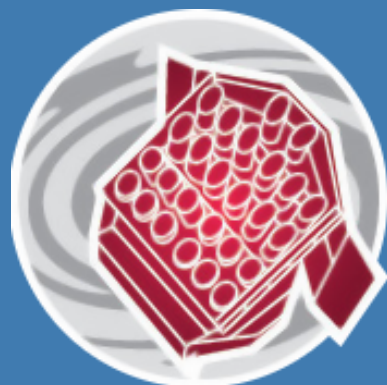


# PLATO

## Data Compression User Manual



# Data Compression User Manual

**Reference:** PLATO-UVIE-PL-UM-0001

**Version:** Issue 1.0, 1. July 2022

**Prepared by:** Dominik Loidolt<sup>1</sup>

**Checked by:** Roland Ottensamer<sup>1</sup>

**Approved by:** Franz Kerschbaum<sup>1</sup>

<sup>1</sup> Department of Astrophysics, University of Vienna

Copyright ©2022

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Front-Cover, no Logos of the University of Vienna.

# Contents

<b>1</b>	<b>Terms, Definitions and Abbreviated Items</b>	<b>6</b>
1.1	Acronyms . . . . .	6
<b>2</b>	<b>Introduction</b>	<b>7</b>
2.1	Purpose of the Document . . . . .	7
2.2	The Data Compression Algorithm . . . . .	7
2.3	The RDCU Hardware Compressor . . . . .	9
2.4	The ICU Software Compressor . . . . .	9
<b>3</b>	<b>Hardware &amp; Software Compression Parameters</b>	<b>11</b>
3.1	Generic Compression Parameters . . . . .	11
3.1.1	Compression Data Product Type (data_type) . . . . .	11
3.1.2	Compression Mode (cmp_mode) . . . . .	11
3.1.3	Model Weighting Parameter (model_value) . . . . .	14
3.1.4	Lossy Rounding Parameter (lossy_par or round) . . . . .	14
3.2	Data Buffers Parameters . . . . .	14
3.2.1	Data to be Compressed Buffer (data_to_compress) . . . . .	14
3.2.2	Data Samples (data_samples) . . . . .	14
3.2.3	Model of Data Buffer (model_of_data) . . . . .	15
3.2.4	Updated/New Model Buffer (updated_model) . . . . .	15
3.2.5	Compressed Data Buffer (compressed_data) . . . . .	15
3.2.6	Compressed Data Buffer Length (compressed_data_len_samples) . . . . .	15
3.2.7	RDCU Addresses (rdcu_data_adr, rdcu_model_adr, rdcu_new_model_adr, rdcu_buffer_adr) . . . . .	16
3.3	RDCU Imagette Specific Compression Parameters . . . . .	16
3.3.1	Golomb Parameter (golomb_par) . . . . .	16
3.3.2	Spillover Threshold Parameter (spillover_par) . . . . .	16
3.3.3	Adaptive Golomb Parameter, Adaptive Spillover Threshold . . . . .	17
3.4	Software Specific Compression Parameters . . . . .	17
3.4.1	Data Type Specific Compression (cmp_par_*) and Spillover Threshold (spillover_*) Parameters . . . . .	17

3.5	Compression Parameter Errors . . . . .	20
3.5.1	Spill Golomb Error . . . . .	21
<b>4</b>	<b>Hardware/RDCU Compression</b>	<b>23</b>
4.1	Configure and Start the Hardware/RDCU Compressor . . . . .	24
4.2	Reading the RDCU Status Register . . . . .	25
4.2.1	HW Compression Status Structure . . . . .	26
4.2.2	RDCU Status Register Read Function . . . . .	27
4.3	RDCU Compression Information Register Read Function . . . . .	28
4.3.1	Compression Information Structure . . . . .	28
4.3.2	Compressor errors (cmp_err) . . . . .	29
4.3.3	Read out the RDCU Hardware Information Registers . . . . .	30
4.4	RDCU SRAM Read Function . . . . .	31
<b>5</b>	<b>Software Compression</b>	<b>32</b>
5.1	Maximum Used Bits . . . . .	32
5.2	How to compress data with the Software Compressor on the ICU . . . . .	33
<b>6</b>	<b>Compression Entity Format</b>	<b>34</b>
6.1	Specific Compression Entity Header . . . . .	34
<b>7</b>	<b>Frame Processing</b>	<b>39</b>
7.1	Chunk Processing . . . . .	39
7.2	1D-Differencing Mode and Model Mode . . . . .	42
7.2.1	1D-Differencing Mode . . . . .	42
7.2.2	Model Mode . . . . .	42
7.3	Chunk Procedure Order . . . . .	43
7.3.1	Optimised Chunk Processing . . . . .	44
7.3.2	Chunk Size . . . . .	44
<b>A</b>	<b>Hardware Compression Example</b>	<b>46</b>
<b>B</b>	<b>Software Compression Example</b>	<b>50</b>

# 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, configuration function), add max bit used section, add max bits used version in compression entity

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 ICU RDCU User Manual, PLATO-IWF-PL-UM-0076 Issue 1.2, 7. October 2021

ID	Title, Reference Number, Revision Number
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

<b>API</b>	Application Programming Interface
<b>CE</b>	Compression Entity 34, 39, 44
<b>CPU</b>	Central Processing Unit
<b>FPGA</b>	Field-Programmable Gate Array 7, 9
<b>HW</b>	Hardware 7, 15–17, 23–25, 27–29, 31, 44
<b>ICU</b>	Instrument Control Unit 7, 9, 20, 23, 33, 39, 44
<b>ISR</b>	Interrupt Service Routine
<b>MMU</b>	Memory Management Unit
<b>PUS</b>	Packet Utilisation Standard
<b>RDCU</b>	Router and Data Compression Unit 7, 9–11, 15, 16, 21, 23–27, 30, 31, 34, 39, 43, 44, 46
<b>RISC</b>	Reduced Instruction Set Computing
<b>RMAP</b>	Remote Memory Access Protocol 9, 11, 23–25
<b>SRAM</b>	Static Random Access Memory 9, 15, 16, 21, 23, 24, 30, 31, 39, 44
<b>SW</b>	Software 7, 15
<b>TBC</b>	To Be Confirmed 12, 19
<b>UVIE</b>	University of Vienna 7, 9, 23

## 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 a [Field-Programmable Gate Array \(FPGA\)](#) on the [Router and Data Compression Unit \(RDCU\)](#) 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 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. This mode calculates the difference to the left neighbouring 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



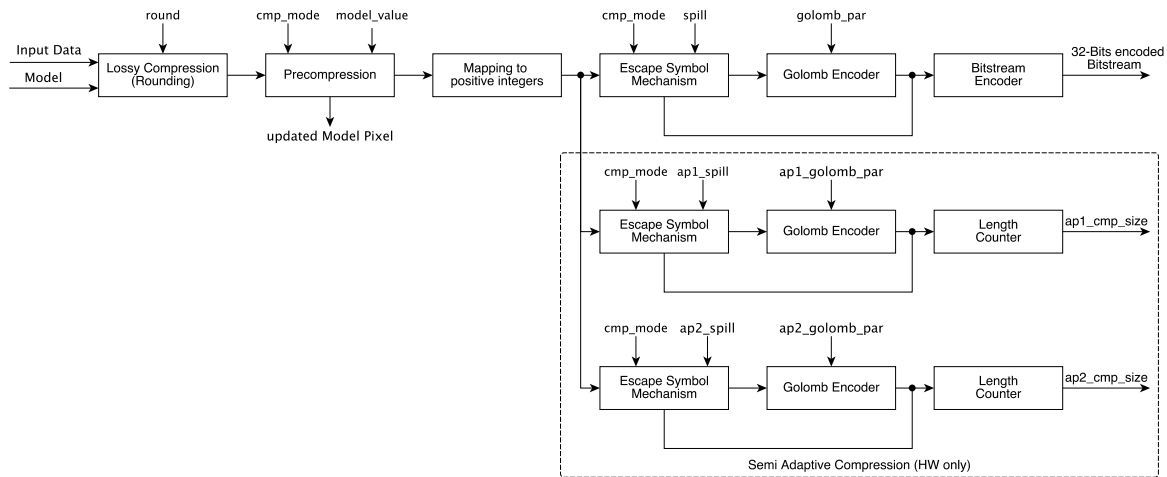


Figure 2.1: Visualization of the compression algorithm.

