P9 NX Gzip Accelerator User's Manual

Version 0.9

Advance

Nov 1, 2017
Public
DRAFT

Printed in the United States of America Nov 1, 2017

Advance Products - Legal Disclaimer

While the information contained herein is believed to be accurate, such information is preliminary, and should not be relied upon for accuracy or completeness, and no representations or warranties of accuracy or completeness are made.

Note: This document contains information on products in the design, sampling and/or initial production phases of development. This information is subject to change without notice. Verify with your IBM field applications engineer that you have the latest version of this document before finalizing a design.

nx_wb_title.fm.0.9
Nov 1, 2017 Public
DRAFT

Table of Contents

List of Figures

List of Tables

Preface

About this Book

This book is a work in progress and includes material from other works in progress and as such may contain references to components and facilities not present in P9, or may contain errors that have been fixed in later versions of the source documents.

Who Should Use this Book

This book is for software application and library developers who need to under the P9 NX Gzip Accelerator operation and programming.

This book does not describe the system software required for enabling P9 Gzip. System software developers should follow the references in the back of this document for most up to date information or contact the document owner.

Although this document has references to other accelerator types EFT, SYM, and RNG, it is not suitable for programming them. Only the Gzip subject is covered.

Document Control

This is a working document and its contents may change at any time. The latest version of this document may be found at https://github.com/abalib/power-gzip

Change bars will be used to highlight the areas which were modified from the previous release of this document. Each release supersedes all previously released versions.

Document Owner

Name	Contact Information
Bulent Abali	abali@us.ibm.com

1. 0.90.9Overview

The P9 Nest Accelerator unit (NX) comprises cryptographic and compression/decompression accelerators (coprocessors) with support hardware.

This document is applicable to the Gzip accelerator. Although residual references to various accelerator types EFT (842), AES, SHA, and RNG may be found, this document does not teach programming those.

The Gzip Compression/Decompression accelerator's characteristics are:
- Industry standard DEFLATE RFC 1951 compliant
- Supports RFC 1950 (zlib) and RFC 1952 (gzip) file formats
- Fixed and Dynamic Huffman Table support
- Assist for Dynamic Huffman Table creation
- Ability to suspend an operation when a byte count limit is hit
  - Software may resume operation from suspend point after adjusting job parameters
- Wrap function for memory to memory data move

To support coprocessor invocation by user code, use of effective addresses, high bandwidth storage accesses, and interrupt notification of job completion, NX includes the following support hardware:

SMP Interconnect Unit (SIU)
- Interfaces to SMP Interconnect and Direct Memory Access (DMA) controller
  - Provides 16B/cycle data bandwidth per direction to both
- Employs SMP Interconnect Common Queue (SICQ) multiple parallel read and write machine architecture to maximize bandwidth
  - 16 DMA read machines, each with one cacheline data buffer storage
  - 16 DMA write machines, each with one cacheline data buffer storage
  - 3 UMAC read machines, each with one cacheline data buffer storage
  - 3 UMAC write machines, each with one doubleword (DW) data buffer storage
- User Mode Access Control (UMAC) coprocessor invocation block
  - Snoops Coprocessor Request Block (CRB, or job) Available notification mastered by Virtual Accelerator Switchboard (VAS)
  - Supports one high & one low priority queue per CT
  - Retrieves CRBs from queues and dispatches CRBs to DMA Controller
- Effective to Real Address Translation (ERAT) table stores 32 recently used translations
  - Interfaces to Nest Memory Management Unit (NMMU) for address translation services
  - Translates all Effective Addresses (EA) from DMA controller to Real Addresses (RA)
  - Returns translation faults to DMA controller

DMA Controller
- Decodes CRB to initiate coprocessor and move data on behalf of coprocessors
- Uses effective addresses for all CRB storage accesses
  - Issues paste command to VAS to dispense CRB with translation fault to per-partition fault queue
- 5-channel data mover, one per each instance of AES/SHA, 842, Gzip accelerators, with buffers for data to and from accelerators
- Two CRB queue positions per channel: one for current CRB (currently executing on a coprocessor) and one for pending CRB (awaiting execution)
  - May prefetch Coprocessor Parameter Block (CPB) and source data for pending CRB
- Provides prefetch hints to memory controller to reduce read latency
- Supports byte-aligned source and target data
- Supports scatter/gather through Data Descriptor List (DDL)
- Supports interrupt notification on CRB completion

- 16B/cycle data bandwidth per direction to/from SIU
- 16B/cycle data bandwidth toward 842 accelerators, 8B/cycle toward AES/SHA, Gzip
- 16B/cycle data bandwidth from 842 accelerators, 8B/cycle from AES/SHA, Gzip

Details on NX invocation, CRB processing and other functions may be found in *Section 2* and *Section 6*.

NX integrates many features to improve Reliability, Availability, and Serviceability (RAS), including the following:

- Error Correction Code (ECC) or parity on all data arrays
- ECC or parity on ERAT and other address-carrying structures
- Parity on key configuration registers
- Control checkers on many state machines and other control structures
- High degree of error tolerance
  - Many errors simply write error CC to CSB and hardware operation continues
  - Ability to unit checkstop on severe error and become benign on the SMP Interconnect interface
- Per-channel watchdog timers to detect and terminate hung coprocessor
- DMA hang timer to detect DMA Controller hang
- Hang timers on SMP Interconnect operations
- Unit checkstop upon VAS or NMMU unit checkstop

2. Understanding the P9 gzip accelerator for software integration

The P9 accelerator unit NX contains a gzip accelerator. The gzip accelerator compresses and decompresses data at a rate of 9 to 16 GByte per second–depending on the processor model. One gzip compress accelerator throughput is equivalent to 70 to 120 cores, and one gzip decompress accelerator throughput is equivalent to 25 to 45 cores running software gzip/zlib/deflate implementations.

This section conceptually describes how to integrate the gzip accelerator in to software applications and libraries. Intended audience is the developers who are expected to encapsulate hardware details in Zlib deflate() and inflate() like interfaces. We use the Zlib software package (http://zlib.org) as an example library since many compression enabled applications link with it (libz), although any other compression enabled application may integrate the gzip accelerator without adhering to the Zlib interfaces.

Also on the P9 chip is a user-mode accelerator interface which will substantially eliminate the kernel and hypervisor software path-lengths of accessing the gzip accelerator (see *Section 6.1* on page 46).

System calls are still required for some tasks; for example one time channel setup with the gzip accelerator. This document does not detail the system software--kernel or hypervisor support for accessing the gzip accelerator. System software developers must follow the references or contact the document owner for more information.

