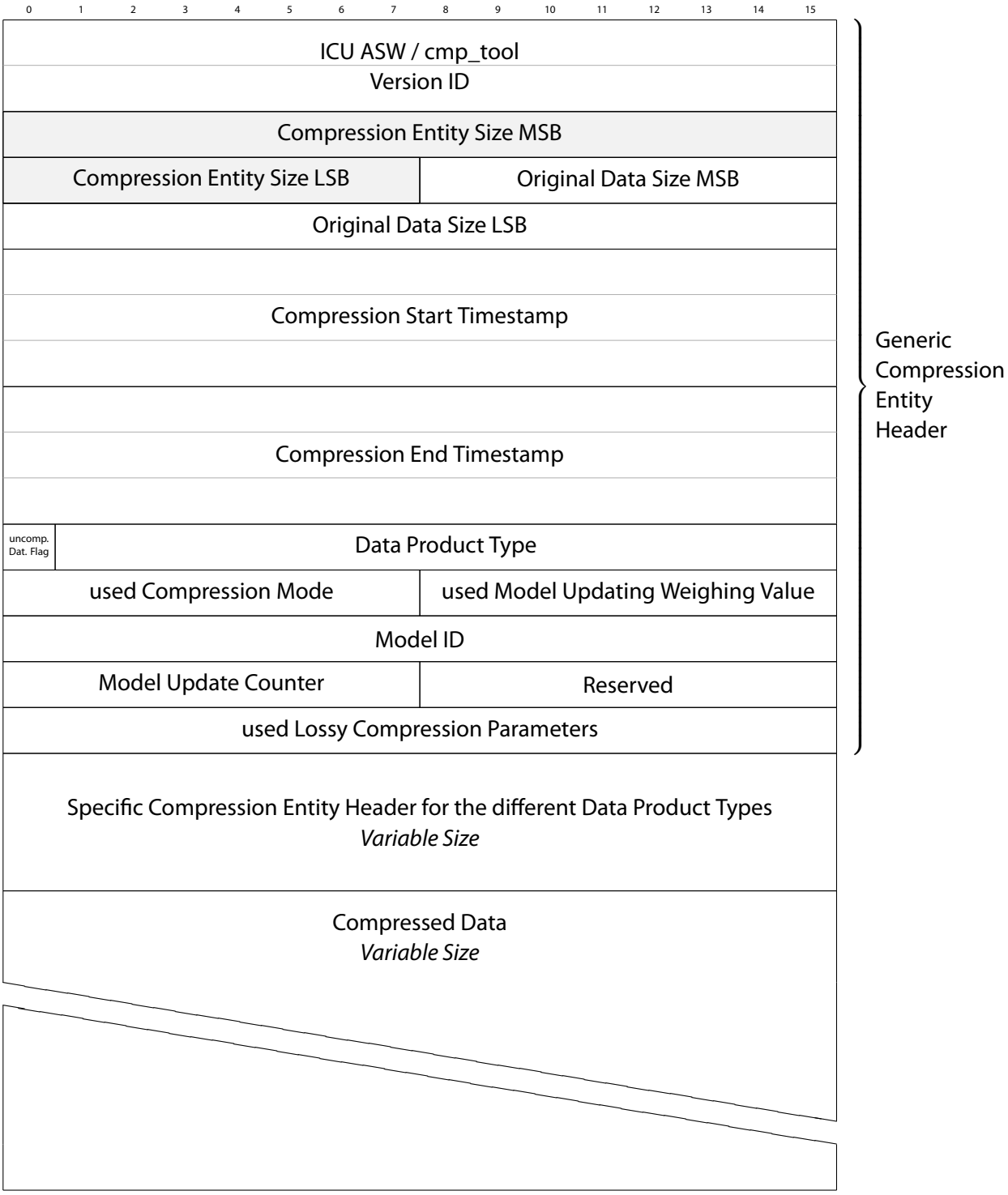


Figure 6.1: Structure of a compression entity consisting of a generic header, a data product type specific header and the compressed data.

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	specify specific CE header, see Table 6.2. The MSB in the data product type is set for uncompressed data.	1, 2, 3, 4, 20, 21, 24
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	Reserved	Unused, expected to be 0	0
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. 6.2, 6.3, 6.4
var.	Compressed Data	Compressed data	custom

Table 6.1: Compression entry header parameters description.

