Fischer Benjamin, BSc

POCKET+
Research and implementation of a new spacecraft telemetry data compression algorithm

Master's Thesis

to achieve the university degree of

Dipl. Ing.

Master's degree programme: Space Sciences and Earth from Space

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl-Ing. Dr.techn.Otto Koudelka

Institute of Communication Networks and Satellite Communications
Head: Univ.-Prof. Dipl-Ing. Dr.techn.Otto Koudelka

Graz, June 2017

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Abstract

The amount of spacecraft telemetry data is constantly increasing, but the available bandwidth is generally bounded. Spacecraft operators have to decide which telemetry to downlink and which not. Furthermore, this is not only causing a loss of information but also a waste of data storage and bandwidth. When compressing telemetry data all these drawbacks can be canceled out. Moreover, data compression can lead to a better signal quality and less transmission power.
The Advanced Operation Concepts Office at the European Space Operations Center has been promoting the compressing of spacecraft telemetry data since 2009 and invented an algorithm specifically for that purpose. It is called Pocket+ and its compression is based on bit differences of two fixed length consecutive packets. Bits which have changed become non-predictable and are added the the compressed packet. A mask tracks all previous bit flips and is updated every packet, which is called a negative update, or cyclically, which is called a positive update. The negative update indicates that more bits became non-predictable and a positive mask update removes non-predictable bits from the mask. Moreover, it uses an integrated feature named protection level to cope with packet loss, which could lead to decompression problems.
The results in terms of compression and execution times of a C implementation have shown that Pocket+ can easily compete with, or outperform, several generic compression algorithms such as LZ4, bzip2, or gzip. Pocket+ almost always showed better a compression in a comparatively short execution time over these generic algorithms. Therefore, Pocket+ can be applied on raw binary files with a repetitive bit pattern. This is demonstrated with a picture generated by the OPS-SAT on-board camera, where Pocket+ surpasses all other compression algorithms distinctly.
In 2017 it was decided to integrate Pocket+ into the on-board software of OPS-SAT, which will raise the technology readiness level up to nine and facilitates the implementation in future space missions.

Contents

List of Figures

1 Introduction

The amount of spacecraft (S/C) data is constantly increasing and in 1966 Goddard Space Flight Center (GSFC) had already operated fourteen satellites and pointed out that fourteen satellites for which GSFC has complete mission responsibility, produce data at the average rate of 60 million data points per day, where a data point is defined as a single complete reading taken from a sensor. In the near future, it is anticipated that this will grow to more than 200 million data points per day. Edmund J. Habib, 1966, p.1

Satellite sensor or telemetry (TM) data is often stored in mnemonics and written to the on-board data pool (OBDP) of the satellite. When a link to a ground station is established the data is transmitted in data streams. These streams can contain between 700 and 12,000 or more mnemonics. These mnemonics must be analyzed to ensure that the satellite is healthy, to assist in resolving any anomalies, and to determine if there are any trends that would indicate a possible future problem. N.L. Crowley, 1997, p.1

Hence this amount of data has an impact on mission control systems (MCS) and S/C operators. As a result, S/C operators have to make hard choices between which parameters to downlink and which not, in which different situations and at which sampling rates. David Evans, 2017, p.1

Data compression would drastically increase the amount of received data, however it would also add additional complexity and demands hardware resources such as CPU and RAM. This chapter will highlight the impacts of data compression and the following chapters are about Pocket+, a TM data compression algorithm invented at the European Space Agency (ESA).

1.1 Compressing TM data

The implementation of data compression leads to new possibilities and drawbacks, which are discussed in the following subsections.

1.1.1 Bandwith, datarate, and signal quality

For S/C communication the digital modulation scheme Phase-Shift Keying (PSK) is used most commonly [Takashi Iida, 2000, p. 402], thus the following calculations refer to PSK modulation schemes. For BPSK, QPSK, 8PSK and 16PSK the required 3.0 dB bandwidth can be calculated with

$$B_{\text{BPSK}} = f_b(1+\alpha), \tag{1.1}$$

$$B_{\text{QPSK}} = \frac{f_b}{2}(1+\alpha), \tag{1.2}$$

$$B_{\text{8PSK}} = \frac{f_b}{3}(1+\alpha), \tag{1.3}$$

$$B_{\text{16PSK}} = \frac{f_b}{4}(1+\alpha), \tag{1.4}$$

where f_b is the channel bit-rate and α is the roll-off factor of the raised-cosine filter. For simplicity reasons, forward error correction (FEC) is not considered. When compressing data, the following two options are available:

1. Use the same data rate, which results in more information in the same time.
2. Decrease the data rate by the average compression, which results in the same amount of information in the same time.

Approach one is straightforward and the whole transmission configuration stays the same.
When decreasing the data rate the energy per bit (E_b) increases, which is defined as

$$E_b = \frac{C}{f_b}, \tag{1.5}$$

where C is the carrier power. Furthermore, the occupied bandwidth of a PSK modulated signal goes down, which leads to less noise on the receiver side.

However, the ground station measures a signal-to-noise ratio (SNR) according to equation 1.6 and experiences a bit-error rate (BER) according to figure 1.1.

$$\mathrm{SNR} = \frac{E_b}{N_0} \cdot \frac{f_b}{B}, \tag{1.6}$$

where N_0 is the spectral noise density. With these equations and figure 1.1 it can be seen that when decreasing the data rate for a given modulation scheme, the SNR increases (less noise / higher E_b) and the BER decreases. This leads to the conclusion that instead of increasing the transmission power the S/C operator could use data compression in order to improve the signal quality. Subsequently the transmission power can be readjusted to save power and meet the SNR requirements for the receiver.

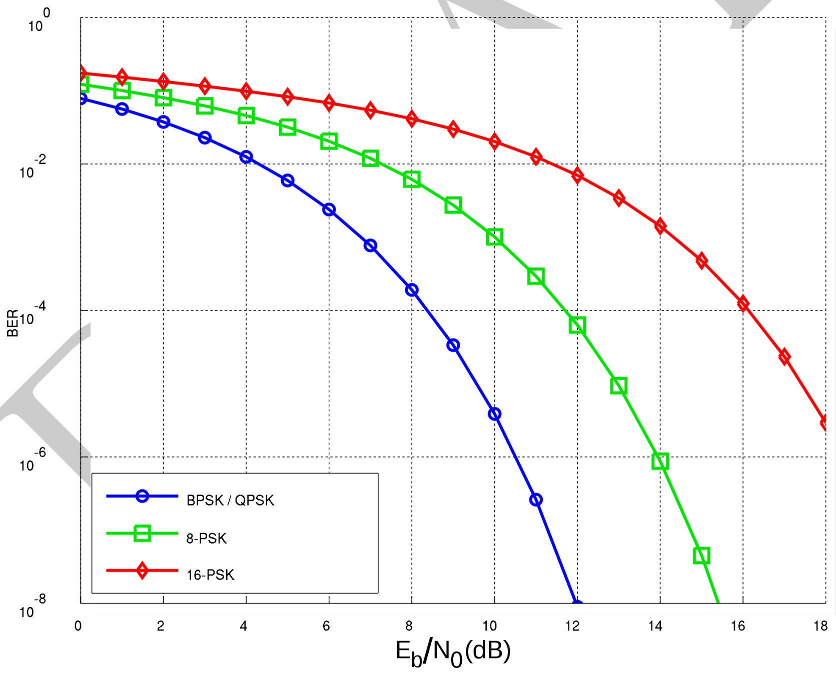


Figure 1.1: BER for BPSK, QPSK, 8PSK and 16PSK. Wikimedia Commons, 2017

1.1.2 Memory storage and data corruption

TM data is typically stored in separate Mass Memory Banks Formatting Units (MMFUs), often also called Solid State Recorders (SSRs), which are fast and large memory areas. Several recorders also provide integrated data compression units and some suppliers offer external separate units. Most of SSRs are based on synchronous dynamic random access memory (SDRAM) technology, however latest recorder generations are based on non volatile flash memory technology such as electrically erasable programmable read-only memory (EEPROM). Eickhoff, 2011, p. 82 - 83
In terms of unwanted bit flips such as Single Event Upsets (SEU), the usage of compression algorithms leads to a higher probability of data loss depending on the compression algorithm. When data is compressed, most of the redundancy is removed and the compressed data is stored in a new data format. SEUs can lead to data corruption within this data structure, which could lead to uncompressable data or data loss. Certainly this disadvantage can be minimized by using error correction and detection methods such as a CRC.

1.1.3 Complexity and Technology Readiness Level

The addition of a compression algorithm induces new complexity. Most S/C use already existing and flight proven systems, which are highly standardized. Space Agencies such as ESA and NASA require a TRL (Technology Readiness Level) of eight for new systems. In order to reach a TRL of eight, the system itself must withstand intensive testing and pass a technology readiness assessment. In order to ensure that a TRA of an element is objective, it is completed by independent expertise in the discipline, i.e. not part of the technology developer engineering team. ECSS-E-HB-11A, 2017, p. 18-23
However, the Advanced Operation Concepts Office at ESA-ESOC and TUG are working on a satellite called OPS-SAT. The satellite's goal is to test and evaluate new software in space. Pocket+ will be part of a new module within the on-board software (OBSW) and demonstrate the advantages of TM data compression.

1.1.4 Processing hardware

The hardware of todays S/C is comparable to a degraded Desktop PC from several decades ago. Since the hardware is flight proven and flew successfully in past space missions, it is normally not desirable to change a functioning system.
S/C software modules are separated into systems blocks which are executed cyclically. Most of these blocks are implemented as individual tasks or threads on the real time operating system (RTOS). In former on-board computer (OBC) generations a perfectly optimized OBSW tasking with respect to CPU load management was essential. Nowadays the situation became more relaxed, but an efficient tuning of OBSW task scheduling is still necessary. Eickhoff, 2011, p. 121
When adding a new system such as a compression algorithm, the runtime for this task must be deterministic and more CPU and RAM resources must be allocated for that task. Otherwise the entire OBSW scheduling table will not fulfill its real time requirements.

2 POCKET+ compression algorithm

2.1 Preamble - The story of POCKET+

The European Space Operations Center (ESOC) has developed and patented several algorithms that when tested reach average compression of between 5 % and 20 %. In 2011 an algorithm that took this one step further was invented. It was called POCKET and can compress individual packets in only a few microseconds on representative flight hardware. This made it suitable for compressing the real time TM streams. In 2012, the complete end to end chain was built and tested in an ESA contract with Spacebel SA and will be tested on PROBA-2 in 2017. Since 2012 the algorithm has undergone some major changes that have resulted in a new patent and a name change to POCKET+. One major improvement is that it is now self-adapting, meaning it adapts to changes in the data without ground intervention for a very small drop in compression and speed performance. To raise the TRL, POCKET+ will be implemented in OPS-SAT, the world's first mission dedicated to in-flight testing of operational technology like this. David Evans, 2017, p. 1

2.2 Functional principle

POCKET+ works on fixed length data packets and uses a bit mask to distinguish between predictable and non-predictable bits, which can be determined by the negative and positive mask update.

2.2.1 A look into the past - The mask updates

A constantly updating bit mask is used to split a packet into bits which have the same value as the last packet and those that may not. It updates every packet negatively (bits become non-predictable) and every cycle (e.g. 20 packets) positively (bits become predictable).

Collect every change - The negative mask update

The negative mask update accumulates bit differences of consecutive packets (the previous packet is termed as reference packet) and updates the mask negatively. The mask comprises all bit positions, which changed in the past. These bit positions tend to change frequently and are named non-predictable bits. All newly added mask bit positions and non-predictable bits are then written to the compressed packet. Mask bit positions are run length encoded and non-predictable bits are put in the clear. The basic principle is shown in figure 2.1.

Compare with the previous cycle - The positive mask update

The positive mask update compares bit differences of two different sampling cycles (e.g 20 packets). Every cycles, the negative mask, which has been built by all bits changes in the past, is compared with the bits, which have changed in the last 20 packets. All bits, which are in the negative mask and not in the positive mask (here are only bits which changed the last 20 packets), become predictable. These bits are removed from the negative mask and their positions are written run length encoded to the compressed packet.The principle is shown in figure 2.2.

Packet	Buffer	Packet (One byte)								Non-predictable bits	Negative mask bits
1	Packet 0 (Reference)	0	0	0	0	0	1	0	0	1	2nd bit
	Packet 1	0	0	0	0	0	1	1	0		
	Negative mask	0	0	0	0	0	0	1	0		
	Positive mask	0	0	0	0	0	0	1	0		
2	Packet 1	0	0	0	0	0	1	1	0	01	3rd bit
	Packet 2	0	0	0	0	0	0	1	0		
	Negative mask	0	0	0	0	0	1	1	0		
	Positive mask	0	0	0	0	0	1	1	0		
3	Packet 2	0	0	0	0	0	0	1	0	1101	1st bit 5th bit
	Packet 3	0	0	0	1	0	1	0	1		
	Negative mask	0	0	0	1	0	1	1	1		
	Positive mask	0	0	0	1	0	1	1	1		

Figure 2.1: Principle of the negative mask update. Every new packet is compared to the last packet. Bit positions, which have changed, are set in the mask and called non-predictable bits (red). When an already set mask bit has changed (orange), the non-predictable bit value itself changes.

Cycle	Buffer	Packet (one byte)								Positive mask bits
1	Negative Mask	0	0	0	1	0	1	1	1	None First iteration
	Positive Mask	0	0	0	1	0	1	1	1	
	New Mask	0	0	0	1	0	1	1	1	
	Reset Positive	0	0	0	0	0	0	0	0	
2	Negative Mask	0	1	0	1	0	1	1	1	5th bit
	Positive Mask	0	1	0	0	0	1	1	1	
	New Mask	0	1	0	0	0	1	1	1	
	Reset Positive	0	0	0	0	0	0	0	0	
3	Negative Mask	1	1	0	0	0	1	1	1	1st bit 3rd bit
	Positive Mask	1	1	0	0	0	0	1	0	
	New Mask	1	1	0	0	0	0	1	0	
	Positive Reset	0	0	0	0	0	0	0	0	

Figure 2.2: Principle of the positive mask update. Every cycle (e.g. 20 packets) the negative mask is compared to the positive mask. Bits which are in the negative mask, and not in the positive mask, are removed and become predictable (red).