2.1 The accelerator user interface: basics

The accelerator may be accessed either via a kernel/hypervisor interface or user-mode interface as described in *Section 6* on page 45. In the user-mode case effective addresses may be used. P9 chip provides the address translation services. User-mode software--applications and libraries need not be concerned with pinning pages which will be handled by the O/S.

The accelerator compresses or decompresses data in the system memory. It moves data from a source to a target buffer.

Software must prepare a Co-processor Request Block (CRB) data structure in the memory (*Figure 6-1* on page 48, *Section 6.3* on page 48). CRB is 128 bytes in length and contains a Function Code indicating the type of accelerator request. There are 13 accelerator FCs. Main function codes are compress and decompress and variations of the two (*Table 6-2* on page 59).

A CRB may also contain direct or indirect addresses of the source and target buffers for scatter-gather type of operations (*Section 6.4* on page 52, *Figure 6-6* on page 53). CRB is submitted to the accelerator using the copy/paste P9 instructions (*Section 6.1* on page 46). Upon job completion, the accelerator writes the status to a Coprocessor Status Block (CSB) in the system memory (see *Section 6.6* on page 54, *Table 6-8* on page 66, and for more detailed explanation of some status in *Table 5-1* on page 37).

The accelerator and software may exchange additional control information via the Coprocessor Parameter Block, which may contain separate, non-overlapping input and output regions called CPBin and CPBout, also in the system memory (*Section 6.8* on page 55, *Section 6.11.2* on page 59, *Section 6.11.3* on page 60). Valid CPB fields differ according to the function code (*Table 6-5* on page 62, *Table 6-6* on page 63).

CPB contains fields including the Adler32 and CRC32 checksums that the accelerator computed, number of bits and bytes that the accelerator consumed and produced, optionally code tables supporting the dynamic Huffman compression algorithm, and additional fields in support of the Suspend and Resume events of the accelerator (*Section 2.6 Suspend and Resume functions*).

2.2 Embedded implementation concepts

This document is written as if the accelerator will be integrated with the Zlib software. For embedded use of the accelerator such as integration with the system software (for example the Linux zswap function and the AIX Active Memory Expansion (AME)) compliance with standards may not be necessary and some function codes may not be necessary. Here, “embedded“ means that the compressed data does not leave the server and it is not visible to the end user. Therefore the software developer can freely organize the compressed data structures and use only a subset of the functions described here.

For example, a checksum need not be stored in a compressed stream trailer but stored elsewhere for example in a page table. Some conditions such as a suspend event may never occur at run time and therefore the resume function codes becomes unnecessary (*Section 6.11.1* on page 59).

Multi block gzip streams need not be implemented when compressing small blocks such as system pages, which will simplify the software development effort.

2.3 Zlib integration concepts

Zlib is a popular compression library used by many applications (http://zlib.net.) The accelerator functions may be wrapped in the Zlib functions listed below (see zlib.h for function descriptions). In one possible approach, a Zlib library integrating the P9 gzip accelerator can dynamically detect the accelerator presence and it can perform compress and decompress functions using the hardware instead of the native Zlib functions, while being transparent to the Zlib linking clients. The Zlib examples/zpipe.c sample code demonstrates some of these functions.

deflate(),

deflateInit2(),

deflateSetDictionary()

inflate()

inflateInit2()

inflateSetDictionary()

The accelerator relevant zlib.h constants are

Z_SYNC_FLUSH,

Z_FULL_FLUSH

Z_BLOCK

Z_FIXED

Z_DEFAULT_STRATEGY

2.4 Understanding DEFLATE, GZIP and ZLIB compressed data formats

DEFLATE (RFC1951) is a compressed data format. ZLIB (RFC1950) and GZIP (RFC1952) are wrapper formats defining their own headers and trailers around a DEFLATE stream. The ZLIB RFC1950 data format should not be confused with the Zlib software package that can process all three data formats. We will refer to the RFCs in capital letters, ZLIB, GZIP, DEFLATE in this document to avoid confusion.

The accelerator can process all three data formats with some software help. In this document, we may loosely use the term “gzip” in reference to any of the three formats.

The software developer will need a high level knowledge of the GZIP and ZLIB formats (a) when assembling multiple DEFLATE blocks in a stream, (b) when decoding GZIP and ZLIB headers, and (c) when wrapping the accelerator output with the GZIP and ZLIB headers.

2.4.1 GZIP (RFC1952) and ZLIB (RFC1950) header and trailer formats

GZIP is a file oriented wrapper of a Deflate stream https://tools.ietf.org/html/rfc1952

ZLIB is memory oriented and has a different header and trailer than GZIP https://www.ietf.org/rfc/rfc1950

[GZIP header] [Deflate Stream] [CRC32] [ISIZE]

[ZLIB header] [Deflate Stream] [Adler32]

Both the ZLIB and GZIP formats wrap the same DEFLATE formatted data. The GZIP header includes a file-name. GZIP uses CRC32 checksum of the original uncompressed data. ISIZE is the data size modulo 2^32.

The accelerator only handles DEFLATE streams, although it does provide an offload engine to compute checksums for GZIP and ZLIB. The accelerator does not produce or consume GZIP or ZLIB headers. Soft-ware is expected to decode the headers which typically comprise tens of bytes.

For compress, software must either write first or allocate a headroom for the GZIP/ZLIB header at the target address. Software then passes the next sequential address to the accelerator as the target buffer address. The accelerator compresses the source data and writes the result to the target.

The accelerator computes both the Adler32 and the CRC32 checksums of the source data and writes them to CPBout (*Figure 6-4* on page 60) for both compress and decompress jobs. For compress operations, the accelerator does not write the checksum to the compressed stream tail. Instead, software must copy one of the two computed checksums from CPBout to the tail. For GZIP, software must also calculate ISIZE and append it after CRC32 (ISIZE is input size modulo 2^32).

For decompress, software decodes the GZIP/ZLIB header. Software then passes the next source address after the header to the accelerator which will decompress data in to the target buffer.

The source stream may have extraneous data at the tail which the accelerator will not process; for example, a checksum or a header of another stream, or simply garbage data in memory. The accelerator will stop decompression at the actual end of the source stream and will not process any bits past the end.

Software may calculate the end address by reading the CPBout.SPBC and SUBC values reported by the accelerator (see *Table 6-4* on page 60). Source Processed Byte Count (SPBC) indicates the number of compressed source bytes read by the accelerator. Source Unprocessed Bit Count (SUBC) indicates the number of source bits that the accelerator discarded, because they were past the stream end.