6.1 Specific Compression Entity Header

For uncompressed data (raw mode) indicated by the uncompressed data bit, no specific compression entity header is used. The uncompressed data are only prefixed with a generic compression entity header.

There are two specific compression entity headers for RDCU compressed imagette data

defined. The one shown in Figure 6.2 includes additional to the imagette specific decompression parameters and 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 Figure 6.3 can be used for compressed imagette data.

For software compressed data, the specific compression entity header shown in Figure 6.4 is used.

The data type field in the generic header specifies which specific compression header is used (when the uncompressed data flag is unset) as given in Table 6.2.

Data Product Type Value	Specific CE Header	Description
2 4 21	RDCU compressed imagette data with adaptive parameters using the header defined in Figure 6.2	N-CAM imagette data Saturated imagette data F-CAM imagette data
1 3 20	RDCU compressed imagette data without adaptive parameters using the header defined in Figure 6.3	N-CAM imagette data Saturated imagette data F-CAM imagette data
24	Software compressed chunk data using the header defined in Figure 6.4	Any chunk data

Table 6.2: Data Product Type to specific header mapping, if the uncompressed data bit is unset. The arbitrary numbering has historical reasons.

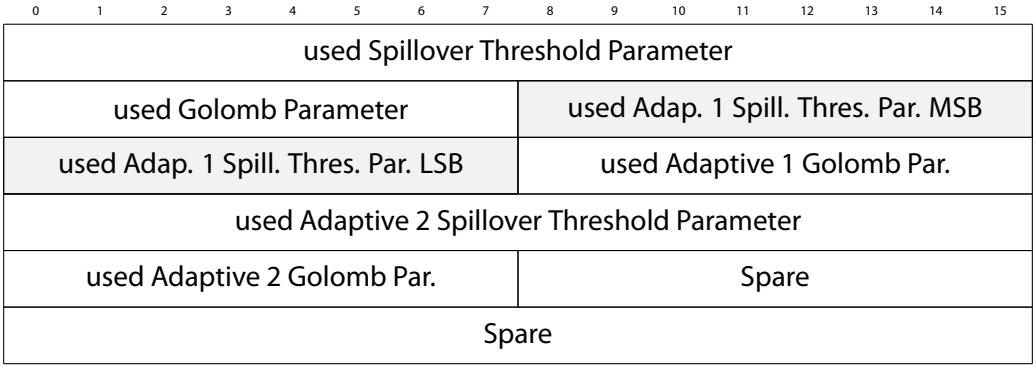


Figure 6.2: Specific compression entity header for RDCU imagette compression containing the semi-adaptive compression feature.

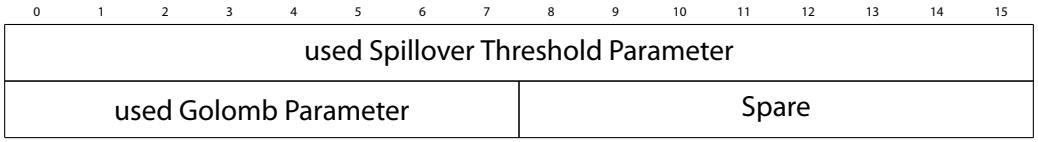


Figure 6.3: Specific compression entity header for RDCU imagette compression without containing the semi-adaptive compression feature.