2.2.2 The compression process

In both mask updates, the new mask bit positions are converted into a relative mask position, termed as delta (see section 2.2.3), run length encoded, and then written to the compressed packet. Subsequently, the non-predictable bits are added to the compressed packet. These two processes are called

- Non-predictable bits and
- relative mask bit positions (deltas) extraction.

The compression process is shown in figure 2.3.

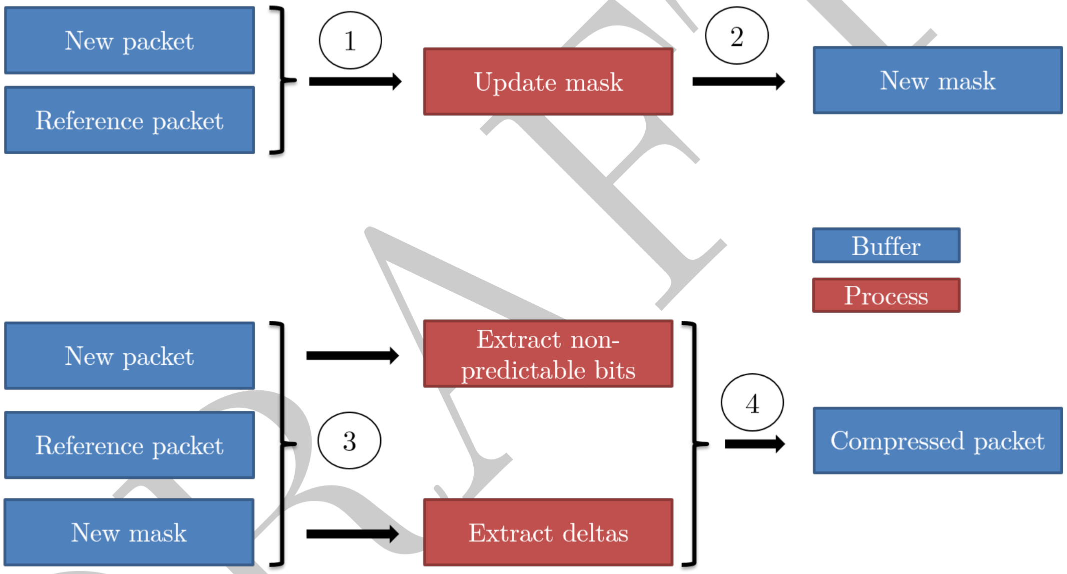


Figure 2.3: Principle of the compression process. First, a new mask is generated (mask update). Afterwards, the deltas and non-predictable bits can be extracted and written to the compressed packet.

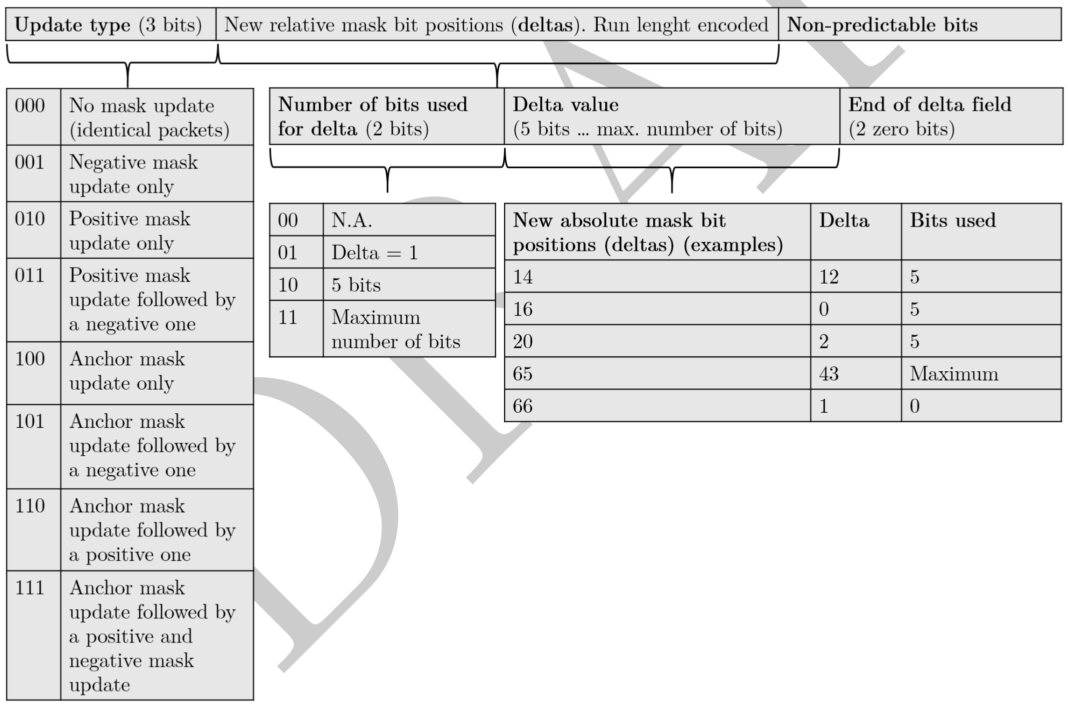
2.2.3 Structure of the compressed packet

All absolute mask bit positions are converted into a relative bit position, termed as delta, and can be determined by

$$\Delta = \begin{cases} (u-v)-2, & \text{if } (u-v) \geq 2 \\ 1, & \text{otherwise} \end{cases} \tag{2.1}$$

where u is a set absolute mask bit position and v the previous one. The usage of deltas, instead of full bit positions, considerably reduce the number of necessary bits for the run length encoding process. In figure 2.4, the structure of a compressed packet and examples for deltas are shown. The maximum number of bits ($\mathrm{n_{max}}$) used for a run length encoded delta value can be determined by the following equation:

$$\mathrm{n_{max}} = \mathrm{ld(Number\ of\ bits\ in\ packet)} \tag{2.2}$$



Update type (3 bits)	New relative mask bit positions (**deltas**). Run lenght encoded	**Non-predictable bits**

000	No mask update (identical packets)
001	Negative mask update only
010	Positive mask update only
011	Positive mask update followed by a negative one
100	Anchor mask update only
101	Anchor mask update followed by a negative one
110	Anchor mask update followed by a positive one
111	Anchor mask update followed by a positive and negative mask update

Number of bits used for delta (2 bits)	**Delta value** (5 bits ... max. number of bits)	**End of delta field** (2 zero bits)

00	N.A.
01	Delta = 1
10	5 bits
11	Maximum number of bits

New absolute mask bit positions (deltas) (examples)	**Delta**	**Bits used**
14	12	5
16	0	5
20	2	5
65	43	Maximum
66	1	0

Figure 2.4: Structure of the compressed packet.

2.2.4 Fields of applications

Since POCKET+ works with the difference of two consecutive packets, the Hamming distance, of these two packets should be low in order to obtain good compression ratios. The higher the Hamming distance, the more mask updates (deltas) and non-predictable bits are necessary to describe these changes. Therefore, POCKET+ should be implemented in systems which show only low dynamic properties.

2.3 Decompression

In order to decompress a compressed packet, a valid mask and non-predictable bit field are required. To obtain a valid mask all received delta fields must be error-free. With a valid mask, the number of non-predictable bits can be determined and the actual bits extracted. The decompression uses the following scheme that is also shown in figure 2.5.

1. Extract the deltas from the compressed packet.
2. Update the old mask.
3. Extract the non-predictable bits from the compressed packet.
4. Add the non-predictable bits to the reference packet, which results in the uncompressed packet.
5. Make the uncompressed packet the new reference packet and the new mask the old mask.

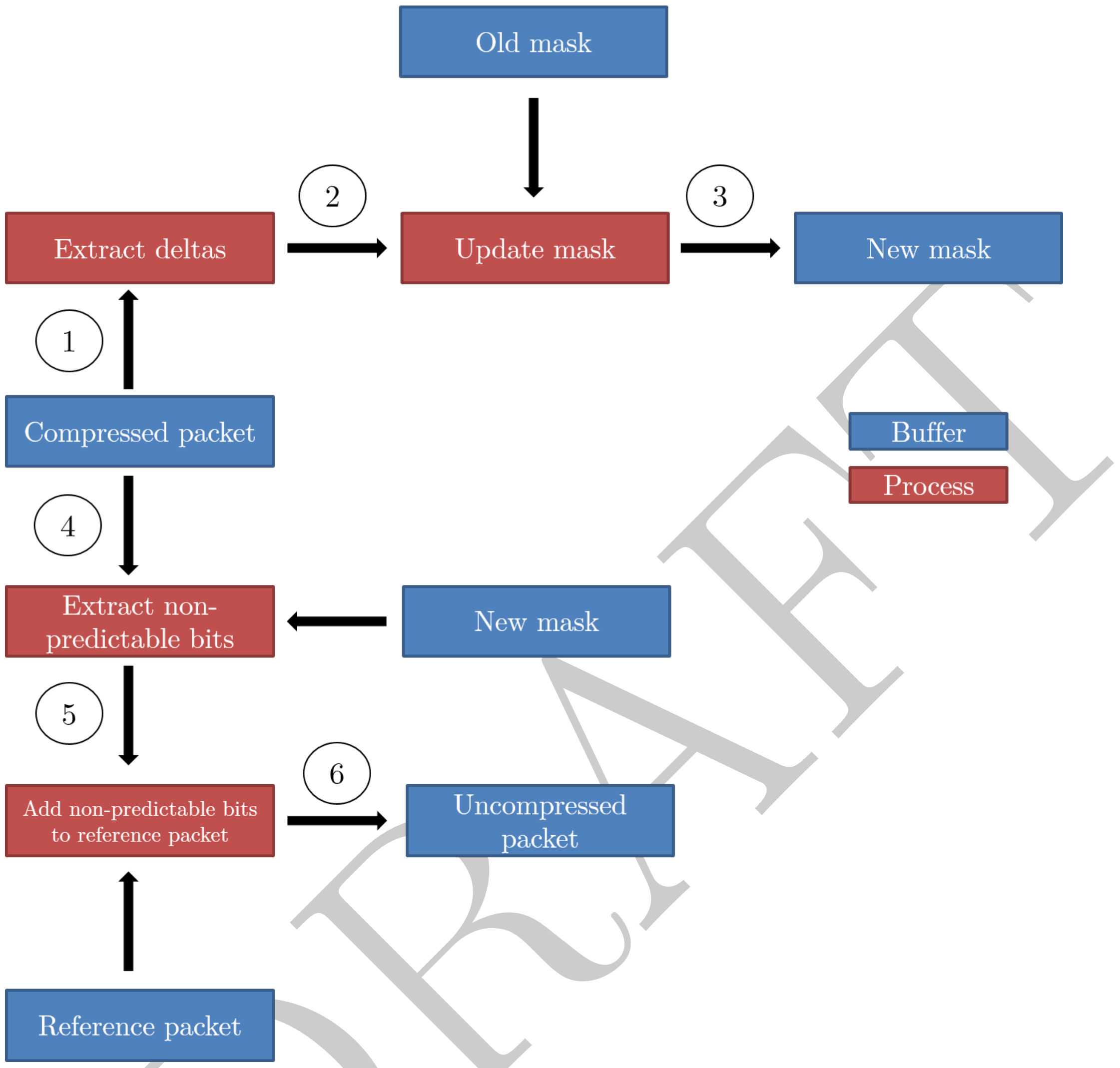


Figure 2.5: Principle of the decompression process. First, the deltas are extracted and used to generate the new mask. Then, the non-predictable bits are extracted and overwrite the bits in the reference packet according to the set mask bit position. The result is the uncompressed packet.

2.4 Reliability - Transmission errors

In order to start the compression algorithm a reference packet, which is an uncompressed original packet, is required. All subsequent mask updates refer to this reference packet.

2.4.1 Bit errors and packet loss

Error in the delta field

A single bit error in one of the delta fields would lead to incorrect, or even unrecoverable, future packets. In other words, all packets after the faulty one are going to be wrong. This is because the original mask can not be reconstructed anymore. Furthermore, two cases regarding the resumption of the decompression process can be distinguished:

1. The reference/previous packet is valid (correctly decompressed) and no positive mask update has been applied. The decompression process is in the same tracking cycle as the compression process. Both of them are linked to the same reference packets.
   a) An anchor mask update must be requested.
2. The reference/previous packet is invalid (incorrectly decompressed) and the decompression missed a positive mask update. The decompression and compression process are in different tracking cycles, meaning that the ground and S/C mask update process is not synchronized any longer.
   a) An anchor mask update and a reference packet must be requested.

The anchor mask update comprises the entire mask build of all previous packets. If the decompression module missed a positive mask update (case two), a reference packet of the current tracking cycle is required. If no reference packet was requested, all future uncompressed packets could be wrong. This error can be detected by unrealistic values or by a cyclic redundancy check (CRC). However, when an anchor mask update and a reference packet have been requested and received, the decompression restarts and even previous un-compressable packets can be decompressed. To minimize the probability of these events, the protection level was invented.

Error in the non-predictable bit field

A bit error in the non-predictable bit field would lead to one incorrectly decompressed packet, but future packets would not be affected. This error can also be detected by error detection methods such as a CRC.

2.4.2 The protection level - security against packet loss

When the protection level is used, the current compressed packet also includes the deltas of the previous n packets, whereby the first delta value is always implemented *wraparound*. The formula for a wraparound delta is

$$\Delta_{\text{wraparound}} = \begin{cases} x - y, & \text{if } x > y \\ z - y, & \text{otherwise} \end{cases} \tag{2.3}$$

where x is the position of the most significant mask bit (MSMB) of the last packet, y is the MSMB of the current packet, and z is the total number of bits in the packet.

2.4.3 The anchor mask update

With an anchor mask update, the entire latest mask can be obtained. It also consists of run length encoded deltas, but the delta extraction process is applied on a "HXORed" mask. Thereby, the actual mask has gone through a XOR gatter with a replica of itself shifted by one bit to the right.