Once the actual stream end address is known, software may read the CRC32 or Adler32 checksum and ISIZE from the source buffer's tail. Software compares the source checksum to the accelerator computed checksum value in CPBout (*Table 6-4* on page 60). If the checksums do not match, the source data is considered corrupt.

When compressing, if software must know the address of the last valid bit in the compressed target stream the Target Ending Bit Count (TEBC) field provides that information (*Table 6-4* on page 60)

2.5 Deflate compressed data format

2.5.1 Deflate stream

A DEFLATE stream consists of one or more compressed data blocks (https://tools.ietf.org/html/rfc1951).

Deflate stream = [BHDR] [Compressed Data Block] … [BHDR] [Compressed Data Blk]

2.5.2 Deflate block header

Each DEFLATE block has a 3 bit header:

BHDR = [BFINAL] [BTYPE]

BHDR contains the BFINAL bit indicating whether it is the last block in the stream. The accelerator sets BFINAL=1 by default. Software must flip this bit to 0 when the block is not final.

Next 2 bits are the block type BTYPE. DEFLATE supports three block types:

BTYPE=00: a literal block (also called stored block) containing raw source bytes

BTYPE=01: a fixed Huffman encoded compressed block

BTYPE=10: a dynamic Huffman encoded compressed block

2.5.3 Bit and byte ordering (when to examining binary dumps)

The DEFLATE bit and byte ordering may be confusing to some since codes start on non-integral byte boundaries and the codes are packed as if the bytes go from right to left. This fact may also be relevant if your software manipulates the compressed data stream such as inserting a SYNC_FLUSH (see *Section 2.5.6* and *Section 5.2.5.2*).

We illustrate the bytes as packed left to right in this document. But within each byte DEFLATE codes are packed from right to left. The most significant bit of any field other than the Huffman codes are on the left. For example, the first byte of a DEFLATE stream that has a header BFINAL=1, BTYPE=01 is written not as 101 but as

ddddd 01 1

where ddddd are the 5 bits that follow the 3 bit header. In another example, the first two bytes of a dynamic Huffman stream with BFINAL=1, BTYPE=10, followed by 3 bit codes aaa, bbb, ccc, ddd, eee are written as

bbaaa101, edddcccb

2.5.4 Literal block (also called the Stored block)

A literal block contains a copy of the source data in uncompressed form. A literal block starts with the 3 bit block header, followed by 0 to 7 bits of padding for aligning the stream to a byte boundary. The pad bits are followed by a 16 bit wide LEN and its 1's complement NLEN indicating the number of bytes in the block. Therefore, the maximum data size is 0 to 65535 bytes. Note that a Literal block with LEN=0000 NLEN=FFFF contains no data.

[BFINAL bit] [BTYPE=00] [0-7bits padding] [LEN] [NLEN] [... Raw Data…]

The accelerator can decompress literal blocks. But the accelerator compress function does not produce Literal blocks. If necessary, software may create a literal block simply by writing a Literal block Header and LEN/NLEN fields to the stream followed by copying data from source to target addresses.

Compression libraries typically use a literal block when the compressed data ends up being larger than the source data, for example when the source data is already compressed or encrypted. If the accelerator does not yield enough compression, optionally software may discard the accelerator produced block and instead write a literal block to the target buffer.

2.5.5 Creating Multiple block streams

Software may break a very long data stream in to multiple blocks for few a reasons. Source data may arrive piecemeal for example when reading from storage or network. Multiple blocks may also result in better compression ratio because each block will have a customized dynamic Huffman table. And sometimes having multiple blocks may help with data recovery. Block size will be specific to each application scenario. A 32KB or larger block might be appropriate for the best throughput and compression ratio. Zlib lets user to choose the block size. At preset intervals or when the Zlib data buffer is full, the current block may be closed and a new one started (see examples/zpipe.c in the Zlib source code and the Z*FLUSH constants in zlib.h).

The accelerator can be used for producing multiple block streams. Software must submit a separate compress job for each block. Software must append the job results one after the other and must make necessary adjustments when transitioning from one block to another (see *Section 2.5.6* on page 16 and *Section 5.2.5.2* on page 40.)

Software must also ensure that the partial checksums are transmitted correctly between multiple blocks appended in sequence. DEFLATE checksums are computed over the entire input stream and the result is independent of the number of compressed blocks. Partial checksums are transmitted across two blocks via the CPB data structure (*Table 6-3* on page 59, *Table 6-4* on page 60)

Before compressing block number K, The accelerator checksum must be initialized with partial checksum result of the compressed block K-1. Likewise the output checksum from block K must be saved and used as the initial checksum of block K+1.

A seldom used multi-block feature is inserting a literal block in to the compressed stream. See *Section 4.2* on page 27 for more details.

The accelerator will decompress a multiple block stream transparently with a single job submission; there is no need to know that it is a multiblock stream nor separate jobs are required.

However, some special software applications, if they require, can request a decompress job to stop at a block boundary (“process single block and suspend” *Table 6-2* on page 59).

2.5.6 Z_SYNC_FLUSH is an empty literal block for byte aligning multiple blocks in a stream

An empty literal block is also referred to as a SYNC FLUSH block mainly used for byte aligning a stream (*Section 5.2.5.2* on page 40).

When multiple blocks are being appended one after the other, software may need to insert an empty literal block called SYNC_FLUSH between two blocks. Likewise, if the compress job is suspended because the source buffer becomes empty, software may need to insert a sync flush before resuming compression.

A sync flush is often required because a compressed block may end on any bit boundary, whereas the accelerator begins to write to memory only on a byte boundary. The sync flush requirement is not specific to the accelerator. Many existing software applications that produce multiple blocks use the same byte alignment method, for example networking utilities such as SSH.

The accelerator assists sync flushing with the Target Ending Bit Count TEBC(0:2) field indicating the actual end of the compressed stream; TEBC is the number of valid bits in the last byte of the compressed data in the target buffer (see *Table 6-4*).

Unused bits in the last compressed byte must be followed by the software appending a SYNC_FLUSH block, whose 3 bit header starts with the first unused bit in the last byte. A SYNC_FLUSH is required only when another Deflate block is to be appended to the end of the current block and TEBC != 0. While inserting a sync flush recall the bit ordering convention within a byte as explained in *Section 2.5.3*

2.5.7 LZ77 and Huffman encoding

The accelerator compresses data first with the LZ77 algorithm followed by the Huffman algorithm. LZ77 replaces repeated strings in raw data with back references (distance and length pairs) to the original strings. An LZ77 string length is maximum 258 bytes. Literals and Lengths are encoded with the same alphabet of symbols 0 to 285. Distances are encoded with a separate alphabet of symbols 0 to 29. A distance can be at most 32768 bytes. LZ77 retains the most recent 32768 bytes of input as the compressor state, often referred to as the “history” or “sliding window”. Note that a back reference may cross block boundaries pointing from one block to a string in another block.