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 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 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 5.2 discusses the provided function for software compression in detail.

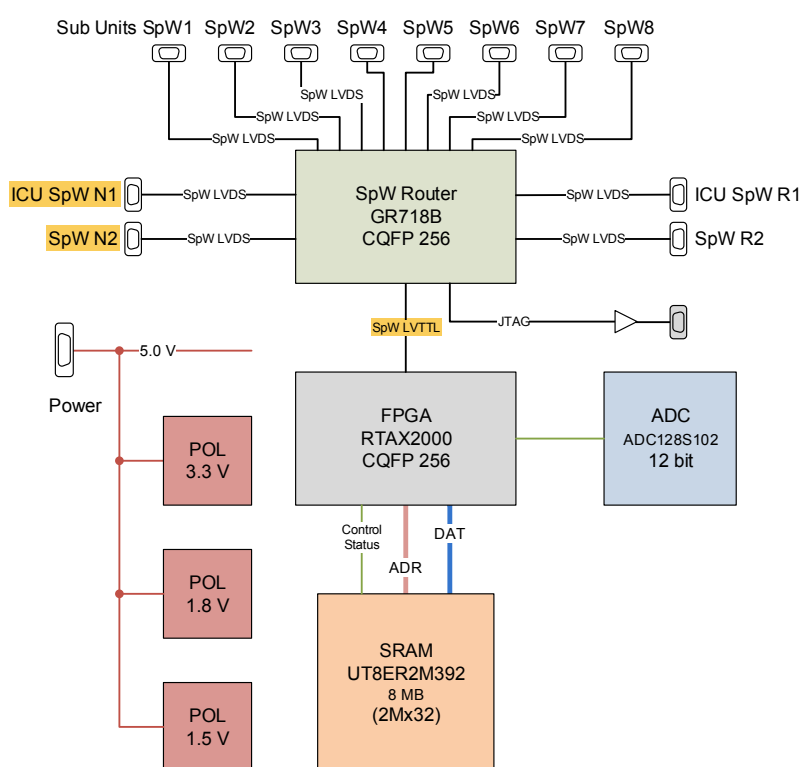


Figure 2.2: RDCU Electrical Concept.

## 3. Hardware & Software Compression Parameters

The compression is controlled by several compression parameters. For software and hardware compression a structure and setter functions are provided to configure all compression parameters. This structure is passed to the software compression function to start a compression.

For hardware compression, this configuration 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 [\[RD-2\]](#).

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 Data Product Type (`data_type`)

The compression data product type specifies which type of scientific data is compressed. The [Table 3.1](#) lists all available compression data product types.

#### 3.1.2 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.

Table 3.1: Defined compression data product types.

Data Product Type	Description/Constant Name	Size of a Sample [Byte]	Specific Compression Header
0	DATA_TYPE_UNKOWN	0	invalid data type
1	DATA_TYPE_IMAGETTE	2	imagette header
2	DATA_TYPE_IMAGETTE_ADAPTIVE	2	adaptive imagette header
3	DATA_TYPE_SAT_IMAGETTE	2	imagette header
4	DATA_TYPE_SAT_IMAGETTE_ADAPTIVE	2	adaptive imagette header
5	DATA_TYPE_OFFSET	8	non-imagette header
6	DATA_TYPE_BACKGROUND	10	non-imagette header
7	DATA_TYPE_SMEARING	8	non-imagette header
8	DATA_TYPE_S_FX	5	non-imagette header
9	DATA_TYPE_S_FX_DFX	9	non-imagette header
10	DATA_TYPE_S_FX_NCOB	13	non-imagette header
11	DATA_TYPE_S_FX_DFX_NCOB_ECOB	25	non-imagette header
12	DATA_TYPE_L_FX	11	non-imagette header
13	DATA_TYPE_L_FX_DFX	15	non-imagette header
14	DATA_TYPE_L_FX_NCOB	27	non-imagette header
15	DATA_TYPE_L_FX_DFX_NCOB_ECOB	39	non-imagette header
16	DATA_TYPE_F_FX	4	non-imagette header
17	DATA_TYPE_F_FX_DFX	8	non-imagette header
18	DATA_TYPE_F_FX_NCOB	12	non-imagette header
19	DATA_TYPE_F_FX_DFX_NCOB_ECOB	24	non-imagette header
20	DATA_TYPE_F_CAM_IMAGETTE	To Be Confirmed (TBC)	imagette header
21	DATA_TYPE_F_CAM_IMAGETTE_ADAPTIVE	TBC	adaptive imagette header
22	DATA_TYPE_F_CAM_OFFSET	TBC	non-imagette header
23	DATA_TYPE_F_CAM_BACKGROUND	0	non-imagette header

### **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].

### 3.1.3 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.

### 3.1.4 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 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.

## 3.2 Data Buffers Parameters

The following section describes the parameters related to the different data buffers.

### 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, including the collection header.

### 3.2.2 Data Samples (`data_samples`)

The data samples parameter describes the length of the data to be compressed. A sample is the size that an entry in a collection has. The size of a collection entry can be seen in Table 3.1.

**NOTE: There is a difference in the `data_samples` parameter when compressing imagette data compared to compressing non-imagette data! When compressing non-imagette data, the compressor expects that the collection header will always prefix the non-imagette**

**data.** Therefore, the `data_samples` parameter is simply the number of entries in the collection. It is not intended to join multiple non-imagette collections and compress them together.

**When compressing imagette data, the length of the entire data to be compressed, including the collection header, is measured in 16-bit samples. The 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 stored in 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. For [SW](#) compression this parameter specifies the length of the `compressed_data` buffer. 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](#). 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 work-around 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 3.5](#) 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 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 3.5.1](#), an ap\_spill parameter can be selected up to a specific value depending on the set ap\_golomb\_par parameter.

## 3.4 Software Specific Compression Parameters

The following section describes the data type specific parameters required for the software compression.

### 3.4.1 Data Type Specific Compression (cmp\_par\_\*) and Spillover Threshold (spillover\_\*) Parameters

The data type specific compression parameters are one or many pairs consisting of a compression parameter with cmp\_par\_ prefix and a spillover threshold parameter with a spillover\_ prefix. These parameters allow adapting the compression to the properties of the different science data. The maximum spillover threshold parameter depends on the chosen compression parameter. The provided function get\_max\_spill() returns the maximum allowed value.

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 [\[RD-6\]](#). Depending on the number of parameters in an entry, a different number of data type specific compression parameter pairs are needed. For the software compression we group the science data types in three

Table 3.2: Specific compression parameters needed for the different flux/COB data types.

Data Type	<i>needed Param.</i>	<i>exp_flags</i>	<i>fx</i>	<i>ncob</i>	<i>efx</i>	<i>ecob</i>	<i>fx_cob_variance</i>
DATA_TYPE_S_FX		x	x	-	-	-	-
DATA_TYPE_S_FX_DFX		x	x	-	x	-	-
DATA_TYPE_S_FX_NCOB		x	x	x	-	-	-
DATA_TYPE_S_FX_DFX_NCOB_ECOB		x	x	x	x	x	-
DATA_TYPE_L_FX		x	x	-	-	-	x
DATA_TYPE_L_FX_DFX		x	x	-	x	-	x
DATA_TYPE_L_FX_NCOB		x	x	x	-	-	x
DATA_TYPE_L_FX_DFX_NCOB_ECOB		x	x	x	x	x	x
DATA_TYPE_F_FX		-	x	-	-	-	-
DATA_TYPE_F_FX_DFX		-	x	-	x	-	-
DATA_TYPE_F_FX_NCOB		-	x	x	-	-	-
DATA_TYPE_F_FX_DFX_NCOB_ECOB		-	x	x	x	x	-

groups: imagette, flux/COB and auxiliary science data types. For every group there is a specific compression parameter configuration function to configure the for the data type needed parameters (`cmp_cfg_icu_imagette()`, `cmp_cfg_fx_cob()`, `cmp_cfg_aux()`, see Listing 3.1). A software configuration is first created with the function `cmp_cfg_icu_create()`.