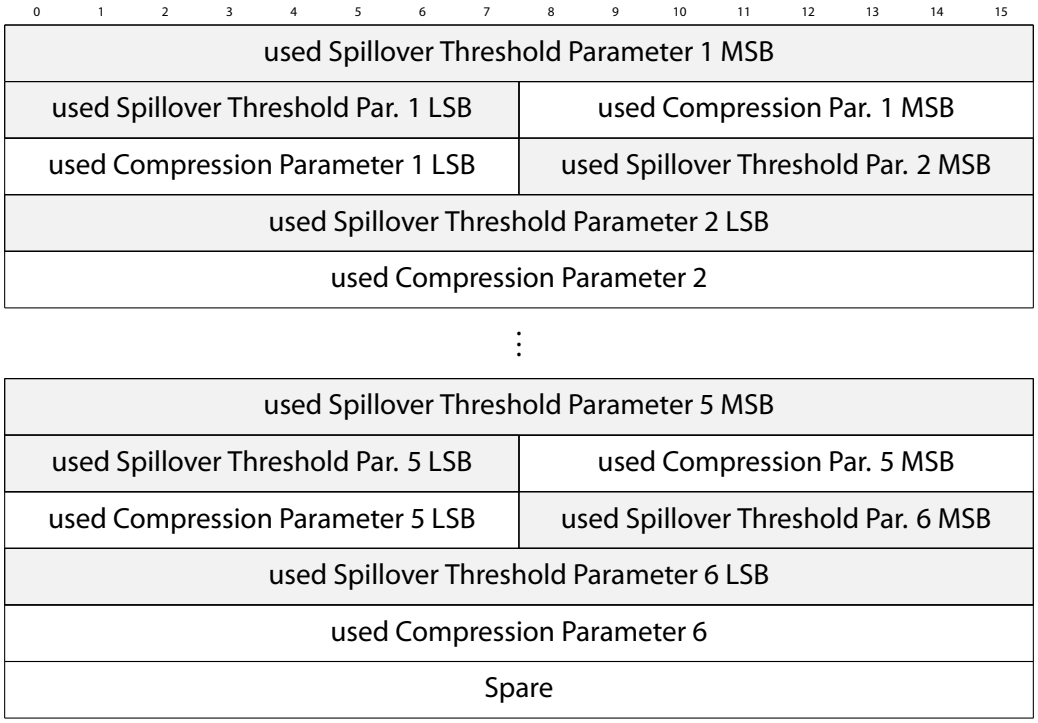


Figure 6.4: Specific compression entity header for software chunk data compression.

6.2 Internal Software Compressed Data Structure

The software compressed data are structured as shown in Figure 6.5. They start with a generic CE header where the uncompressed data bit is 0, and the data product type is 24. This is followed by the software compression specific CE header, as defined in Figure 6.4, containing the additional parameters needed for decompression.

Next comes the compressed data. The internal structure of the software compressed data is as follows: For every compressed collection of the chunk, there is a 2-byte field containing the size of the compressed collection data (without the collection header size), the uncompressed header of the collection, and finally, the compressed collection data.

If the compressed collection size of a collection is equal to its collection length, the data compression was not able to compress the collection data with the given configuration, and this collection data are placed uncompressed after its header.

The RDCU data do not have such an internal structure as the software compression.