Huffman codes are variable length codes. Fewer bits encode more frequent LZ77 symbols. For example, the LZ77 symbol, the literal ‘A’ may be encoded with 6 bits, whereas the less frequent literal ‘Q’ may be encoded with 9 bits.

We provide a software utility for producing a Dynamic Huffman code table (called DHT). DHT is submitted together with the source data to the accelerator when compressing with the dynamic Huffman codes (see *Table 6-2* on page 59, DHT and DHTlen in *Table 6-3* on page 59).

2.5.8 Fixed Huffman block

A fixed Huffman block uses a preset Huffman encoding table, where variable length symbols are 7 to 9 bits long. The End of Block (EOB) LZ77 symbol 256 is the last symbol in each block (7 bits of 0000000 in encoded form).

[BFINAL] [FH Block Header (01)] […Fixed Huffman encoded LZ77 stream…] [EOB]

The fixed Huffman mode typically compresses worse than the dynamic mode, however software may sometimes use the fixed mode when block size is small.

2.5.9 Dynamic Huffman block

The dynamic Huffman block type is the default block type in gzip and zlib since typically it gives the best compression ratio. A dynamic Huffman (DH) block contains a Dynamic Huffman Table (DHT) before the data.

[BFINAL] [DH Block Header(10)] [**DHT**] [dynamic Huffman encoded LZ77 stream] [EOB]

DHT is a code table that maps LZ symbols 0 to 285 to variable length Huffman codes each 1 to 15 bits long. It is called dynamic because the code lengths are customized according to symbol probabilities (frequencies). The DH method typically achieves better compression than FH.

The accelerator can decompress DH blocks. It also provides support for creating DH blocks. Creating a DH block is a two-step operation in the accelerator:

(1) The whole or a subset of the source data is compressed once to sample the symbol probabilities. The accelerator collects the symbol counts (LZcounts, *Table 6-4*); software may discard the compressed data. Software then builds a DHT based on the LZcounts to determine which symbol is assigned how many code bits.

(2) Software submits a second compress job with the same source data (the second pass) along with the DHT supplied in CPBin (DHT, *Table 6-3*), although throughput optimizations are possible to avoid this second pass (*Section 2.5.9.4*).

In the DHT, software may provide a superset of the LZ symbols that the accelerator will actually use (see canned and cached DHTs; see *Section 2.5.9.4*).

Note that sometimes software erroneously may supply a DHT with missing codes. The accelerator will throw an exception (see CC=66 in *Table 5-1* on page 37). To avoid the error, the software supplied DHT must contain a Huffman code for every non-zero count indicated in the LZcount array.

2.5.9.1 Counting LZ77 symbols for compressing with the dynamic Huffman functions

Four CRB function codes have a mode called “Count LZ77 symbols” (*Section 6.11.1* on page 59). When the Count mode is enabled, the CPBout:LZcount array contains the number of occurences of all Literal and Length symbols 0 to 285 and Distance symbols 0 to 29 in the compressed stream (*Section 6.11.3* on page 60). Next step is to generate a DHT using these LZcounts.

2.5.9.2 How to make a DHT code table

For the convenience of the developer, we provide two methods of making a DHT. We modified the Zlib source code to make a DHT given the LZcounts. A standalone DHT maker code is also provided.

See https://github.com/abalib/power-gzip

2.5.9.3 Compression with dynamic Huffman functions

Once the DHT is made, software copies it to the CPBin:DHT field (*Section 6.11.2* on page 59). Then, software submits a CRB with a Compress with a dynamic Huffman function code. The accelerator compresses the source data and writes the result to the target address.

2.5.9.4 Throughput optimizations when using DH compress functions

It may not be necessary to make a custom DHT for every single block as both the two pass operation and the software overhead of making a DHT may reduce the throughput. Throughput optimizations are possible essentially by reusing the same DHT or a set of DHTs.

In the “canned” DHT approach, the software developer may produce a DHT or a set of DHTs offline, and either statically compile them as constants in to the software, or read them from a config file at initialization. If the source data patterns do not show much variability and if “good enough” compression ratio is achieved this method may be used.

In the “cached” DHT approach, the DHTs can be made on-the-fly as required, then saved in memory for later use. Therefore, the DHT making software overhead is amortized across multiple jobs.

When multiple DHTs are in memory, canned or cached, how do we know which DHT to use?

If the frequencies of symbols relative to each other do not change significantly from one block to the next, software may reuse the same DHT. The compress jobs may be submitted with the “Count LZ77 symbols” mode enabled. CPBout.LZcounts are produced with every compress job.

A small sample of symbols may be selected as signature symbols representative of the source data. In one approach, the top two to three most frequent symbols may help determine when to switch the DHT--either making a new one or using an old one from canned or cached list of DHTs.

One software implementation that we are aware of is considering a “1.1-pass” method. First 32KB of the source input is compressed first to sample the symbols. DHT is either made on-line or read from a table. In the second pass the entire source input is compressed. The overall throughput is nearly same as the 1-pass throughput for large data, e.g. 256KB or larger.

Some of these throughput optimizations may result in some degradation in the compression efficiency. There is a tradeoff between the compression ratio and throughput.

2.5.9.5 Canned and Cached DHTs: Don’t forget to initialize zero symbol counts to a non-zero value

Note that canned/cached DHTs will require having a Huffman code for every possible LZ symbol which may be encountered in the future. If the software supplied DHT is missing a Huffman code, then the accelerator will throw an exception because the data cannot be correctly compressed (see CC=66 in *Table 5-1* on page 37).

This is avoided in the code samples we provide using the “fill zero counts” option which will initialize all 0 count values to 1. The fill-zero option forces the DHT making code samples to produce a Huffman code for all symbols, Literals/Lengths 0 to 285 and Distances 0 to 29.

2.5.9.6 Making DHTs in the two pass mode: consider Initializing Length and Distance symbols counts to a non-zero value

In rare occasions, the accelerator may produce a different set of Length and Distance symbols for the same input, since the internal hash tables are initialized with random values. A symbol not present in the first pass may appear in the second pass for the same input data. A DHT made from the first pass LZcounts may cause an exception if a new symbol appears in the second pass.