3 Implementation

3.1 General software requirements specifications

- Copyrighted intellectual property by others must not be used.
  - Action: Only the C standard library and open source software will be used.
  - Testing: Code inspection.
- Software must be portable to other systems such as a micro-controllers, Windows/Linux PCs, and a ARM architectures.
  - Action: A CMake file will be provided, which in turn creates a Makefile for these systems.
  - Testing: All executables have to produce the same data output.
- Implementation must be generic for any kind of input data.
  - Action: Implementation works on the bit-level.
  - Testing: CCSDS packets and a raw binary files have to be compressed without any code changes to the compression module.
- Up to 8,191 kilobyte of data per packet must be processable.
  - Action: Code is adjusted to at least 16 bit variables.
  - Testing: Test procedures with packets up to 8,191 kilobyte.

3.2 Compression (C)

The compression algorithm was written in C since it is the most common programming language for the OBSW of satellites. Eickhoff, 2011, p.136 This sections describes the generic implementation, which uses dynamic memory allocation, and is optimised for Desktop PCs. The implementation used for the OBSW of OPS-SAT, which fullfills the ECSS software standards for memory allocation are covered in section 3.4.

3.2.1 Software architecture

A header file (*compression.h*) and the related source file (*compression.c*) contain the source code for the compression module. The software comes with an API, which shall be used by any user or developer. Static functions declared in *compression.src* are needed for the compression process itself and are called by the API methods in the background. The according function call diagram is shown in figure 3.2. The compression process itself is fully generic and works on byte level. The only requirement is a constant input block size. For example, the compression of a CCSDS datafield would require a *packetizer*, which splits the CCSDS packets and inputs the desired datafields into the compression module. Afterwards, it builds the new CCSDS packet out of the compressed memory block. This principle is shown in figure 3.1. The algorithm needs a 32 bit memory block as input, and outputs a compressed 8 bit memory block. Whereas, the maximum size of the input memory block is 8.188 kilobyte. This limit arises from the range of 16 bit variables and the usage of 32 bit input data, and was calculated with the following formula.

$$\text{input}_{\text{max}} = \left\lfloor \frac{\left\lfloor \frac{2^{16}-1}{8} \right\rfloor}{4} \right\rfloor \cdot 4 = 8,188 \tag{3.1}$$

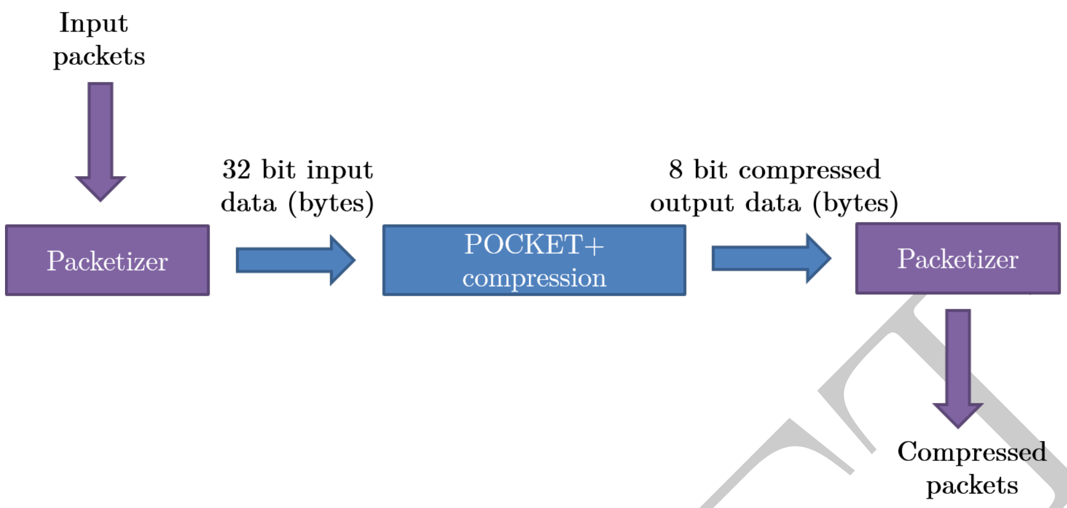


Figure 3.1: Software architecture of the POCKET+ compression. Input and output data are independent of the compression process.

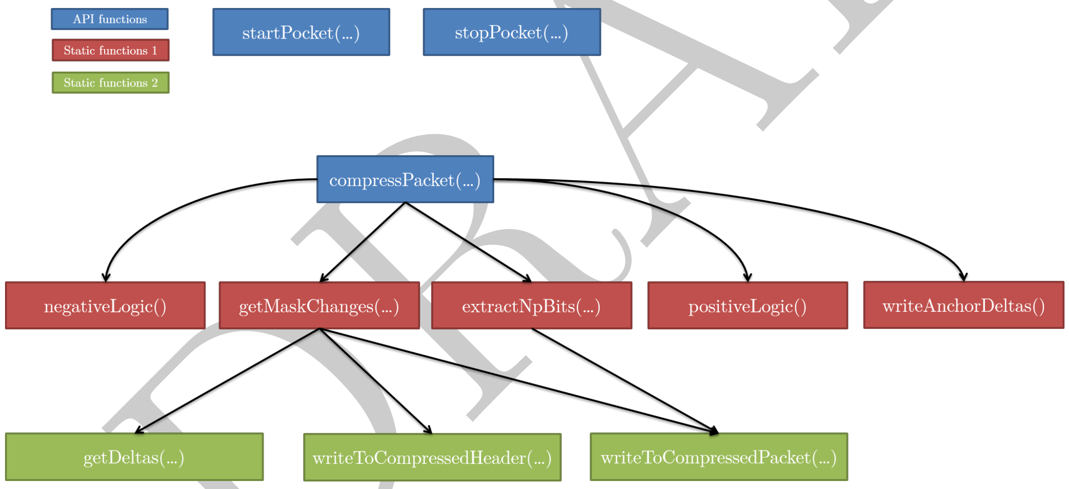


Figure 3.2: Function call diagram of the POCKET+ compression.

3.2.2 Configuration parameters

Five configuration parameters are required to initialize Pocket+.

- Size of the input data. (bytes)
- Positive update rate
  - Typical values = 20 ... 100
- Anchor mask update rate
  - Never output any anchor mask updates = 0
  - Always output anchor mask update = 1
  - Output an anchor mask update every 5 packets = 5
- Maximum protection level used
  - No protection level = 1
  - Add mask updates from the previous three packets (up to 3 packets can be lost) = 4
- Non-predictable word offset
  - All non-predictable bits are added to the compressed packet (default) = 0
  - Do not include the non-predictable bits of the the first word (4 bytes) = 1
  - Do not include the non-predictable bits of first three words (12 bytes) = 3

3.2.3 Application Programming Interface (API)

The following functions shall be used by any user or developer to interact with Pocket+.

Start and configure POCKET+

```
uint_fast16_t startPocket( uint8_t ** compressedPacket ,
    uint32_t * referencePacket , uint_fast16_t
    referencePacketSize , uint_fast16_t positiveUpdateRate ,
    uint_fast8_t maximumProtectionLevel , uint_fast8_t
    anchorUpdateRate , int_fast16_t npOffset );
```

Listing 3.1: Pocket+ initialization function in compression.h

Description: Initialize variables and allocate memory. This function must be called in order to compress any data.
Return values:

- Success: <referencePacketSize >
- Failure: 0

Arguments:

- uint8_t ** compressedPacket ... pointer to the memory pointer in main(). Memory will be allocated in startPocket(...).
- uint32_t * referencePacket ... pointer to the 32 bit reference packet.
- uint_fast16_t referencePacketSize ... size of the reference / input packets (bytes).
- uint_fast16_t positiveUpdateRate ... positive update rate.
- uint_fast8_t maximumProtectionLevel ... maximum protection level used.
- uint_fast8_t anchorUpdateRate ... anchor mask update rate.
- int_fast16_t npOffset ... offset for the non-predictable bits extraction process.

Compress data

```
int_fast16_t compressPacket(uint8_t * compressedPacket,
    uint32_t * uncompressedPaket)
```

Listing 3.2: Main compression function in compression.h

Description: Compress a 32 bit input data buffer and output a 8 bit compressed data buffer. The input data must have the same size as the reference packet defined in startPocket(. . .).
Return values:

- Success: Size of the compressed packet (bytes)
- Failure: -1 (Not enough memory available for the compressed packet)

Arguments:

- uint8_t * compressedPacket . . . pointer to the compressed packet (output).
- uint32_t * uncompressedPaket . . . pointer to the uncompressed packet (input).

Stop and terminate POCKET+

```
void stopPocket(void)
```

Listing 3.3: Memory release and Pocket+ termination function in compression.h

Description: Deallocate the dynamic memory which POCKET+ has used. It shall be called when POCKET+ has to be terminated.
Return values:

- None.

Arguments:

- None.

3.2.4 Allocated dynamic memory

For the primary compression process, the following dynamic memory buffers are required. Their size (bytes) is equal to the variable <referencePacketSize > defined in startPocket(...).

- static uint32_t * positive ... pointer to the positive mask.
- static uint32_t * mask ... pointer to the current mask.
- static uint32_t * oldPacket ... pointer to the old input packet.
- static uint32_t * newPacket ... pointer to the new input packet.

The implementation of the protection level leads to the following five additional buffers. For performance and configuration reasons they are integrated and part of the compression module. However, if the protection level is not desired, these buffers are also not required anymore.

- static uint16_t * protectionLevelBufferNegative ... negative deltas of the last n packets
- static uint16_t * protectionLevelBufferPositive ... positive deltas of the last n packets.
- static uint16_t * newDeltaBuffer ... buffer for the new deltas of the current packet
- static uint16_t * protectionLevelNumberOfDeltasNegative ... number of negative deltas (counted) for each of the last n packets.
- static uint16_t * protectionLevelNumberOfDeltasPositive ... number of positive deltas (counted) for each of the last n packets.

In order to determine the total allocated dynamic memory, the following formula can be used.

$$52 \cdot (< \text{referencePacketSize} > + < \text{maximumProtectionLevel} >) \tag{3.2}$$

Without the protection level the required memory would go down to

$$4 \cdot < \text{referencePacketSize} >. \tag{3.3}$$

The primary cause of this is that the process has to store all bit positions, which flipped in the previous n packets. For more information about the implementation of the protection level feature itself, refer to section 3.2.5.

3.2.5 Protection level

As already pointed out in the end of section 3.2.4, the protection level implementation requires approximately 50 times more memory then without. Before the implementation, two different implementation approaches were researched.

- Store the last n packets, process them again, and write these deltas to the compressed packet.
- Store the deltas of the last n packets and write them to the compressed packet.

Approach one would require n additional memory buffers for the last n packets. Each buffer would have the size <referencePacketSize >. Compared to method two, approach one requires less memory, however the packets have to be processed again. This approach mainly outsources the new task to the CPU, which would slow down the compression process by a factor of approximately n.
The second technique stores all deltas of the last n packets in new memory buffers. As OBCs are not allowed to allocate memory on the fly, the worst case scenario of required memory has to be allocated. Every single bit of the input packet flips, meaning that a delta for each bit is necessary. This means that for every possible bit position a 16 bit variable (max. packet size is 8.191 bytes) has to be allocated. The natural probability for this event is quite low, but an incorrect memory access from another process such as a wrong pointer could trigger this issue. To prevent an internal memory error, this case has to be taken into account. As faster processing is more important than memory conservation, method two was chosen.

Implementation of the protection level

The protection level is integrated within the function getMaskChanges(. . .) and is portrayed in figure 3.3. First, the initialization function startPocket(. . .) allocates the required memory. Once a packet is received, the total number of deltas are counted and written into the corresponding buffers. The process for new incoming packets is similar. The total number of new deltas are counted, the first delta of the previous deltas is converted into a wraparound and the entire delta field is shifted to the right. Last but not least, the new deltas are inserted into the leftmost position. Furthermore, to reuse the memory, new deltas can overwrite old delta values. The total number of deltas are stored in the buffer <protectionLevelNumberOfDeltasNegative > or <protectionLevel-NumberOfDeltasPositive >. This process is applied separately on both types, negative and positive deltas.

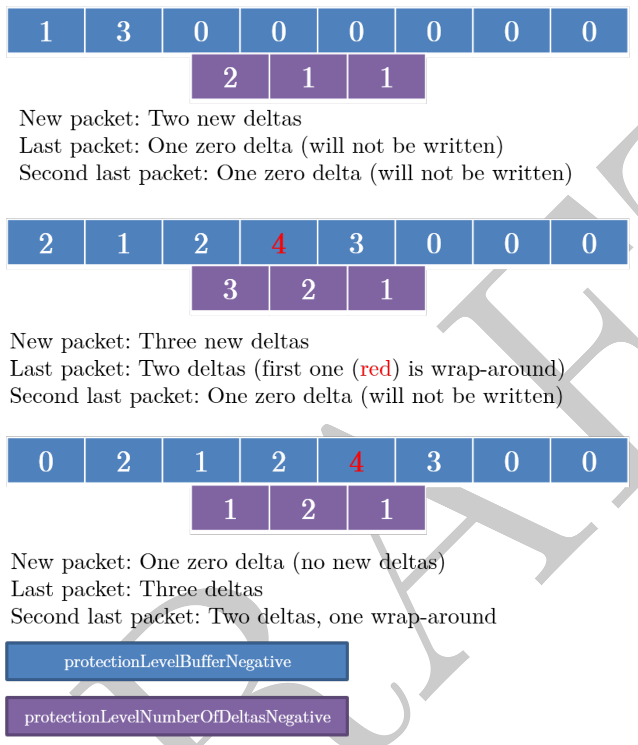

1 3 0 0 0 0 0 0
2 1 1
New packet: Two new deltas
Last packet: One zero delta (will not be written)
Second last packet: One zero delta (will not be written)
2 1 2 4 3 0 0 0
3 2 1
New packet: Three new deltas
Last packet: Two deltas (first one (red) is wrap-around)
Second last packet: One zero delta (will not be written)
0 2 1 2 4 3 0 0
1 2 1
New packet: One zero delta (no new deltas)
Last packet: Three deltas
Second last packet: Two deltas, one wrap-around
protectionLevelBufferNegative
protectionLevelNumberOfDeltasNegative

Figure 3.3: Implementation and demonstration of the buffers used for the protection level.