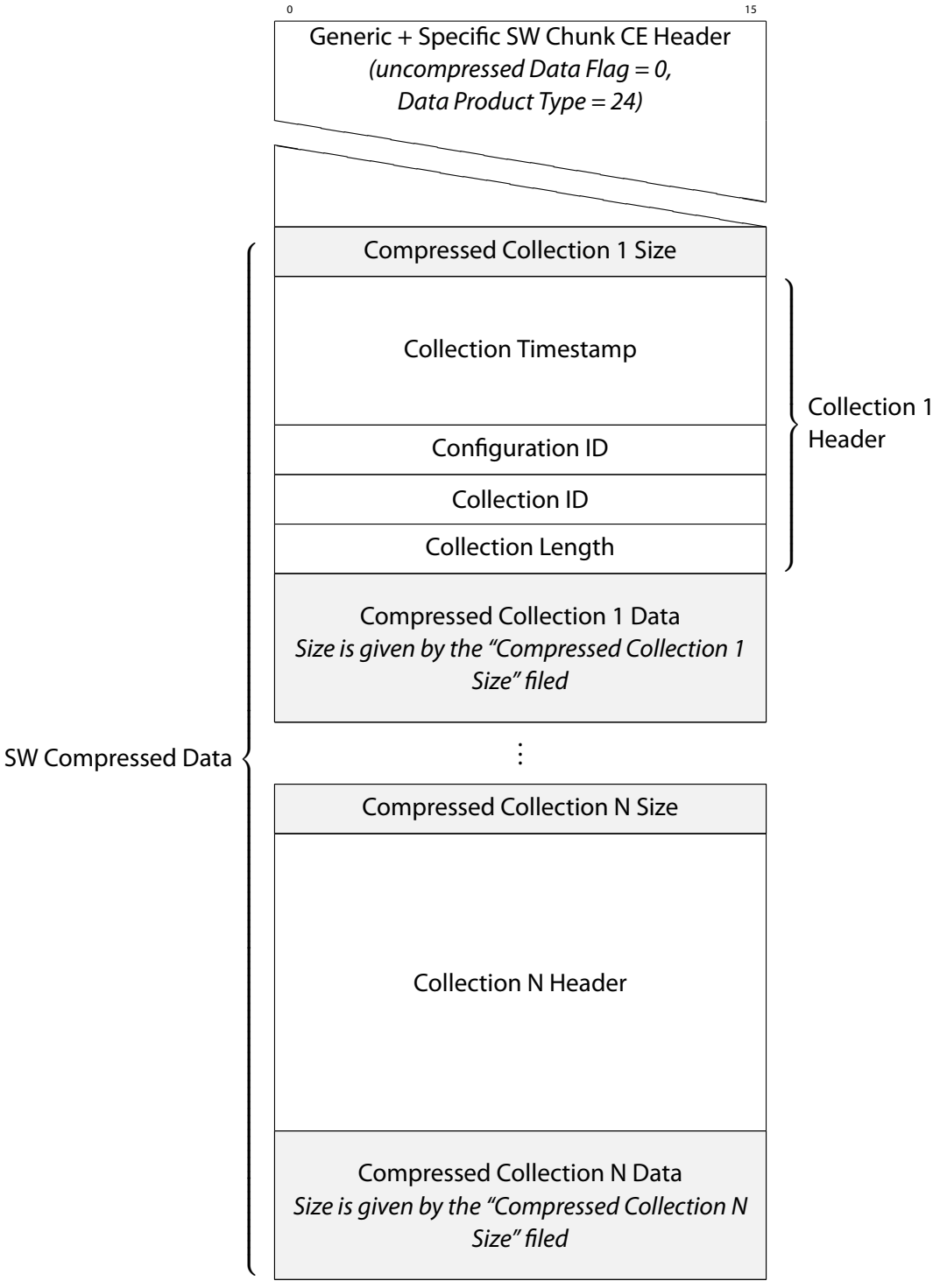


Figure 6.5: Data structure of software compressed data.

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 CE header is added to the compressed data. This CE header contains the necessary information to decompress the data again. The header is described in Chapter 6.

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 letting 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.

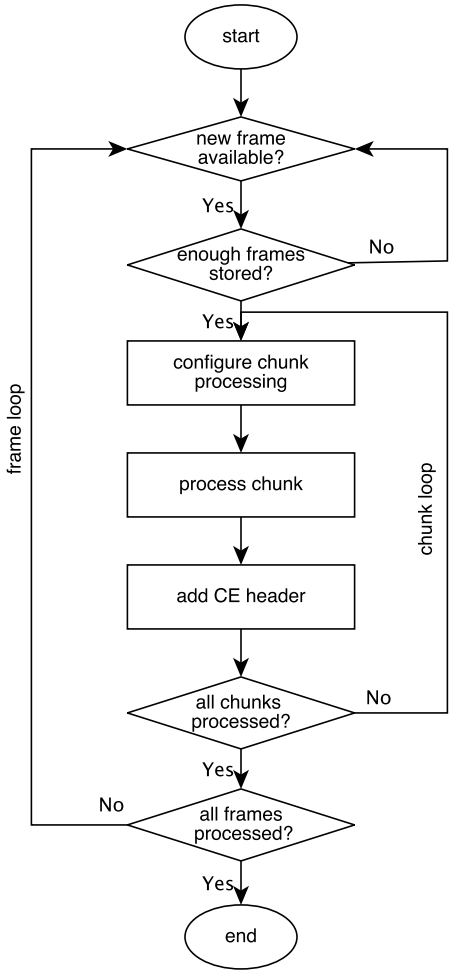


Figure 7.1: Frame processing workflow.