To avoid this, all the Length (257-285) and Distance (0-29) symbol counts may be changed to non-zero values before making the DHT, therefore avoiding the exception (similar to the canned DHT approach). In this case, it is not necessary to initialize Literal (0 -255) counts to nonzero. Because a Literal not present in the first pass cannot appear in the second pass. The code samples we provide has an example where Length and Distances counts may be changed to non-zero.

2.6 Suspend and Resume functions

The accelerator may suspend a job for a few reasons: not enough source data (*Section 2.7* and *Section 2.9*), to provide fair access to multiple users (*Section 4.4*), and on compressed block boundaries if specified in FC (*Section 2.11*).

The accelerator does not require all data to be present in the system memory before starting a compress or decompress job. Source data may be processed piecemeal with multiple accelerator jobs, for example when reading small blocks of data from storage or network.

When the accelerator suspends the execution, it dumps its internal state to CPBout in memory. Software can resume the input stream from where it was suspended. The accelerator state is restored via the CPBin data structure. (Zlib functions in the same manner; see the examples/zpipe.c sample code in the Zlib distribution in which multiple calls are made to inflate() (or deflate()) to process large data and the software state is saved in a z_strm structure.)

A job may also be suspended when the job reaches the autonomic job length limits (see *Section 6.11.5* on page 64). The P9 limit registers Max Byte Count Low and Max Byte Count High enable fair scheduling of multiple jobs. The accelerator has two hardware managed job queues. Normal priority jobs get an execution slice dictated by the Max Byte Count High limit register (for example to process a maximum of 1MB of data per job). When the job limit is reached AND if another job is in the Normal priority queue, the current job is suspended to make way for the waiting job.

The two registers also enable high priority jobs to preempt normal priority jobs. The Low limit register (e.g. a 64KB limit) is applied when the high priority job queue is not empty. Software may use the high priority queue for jobs with real-time requirements for example an interrupt handler needing the accelerator quickly.

2.7 Suspending a compress job

When the source data runs out and the accelerator suspends the job, it closes the current block by writing an end of block (EOB) symbol as the last item to the compressed stream in the target buffer. Next, software must decide if this is the final block or if more source data is to follow.

For the accelerator purposes, suspending and finalizing the job are essentially the same. The accelerator does not distinguish between the two scenarios. Software must take a different action in either case.

2.7.1 Input was final and the accelerator has written the final block

If the suspended block was the last input in this compressed stream, software sets BFINAL=1 by directly writing to memory; software knows the address of the first byte of the block containing the BFINAL bit in the header (*Section 2.5.2* on page 14, *Section 5.2.5.1* on page 40). Software then copies one of CRC32 or Adler32 from the CPBout to the tail of the compressed stream depending on the chosen data format ZLIB or GZIP (*Table 6-4* on page 60).

2.7.2 Input was not final and more jobs are to follow

After the suspended block, if more blocks are to be appended to the target buffer, then software sets BFINAL=0 of the last block, then byte aligns the last block's tail using a SYNC_FLUSH block (*Section 2.5.6* on page 16, *Section 5.2.5.2* on page 40). Software must also save certain CPBout fields needed when resuming the compression job.

2.8 Resuming a compress job

2.8.1 Resume Compress function codes

When the accelerator suspends, all of its internal state will be lost afterwards because other jobs might have used the accelerator interim. However, CPBout of the suspended job contains all the information needed for restoring the accelerator state on a resume.

Software submits a CRB with a Resume Compress function code (*Table 6-2* on page 59) along with the CPBin data structure, which restores the compression history in the accelerator (*Table 6-3* on page 59). This is also called “seeding the history” (*Section 5.2.5.3* on page 41). The history is stored in a high speed SRAM in the accelerator hardware.

The resumed job cannot produce the LZ77 back references unless the accelerator state is restored. In essence, a maximum of 32KB of source data that created the previous target block(s) are read a second time to restore the accelerator internal state to the time of suspension (or fewer bytes if the suspended job read less than 32KB of source data). Software must calculate the amount source data that the accelerator must read-again; the CPBout.SPBC field of the suspended job may be used.

Seeding the history is not required however it will give higher compression ratio at the expense of reading an a max 32KB of the source data a second time. The software developer can make the necessary tradeoffs to seed the history or not (see Z_FULL_FLUSH *Section 2.8.2* on page 21). If job suspensions occur rarely and if the source data is much larger than the 32KB history, then not seeding the history will have negligible impact on the compression ratio.

The CPBin.HistLen field (*Table 6-3* on page 59) instructs the accelerator the amount source data to be read-again for restoring the compression history in 16 byte quadwords. The accelerator does not compress the same source data again; they are read only for restoring the accelerator internal state.

The HistLen field is defined in terms of quadwords. If the amount of history is not a multiple of 16 bytes, the leading bytes should be dropped until the length is a multiple of 16 bytes, rounding down the amount of history and rounding up the starting address. It is acceptable to omit the first 1 to 15 bytes; they are not required for correctness. Their omission will negligibly effect the compression ratio. History starting address can be on any boundary; 16 byte alignment is not required.

When a job is to be resumed, the partial CRC32 and Adler32 checksums output by the suspended job must be copied the next job’s CPBin (*Table 6-3* on page 59). Because the checksum is defined for the entire source data.

Note that the same Compress Resume function code may be used to implement the Zlib dictionary functions as well (*Section 2.12* on page 23).

2.8.2 Implementing Z_FULL_FLUSH with Compress Resume

For some applications it is desirable to prevent any back references at certain points in the compressed stream. It is basically a DEFLATE checkpoint that will allow decompression to start from middle of the stream, if needed by the application. This functionality is called a Z_FULL_FLUSH in the Zlib terminology.

A full flush can be implemented with the accelerator. When a compression job is suspended, software writes the SYNC_FLUSH bits to the target buffer, and then submits a Compress with Resume job, however with CPBin.HistLen=0. In other words, it will be resuming with no history, therefore no back references from the resumed block to a prior block is possible. Here is the related zlib.h paragraph:

“If flush is set to Z_FULL_FLUSH, all output is flushed as with Z_SYNC_FLUSH, and the compression state is reset so that decompression can restart from this point if previous compressed data has been damaged or if random access is desired. Using Z_FULL_FLUSH too often can seriously degrade compression.”

2.9 Suspending a decompress job

Suspend and resume of a decompress job is more involved since source data may terminate in the middle of a compression data structure which cannot be fully decoded. Therefore, it may be necessary to rewind the input stream by a small amount to an earlier boundary.

When a decompress job suspends, the accelerator writes sufficient information in CPBout to be able to resume the job (*Section 6.11.3* on page 60). The Adler32 and CRC32 fields contains partial checksums of the source blocks up to the suspend point.