3.3 Decompression (C++)

Since the decompression module will not run on a S/C, but rather on ground hardware, C++ was chosen as the programming language. This allows future programmers to more easily add additional functionalities such as a graphical user interface or third party libraries.

3.3.1 Software architecture

The decompression module consists of the class *PocketDecompression*, which is located in *decompression.hpp*. It contains the API functions

- decompressPacket(. . .),
- and getDecompressedPacket(. . .).

The according private functions are

- recoverMask(. . .),
- addNpBits(. . .),
- and getValueFromCompressedPacket(. . .).

In figure 3.4 the corresponding function call diagram is shown.
Similar to the compression module, a packetizer is necessary to handle the input packets. Then, the packetizer processes and forwards the desired compressed data into Pocket+. This is demonstrated in figure 3.5. The decompression is done on byte level and processes an input packet until the end of the compressed packet has been reached. Thus the size of the compressed packet is not required, only the size of the reference/output packet must be known.

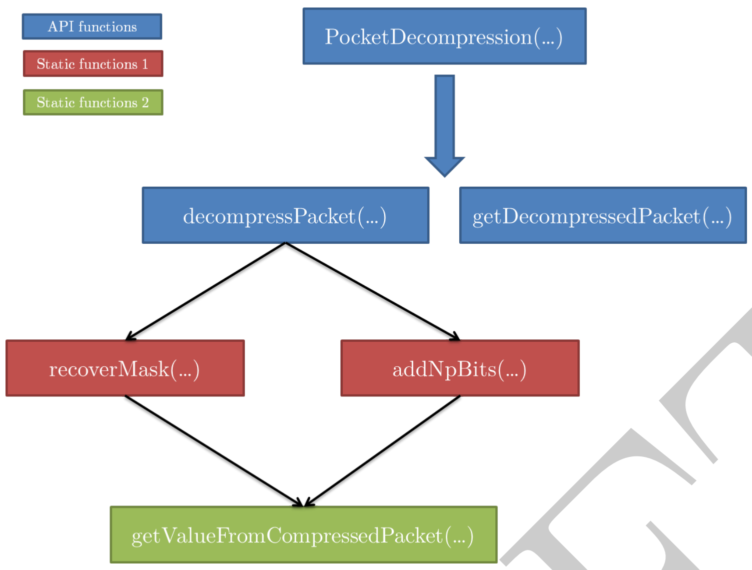


Figure 3.4: Function call diagram for the C++ decompression class.

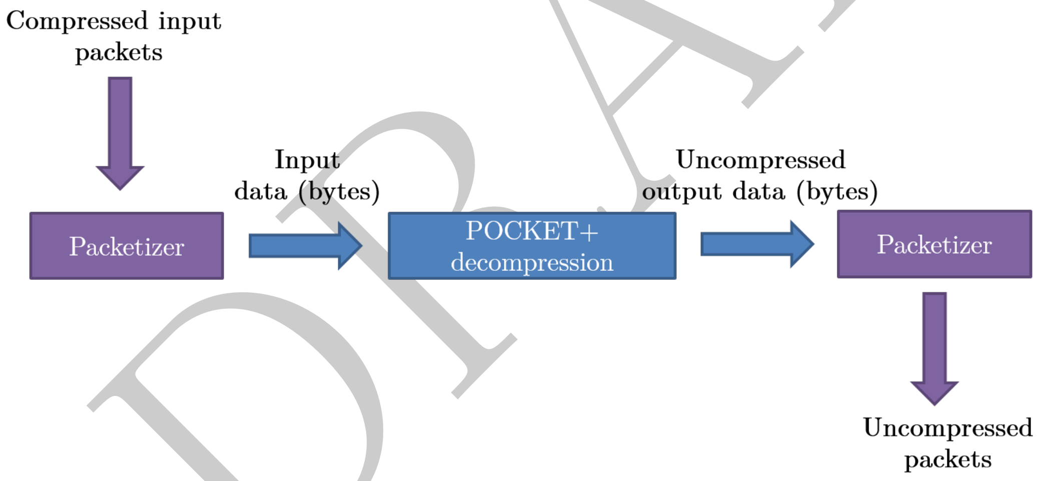


Figure 3.5: Software architecture for the decompression process.

3.3.2 Application Programming Interface (API)

The following functions shall be used by any user or developer to interact with the decompression module of Pocket+.

Initialize the decompression (class constructor)

```
PocketDecompression(uint8_t * referencePacket, uint_fast16_t
    referencePacketSize)
```

Listing 3.4: Constructor for the POCKET+ decompression in decompression.hpp

Description: Allocate the required memory and initialize variables.
Return values:

- None (Constructor)

Arguments:

- uint8_t * referencePacket . . . pointer to the reference packet.
- uint_fast16_t referencePacketSize . . . size of the reference / original packets (bytes).

Decompress the input data

```
uint_fast16_t decompressPacket(uint8_t * inputPacket)
```

Listing 3.5: Function used to decompress data.

Description: Decompress a compressed input packet (8 bit) and return the size of the compressed input packet. The input data will be processed until the end of the input packet has been reached. Due to safety reasons, the input packet should be terminated with at least three zero bytes.
Return values:

- Success: Size of the compressed packet.
- Failure: Not available.

Arguments:

- uint8_t * inputPacket . . . pointer to the compressed input packet.

Output the decompressed packet

```
void getDecomressedPacket(uint_fast8_t *
    decompressedPacketOutputBuffer)
```

Listing 3.6: Function used to output the decompressed data.

Description: Output the latest decompressed packet.
Return values:

- None

Arguments:

- uint8_t * decompressedPacketOutputBuffer . . . pointer to the compressed output buffer. The uncompressed packet will be copied to that location.

3.3.3 Allocated dynamic memory

For the decompression process the following two dynamic memory buffers are required.

- uint8_t * mask . . . current mask,
- uint8_t * decompressedPacket . . . pointer to the decompressed packet.

Their total size can be calculated with formula 3.4.

$$\text{dynamic memory} = <\text{referencePacketSize}> \cdot 2 \tag{3.4}$$

3.4 OPS-SAT

3.4.1 Introduction - The mission

ESA's and TUG's new cube-sat named OPS-SAT will test and validate new techniques in mission control and on-board systems. OPS-SAT is devoted to demonstrate drastically improved mission control capabilities that will arise when satellites can fly more powerful on-board computers. It is 30 cm high but comprises an experimental computer that is ten times more powerful than any current ESA spacecraft. In general, it is very difficult to perform live testing in the domain of mission control systems. No-one wants to take any risk with an existing, valuable satellite, so it is very difficult to test new procedures, techniques or systems in orbit. The OPS-SAT solution is to design a low-cost satellite that is rock-solid safe and robust even if there are any malfunctions due to testing. The robustness of the basic satellite itself, will give ESA flight control teams the confidence they need to upload and try out new, innovative control software submitted by experimenters; the satellite can always be recovered if something goes wrong. The outcome is an open, flying laboratory that will be available for in-orbit demonstration of new control systems and software that would be too risky to attempt on a real satellite. Over 100 companies and institutions from 17 European countries have registered experimental proposals to fly on OPS-SAT. ESA, 2017

3.4.2 System description

For OPS-SAT, it was decided that only one specific CCSDS SID packet will be compressed. The compression itself is done by the Pocket+ compression module, which can be controlled by the OBSW through a given API. The output of the Pocket+ compression module shall be handled, processed and transmitted like an ordinary CCSDS SID packet. Compressed packets can be distinguished from uncompressed packets by their Pocket+ APID. Furthermore, several new TCs and adaptions of the current MCS are required to control Pocket+ as S/C operator.

3.4.3 Compression module for the OBSW

In 2017, ESA decided that Pocket+ will be integrated into the OBC of OPS-SAT. The OBC used is a Nanomind A3200, which is comprised of

- an AVR32 MPU (8 MHz . . . 64 MHz),
- 512 kB of build-in flash / 128 MB of NOR flash,
- and 32 MB of SDRAM.

In order to integrate the Pocket+ module into a S/C OBSW, several adaptations and new features of the generic version of Pocket+ are required. These include

- static memory allocation,
- a state-machine,
- a configuration data pool and a
- API for the OBSW.

All these modifications and implementations are explained in detail and are covered in the following subsections.

Resources management: Memory budget

According to [ECSS-E-HB-40A, 2013, p. 160], the following verification methods must be used for memory budget analysis.

- Verify that the budget presented is based on compiled object files.
- In addition, the verification of the volatile memory budget intends, at CDR and following milestones, to verify that the stack allocation has been analyzed based on the call graph and pre-emption phenomena and that a margin for it is specified.
- Generally, for flight software there is no dynamic memory mechanism implementation (state of the art)

For this reason all dynamic memory buffers are replaced by static memory buffers whose size can be chosen at compile time. Within the compression module, which is located in *compression.c*, the following memory buffers are defined.

- static uint32_t mask[WORDS]

- static uint32_t positive [WORDS]
- static uint32_t newPacket[WORDS]
- static uint32_t oldPacket[WORDS]
- static uint16_t protectionLevelBufferNegative[WORDS $\cdot 32 + 5$]
- static uint16_t protectionLevelBufferPositive[WORDS $\cdot 32 + 5$]
- static uint16_t newDeltaBuffer[WORDS $\cdot 32 + 5$]
- static uint16_t protectionLevelNumberOfDeltasNegative[5]
- static uint16_t protectionLevelNumberOfDeltasPositive[5]

Since Pocket+ can only handle 32 bit input, one additional buffer, which stores the 32 bit conversion of the 8 bit input, is required. It is located within the static structure of PACKETISER_CONFIGURATION in *pocket.c*.

- static uint32_t POCKET_INPUT_DATA_POINTER[WORDS]

The variable WORDS can be calculated with equation 3.5 and the total size of the required static memory buffers (bytes) can be determined with equation 3.6 .

$$\text{WORDS} = \left\lfloor \frac{\text{MAX\_PACKET\_INPUT\_SIZE} + 3}{4} \right\rfloor \tag{3.5}$$

Where MAX_PACKET_INPUT_SIZE is a given preprocessor-define at compile time, which defines the entire CCSDS input packet size (header + data field). When no preprocessor-define argument is used, the default value is set to 200.

$$\text{static memory} = 212 \cdot \text{WORDS} + 50 \tag{3.6}$$

The compiling was done with the avr32-gcc (4.4.7) compiler and the following configuration:

- –std=c99 ... use the c99 standard,
- -Os ... optimise the code for size,
- -c ... compile, but don't link,
- -g ... add debugging information,
- -fno-exceptions ... no exceptions,
- -ffunction-sections ... put each function in a seperate section,
- -masm-addr-pseudos ... output the pseudo instructions (AVR specific)
- -mpart=uc3c0512c ... name of the MCU
- -DMAX_PACKET_INPUT_SIZE=200 ... maximum size of the input CCSDS packet (Pocket+ specific)

The results of the object files memory analysis are shown in table 3.1. The stack analysis is based on a typical configuration where only the mode ACTIVE is enabled. Two function call diagrams, which were determined with the tools *valgrind* and *kcachegrind* are shown in figure 3.6 (data compresses well) and figure 3.7 (data compresses badly). In these two figures one can see which functions are called, how often, and how much runtime they consume relative to each other. The worst case stack usage was estimated with the tool *nm (list symbols from object files)* provided by AVR and sums up to **3,192 bytes** including a safety factor of 1.5.

Table 3.1: Overview of the memory budget of the individual object files when MAX_PACKET_INPUT_SIZE is set to 200 bytes.

Object file	**text (bytes)**	**data (bytes)**	**bss (bytes)**
packetiser.o	232	0	0
compression.o	2,204	36	10,268
pocket.o	1,068	10	232
total	3,504	46	10,500

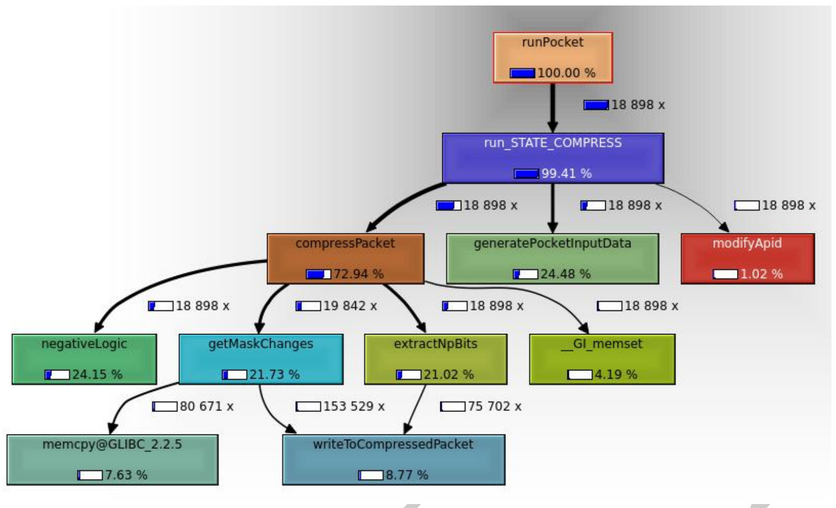


Figure 3.6: Function call diagram of TM data that compresses well. Figure created with valgrind and kcachegrind.