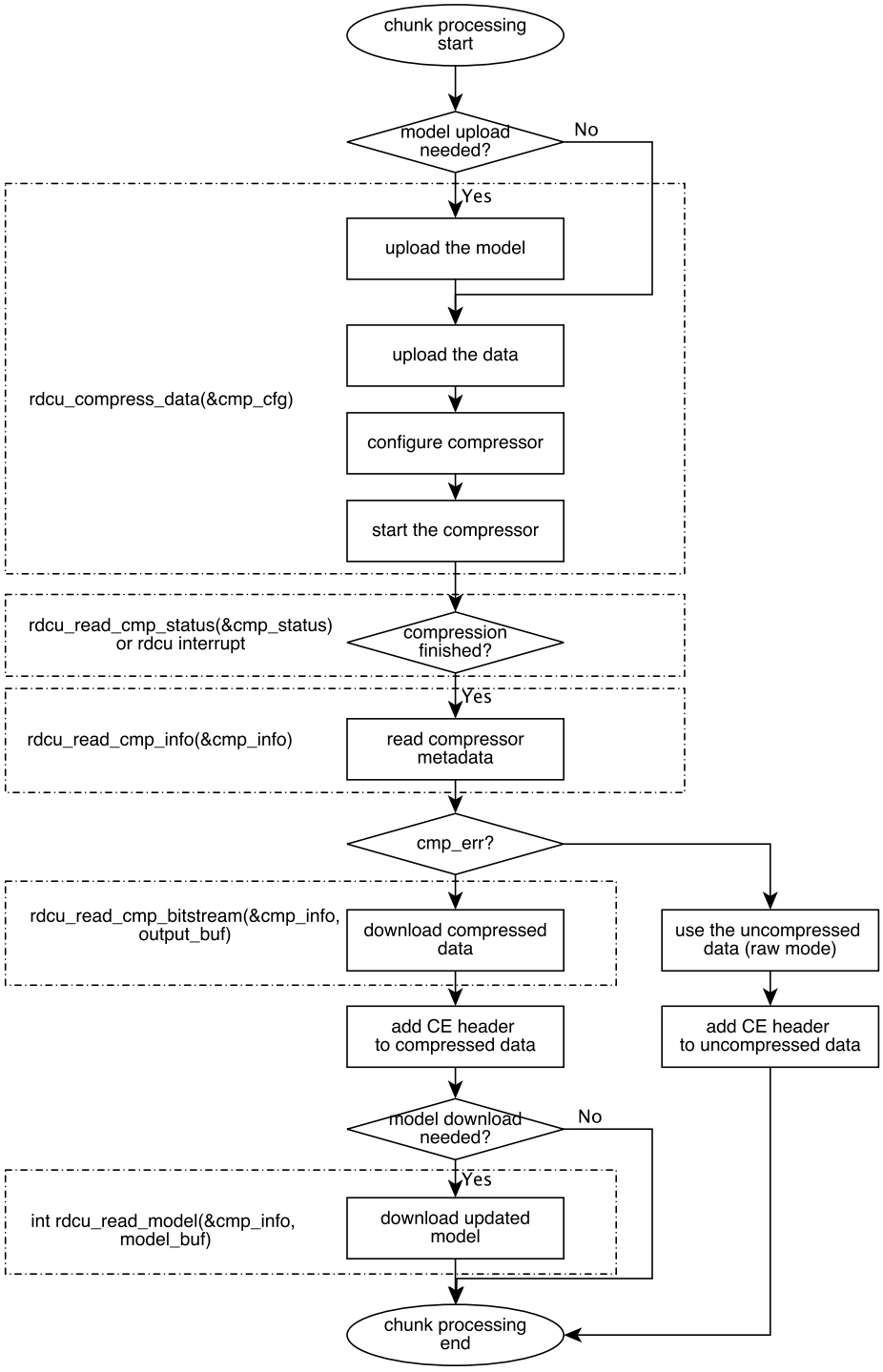


Figure 7.2: Chunk processing workflow.