Not every data type needs the full set of parameter pairs. The table 3.2 shows which parameter pairs are needed to compress the different flux/COB compression data types and Table 3.3 shows the parameter pairs needed for the auxiliary science data types. For imagette data, only one pair of compression and spillover threshold parameters is needed.

```

1 /**
2  * @brief create an ICU compression configuration
3  *
4  * @param data_type compression data product type
5  * @param cmp_mode compression mode
6  * @param model_value model weighting parameter (only needed for model compression mode)
7  * @param lossy_par lossy rounding parameter (use CMP_LOSSLESS for lossless compression)
8  *
9  * @returns a compression configuration containing the chosen parameters;
10 * on error the data_type record is set to DATA_TYPE_UNKOWN
11 */
12
13 struct cmp_cfg cmp_cfg_icu_create(enum cmp_data_type data_type, enum cmp_mode cmp_mode,
14                                uint32_t model_value, uint32_t lossy_par)
15
16
17 /**
18 * @brief setup the different data buffers for an ICU compression
19 *

```

Table 3.3: Specific compression parameters needed for the different auxiliary science data types.

Data Type	<i>needed Param.</i>	<i>mean</i>	<i>variance</i>	<i>pixels_error</i>
DATA_TYPE_OFFSET		x	x	-
DATA_TYPE_BACKGROUND		x	x	x
DATA_TYPE_SMEARING		x	x	x
DATA_TYPE_F_CAM_OFFSET		TBC		
DATA_TYPE_F_CAM_BACKGROUND		TBC		

```

20 * @param cfg      pointer to a compression configuration (created
21 *                with the cmp_cfg_icu_create() function)
22 * @param data_to_compress pointer to the data to be compressed
23 * @param data_samples  length of the data to be compressed measured in
24 *                data samples/entities (collection header not
25 *                included by imagette data)
26 * @param model_of_data pointer to model data buffer (can be NULL if no
27 *                model compression mode is used)
28 * @param updated_model pointer to store the updated model for the next
29 *                model mode compression (can be the same as the model_of_data
30 *                buffer for in-place update or NULL if updated model is not needed)
31 * @param compressed_data pointer to the compressed data buffer (can be NULL)
32 * @param compressed_data_len_samples length of the compressed_data buffer in
33 *                measured in the same units as the data_samples
34 *
35 * @returns the size of the compressed_data buffer on success; 0 if the
36 *                parameters are invalid
37 */
38
39 size_t cmp_cfg_icu_buffers(struct cmp_cfg *cfg, void *data_to_compress,
40                          uint32_t data_samples, void *model_of_data,
41                          void *updated_model, uint32_t *compressed_data,
42                          uint32_t compressed_data_len_samples)
43
44 /**
45 * @brief set up the configuration parameters for an ICU imagette compression
46 *
47 * @param cfg      pointer to a compression configuration (created
48 *                by the cmp_cfg_icu_create() function)
49 * @param cmp_par  imagette compression parameter (Golomb parameter)
50 * @param spillover_par imagette spillover threshold parameter
51 *
52 * @returns 0 if parameters are valid, non-zero if parameters are invalid
53 */
54
55 int cmp_cfg_icu_imagette(struct cmp_cfg *cfg, uint32_t cmp_par,
56                        uint32_t spillover_par)
57
58
59 /**
60 * @brief set up the configuration parameters for a flux/COB compression
61 * @note not all parameters are needed for every flux/COB compression data type

```

```

62 *
63 * @param cfg      pointer to a compression configuration (created
64 *                by the cmp_cfg_icu_create() function)
65 * @param cmp_par_exp_flags exposure flags compression parameter
66 * @param spillover_exp_flags exposure flags spillover threshold parameter
67 * @param cmp_par_fx      normal flux compression parameter
68 * @param spillover_fx      normal flux spillover threshold parameter
69 * @param cmp_par_ncob     normal center of brightness compression parameter
70 * @param spillover_ncob   normal center of brightness spillover threshold parameter
71 * @param cmp_par_efx      extended flux compression parameter
72 * @param spillover_efx    extended flux spillover threshold parameter
73 * @param cmp_par_ecob     extended center of brightness compression parameter
74 * @param spillover_ecob   extended center of brightness spillover threshold parameter
75 * @param cmp_par_fx_cob_variance flux/COB variance compression parameter
76 * @param spillover_fx_cob_variance flux/COB variance spillover threshold parameter
77 *
78 * @returns 0 if parameters are valid, non-zero if parameters are invalid
79 */
80
81 int cmp_cfg_fx_cob(struct cmp_cfg *cfg,
82                  uint32_t cmp_par_exp_flags, uint32_t spillover_exp_flags,
83                  uint32_t cmp_par_fx, uint32_t spillover_fx,
84                  uint32_t cmp_par_ncob, uint32_t spillover_ncob,
85                  uint32_t cmp_par_efx, uint32_t spillover_efx,
86                  uint32_t cmp_par_ecob, uint32_t spillover_ecob,
87                  uint32_t cmp_par_fx_cob_variance, uint32_t spillover_fx_cob_variance)
88
89
90 /**
91 * @brief set up the configuration parameters for an auxiliary science data compression
92 * @note auxiliary compression data types are: DATA_TYPE_OFFSET, DATA_TYPE_BACKGROUND,
93 * DATA_TYPE_SMEARING, DATA_TYPE_F_CAM_OFFSET, DATA_TYPE_F_CAM_BACKGROUND
94 * @note not all parameters are needed for the every auxiliary compression data type
95 *
96 * @param cfg      pointer to a compression configuration (
97 *                created with the cmp_cfg_icu_create() function)
98 * @param cmp_par_mean     mean compression parameter
99 * @param spillover_mean   mean spillover threshold parameter
100 * @param cmp_par_variance variance compression parameter
101 * @param spillover_variance variance spillover threshold parameter
102 * @param cmp_par_pixels_error outlier pixels number compression parameter
103 * @param spillover_pixels_error outlier pixels number spillover threshold parameter
104 *
105 * @returns 0 if parameters are valid, non-zero if parameters are invalid
106 */
107
108 int cmp_cfg_aux(struct cmp_cfg *cfg,
109                uint32_t cmp_par_mean, uint32_t spillover_mean,
110                uint32_t cmp_par_variance, uint32_t spillover_variance,
111                uint32_t cmp_par_pixels_error, uint32_t spillover_pixels_error)

```

Listing 3.1: Declaration of the ICU software configuration compression function.

### 3.5 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 com-

pression 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 3.4 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 3.4: 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 3.5.1]
Adaptive Golomb Parameter 1/2	ap1/2_golomb_par	[1,63]
Adaptive Spillover Threshold 1/2	ap1/2_spill	[2, see Section 3.5.1]
RDCU SRAM Addresses	rdcu_***_adr	[0x000000, 0x7FFFFFFF]

### 3.5.1 Spill Golomb 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 3.5 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 3.5 are the same for ap1\_spill and ap2\_spill parameters.

Table 3.5: Valid spillover threshold parameter (spill) range in relation to the used Golomb parameter (golomb\_par).

golomb_par	spill $\leq$	golomb_par	spill $\leq$	golomb_par	spill $\leq$	golomb_par	spill $\leq$
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		

## 4. Hardware/RDCU Compression

The [UVIE](#) team provides for the [ICU](#) a set of software to simplify the set up of an [RDCU](#) hardware compression. The [HW](#) compressor can only compress imagerie 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 set into 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. 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 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. Appendix [A](#) shows a detailed example of the whole hardware compression process.

In the end, the compressed data is 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, see Section 3.1.

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 `texttrdcu_rmap_init()`.

```

1 /**
2  * @brief create an RDCU compression configuration
3  *
4  * @param data_type compression data product type
5  * @param cmp_mode compression mode
6  * @param model_value model weighting parameter (only needed for model compression mode)
7  * @param lossy_par lossy rounding parameter (use CMP_LOSSLESS for lossless compression)
8  *
9  * @returns a compression configuration containing the chosen parameters;
10 * on error the data_type record is set to DATA_TYPE_UNKOWN
11 */
12
13 struct cmp_cfg rdcu_cfg_create(enum cmp_data_type data_type, enum cmp_mode cmp_mode,
14                               uint32_t model_value, uint32_t lossy_par)
15
16
17 /**
18 * @brief setup of the different data buffers for an RDCU compression
19 *
20 * @param cfg pointer to a compression configuration (created
21 * with the rdcu_cfg_create() function)
22 * @param data_to_compress pointer to the data to be compressed (if NULL no
23 * data transfer to the RDCU)
24 * @param data_samples length of the data to be compressed measured in
25 * 16-bit data samples (ignoring the collection header)
26 * @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.