Source Unprocessed Bit Count (SUBC(0:15)) indicates the number of bits that the accelerator read but could not decode from the source buffer’s end. SUBC indicates the source bits amount to rewind. Software can calculate the resume address by subtracting SUBC from SPBC (*Section 5.2.5.4* on page 42). SUBC may also be non-zero when additional source data is received past the EOB symbol of the final block.

When a job suspends, the accelerator also writes to CPBout:SFBT the type of block that was being decompressed at the time: Source Final Block Type (SFBT) indicates one of the three block types, BTYPE and whether this was the final block, BFINAL. (*Table 6-4* on page 60 and *Section 5.2.5.4* on page 42)

If SFBT indicates a literal block was suspended, the RemByteCount field indicates the number of bytes remaining in the block at the time of suspend. Literal blocks do not have an EOB termination symbol, but the LEN and NLEN fields at the beginning of the block indicates the block length. RemByteCount is the bytes still to go in the suspended block.

When suspended in a DH (BTYPE=10) block, the DHT present in the source block must be saved and restored on a resume. The CPBout DHT and DHTlen fields contain the DHT of the suspended block (*Table 6-4* on page 60).

2.10 Resuming a decompress job

Resuming requires restoring the accelerator internal state. Software must copy the Adler32 and CRC32 partial checksums from CPBout of the suspended job to the CPBin of the resumed job. Software indicates in the CPBin.SFBT field what type of block the accelerator is about to resume. Software supplies a DHT and DHTlen in the CPBin if the SFBT type is for a DH block; or RemByteCount if the SFBT type is for a literal block (*Table 6-3* on page 59)

The SUBC(0:2) field indicates the bit offset in the first source byte that the actual data starts from. This is because a suspended decompress job may have processed fraction of a byte in the last source byte. Notice the difference in field widths: the CPBout.SUBC is 16 bits wide but CPBin.SUBC is only 3 bits wide. This is because the software calculates how many bytes of source to rewind using the information in the 16 bit wide CPBout.SUBC. However, within the last byte to be resumed, the bits may start on any bit boundary which software must indicate with the CPBin.SUBC(0:2) field.

For resume decompress, the CPBin.HistLen field seeds the accelerator history similar to the resume compress function, although there are important differences:

(a) The history seeding will be done with data not from the source side but from the target side; a maximum of 32KB of accelerator produced target data must be read as the history, which software indicates with HistLen. Therefore, most recently produced target data--up to a maximum of 32KB must be kept in memory to be able to seed the history in case the job suspends.

Software developer must ensure that the target data does not disappear for example after flushing it to a disk or network (If you are porting the accelerator to Zlib and modifying the inflate() call, consider updating the strm.avail_out variable late such that the last 32KB of output always stays in the memory; see the inf() routine in examples/zpipe.c.)

(b) Secondly, HistLen cannot be set to an arbitrary value (as in the compress Z_FULL_FLUSH case of compress; *Section 2.8.2* on page 21). To be able to decompress correctly, the accelerator history must be seeded with the smaller of all or the last 32KB of the target data that the accelerator has produced. Because the source data may have LZ77 back references to the accelerator history. Otherwise, the decompress job may terminate with an error.

Note that the HistLen field is defined in terms of quadwords. If the amount of history data is not a multiple of 16 bytes, the leading bytes should be padded to a multiple of 16 bytes, rounding up the amount of history and rounding down the starting address. It is acceptable to pad input with arbitrary byte values because they will not be referenced; padding is done only for satisfying the hardware quadword requirement.

While padding the history, software developer must ensure that no page access violations occur. If the pad bytes come from the suspended target buffer, no access violations may occur as it is already part of the user address space. An exception is when the target block starts on a page boundary. Then, add pad bytes from anywhere in the accessible address space by using a DDE indirect entry (*Figure 6-6* on page 53).

2.11 Decompress single block and suspend (Z_BLOCK)

The Z_BLOCK option in the Zlib software forces inflate (decompress) function to stop whenever a new block is about to start in the compressed stream. Here is the zlib.h paragraph:

Z_BLOCK requests that inflate() stop if and when it gets to the next deflate block boundary. When decoding the zlib or gzip format, this will cause inflate() to return immediately after the header and before the first block. When doing a raw inflate, inflate() will go ahead and process the first block, and will return when it gets to the end of that block, or when it runs out of data. The Z_BLOCK option assists in appending to or combining deflate streams.

Z_BLOCK is not a commonly used flag, although on the accelerator it may be implemented with the “process single block and suspend” function codes (*Section 6-2* on page 59). The accelerator suspends whenever it detects an end of block symbol.

2.12 Zlib dictionary operations

The Zlib software and the ZLIB data format provides a preset dictionary method for compress and decompress operations. DeflateSetDictionary() and inflateSetDictionary() are the relevant functions in Zlib.

A preset dictionary is a common document that both the sender (compress side) and the receiver (decompress side) have previously agreed on. The dictionary is not transmitted along with the compressed data because both sides have a copy of it. Small blocks of data benefit from preset dictionary the most because a dictionary need not be transmitted across, unlike the DEFLATE format which builds a dynamic dictionary in each block.

Dictionaries are used in few applications; for example, a version of the SPDY network protocol uses a dictionary comprised of commonly used terms in an HTTP header.

The ZLIB format allows for multiple preset dictionaries https://www.ietf.org/rfc/rfc1950.txt. The FLG.FDICT bit in the ZLIB header indicates that the block is compressed with a preset dictionary and the Adler-32 checksum identifies the which preset dictionary must be used by the decompressor.

Here is the related excerpt from the standard:

> FDICT (Preset dictionary)
> If FDICT is set, a DICT dictionary identifier is present
> immediately after the FLG byte. The dictionary is a sequence of
> bytes which are initially fed to the compressor without
> producing any compressed output. DICT is the Adler-32 checksum
> of this sequence of bytes (see the definition of ADLER32
> below). The decompressor can use this identifier to determine
> which dictionary has been used by the compressor.

For the accelerator purposes, the sentence “The dictionary is a sequence of bytes initially fed to the compressor” is equivalent to seeding the accelerator history when resuming compression.

To implement Zlib deflateSetDictionary (for compress) the software developer must place the dictionary in the source buffer, and set CPBin.HistLen field to the size of the dictionary. The actual source data to be compressed is placed after the dictionary in the source buffer (the dictionary and the source data need note be contiguous in memory; see the DDE indirect addressing (*Section 6.4* on page 52,*Figure 6-6* on page 53)

Software then submits a job with one of the Resume Compress function codes. The target buffer will receive the compressed data, minus the dictionary itself since the history portion of the source buffer is neither compressed nor written to the target by the resume function.