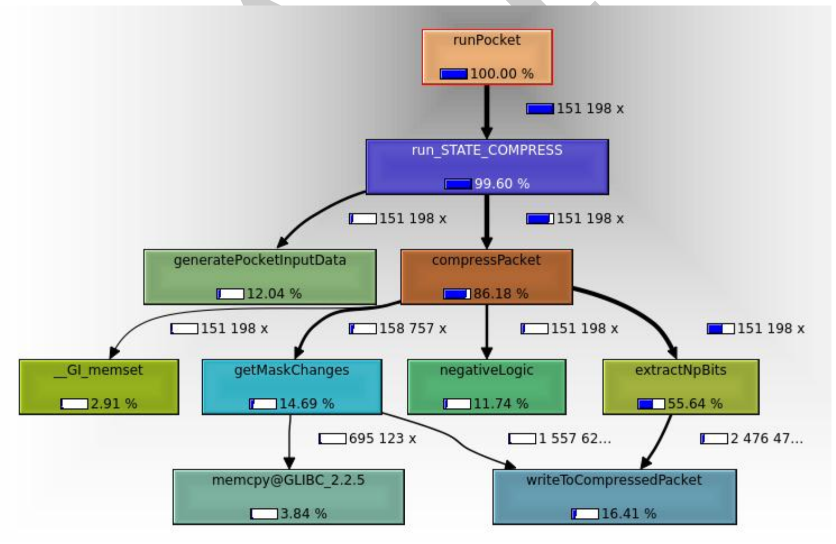


Figure 3.7: Function call diagram of TM data that compresses badly. Figure created with valgrind and kcachegrind.

Statemachine: Modes and States

In order to control Pocket+ with the OBSW through the given API, the compression and packetizer modules are embedded into a state machine, which has three states, or more specifically four modes.

1. STANDBY: Do not compress. Output and input packet is the same.
2. REFERENCE: Do not compress, but change the APID of the input packet.
3. COMPRESS
    - ABSOLUTE: Compress and include the anchor deltas.
    - ACTIVE: Compress and do not include anchor deltas.

Every state is also a mode except for the state COMPRESS, which is split up into the two sub-modes ABSOLUTE and ACTIVE. All state transistions are done automaticly in the background, forced state transitions are not possible. These automatic transitions are done by comparing the current mode with the target mode, which can be set with the OBSW via the Pocket+ API. As long as the current mode is not the target mode, the statemachine goes to the next mode. When both parameters are equal, the transitions will stop. However, a compression error is the only exception, which would cause a transition from COMPRESS back to REFERENCE. This transition can bee seen as a reset loop. When a compression error occurs, Pocket+ goes into REFERENCE and then back into COMPRESS. One can see the interactions between the OBSW and Pocket+ in table 3.2. The architecture of the statemachine is shown in figure 3.8. Examples for transitions are given afterwards.

Table 3.2: Operational modes and interactions between the OBSW and Pocket+. The OBSW only has to call runPocket() and check the output buffer. The configuration data pool has to be initialized before.

Mode name	**OBSW**	**Pocket+ and mode description**
STANDBY	Copy the CCSDS SID packet into the input buffer and call runPocket(). Use the packet in the output buffer for further processing.	Copy the input packet into the output buffer and return the size of the packet (header + data field).
REFERENCE	Copy the CCSDS SID packet into the input buffer and call runPocket(). Use the packet in the output buffer for further processing.	Run the Pocket+ logic, copy the input packet into the output buffer, modify the APID and return the size of the packet (header + data field).
ABSOLUTE	Copy the CCSDS SID packet into the input buffer and call runPocket(). Use the packet in the output buffer for further processing.	Run the Pocket+ compression and add anchor deltas. Copy the original CCSDS header and the compressed packet into the output buffer, modify the APID and the data field length and return the size of the compressed packet (header + data field)
ACTIVE	Copy the CCSDS SID packet into the input buffer and call runPocket(). Use the packet in the output buffer for further processing.	Run the Pocket+ compression. Copy the original CCSDS header and the compressed packet into the output buffer, modify the APID and the data field length and return the size of the compressed packet (header + data field)

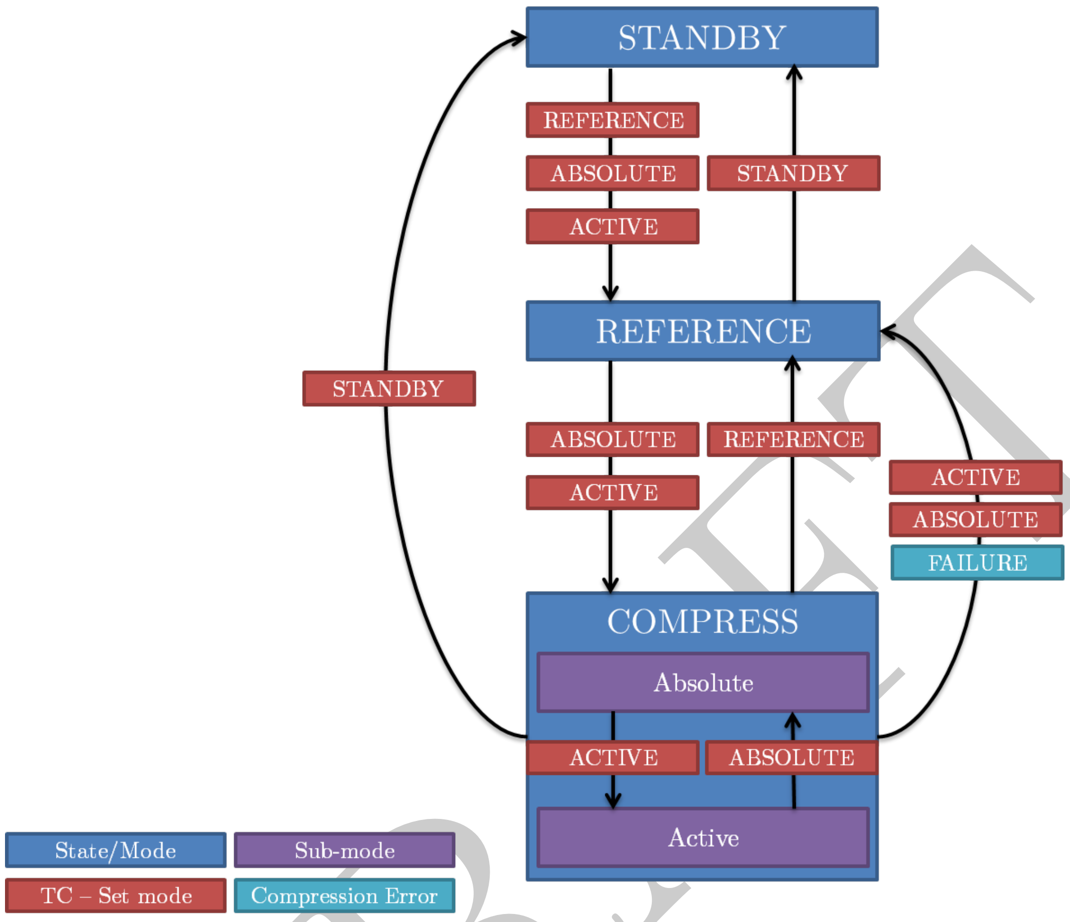


Figure 3.8: Statemachine of the Pocket+ module.

Example 1: The statemachine is in state/mode STANDBY and the API method setMode(ACTIVE) is called.

1. Finish the current iteration in state/mode STANDBY.
2. Go to state/mode REFERENCE and stay there for one iteration.
3. Transit to state COMPRESS and go into mode ABSOLUTE. Stay there for one iteration.
4. Stay in COMPRESS and go into mode ACTIVE.

Example 2: The statemachine is in mode ACTIVE and receives a TC that it should go to mode ACTIVE. This command shall be used when the decompression fails due to packet loss.

1. Finish the current iteration in mode ACTIVE.
2. Go to state/mode REFERENCE and stay there for one iteration.
3. Transit to state COMPRESS and go into mode ABSOLUTE. Stay there for one iteration.
4. Stay in COMPRESS and go into mode ACTIVE.

Example 3: The statemachine is in mode ACTIVE and the compression fails.[1]

1. Finish the current iteration in mode ACTIVE.
2. Go to state/mode REFERENCE and stay there for one iteration.
3. Transit to state COMPRESS and go into mode ABSOLUTE. Stay there for one iteration.
4. Stay in COMPRESS and go into mode ACTIVE.

[1]Instead of the compressed packet, the original packet is copied to the output memory.

Control Pocket+: Required Telecommands (TC)

Six telecommands are required to control Pocket+ and are listed in table 3.3. Every TC refers to a respective API method, which shall be called after receiving it.

Example 1:

1. MC sents the TC to change the mode of Pocket+.
2. The OBSW calls the API method setMode(newValue)

Example 2:

1. MC sents the TC to change the positive update rate of Pocket+.
2. The OBSW calls the API method setPositiveUpdateRate(newValue).

Table 3.3: List of TCs required to use the Pocket+ compression module with the given API

TC Name	**Possible values**
Set target mode	0 = STANDBY 1 = REFERENCE 2 = ABSOLUTE 3 = ACTIVE
Set positive update rate	6 . . . 32.767
Set maximum protection level used	1 . . . 5
Set new APID for Pocket+ packets	0 . . . 2047 (11 bits)
Set a new non-predictable word offset	0 (default) . . . WORDS
Activate 16 bit CRC	0 = OFF 1 = ON

Configure Pocket+: Configuration data pool

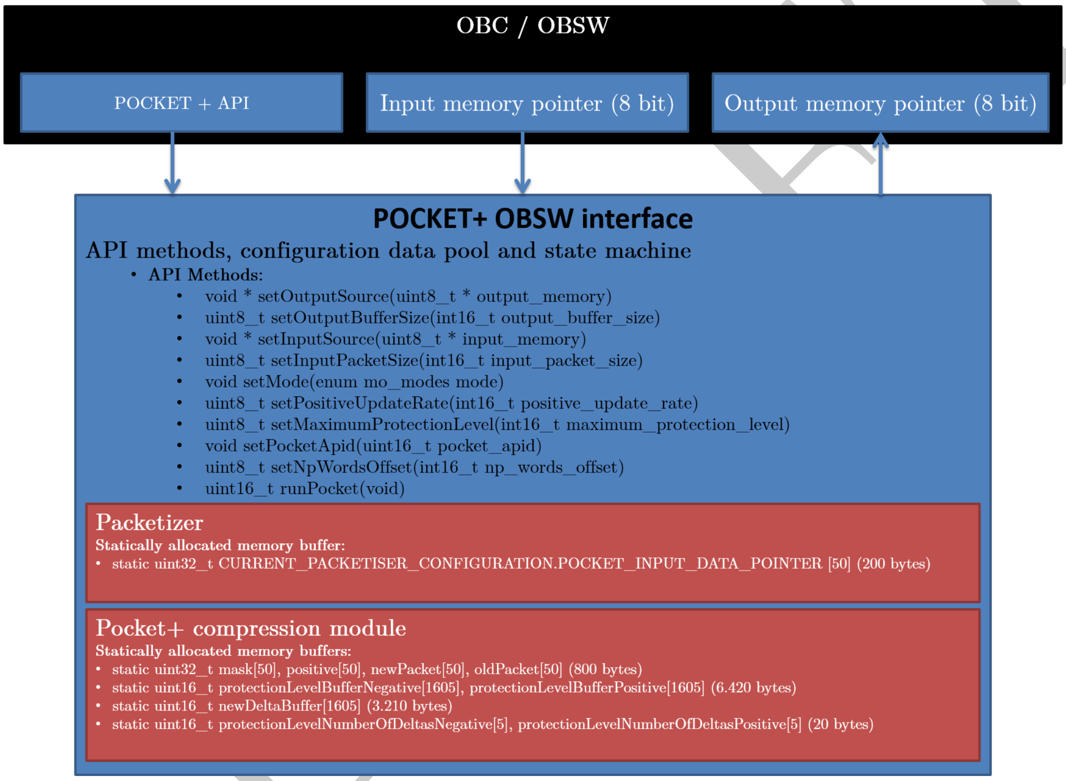
The configuration data pool is located in pocket.c and uses the following static structures.

1. struct STATEMACHINE_CONFIGURATION
   - enum state_names CURRENT_STATE;
   - enum state_names TARGET_STATE;
   - enum mo_modes TARGET_MODE;
   - uint8_t IDENTIFY;
2. Struct packetizer_CONFIGURATION
   - uint8_t * INPUT_PACKET_POINTER
   - int16_t INPUT_PACKET_SIZE;
   - uint8_t * OUTPUT_PACKET_POINTER;
   - int16_t OUTPUT_BUFFER_SIZE;
   - int16_t OUTPUT_PACKET_SIZE;
   - uint32_t POCKET_INPUT_DATA_POINTER[WORDS];
   - uint16_t POCKET_APID;
3. Struct COMPRESSION_CONFIGURATION
   - int16_t POSITIVE_UPDATE_RATE
   - int16_t ANCHOR_UPDATE_RATE
   - int16_t MAXIMUM_PROTECTION_LEVEL
   - int16_t NP_WORDS_OFFSET
   - int16_t SIZE_POCKET_INPUT_DATA

Most of these configurations can be set through the API. **In particular the I/O buffers INPUT_PACKET_POINTER and OUTPUT_PACKET_POINTER and their sizes INPUT_PACKET_SIZE and OUTPUT_BUFFER_SIZE must be set before. Otherwise Pocket+ will not produce any output**.

Interact with Pocket+: The OBSW API

The configuration data pool represents the current status of Pocket+ and most of the parameters can be configured via the Pocket+ API. Figure 3.9 illustrates the system architecture and shows the size of the static memory buffers when MAX_PACKET_INPUT_SIZE is set to 200. All API functions are covered in detail after the figure. **Functions marked with (R) are required** for proper configuration, meaning that they must be called before the actual compression process. Functions marked with (O) are optional and can be used to change parameters within the configuration data pool.


OBC / OBSW
POCKET + API
Input memory pointer (8 bit)
Output memory pointer (8 bit)
POCKET+ OBSW interface
API methods, configuration data pool and state machine
Packetizer
Pocket+ compression module

- API Methods:
  - void * setOutputSource(uint8_t * output_memory)
  - uint8_t setOutputBufferSize(int16_t output_buffer_size)
  - void * setInputSource(uint8_t * input_memory)
  - uint8_t setInputPacketSize(int16_t input_packet_size)
  - void setMode(enum mo_modes mode)
  - uint8_t setPositiveUpdateRate(int16_t positive_update_rate)
  - uint8_t setMaximumProtectionLevel(int16_t maximum_protection_level)
  - void setPocketApid(uint16_t pocket_apid)
  - uint8_t setNpWordsOffset(int16_t np_words_offset)
  - uint16_t runPocket(void)

Statically allocated memory buffer:
- static uint32_t CURRENT_PACKETISER_CONFIGURATION.POCKET_INPUT_DATA_POINTER [50] (200 bytes)

Statically allocated memory buffers:
- static uint32_t mask[50], positive[50], newPacket[50], oldPacket[50] (800 bytes)
- static uint16_t protectionLevelBufferNegative[1605], protectionLevelBufferPositive[1605] (6.420 bytes)
- static uint16_t newDeltaBuffer[1605] (3.210 bytes)
- static uint16_t protectionLevelNumberOfDeltasNegative[5], protectionLevelNumberOfDeltasPositive[5] (20 bytes)

Figure 3.9: Interface between the OBSW and Pocket+. The pre-proceser define MAX_PACKET_INPUT_SIZE is set to 200.