```

1 /**
2  * @brief The cmp_status structure can contain the information of the
3  *        compressor status register from the RDCU, see RDCU-FRS-FN-0632,
4  *        but can also be used for the SW compression.
5  */
6
7 struct cmp_status {
8     uint8_t cmp_ready; /* Data Compressor Ready; 0: Compressor is busy 1: Compressor is
9                          ready */
10    uint8_t cmp_active; /* Data Compressor Active; 0: Compressor is on hold; 1: Compressor
11                          is active */
12    uint8_t data_valid; /* Compressor Data Valid; 0: Data is invalid; 1: Data is valid */
13    uint8_t cmp_interrupted; /* Data Compressor Interrupted; HW only; 0: No compressor
14                              interruption; 1: Compressor was interrupted */
15    uint8_t rdcu_interrupt_en; /* RDCU Interrupt Enable; HW only; 0: Interrupt is disabled;
16                                1: Interrupt is enabled */
17 };

```

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_interrupt_signal()` respectively `rdcu_disable_interrupt_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 time a [HW](#) 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:

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

1-D Differencing mode:

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

```

1 /**
2  * @brief read out the status register of the RDCU compressor
3  *
4  * @param status  compressor status contains the stats of the HW compressor
5  *
6  * @note access to the status registers is also possible during compression
7  *
8  * @returns 0 on success, error otherwise
9  */
10
11 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 3.5.

```

1 /* The cmp_info structure can contain the information and metadata of an
2  * executed RDCU compression.
3  */
4
5 struct cmp_info {
6     uint32_t cmp_mode_used;           /* Compression mode used */
7     uint32_t spill_used;              /* Spillover threshold used */
8     uint32_t golomb_par_used;         /* Golomb parameter used */
9     uint32_t samples_used;            /* Number of samples (16 bit value) to be stored */
10    uint32_t cmp_size;                 /* Compressed data size; measured in bits */
11    uint32_t ap1_cmp_size;             /* Adaptive compressed data size 1; measured in bits */
12    uint32_t ap2_cmp_size;             /* Adaptive compressed data size 2; measured in bits */
13    uint32_t rdcu_new_model_adr_used; /* Updated model start address used */
14    uint32_t rdcu_cmp_adr_used;        /* Compressed data start address */
15    uint8_t  model_value_used;         /* Model weighting parameter used */
16    uint8_t  round_used;               /* Number of noise bits to be rounded used */
17    uint16_t cmp_err;                  /* Compressor errors
18
19        * [bit 0] small_buffer_err; The length for the compressed data buffer is
20        too small
21
22        * [bit 1] cmp_mode_err; The cmp_mode parameter is not set correctly
23        * [bit 2] model_value_err; The model_value parameter is not set correctly
24        * [bit 3] cmp_par_err; The spill, golomb_par combination is not set
25        correctly
26        * [bit 4] ap1_cmp_par_err; The ap1_spill, ap1_golomb_par combination is
27        not set correctly (only HW compression)
28        * [bit 5] ap2_cmp_par_err; The ap2_spill, ap2_golomb_par combination is
29        not set correctly (only HW compression)
30        * [bit 6] mb_err; Multi bit error detected by the memory controller (only
31        HW compression)
32        * [bit 7] slave_busy_err; The bus master has received the "slave busy"
33        status (only HW compression)
34        * [bit 8] slave_blocked_err; The bus master has received the "slave
35        blocked status (only HW compression)
36        * [bit 9] invalid_address_err; The bus master has received the "invalid
37        address status (only HW compression)
38    */
39 };

```

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:

$$CR = \frac{\text{model\_length\_used} \cdot 16 \text{ Bit}}{\text{cmp\_size}} \quad (4.3)$$

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 3.5.

### 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.

```
1 /**
2  * @brief read out the metadata of an RDCU compression
3  *
4  * @param info  compression information contains the metadata of a compression
5  *
6  * @note the compression information registers cannot be accessed during a compression
7  *
8  * @returns 0 on success, error otherwise
9  */
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

### 5.1 Maximum Used Bits

For most scientific parameters, the entire possible range of values is not used. This means that the maximum value can be represented with fewer bits than it is actually stored. The software compression tries to take advantage of this and therefore has a registry that stores the maximum used bits of each scientific parameter. The problem is that the maximum used bits are based on assumptions about the instrument (maximum value of a parameter, conversion factors from float to integer, etc.) and may change during instrument calibration. To adapt to changes it is possible to read the parameters with the `cmp_get_max_used_bits()` function. This function returns a structure containing the maximum bit parameters for the various scientific parameters. These parameters can be changed. If the registry is changed, the version record of the registry must also be changed to another unique value. The structure is the input for the `cmp_set_max_used_bits()` function which changes the values in the maximum used bits registry.

During software compression, a check is made whether a parameter value is greater than the corresponding maximum used bit value allows. If the value is greater, the software compression function fails and returns `CMP_ERROR_HIGH_VALUE`.

```
1 /**
2  * @brief get the maximum length of the different data product types
3  *
4  * @returns a structure with the used maximum length of the different data
5  * product types in bits
6  */
7
8 struct cmp_max_used_bits cmp_get_max_used_bits(void)
9
10
11 /**
12  * @brief sets the maximum length of the different data product types
13  *
14  * @param set_max_used_bits pointer to a structure with the maximum length
15  * of the different data product types in bits
16  */
17
18 void cmp_set_max_used_bits(const struct cmp_max_used_bits *set_max_used_bits)
```

Listing 5.1: Getter and setter function for the max used bits registry.

## 5.2 How to compress data with the Software Compressor on the ICU

We provide the function `icu_compress_data()` to compress data with the software compressor, see Listing 5.2. The function takes as an input parameter a compression configuration. It contains all parameters that control the compression. The process to create and setting up a configuration is explained in more detail in Chapter 3.

The function takes the data and model (if necessary) from the buffers referenced in the configuration and compresses them. The compressed bitstream is written to the `compressed_data` buffer. The length of the compressed bitstream is the return value of the `icu_compress_data()` function. After each compression, it is necessary to check that the return value is not negative. If it is negative, an error has occurred and the compression data and the updated model are invalid.

Appendix B shows a detailed example of how to set up and run a software compression.