To implement inflateSetDictionary (for decompress) software developer must place the dictionary in the source buffer again and set CPBin.HistLen field to the size of the dictionary. The compressed data as received in the ZLIB formatted block is placed after the dictionary (again not necessarily contiguously). Software then submits a job with one of Resume Decompress function codes. The target will then contain the original decompressed data.

Sender must put the Adler32 checksum of the dictionary in the DICT field of the ZLIB header.

History bytes do not participate in Adler32 computation. Since there is no partial checksum in the dictionary scheme, the initial value of the CPBin:Adler32 must be set to 0x00000001.

2.12.0.1 Compressing with a preset dictionary (additional details)

Software may use the accelerator to compute the Adler32 dictionary checksum: compress the dictionary; save the Adler32 checksum; discard the compressed output. This step is done once per dictionary. Note also that software may obtain the preset dictionary checksum by other means for example by copying it from an offline source such as a dictionary specification.

Software prepares the ZLIB header to tell the receiver that a dictionary will be required to decompress the data. Sets the fields FLG.DICT := 1 and FDICT := Adler32 as computed in the previous paragraph. A decompressor will later use the checksum to identify the dictionary that it must use.

Software formats the dictionary in its memory before submitting the resume compress operation: the accelerator HistLen field is defined in units of quadwords. If the dictionary size is not a multiple of quadwords, then 1 to 15 bytes at the beginning of the dictionary may be discarded by rounding down the dictionary length to quadwords. It is acceptable to omit the first 1 to 15 dictionary bytes; they will be not be required for correctness. Their omission may negligibly degrade the compression ratio. This step is done once per dictionary. The dictionary starting address and the dictionary length are written to the first Source Indirect DDE entry.

The dictionary quadword length is written to the CPB Input HistLen(0:11) field. HistLen instructs the accelerator to use as many bytes to initialize its history array and hash arrays.

CPBIn Adler(0:31) is initialized to 0x00000001.

Source address of the data to be compressed is placed in the next set of Source Indirect DDE entries. Target addresses are also entered to DDEs as needed.

Compression operation is started with the Compress-Resume function code.

2.12.0.2 Decompressing with a preset dictionary (additional details)

Software examines the ZLIB header: If FLG.DICT = 1, software reads the FDICT checksum from the ZLIB header and locates the corresponding preset dictionary in its memory.

Software formats the dictionary (possibly ahead of time and not at the time of decompression) in its memory in preparation for the resume decompress operation: HistLen field is defined in units of quadwords. If the dictionary size is not a multiple of quadword, then 1 to 15 bytes at the beginning of the dictionary are padded with zeros and the dictionary length in quadwords is incremented accordingly. It is acceptable to pad the first 1-15 dictionary bytes; they are not going to be referenced by the compressed data. This step is done once per dictionary.

The dictionary starting address and the dictionary length are written to the first Source Indirect DDE entry.

The dictionary quadword length is written to the CPBIn HistLen(0:11) field. HistLen instructs the accelerator to use as many bytes to initialize its history array.

CPBIn Adler(0:31) is initialized to 0x00000001.

Address of the data to be decompressed is placed in the second and following Source Indirect DDE entries.

Target addresses entered to DDEs.

The Decompress-Resume job is submitted.

3. Exception handling

The accelerator detects violations of the DEFLATE standard. Corrupt compressed streams will be detected and exceptions will be thrown (see the condition codes in *Table 5-1* on page 37 and *Table 6-8* on page 66). The CRC32 and Adler32 checksums provide additional detection of corrupt data streams.

In the event that the accelerator cannot decompress a particular input because of an internal error, it will be beneficial to use a software library such as Zlib, as a backup decompressor.

3.1 Checking forward progress with certain corrupt input

This corrupt input scenario was hypothesized by the verificaton engineers. Consider a corrupt compressed stream that ends with an invalid DEFLATE code. When sufficient number of input bits are not available to fully detect the invalid code, the current accelerator behavior is to suspend the decompression. If no more input bits arrive, resuming the stream will result in a second suspend.

Software must test that forward progress is being made and not caught in an infinite loop. Here is one way to check for progress.

Suppose you give the accelerator InputBytecnt bytes to resume decompress with. This is total bytes of source minus history (HistLen*16), so just the data to be decompressed.

From SUBCin(0:2), determine how many bits have already been processed in the first byte. 000=0, 001=7, 010=6, 111=1. Call this StartBitcnt

The job is suspended, CC=3, with SUBCout(0:15) unprocessed bits.

If StartBitcnt + SUBCout = InputBytecnt*8, nothing was processed. If no more input is available, flag an error and stop. If more inputs are available resume with the longer stream; this should throw an exception when the invalid code is fully decoded.

4. Programming Notes

4.1 HLIT and HDIST field treatment

The DEFLATE specification is ambiguous when the Dynamic Huffman header has one of these values HLIT=30 or 31, and HDIST=30 or 31. These specify non-existent Length codes 286 and 287, and Distance codes 30 and 31. Zlib software rejects compressed data with those values https://github.com/madler/zlib/issues/82.

The accelerator will throw an exception for the same

4.2 Computing checksums when creating a literal block

This section explains how to handle partial checksums when creating literal blocks. Some data does not compress well and may expand instead. For example, an encrypted or already compressed source data block may actually expand when compressed. Few compression software libraries (such as Zlib) will insert a literal block in to the compressed data stream when the source data expands. Main benefit of a literal block is to limit data expansion to few bytes, namely the 3 bit block type and the LEN/NLEN field in the literal block header.

For the accelerator, a literal block is created by copying data from source address to destination address, either using a library function such as memcpy(), or the Wrap FC (*Table 6-2* on page 59). Software developer must ensure that the partial checksums are handled properly when creating multiple compressed blocks (see *Section 2.5.5* on page 15 for multiple block streams).

If the compressed data consists of a single literal block, the Wrap function checksum output may be used (*Table 6-5* on page 62).

For multiple block streams, the problem is that the Wrap function cannot be initialized with partial checksums because there is no such field in the CPB input data structure (see the Wrap function input column in *Table 6-5* on page 62.) At the same time computing checksums with software is slow.

A work-around exists to this problem: That input data expands cannot not be known in advance in general. To determine if a literal block is needed, source data needs to be compressed. Therefore, during that compress operation the partial checksum will have been computed.