Set the address of the output buffer. (R)

```
void *setOutputSource(uint8_t * output_memory);
```

Listing 3.7: This function shall be used to set the address for the output buffer.

Description: Set the address of the output buffer and return the address.
Return values: Address of the output buffer.
Arguments:

- uint8_t * output_memory . . . pointer to the output buffer.

Set the size of the output buffer. (R)

```
uint8_t setOutputBufferSize(int16_t output_buffer_size);
```

Listing 3.8: This function shall be used to set the size of the output buffer.

Description: Set the size (bytes) of the output buffer. The size must be greater than the input packet size.
Return values:

- SUCCESS: 0
- FAILURE: 1

Arguments:

- int16_t output_buffer_size . . . size of the output buffer. (bytes)

Set the adress of the input buffer (R)

```
void *setInputSource(uint8_t * input_memory);
```

Listing 3.9: This function shall be used to set the address for the input buffer.

Description: Set the address of the input buffer and return the address.
Return values: Address of the input buffer.
Arguments:

- uint8_t * input_memory . . . pointer to the input buffer.

Set the size of the input CCSDS packet (header + data field) (bytes) (R)

```
uint8_t setInputPacketSize(int16_t input_packet_size);
```

Listing 3.10: This function shall be used to set the size of the input CCSDS packet.

Description: Set the size (bytes) of the CCSDS packet (header + data field), which should be processed. The size must be greater than six bytes and smaller than the output buffer size.
Return values:

- SUCCESS: 0
- FAILURE: 1

Arguments:

- int16_t input_packet_size . . . size of the input CCSDS packet within in the input buffer (bytes).

Set or change the Pocket+ APID (O)

```
void setPocketApid(uint16_t pocket_apid);
```

Listing 3.11: Function which sets the APID for the Pocket+ specific CCSDS packet.

Description: Set the APID for Pocket+ packets. The default value is zero.
Return values: None
Arguments:

- uint16_t pocket_apid . . . value for the Pocket+ APID

Set the target mode of the statemachine of Pocket+ (O)

```
void setMode(enum mo_modes mode);
```

Listing 3.12: This function shall be used to set the target mode of the statemachine.

Description: Set the target mode of Pocket+. The statemachine will do the state transitions automaticly. The default (entry state/mode) is STANDBY.
Return values: None
Arguments:

- enum mo_modes mode . . . name or value of the target mode
  - 0 . . . STANDBY
  - 1 . . . REFERENCE
  - 2 . . . ABSOLUTE
  - 3 . . . ACTIVE

Set or change the positive update rate (O)

```
uint8_t setPositiveUpdateRate(int16_t positive_update_rate);
```

Listing 3.13: This function shall be used to set the positive update rate of Pocket+.

Description: Set the positive update rate of Pocket+. This value must be greater than value for the maximum protection level. The default value is 20.
Return values:

- SUCCESS: 0
- FAILURE: 1

Arguments:

- int16_t positive_update_rate . . . value for the positive update rate

Set or change the maximum protection level (O)

```
uint8_t setMaximumProtectionLevel(int16_t
    maximum_protection_level);
```

Listing 3.14: This function shall be used to set the maximum protection leve of Pocket+.

Description: Set the maximum protection level of Pocket+. This value has to be between one and five, and must be less than the positive update rate. The default value is three.
Return values:

- SUCCESS: 0
- FAILURE: 1

Arguments:

- int16_t maximum_protection_level . . . value for the positive update rate

Set or change the offset for the non-predictable bits (O)

```
uint8_t setNpWordsOffset(int16_t np_words_offset);
```

Listing 3.15: Function which sets the offset (32 bit words) for the non-predictable bits.

Description: Set the offset (32 bit words) for the non-predictable bits. A offset of one means that the first 32 bits of a packet are not processed by the non-predictable bit extraction process. The default value is zero. The value must no be greater than the variable WORDS (equation 3.5)
Return values:

- SUCCESS: 0
- FAILURE: 1

Arguments:

- int16_t np_words_offset . . . value for the non-predictable bits offset

Run the statemachine, process the input packet, and issue the output packet. (O)

```
uint16_t runPocket(void);
```

Listing 3.16: Function which processes the input buffer and writes the corresponding data to the output buffer.

Description: When Pocket+ has been configured, this function has to be called in order to output any data. It comprises the statemachine, the packetizer, and the compression module. The return value is the size of the CCSDS packet within the output buffer. When the configuration is wrong, or has not been done, it returns zero.
Return values:

- SUCCESS: Size of the CCSDS (header + data field) (bytes) packet within the output buffer.
- FAILURE: 0

Arguments: None

Usage: Files and code example

The Pocket+ compression module consists of three file couples.

1. The interface between the OBSW and Pocket+,
   - pocket.c and pocket.h
2. the packetizer module,
   - packetizer.c and packetizer.h
3. and the compression module.
   - compression.c and compression.h

In order to use Pocket+ in a given project, only the header file pocket.h has to be included. An example is given in listing 3.17.

```
#include "pocket.h"

static uint8_t input_memory[200]; //input memory
static uint8_t output_memory[800]; //output memory

int main(){

  /* Set the address and size of the output memory. REQUIRED */
  setOutputSource(output_memory);
  if(setOutputBufferSize(800)){
    return 1; // could not set the size of the output buffer
  }

  /* Set the address of the input memory. REQUIRED */
  setInputSource(input_memory);

  /* Set the size of the CCSDS packet inside the input memory.
   REQUIRED */
  if(setInputPacketSize(128)){
    return 1; // could not set the size of the input packet
   size
  }

  /* Set the APID for the Pocket+ CCSDS packet. The default
   value is zero. */
  setPocketApid(1365);

  /* BASIC SETUP DONE. Now Pocket+ produces an output. */

  /* +++ Let's do some configuration! +++ */

  /* Set the mode to REFERENCE. Default is STANDBY. */
  setMode(REFERENCE);

  /* Set the maximum protection level to 3. */
  setMaximumProtectionLevel(3);

  /* Process the input packet. */
  int16_t outputSize = runPocket();
  if(outputSize){ // size (bytes) of the CCSDS SID packet
   within output_memory.
    /* Do something with the output_memory  */
  }
  /* New packet received. Process the input packet */
  int16_t outputSize = runPocket();
  if(outputSize){
    /* Do something with the output_memory */
  }

  /*
    .
    .
  */

  setMode(ACTIVE); // A TC has been received! Go into mode
    ACTIVE

  /* New packet received. Process the input packet */
  int16_t outputSize = runPocket();
  if(outputSize){
    /* Do something with the output_memory */
  }

  /* ... thats all you have to know! */
}
```

Listing 3.17: An example how to use the compression module.

3.4.4 Decompression module for Mission Control Systems

The implementation of Pocket+ into the OPS-SAT OBC has an impact on the current mission control system and requires new specifications. SCOS2000, which is the main mission control system used by ESA, has no functionality to support compressed packets currently. Thus the following two options were researched in order to integrate the Pocket+ decompression functionality into the mission control system. To keep the system as simple as possible, it was decided to use the APID of a SPP packet to identify compressed Pocket+ packets.

Decomression module before SCOS2000

When no modification of the current MCS is desired, only a Pocket+ module before SCOS2000 would give the desired decompression functionality. This concept has to deframe the input frame, look and filter for compressed SPP packets, decompress these packets and create two new TM frames. Whereas Pocket+ packets get their own virtual channel and forward them to SCOS2000. Both types of packets are stored in SCOS2000. This concept is shown in figure 3.10.

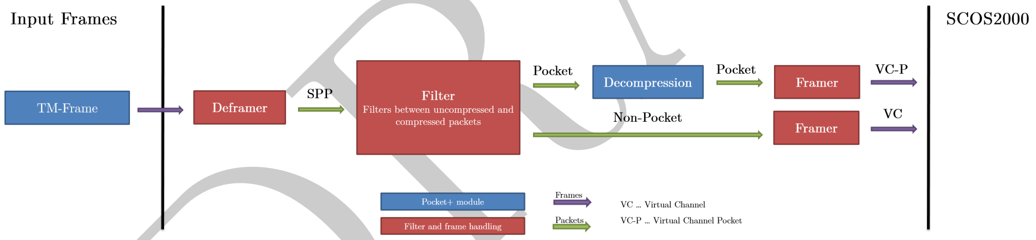


Figure 3.10: The incoming TM frame is deframed and the SPP packets go into a filter. The filter looks for Pocket+ packets and decompresses them. Afterwards, two new frames are created. One for normal TM and one for Pocket+ TM.

Decompression module inside SCOS2000

Incoming CCSDS packets, which are recognised as Pocket+ packets, go directly into the TM packets database table for compressed packets. The decompression module has access to this table and decompresses these packets. After the decompression process, the APID is reversed to the original APID and the uncompressed packet is sent to the database table for uncompressed packets. Figure 3.11 shows this mission operations extension for Pocket+.

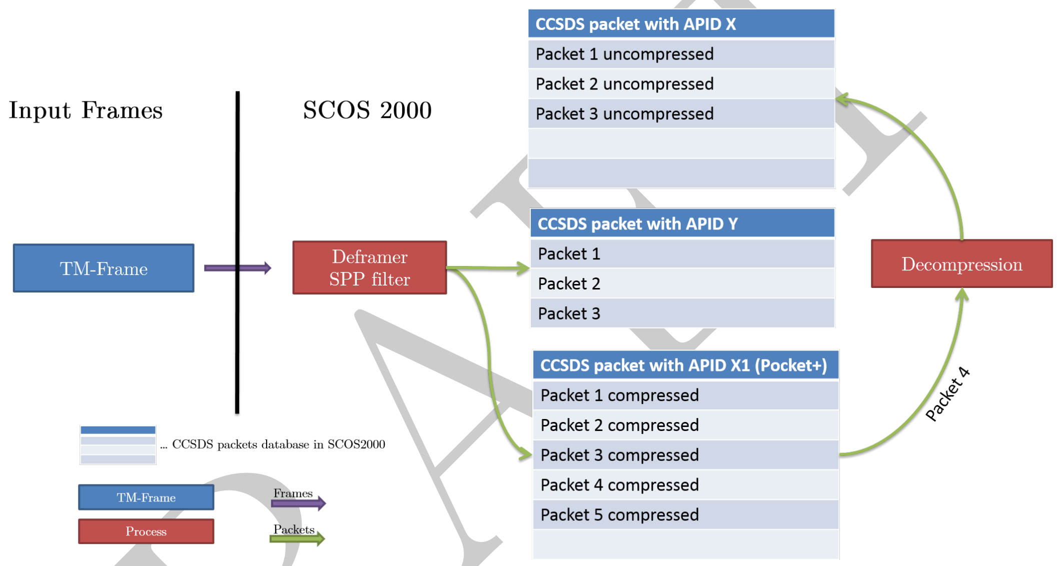


Figure 3.11: All SPP packets are stored into their corresponding database tables. The packets can be distinguished by their APIDs. All packets with Pocket+ specific APIDs go into the decompression module, the APIDs are reverted and the resulting uncompressed packets are stored into their respective database table.

4 Results and conclusions

4.1 Evaluation of the Pocket+ compression

In order to evaluate Pocket+, the execution time and the compression were compared to other compression algorithms such as LZ4, Gzip, and bzip. Real TM data of Venus Express (VEX), GAIA, Rosetta, GOCE and Herschel was used as input data. Furthermore, the algorithm was also tested on RAW pictures of the OPS-SAT camera. The test setup was

- a 64 bit Linux machine - Debian testing with kernel 4.9.0-3,
- 4.0 GB of RAM,
- Intel i7 CPU with 1.60 GHz and
- a 120 GB SSD.

The following subsections deal with finding the optimal configuration for best compression results and compare Pocket+ with other compression algorithms in terms of compression and execution time. The compression itself is defined as

$$\text{Compression (\%)} = \frac{\text{Compressed size}}{\text{Uncompressed size}}. \tag{4.1}$$

4.1.1 Finding the best compression configuration

In order to obtain the best compression, the parameter maximum protection level must be set to one, and the parameter positive update rate has to be varied. The value of the positive update rate depends on the dynamic properties and sampling rate between consecutive packets. For example, when the data of packets tend to follow a fast trend and the sampling rates are low, some bits, which flipped in the the last ten packets, will not change anymore. A positive update value of ten removes these bits from the mask and the non-predictable bits field in the compressed packet. However, one can see in figure 4.1 that for most of the tested TM data the compression fluctuates slightly when a positive update rate between 5 and 40 was chosen, which turned out to be a good initial guess.

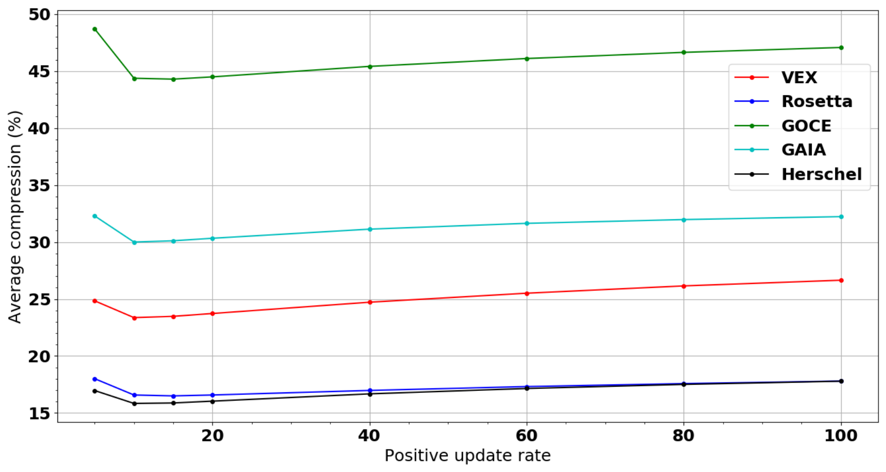


Figure 4.1: Compression values of real TM data of several space missions when the positive update rate is changed and the protection level is set to one. Only the data field of a CCSDS packet was compressed.