```
1 /**
2  * @brief compress data on the ICU in software
3  *
4  * @param cfg pointer to a compression configuration (created with the
5  *       cmp_cfg_icu_create() function, setup with the cmp_cfg_xxx() functions)
6  *
7  * @note the validity of the cfg structure is checked before the compression is
8  *       started
9  *
10 * @returns the bit length of the bitstream on success; negative on error,
11 *       CMP_ERROR_SAMLL_BUF (-2) if the compressed data buffer is too small to
12 *       hold the whole compressed data, CMP_ERROR_HIGH_VALUE (-3) if a data or
13 *       model value is bigger than the max_used_bits parameter allows (set with
14 *       the cmp_set_max_used_bits() function)
15 */
16
17 int icu_compress_data(const struct cmp_cfg *cfg)
```

Listing 5.2: Declaration of the ICU software compress function.

## 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* **Compression Entity (CE)**. 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 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**.

**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.

### 6.1 Specific Compression Entity Header

There are two specific compression entity headers for 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 non-imagette data, the specific compression entity header shown in Figure 6.4 should be used. Table 3.1 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.

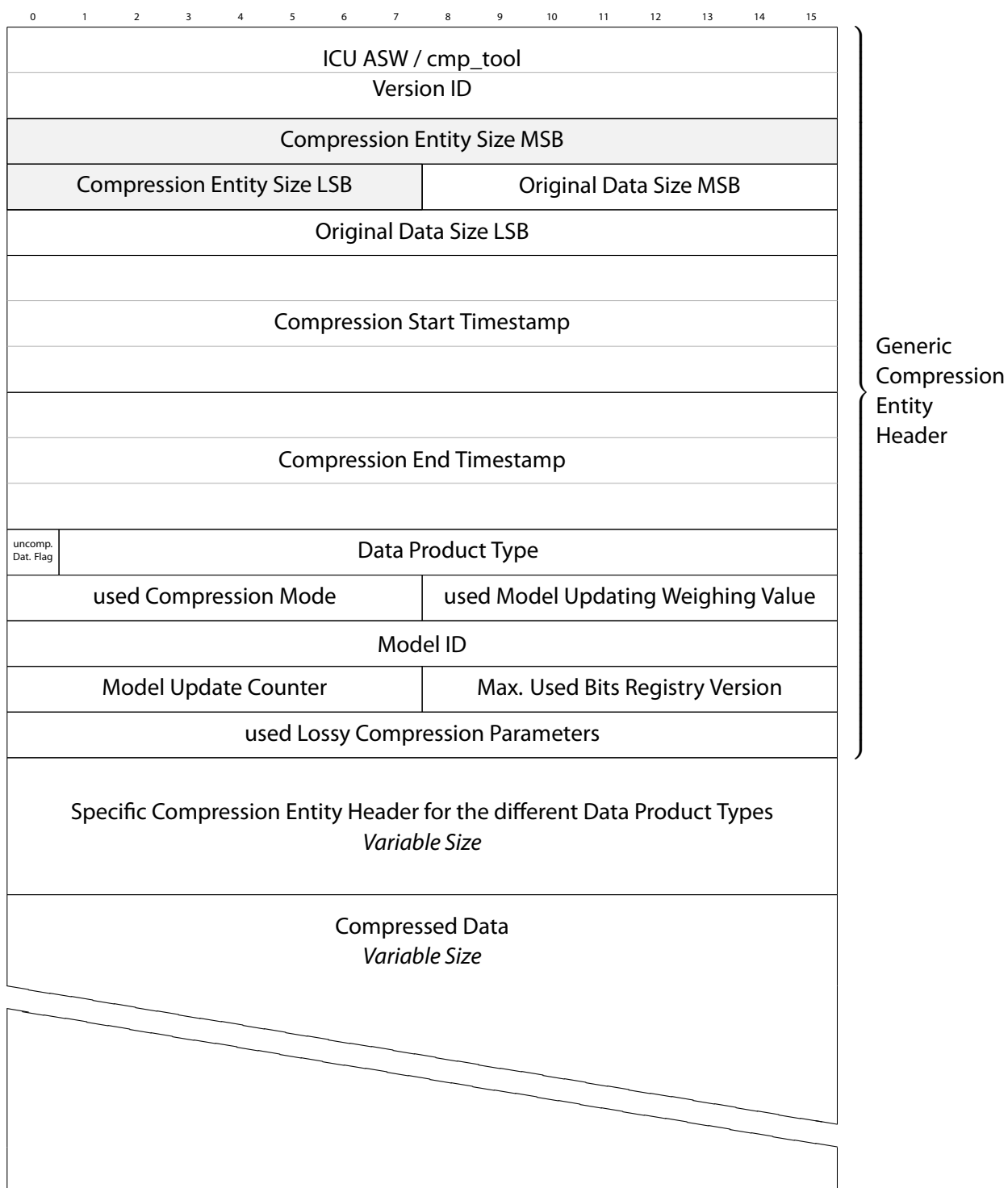


Figure 6.1: Structure of a compression entity consisting of a generic header, a data product type specific header and the compressed data.

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 <sup>24</sup> [
24	Original Data Size	Size of the data before compression in bytes	[0..2 <sup>24</sup> [
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 3.1. 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	Maximum Used Bits Registry Version	Version identifier for the max. used bits registry, see Section 5.1	uint8_t
16	used Lossy Comp. Par.	Parameter controlling the lossy compression	uint16_t
96, 32, 256, 0 var.	Specific Entity Header	Data product type specific header for imagette and non-imagette data	custom see Fig. 6.2, 6.3, 6.4
	Compressed Data	Compressed data	custom

Table 6.1: Compression entry header parameters description.

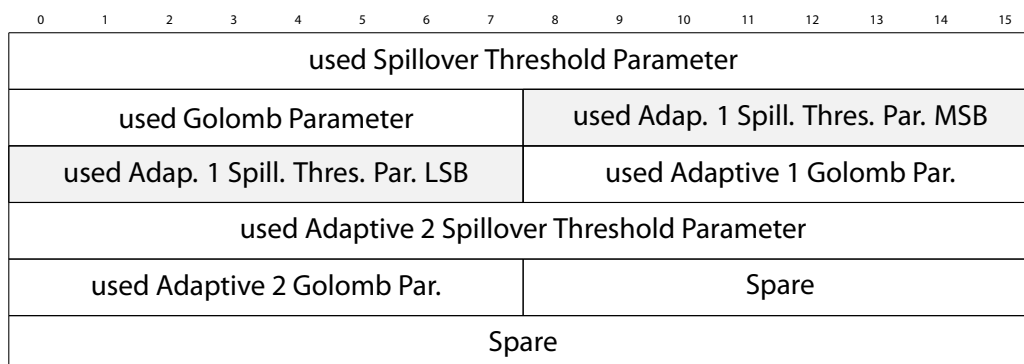