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:

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

$$\mathrm{output}t_i = \mathrm{input}_i - \mathrm{input}_{i-1} \qquad i = 1, \ldots, n \tag{7.2}$$

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:

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

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:

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

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

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

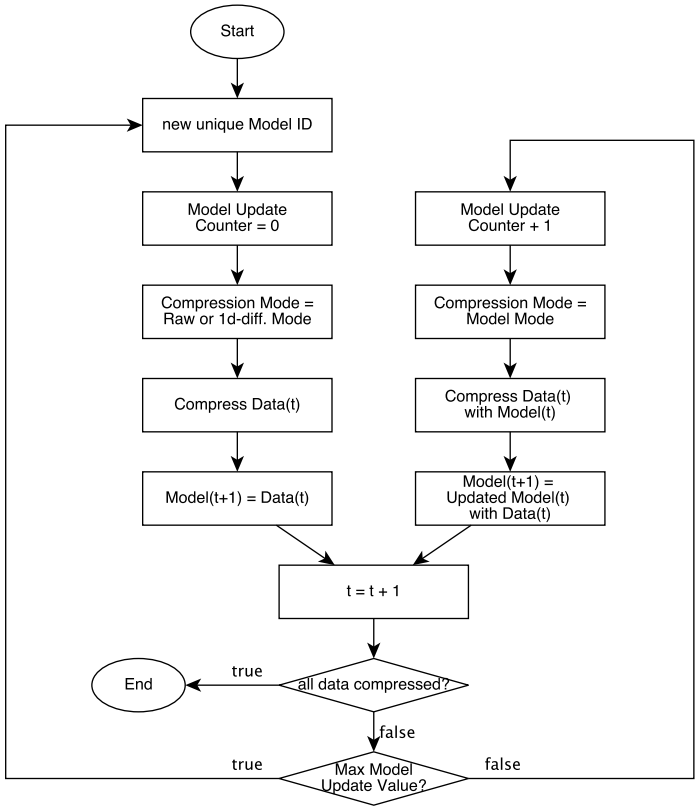


Figure 7.3: Flowchart of the 1d-differencing and model mode.

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 Chapter 6) 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 frames of imagettes. We call these frames N and N+1. First, the imagettes are divided into chunks. Then a chunk from the frame 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 frame 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 frame. Then the process starts from the beginning and the next chunks of the N+2 and N+3 frame 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 CE header. So it is up to the ICU team to decide oon larger chunks and a few compressions per frame or small chunks and more compressions.

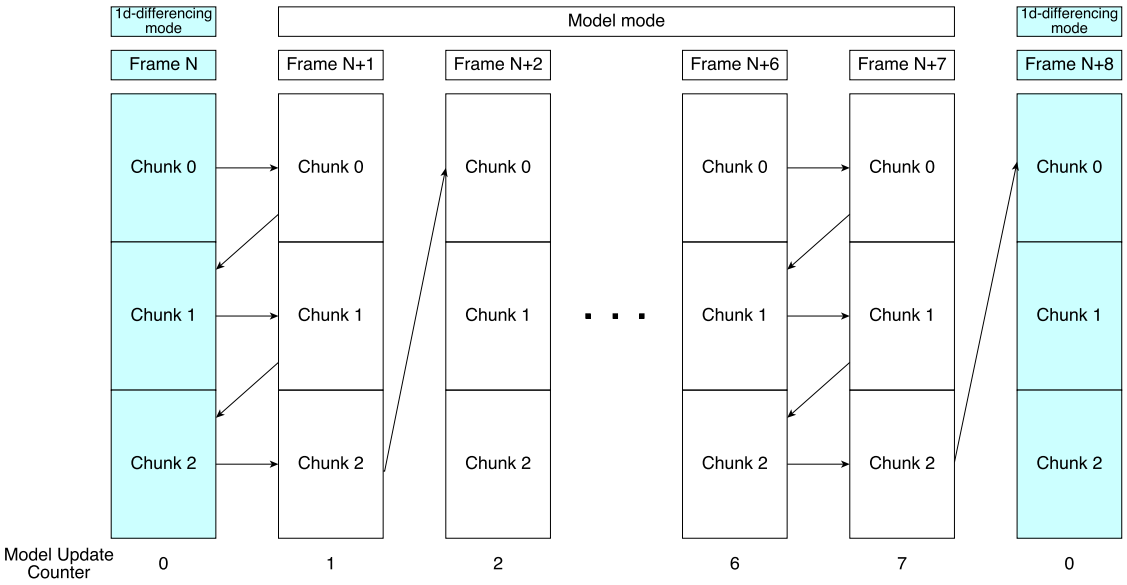


Figure 7.4: Visualization of the optimised chunk processing order.