4.1.2 The influence of the protection level

In figure 4.2 the average compression of several space mission is shown when the parameter maximum protection level is changed and the parameter positive update rate is set to 15. It can be seen that the compression increases linearly, which results from the additional deltas of the previous n packets.

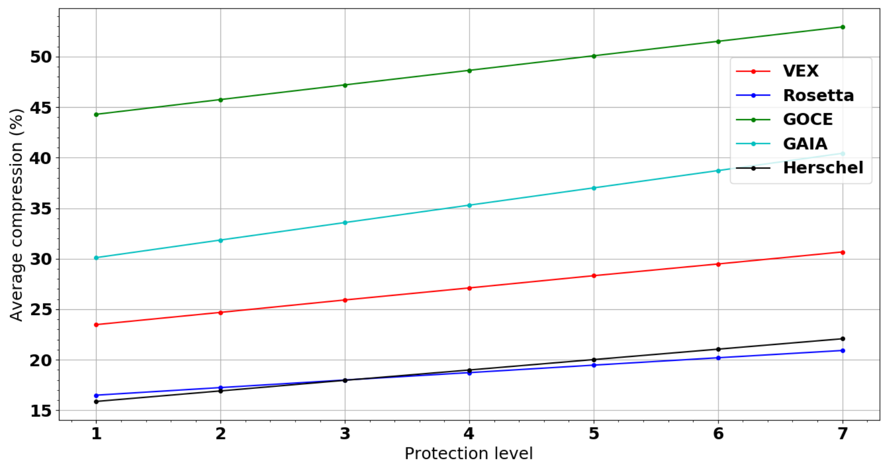


Figure 4.2: Compression of real TM data of several space missions when the positive update rate is set to 15 and the protection level is changed. Only the data field of a CCSDS packet was compressed.

4.1.3 Comparison with other compression algorithms

The following generic compression algorithms were chosen for compression, since all of them are open source and widely used. For this purpose, several CCSDS binary files full of CCSDS packets (headers and data fields) were compressed. Pocket+ used a maximum protection level of one and positive update rate of 15.

- LZ4 (fast mode),
- Gzip (fast mode),
- bzip2 (fast mode).

In the following tables the results of the average compression and execution times of several space missions are shown.

Table 4.1: Comparison of several compression algorithms for telemetry data of Venux Express. 132.50 MB were processed.

Algorithm	**Average compression (%)**	**Average runtime (s)**
Pocket+	19.30	3.84
LZ4	33.70	1.91
Gzip	26.06	3.61
Bzip2	19.84	24.68

Table 4.2: Comparison of several compression algorithms for telemetry data of Rosetta. 55.60 MB were processed.

Algorithm	**Average compression (%)**	**Average runtime (s)**
Pocket+	13.89	1.45
LZ4	22.67	0.71
Gzip	17.73	1.46
Bzip2	12.00	11.26

Table 4.3: Comparison of several compression algorithms for telemetry data of GOCE. 988.50 MB were processed.

Algorithm	Average compression (%)	Average runtime (s)
Pocket+	39.79	34.91
LZ4	64.93	9.81
Gzip	55.82	38.34
Bzip2	52.58	204.89

Table 4.4: Comparison of several compression algorithms for telemetry data of GAIA. 346.7 MB were processed.

Algorithm	Average compression (%)	Average runtime (s)
Pocket+	27.36	8.14
LZ4	45.66	1.79
Gzip	38.26	8.34
Bzip2	30.75	59.74

Table 4.5: Comparison of several compression algorithms for telemetry data of Herschel. 763.8 MB were processed.

Algorithm	Average compression (%)	Average runtime (s)
Pocket+	14.52	13.01
LZ4	28.84	5.79
Gzip	23.14	17.24
Bzip2	14.91	127.97

4.1.4 A look on Venus Express

Venus Express was launched in 2005, orbited Venus in a highly elliptical 24:00 h polar orbit, and observed the atmosphere. Pocket+ was applied on TM data generated in a week, and used a positive update rate of 15 and a maximum protection level of three.

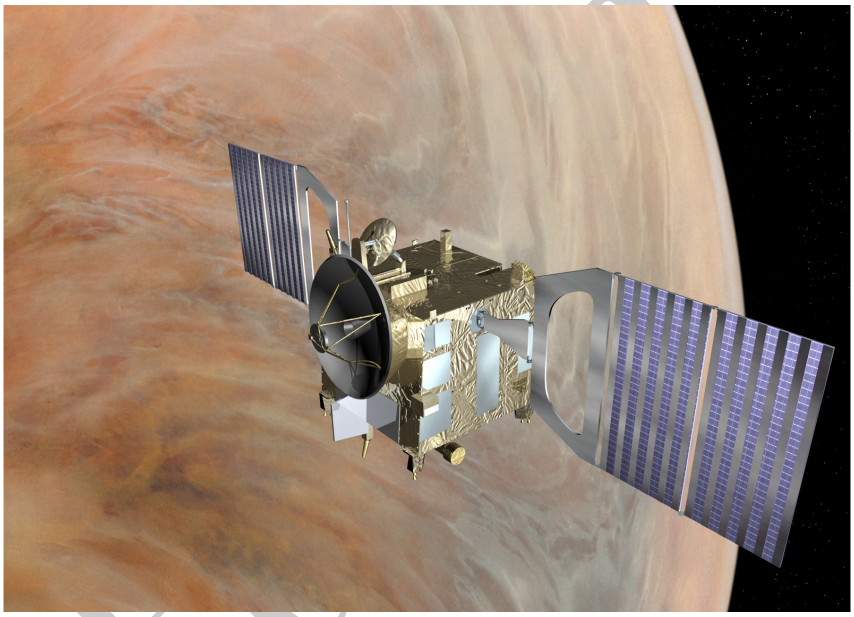


Figure 4.3: Artist's impression of Venus Express orbiting Venus. ESA, 2003

TM data, which compresses well

Figure 4.4, 4.5 and 4.6 show the compression values of every single packet, the likelihood of these compression values and the number of non-predictable bits for TM data which compresses well. The CCSDS packet was sampled with a rate of 32.0 s and has a data field size of 120.0 bytes.

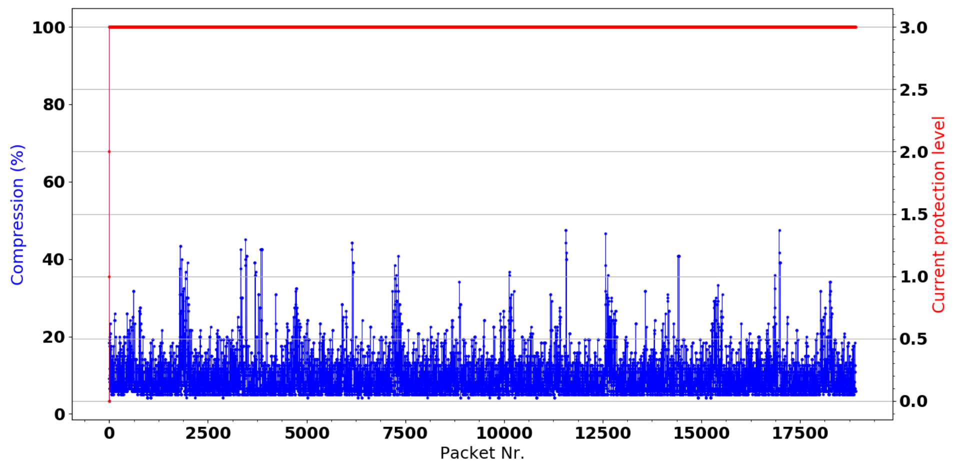


Figure 4.4: Compression values, of every single CCSDS packet of VEX TM data, that compress well. The sampling rate is 32.0 s and the size of one packet is 120.0 bytes. The protection level stays constant, since the compression is never greater then 100 %.

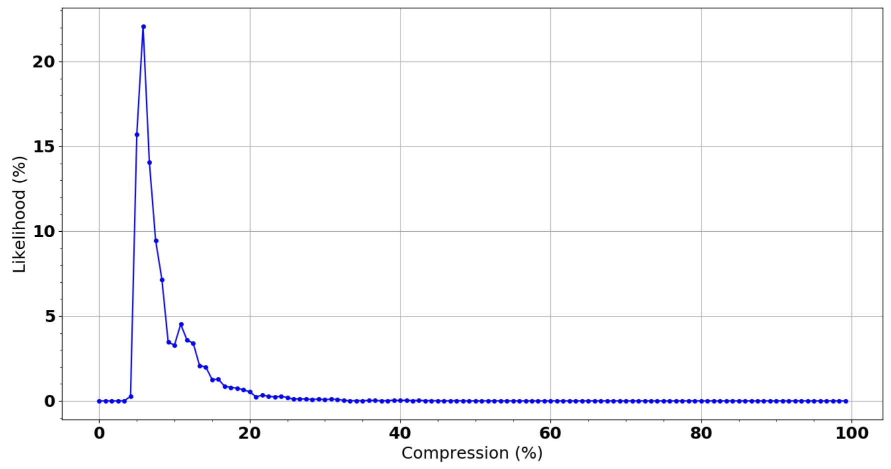


Figure 4.5: Likelihood for compression values of VEX TM data, which compresses well. The probability curve is sharp, tight, and far to left of the graph, which indicates a good compression.

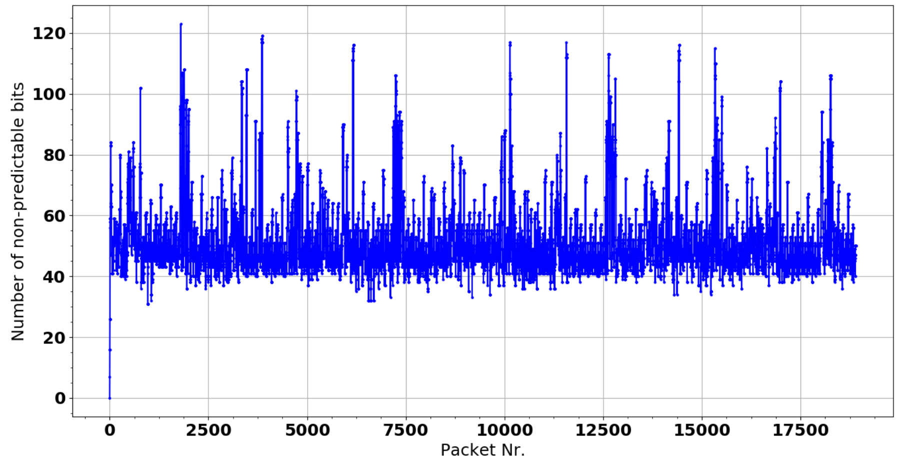


Figure 4.6: Number of non-predictable bits of VEX TM data, which compresses well. The CCSDS packet was sampled with a rate of 32.0 s and has a data field size of 120.0 bytes.

TM data, which compresses badly

Figure 4.7, 4.8 and 4.9 show the results of TM data, which compresses badly. The protection level decreases occasionally, which occurs when the compression becomes greater then 100 %. Figure 4.10 shows this process in more detail. The CCSDS packet was sampled with a rate of 4.0 s and has a data field size of 84.0 bytes.

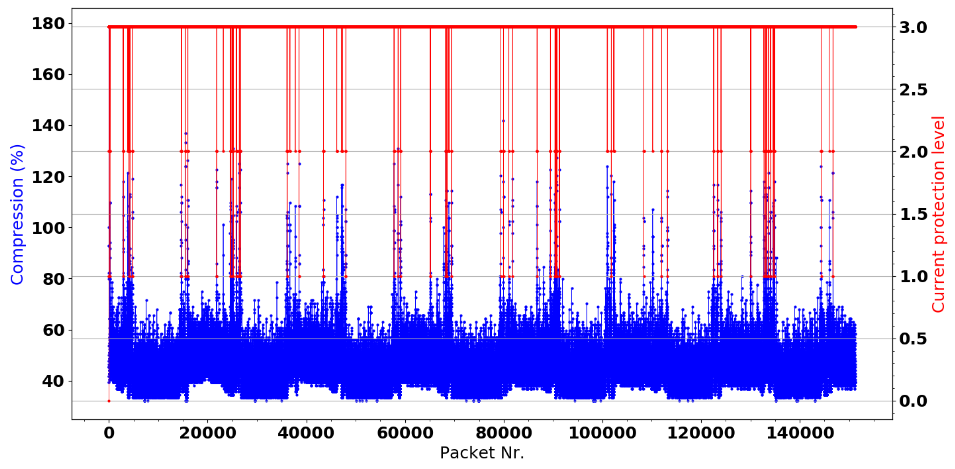


Figure 4.7: Compression values of every single CCSDS packet of VEX TM data, which compresses badly. The sampling rate is 4.0 s and the size of one packet 84.0 bytes.

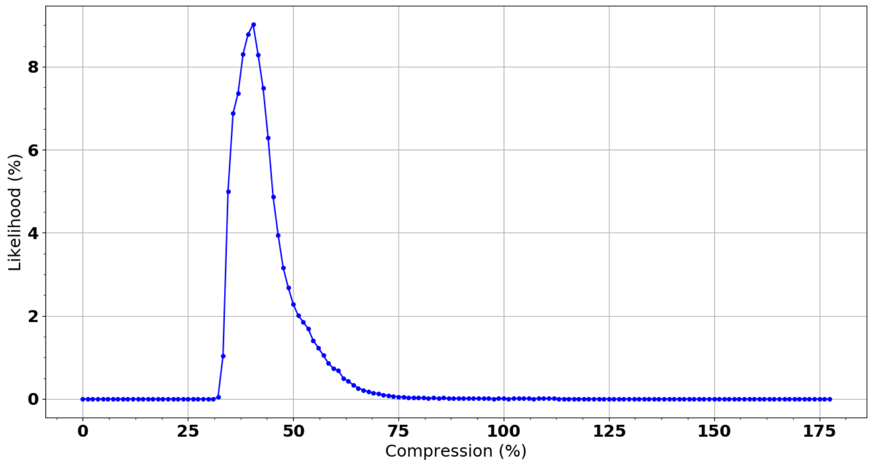


Figure 4.8: Likelihood for compression values of VEX TM data, which compresses badly. The probability curve is spread and shifted to the right.