Figure 6.2: Specific compression entity header for RDCU imagette compression containing the semi-adaptive compression feature.

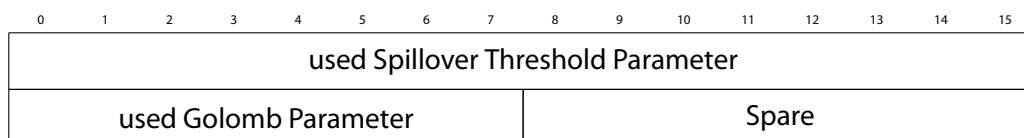


Figure 6.3: Specific compression entity header for RDCU (or ICU) imagette compression without containing the semi-adaptive compression feature.

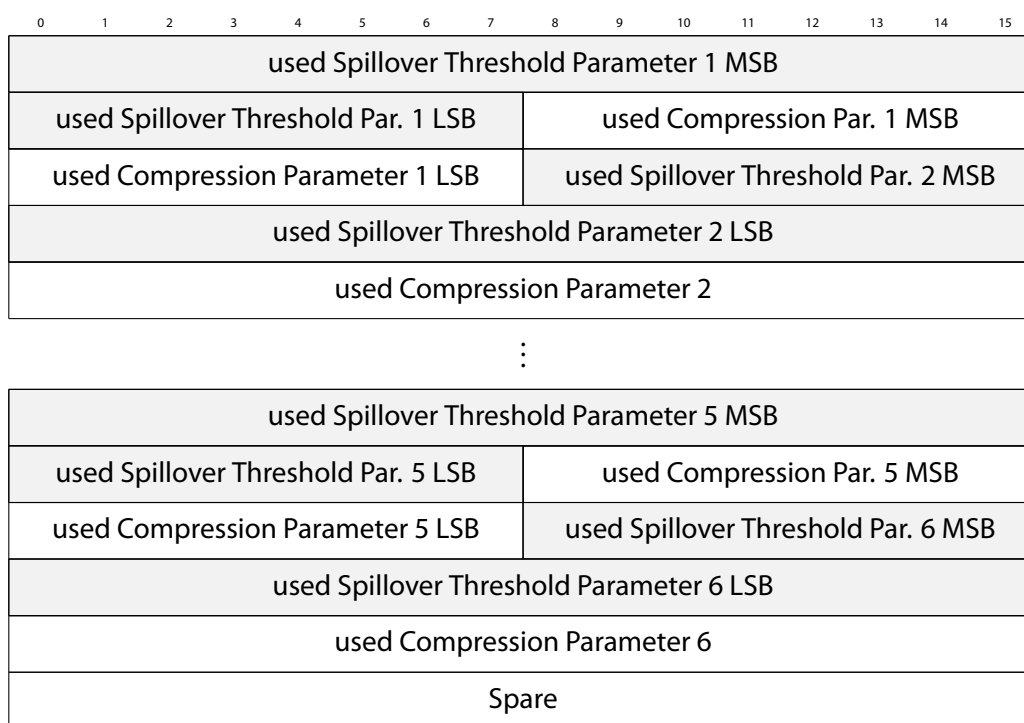


Figure 6.4: Specific compression entity header for non-imagette data compression.

## 7. Frame Processing

There's a problem with compressing imagerettes: the memory of the [RDCU SRAM](#) is much smaller than the sum of all imagerettes of a readout cycle. For this reason, all imagerettes must be divided into several chunks and each chunk must be individually compressed. Note that the imagerettes in a chunk must be in the same order over time. The sum of all imagerettes 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.



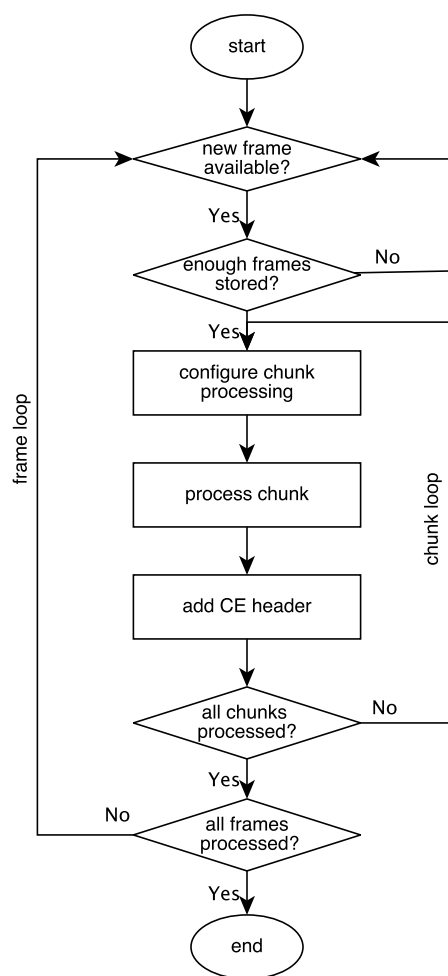


Figure 7.1: Frame processing workflow.

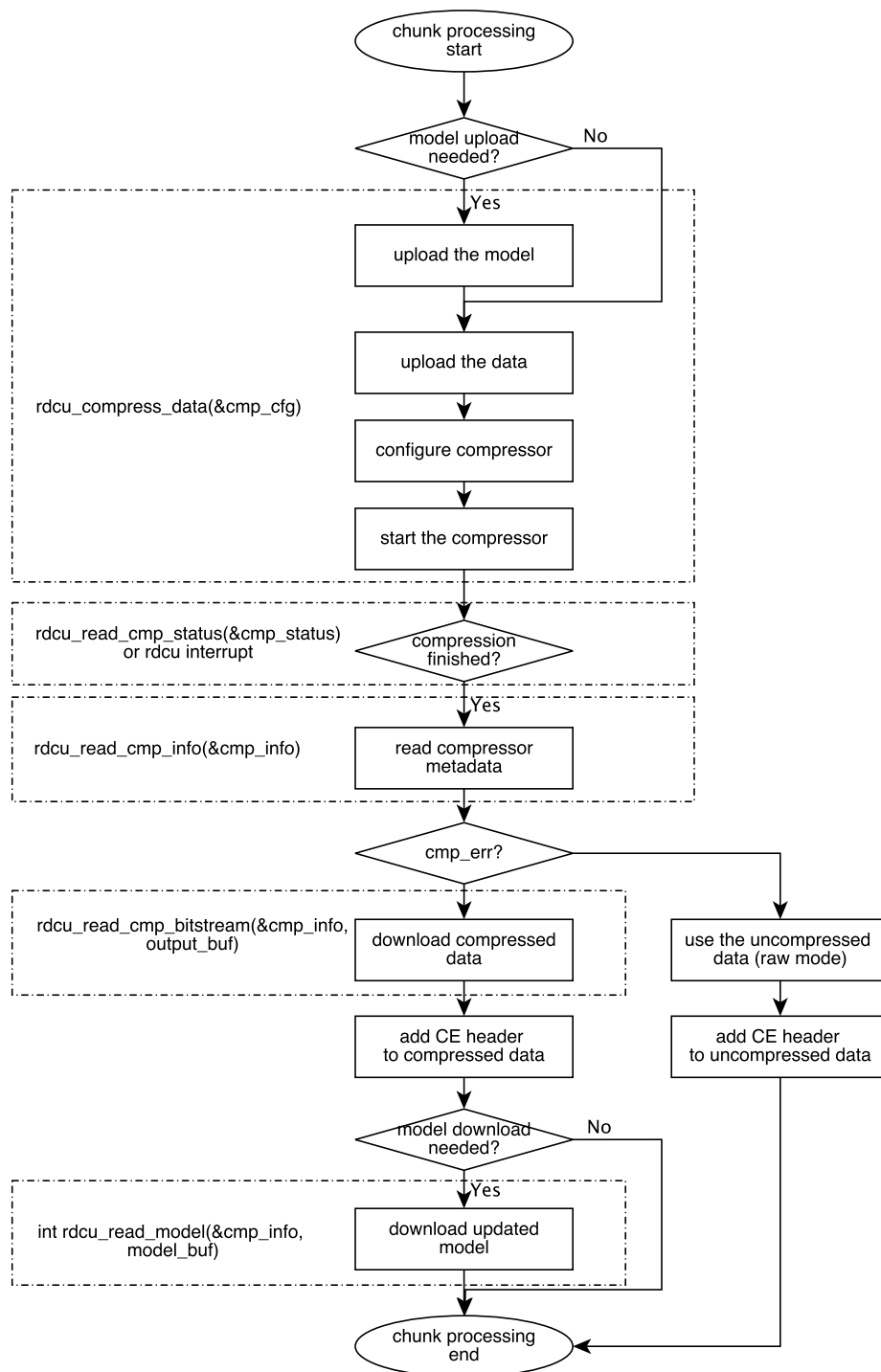


Figure 7.2: Chunk processing workflow.

## 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:

$$\text{output}_0 = \text{input}_0 \quad (7.1)$$

$$\text{output}_i = \text{input}_i - \text{input}_{i-1} \quad i = 1, \dots, n \quad (7.2)$$

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:

$$\text{output}_i = \text{input}_i - \text{model}_i \quad (7.3)$$

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:

$$\text{model}_1 = \text{input}_0 \quad (7.4)$$

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

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

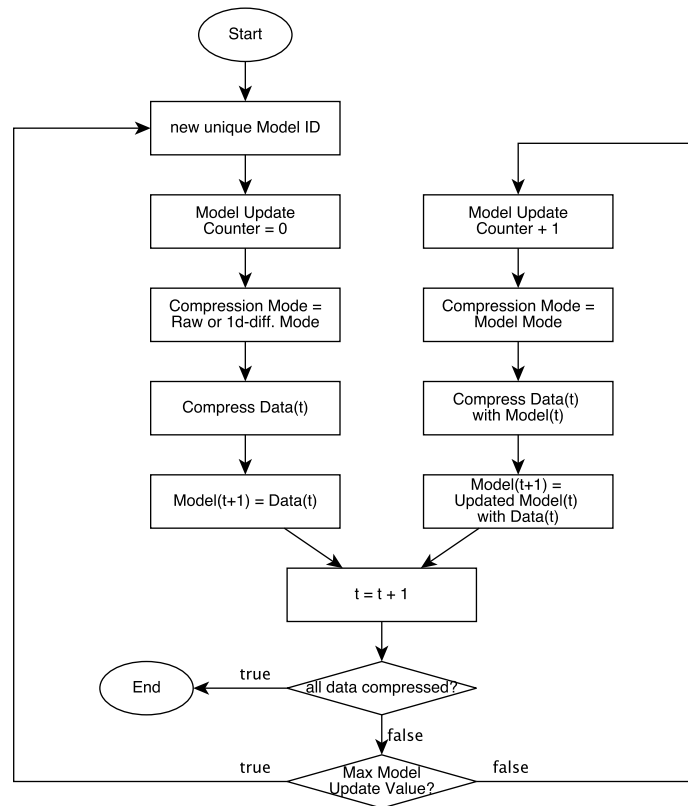


Figure 7.3: Flowchart of the 1d-differencing and model mode.

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 imagerettes (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 imagerettes. We call these frames  $N$  and  $N+1$ . First, the imagerettes 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 on larger chunks and a few compressions per frame or small chunks and more compressions.

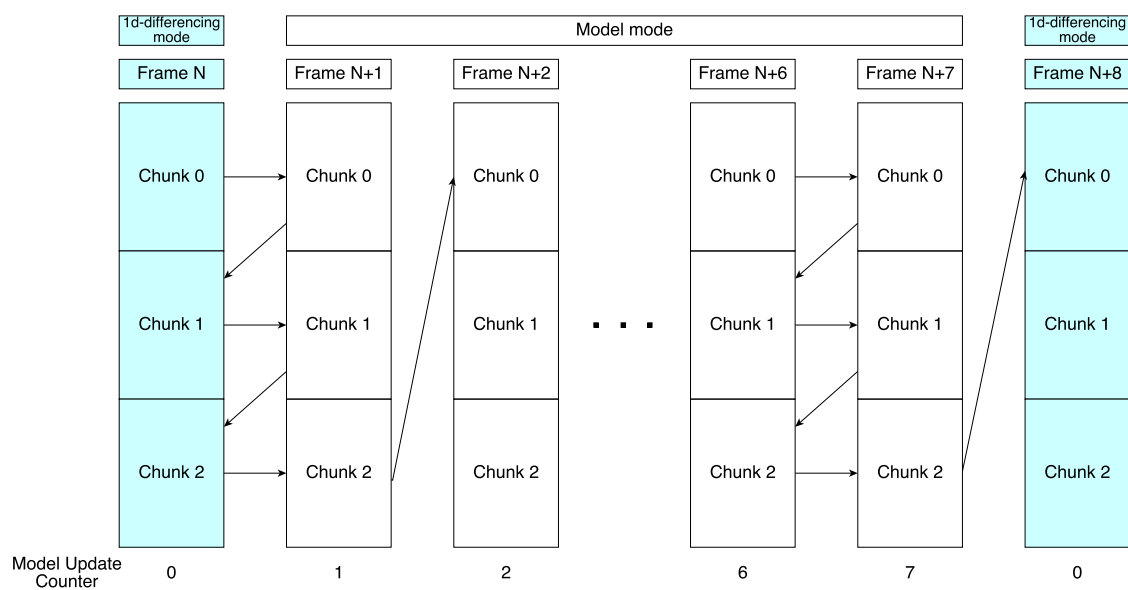


Figure 7.4: Visualization of the optimised chunk processing order.

## A. Hardware Compression Example

Listing A.1 shows a sample compression of the RDCU hardware compressor, it demonstrates how the different functions play together to achieve a compression of the data.

```

1#include <stdint.h>
2#include <stdlib.h>
3#include <stdio.h>
4
5#include <cmp_data_types.h>
6#include <cmp_rdcu.h>
7#include <cmp_entity.h>
8
9#define MAX_PAYLOAD_SIZE 4096
10#define DATA_SAMPLES 6 /* number of 16 bit samples to compress */
11#define CMP_ASW_VERSION_ID 1
12#define CMP_BUF_LEN_SAMPLES DATA_SAMPLES /* compressed buffer has the same sample size as
    the data buffer */
13/* The start_time, end_time, model_id and counter have to be managed by the ASW
14 * here we use arbitrary values for demonstration */
15#define START_TIME 0
16#define END_TIME 0
17#define MODEL_ID 42
18#define MODEL_COUNTER 1
19
20uint32_t rmap_rx(uint8_t *pkt);
21
22int32_t rmap_tx(const void *hdr, uint32_t hdr_size, const uint8_t non_crc_bytes,
    const void *data, uint32_t data_size);
23
24void demo_rdcu_compression(void) {
25    int cnt = 0;
26
27    /* declare configuration, status and information structure */
28    struct cmp_cfg example_cfg;
29    struct cmp_status example_status;
30    struct cmp_info example_info;
31
32    /* declare data buffers with some example data */
33    enum cmp_data_type example_data_type = DATA_TYPE_IMAGETTE_ADAPTIVE;
34    uint16_t data[DATA_SAMPLES] = {42, 23, 1, 13, 20, 1000};
35    uint16_t model[DATA_SAMPLES] = {0, 22, 3, 42, 23, 16};
36
37    /* initialise the libraries */
38    rdcu_ctrl_init();
39    rdcu_rmap_init(MAX_PAYLOAD_SIZE, rmap_tx, rmap_rx);
40
41    /* set up compressor configuration */
42    example_cfg = rdcu_cfg_create(example_data_type, CMP_DEF_IMA_MODEL_CMP_MODE,
    CMP_DEF_IMA_MODEL_MODEL_VALUE, CMP_DEF_IMA_MODEL_LOSSY_PAR);
43    if (example_cfg.data_type == DATA_TYPE_UNKOWN) {
44        printf("Error occurred during rdcu_cfg_create()\n");
45        return;
46    }
47
48    if (rdcu_cfg_buffers(&example_cfg, data, DATA_SAMPLES, model,
    CMP_DEF_IMA_MODEL_RDCU_DATA_ADR,

```

```

51     CMP_DEF_IMA_MODEL_RDCU_MODEL_ADR, CMP_DEF_IMA_MODEL_RDCU_UP_MODEL_ADR,
52     CMP_DEF_IMA_MODEL_RDCU_BUFFER_ADR, CMP_BUF_LEN_SAMPLES)) {
53     printf("Error occurred during rdcu_cfg_buffers()\n");
54     return;
55 }
56 if (rdcu_cfg_imagette(&example_cfg,
57     CMP_DEF_IMA_MODEL_GOLOMB_PAR, CMP_DEF_IMA_MODEL_SPILL_PAR,
58     CMP_DEF_IMA_MODEL_AP1_GOLOMB_PAR, CMP_DEF_IMA_MODEL_AP1_SPILL_PAR,
59     CMP_DEF_IMA_MODEL_AP2_GOLOMB_PAR, CMP_DEF_IMA_MODEL_AP2_SPILL_PAR)) {
60     printf("Error occurred during rdcu_cfg_imagette()\n");
61     return;
62 }
63
64 /* start HW compression */
65 if (rdcu_compress_data(&example_cfg)) {
66     printf("Error occurred during rdcu_compress_data()\n");
67     return;
68 }
69 /* start polling the compression status */
70 /* alternatively you can wait for an interrupt from the RDCU */
71 do {
72     /* check compression status */
73     if (rdcu_read_cmp_status(&example_status)) {
74         printf("Error occurred during rdcu_read_cmp_status()");
75         return;
76     }
77     cnt++;
78     if (cnt > 5) { /* wait for 5 polls */
79         printf("Not waiting for compressor to become ready, will "
80             "check status and abort\n");
81         /* interrupt the data compression */
82         rdcu_interrupt_compression();
83         /* now we may read the compression info register to get the error code */
84         if (rdcu_read_cmp_info(&example_info)) {
85             printf("Error occurred during rdcu_read_cmp_info()");
86             return;
87         }
88         printf("Compressor error code: 0x%02X\n", example_info.cmp_err);
89         return;
90     }
91 } while (!example_status.cmp_ready);
92
93 printf("Compression took %d polling cycles\n\n", cnt);
94
95 printf("Compressor status: ACT: %d, RDY: %d, DATA VALID: %d, INT: %d, INT_EN: %d\n",
96     example_status.cmp_active, example_status.cmp_ready, example_status.data_valid,
97     example_status.cmp_interrupted, example_status.rdcu_interrupt_en);
98
99 /* now we may read the compressor registers */
100 if (rdcu_read_cmp_info(&example_info)) {
101     printf("Error occurred during rdcu_read_cmp_info()");
102     return;
103 }
104
105 printf("\n\nHere's the content of the compressor registers:\n"
106     "=====\n");
107 print_cmp_info(&example_info);
108
109 /* check if data are valid or a compression error occurred */
110 if (example_info.cmp_err != 0 || example_status.data_valid == 0) {
111     printf("Compression error occurred! Compressor error code: 0x%02X\n",
112         example_info.cmp_err);

```



```

113 return;
114 }
115
116 /* build a compression entity and put compressed data from the RDCU into it and print */
117 if (1) {
118     struct cmp_entity *cmp_ent;
119     void *cmp_ent_data;
120     size_t cmp_ent_size;
121     uint32_t i, s;
122
123     /* get the size of the compression entity */
124     cmp_ent_size = cmp_ent_build(NULL, CMP_ASW_VERSION_ID,
125                                START_TIME, END_TIME, MODEL_ID, MODEL_COUNTER,
126                                &example_cfg, example_info.cmp_size);
127     if (!cmp_ent_size) {
128         printf("Error occurred during cmp_ent_build()\n");
129         return;
130     }
131
132     /* get memory for the compression entity */
133     cmp_ent = malloc(cmp_ent_size);
134     if (!cmp_ent) {
135         printf("Error occurred during malloc()\n");
136         return;
137     }
138
139     /* now let us build the compression entity */
140     cmp_ent_size = cmp_ent_build(cmp_ent, CMP_ASW_VERSION_ID,
141                                START_TIME, END_TIME, MODEL_ID, MODEL_COUNTER,
142                                &example_cfg, example_info.cmp_size);
143     if (!cmp_ent_size) {
144         printf("Error occurred during cmp_ent_build()\n");
145         return;
146     }
147
148     /* get the address to store the compressed data in the
149     * compression entity */
150     cmp_ent_data = cmp_ent_get_data_buf(cmp_ent);
151     if (!cmp_ent_data) {
152         printf("Error occurred during cmp_ent_get_data_buf()\n");
153         return;
154     }
155
156     /* now get the compressed data form RDCU and copy it into the
157     * compression entity */
158     if (rdcu_read_cmp_bitstream(&example_info, cmp_ent_data) < 0) {
159         printf("Error occurred while reading in the compressed data from the RDCU\n");
160         return;
161     }
162
163     s = cmp_ent_get_size(cmp_ent);
164     printf("\n\nHere's the compressed data including the header (size %lu):\n"
165           "=====\\n", s);
166     for (i = 0; i < s; i++) {
167         uint8_t *p = (uint8_t *)cmp_ent;
168         printf("%02X ", p[i]);
169         if (i && !((i+1) % 40))
170             printf("\\n");
171     }
172     printf("\\n");
173
174     /* now have a look into the compression entity */

```