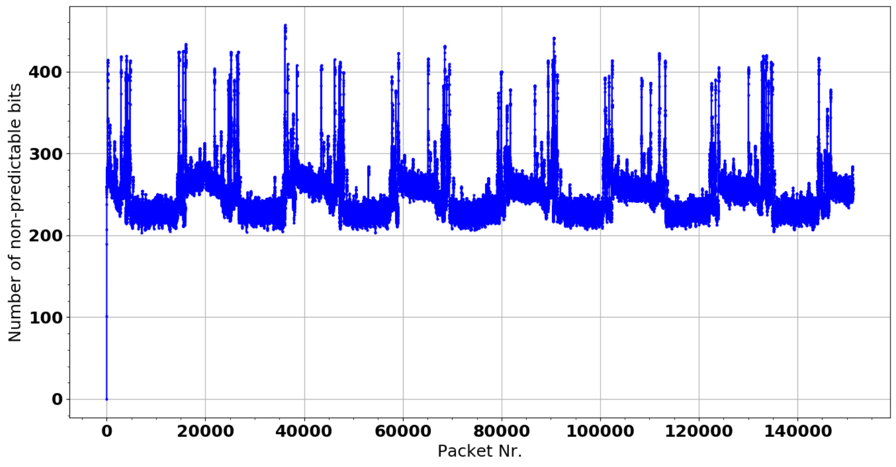


Figure 4.9: Number of non-predictable bits of VEX TM data, which compresses badly. The CCSDS packet was sampled with a rate of 4.0 s and has a data field size of 84.0 bytes.

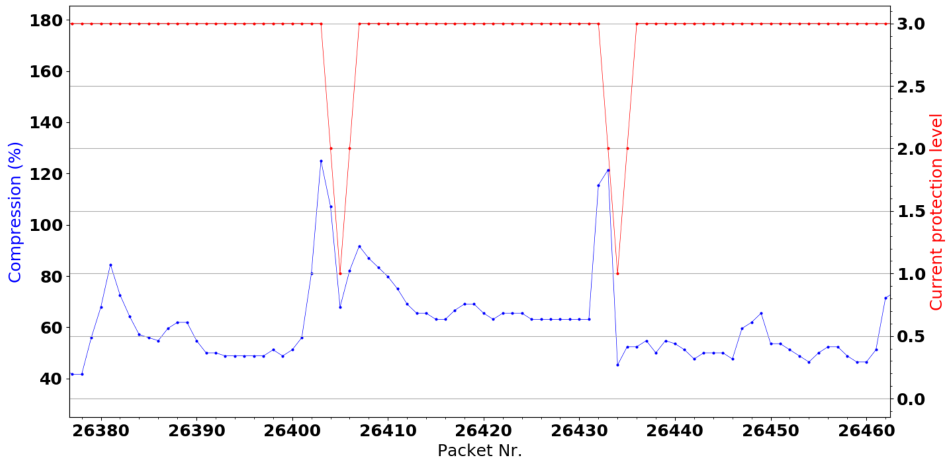


Figure 4.10: Automatic reduction of the protection level when the compression becomes greater then 100 %.

Mask updates

Several mask updates and the initialization phase of Pocket+ can be seen in figure 4.11. When the number of non-predictable bits increase, a negative mask update was done. Conversely, when the non-predictable bits decrease, a positive mask update was done.

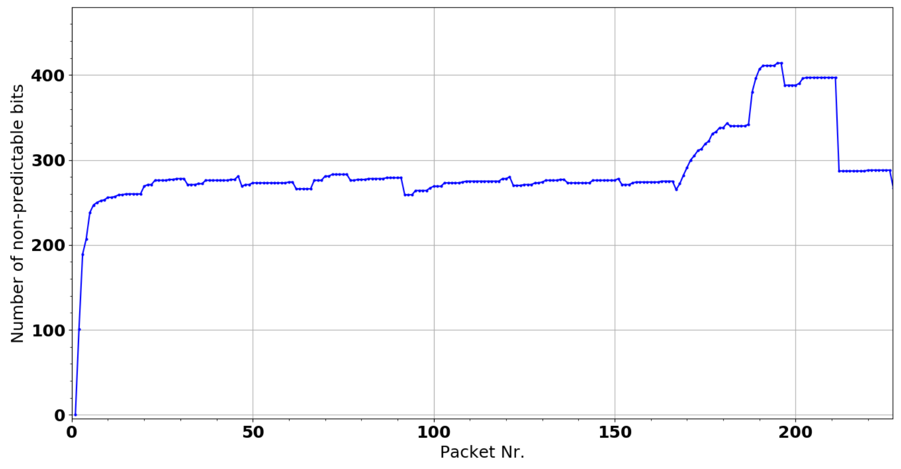


Figure 4.11: Negative and positive mask updates in more detail. When the number of non-predictable bits increase, a negative mask update was done. A reduction indicates a positive mask update.

Periodic trends

A frequency analysis (FFT) was applied on the change of non-predictable bits and is shown in figure 4.12 and figure 4.13. One can see that several peaks can be found at 6.0, 12.0, and 24.0 hours. When the S/C is at the apocenter it moves slowly, most of the sensor data stays the same and Pocket+ compresses well. When the S/C is approaching the pericenter it becomes faster, the sensor data change and the compression becomes worse. Since Venus Express is in a highly elliptical 24:00 hour orbit, this circumstance can be seen in the FFT. The other peaks likely originate from operational procedures, which were scheduled and executed every 6.0 and 12.0 hours.

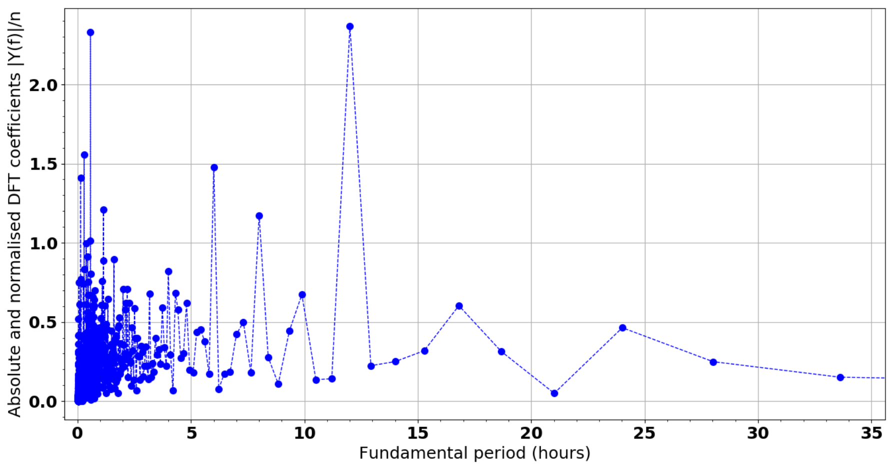


Figure 4.12: FFT applied on the number of non-predictable bits of VEX TM data, which compresses well. The sampling rate is 32.0 s. The 12.0 hour peak could indicate a scheduled operational procedure. The sampling rate is 32.0 s.

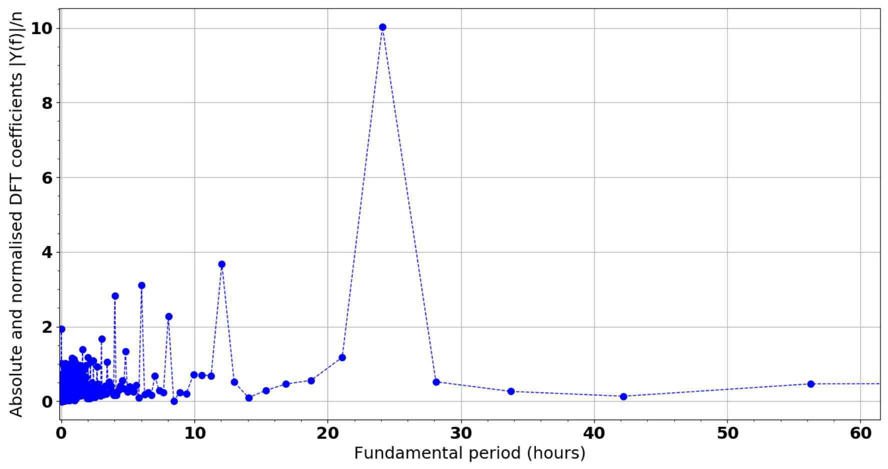


Figure 4.13: FFT applied on the number of non-predictable bits of VEX TM data, which compresses badly. The sampling rate is 4.0 s. The 24.0 hour peak could reflect the 24:00 hour orbit and the peaks at 12.0 and 6.0 could indicate scheduled operational procedures.

4.1.5 Pictures

Since the Pocket+ compression comes from the difference of two consecutive packets it can also be applied on raw binary files, which contain similar bit patterns. For example, the RAW format of the OPS-SAT camera is a Bayer matrix (R-G-R-G) with two bytes per pixel and a default resolution of 2,048 x 1,944 pixel. Thus each horizontal line of a RAW picture can be interpreted as a data packet with a a size of 4,096 bytes. So for the OPS-SAT camera, there are 1,944 data packets with a size of 4,096 bytes. A sample picture is shown in figure 4.14 and table 4.6 shows the compression results of several algorithms applied on that picture.

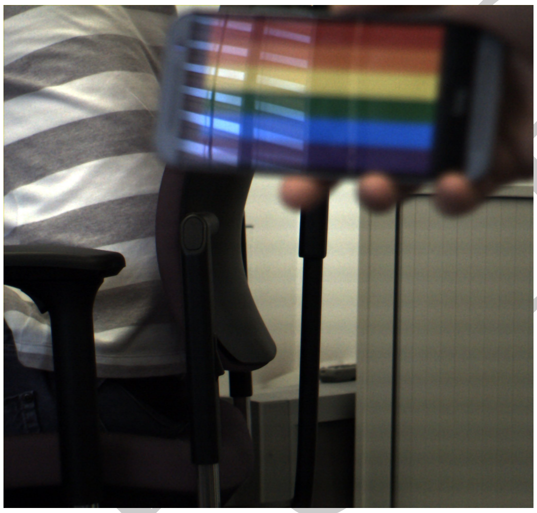


Figure 4.14: Sample image taken from the OPS-SAT CAM.

Table 4.6: Results of the compression of a RAW picture of OPS-SAT.

Algorithm	**Compression (%)**	**Average runtime (s)**
Pocket+	48.39	0.269
LZ4	91.36	0.06
Gzip	55.85	0.36
Bzip2	41.90	1.08

5 Concluding words

Pocket+ has turned out to be a robust and fast way of compressing data, which can compete with, and outperform other generic compression algorithms, in terms of compression ratios and execution times. Although the Pocket+ compression is not optimized for multi-core processing or does not have any integrated parallelization yet, the execution times showed remarkable results. LZ4, which was always the fastest algorithm, also had the worst compression values. Pocket+ had similar execution times as Gzip, but compressed much better. However, it should be pointed out that a whole binary file was used for the generic compression algorithms. These algorithms were not aware of the single CCSDS packets within the binary file. Since those algorithms look for similarities and repeating byte patterns, it can be assumed that their compression performances are getting worse when only small CCSDS packets are used as input.
The research on Venus Express telemetry showed that Pocket+ also reveals operational procedures and orbital properties, which could again be used as input for mission control systems.
Moreover, it turned out that Pocket+ can be used on raw binary files with a known bit pattern. For this case, a packetizer splits and puts the data into an appropriate structure and inputs them into Pocket+. Only the difference of two consecutive packets are decisive. This means that Pocket+ will not work with different types of packets. For every type of data an instance of Pocket+ must be created. However, since current S/C OBCs and OBSWs are highly deterministic, Pocket+ would perfectly fit into this pattern.

Bibliography

David Evans, Benjamin Fischer (2017). “The ESA POCKET+ Housekeeping Telemetry Compression and Decompression Algorithm.” In: *Proceeding Dasia 2017* (cit. on pp. 1, 7).

ECSS-E-HB-11A (2017). *Space engineering: Technology readiness level (TRL) guidelines*. Tech. rep. ESA-ESTEC (cit. on p. 4).

ECSS-E-HB-40A (2013). *Space engineering: Software engineering handbook*. Tech. rep. ESA-ESTEC (cit. on p. 31).

Edmund J. Habib Frank A. Keipert, Richard C. Lee (1966). *Telemetry Processing for NASA Scientific Satellites*. Tech. rep. Goddard Space Flight Center (cit. on p. 1).

Eickhoff, Jens (2011). *Onboard Computers, Onboard Software and Satellite Operations* (cit. on pp. 4, 5, 18).

ESA (2003). *Venus Express*. URL: http://www.esa.int/spaceinimages/Images/2003/05/Venus_Express (cit. on p. 56).

ESA (2017). *Website about OPS-SAT*. URL: http://www.esa.int/Our_Activities/Operations/OPS-SAT (cit. on p. 30).

N.L. Crowley, V. Apodaca (1997). “Analysis of satellite telemetry data.” In: *Proceeding Aerospace Conference, 1997* (cit. on p. 1).

Takashi Iida (2000). *Satellite Communications: System and Its Design Technology* (cit. on p. 2).

Wikimedia Commons (2017). *Bit-error rate curves for phase-shift keying: BPSK, QPSK, 8-PSK and 16-PSK. Drawn in MATLAB, finished in Inkscape.* Ed. by User: Splash. URL: https://de.wikipedia.org/wiki/Datei:PSK_BER_curves.svg (visited on 09/01/2017) (cit. on p. 3).