```
175 printf("\n\nParse the compression entity header:\n"
176        "=====\\n");
177 cmp_ent_parse(cmp_ent);
178
179 free(cmp_ent);
180 }
181
182 /* read updated model to some buffer and print */
183 if (1) {
184     uint32_t i;
185     uint32_t s = cmp_cal_size_of_data(DATA_SAMPLES, example_data_type);
186     uint8_t *mymodel = malloc(s);
187
188     if (!mymodel) {
189         printf("malloc failed!\\n");
190         return;
191     }
192
193     if (rdcu_read_model(&example_info, mymodel) < 0)
194         printf("Error occurred while reading in the updated model\\n");
195
196     printf("\n\nHere's the updated model (size %lu):\\n"
197           "=====\\n", s);
198     for (i = 0; i < s; i++) {
199         printf("%02X ", mymodel[i]);
200         if (i && !((i+1) % 40))
201             printf("\\n");
202     }
203     printf("\\n");
204
205     free(mymodel);
206 }
207 }
```

Listing A.1: Example of a hardware compression.

## B. Software Compression Example

Listing B.1 shows an example of using the software compressor.

```

1 #include <stdint.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 #include <cmp_icu.h>
6 #include <cmp_data_types.h>
7 #include <cmp_entity.h>
8
9 #define DATA_SAMPLES 6 /* number of 16 bit samples to compress */
10 #define CMP_BUF_LEN_SAMPLES DATA_SAMPLES /* compressed buffer has the same sample size as
    the data buffer */
11 #define CMP_ASW_VERSION_ID 1
12 /* The start_time, end_time, model_id and counter have to be managed by the ASW
13 * here we use arbitrary values for demonstration */
14 #define START_TIME 0
15 #define END_TIME 0
16 #define MODEL_ID 42
17 #define MODEL_COUNTER 1
18
19 void demo_icu_compression(void) {
20     struct cmp_max_used_bits max_used_bits;
21     struct cmp_cfg example_cfg;
22     struct cmp_entity *cmp_entity;
23     uint32_t i, cmp_buf_size, entity_size;
24     int cmp_size_bits;
25     void *ent_cmp_data;
26
27 /* declare data buffers with some example data */
28 enum cmp_data_type example_data_type = DATA_TYPE_IMAGETTE;
29 uint16_t example_data[DATA_SAMPLES] = {42, 23, 1, 13, 20, 1000};
30 uint16_t example_model[DATA_SAMPLES] = {0, 22, 3, 42, 23, 16};
31 uint16_t updated_model[DATA_SAMPLES] = {0};
32
33 /* change the max_used_bit parameter for N-CAM imagette data */
34 max_used_bits = cmp_get_max_used_bits();
35 max_used_bits.version = 0;
36 max_used_bits.nc_imagette = 16; /* an imagette value uses a maximum of 16 bits */
37 cmp_set_max_used_bits(&max_used_bits);
38
39 /* create a compression configuration with default values */
40 example_cfg = cmp_cfg_icu_create(example_data_type, CMP_DEF_IMA_MODEL_CMP_MODE,
41     CMP_DEF_IMA_MODEL_MODEL_VALUE, CMP_LOSSLESS);
42 if (example_cfg.data_type == DATA_TYPE_UNKOWN) {
43     printf("Error occurred during cmp_cfg_icu_create()\n");
44     return;
45 }
46 /* configure imagette specific compression parameters with default values */
47 if (cmp_cfg_icu_imagette(&example_cfg, CMP_DEF_IMA_MODEL_GOLOMB_PAR,
48     CMP_DEF_IMA_MODEL_SPILL_PAR)) {
49     printf("Error occurred during cmp_cfg_icu_imagette()\n");
50     return;
51 }
52

```

```

53 /* calculate the size of the buffer for the compressed data in bytes */
54 cmp_buf_size = cmp_cal_size_of_data(CMP_BUF_LEN_SAMPLES, example_data_type);
55 if (!cmp_buf_size) {
56     printf("Error occurred during cmp_cal_size_of_data()\n");
57     return;
58 }
59 /* create a compression entity */
60 #define NO_CMP_MODE_RAW_USED 0
61 entity_size = cmp_ent_create(NULL, example_data_type, NO_CMP_MODE_RAW_USED, cmp_buf_size);
62 if (!entity_size) {
63     printf("Error occurred during cmp_ent_create()\n");
64     return;
65 }
66 cmp_entity = malloc(entity_size); /* allocated memory for the compression entity */
67 if (!cmp_entity) {
68     printf("malloc failed!\n");
69     return;
70 }
71 entity_size = cmp_ent_create(cmp_entity, example_data_type, NO_CMP_MODE_RAW_USED,
72                             cmp_buf_size);
73 if (!entity_size) {
74     printf("Error occurred during cmp_ent_create()\n");
75     return;
76 }
77 /* Configure the buffer related settings. We put the compressed data directly into
78 * the compression entity. In this way we do not need to copy the compressed data
79 * into the compression entity */
80 ent_cmp_data = cmp_ent_get_data_buf(cmp_entity);
81 if (!ent_cmp_data) {
82     printf("Error occurred during cmp_ent_get_data_buf()\n");
83     return;
84 }
85 cmp_buf_size = cmp_cfg_icu_buffers(&example_cfg, example_data, DATA_SAMPLES,
86                                   example_model, updated_model,
87                                   ent_cmp_data, CMP_BUF_LEN_SAMPLES);
88 if (!cmp_buf_size) {
89     printf("Error occurred during cmp_cfg_icu_buffers()\n");
90     free(cmp_entity);
91     return;
92 }
93
94 /* now we compress the data on the ICU */
95 cmp_size_bits = icu_compress_data(&example_cfg);
96 if (cmp_size_bits < 0) {
97     printf("Error occurred during icu_compress_data()\n");
98     if (cmp_size_bits == CMP_ERROR_SAMLL_BUF)
99         printf("The compressed data buffer is too small to hold all compressed data!\n");
100     if (cmp_size_bits == CMP_ERROR_HIGH_VALUE)
101         printf("A data or model value is bigger than the max_used_bits parameter allows (set
102             with the cmp_set_max_used_bits() function)!\n");
103     free(cmp_entity);
104     return;
105 }
106 /* now we set all the parameters in the compression entity header */
107 entity_size = cmp_ent_build(cmp_entity, CMP_ASW_VERSION_ID, START_TIME, END_TIME,
108                             MODEL_ID, MODEL_COUNTER, &example_cfg, cmp_size_bits);
109 if (!entity_size) {
110     printf("Error occurred during cmp_ent_build()\n");
111     free(cmp_entity);

```

```
112     return;
113 }
114
115 printf("Here's the compressed entity (size %u):\n",
116        "=====\n", entity_size);
117 for (i = 0; i < entity_size; i++) {
118     uint8_t *p = (uint8_t *)cmp_entity; /* the compression entity is big-endian */
119     printf("%02X ", p[i]);
120     if (i && !((i+1) % 40))
121         printf("\n");
122 }
123 printf("\n\nHere's the updated model (samples=%u):\n",
124        "=====\n", DATA_SAMPLES);
125 for (i = 0; i < DATA_SAMPLES; i++) {
126     printf("%04X ", updated_model[i]);
127     if (i && !((i+1) % 20))
128         printf("\n");
129 }
130 printf("\n");
131
132 free(cmp_entity);
133 }
```

Listing B.1: Example of a software compression.