In more detail:

1. Software sends a compress job to the accelerator with the checksum registers initialized to the partial checksum result computed for the previous block K-1 (or initialized to default values if this is the first block)
2. If the output data expands, software discards the output but saves the checksum value computed in Step 1
3. Assuming software decides to use a literal block, it writes a literal block header to the target adress memory, copies source data to the target address–either using memcopy() or the accelerator wrap function, therefore creating the literal block
4. Software initializes the checksum register of the next block K+1 with the partial checksum value saved in Step 2.

4.2.1 Computing checksums for non-compression applications

One may need to compute CRC32 or Adler32 for other than data compression purposes. The Wrap function code may be used for this purpose. See *Table 6-5* on page 62.

4.3 User mode vs kernel mode

P9 implements a user-mode interface that largely eliminates the system calls for accessing the accelerator. Especially for small data sizes consider exploiting this interface to achieve the best possible performance.

To put it in perspective, in 1 microseconds the accelerator will process as much as 16KB of data (model dependent). If you burden each compress or decompress operation with a kernel and hypervisor call, then the performance may be lost in the system calls, in particular for small data sizes.

On the P8 chip which does not have a user-mode interface to NX accelerators, using another accelerator nx-842 we measured throughput when using 4KB and 64KB size pages. We measured a 10x drop in throughput for 4KB pages compared to 64KB, which is largely due to the system software path lengths.

On P9, we estimate that as much as 4.8x latency reduction is possible with the user-mode interface (see *Section 6.1* on page 46 for the copy/paste instructions). With lower latencies, the user-mode software is expected to reach the accelerator peak throughput with smaller data sizes.

4.4 Fairness and Queue management

The accelerator provides two hardware managed FIFO job queues in support of high and normal priority jobs (*Section 6.1* on page 46, *Section 6.11.5* on page 64). Queue sizes are configurable up to a maximum of 256 CRBs per queue. The actual queue size may be determined either by the O/S developers or the system administrators. At the time of this writing IBM's hypervisor developers were considering 8 deep queues. A benefit of shorter queues is receiving the busy feedback quickly, however at the expense of more frequent polling of the queue for an empty slot.

Per job limits provide fair access to the accelerator via the Maximum Byte Count High and Maximum Byte Count Low limit registers. A normal priority job is suspended if any of the two job queues is not empty AND when the job's data consumption reaches one of the two limit register values.

When the high priority queue is not empty, the smaller limit register is applied therefore quickly preempting the currently running job, whether high or normal priority.

When the high priority queue is empty, the larger limit register is applied therefore the accelerator runs more efficiently with fewer suspend/resume operations.

Software must resume a suspended job by submitting a CRB again from where the job was suspended.

High priority software applications, such as kernel mode drivers, and page fault handlers may consider using the high priority FIFO queue.

Here are the excerpts from the design document

> Maximum Byte Count Low: The accelerator uses this value as the byte count limit if at least one high priority gzip job is queued behind the current job. On compression or data move, the byte count limit is compared to the Source Processed Byte Count for the current CRB. On decompression, the byte

count limit is compared to the Target Processed Byte Count for the current CRB. If the limit is exceeded the job is suspended.

Maximum Byte Count High: The accelerator uses this value as the byte count limit if there are no high priority gzip jobs queued behind the current job. On compression or data move, the byte count limit is compared to the Source Processed Byte Count for the current CRB. On decompression, the byte count limit is compared to Target Processed Byte Count for the current CRB. If the limit is exceeded the job is suspended.

4.4.1 What are good values for the Max Byte Count High and Low registers?

We propose a 1MB High register limit. We also propose that O/S developers expose the limit registers through /sysfs, so that admins can tune their systems if needed.

When a suspended job resumes, the 32KB history is also applied to the jobs budget as determined by the limit registers. If the High Limit is H KBytes, then the accelerator efficiency will be

(H - 32) / H

For example, assume that the High limit is set very low H = 64KB. Then, the accelerator efficiency will be 50% for large data, meaning 50% of the input bytes go to restoring the accelerator state. Whereas when the High limit register is configured for 1 MB, the accelerator efficiency will be around 97%. These examples do not apply to jobs smaller than the limit register value; they will not be suspended.

The Low limit register applies when high priority jobs are waiting in their respective queue. We propose a 64KB Low limit register value, which may be good a balance between quick preemption vs accelerator efficiency.

Note that the limit registers may be disabled by setting them to 0. In that case, a job is not subject to any limits and therefore cannot be preempted.

4.4.2 When to switch between software and hardware gzip?

The accelerator throughput is couple orders of magnitude greater than a single thread of Zlib running on a general purpose core. We estimated that the break-even data size for which the accelerator and Zlib is around 100 bytes assuming a 370 ns startup overhead. For almost all practical data sizes, the accelerator will be faster than software running on a general purpose core.

Next concern is whether to use multiple general purpose cores or the accelerator. Preliminary estimates show that the accelerator throughput is equivalent to 25 to 45 general purpose cores for decompress, and 70 to 120 general purpose cores for compress. From aggregate throughput perspective it makes sense to use the accelerator for all compression tasks instead of running software compression on a chip full of cores.

Another concern expressed by a software developer is when small size jobs are mixed in with large jobs. the accelerator consumes the job queues in FIFO order. The developer is concerned with a small job, for example 4KB in size, being queued behind a large job for example 1MB in size. While the P9 chip has some support for checking the queue fullness, we don’t know at the time of writing whether the kernels and hypervisors will expose that function. We do not believe that this problem is a practical concern unless hundreds of

software clients are sharing the accelerator. But here are a couple of potential solutions: (a) set the High limit register to a small value at the expense of reduced acceleration efficiency for large jobs; or (b) estimate the queue size and switch between hardware and software methods of compression:

Each user process can estimate the queue length for itself by measuring and recording the delays of its previously processed jobs and accordingly decide whether to use the accelerator or software on a core. The user process may keep this information in its private memory in the form of a moving average, then use the average delay for the next job submission). This method uses no kernel calls, and easy to implement.

In more detail, let your job's length be L bytes. Let S and H are the software and hardware rates of compression (decompression). Then the condition

L/S > (L/H + Avg. Queue Delay)

indicates when the job will process faster on the accelerator than a software thread on a core. Avg. Queue delay is the time spent waiting in the job queue excluding the actual job run time, L/H. Moving averages can be conveniently calculated with the formula below, where the constant K can be set as desired to filter out fluctuations. Once switched to the software method, occasionally the application must use the accelerator to get an update on most recent queue delays.

Avg := (K * Previous Avg + New) / (K+1)

4.5 Handling target buffer full condition

During decompression source data may expand more than expected and may not fit in the target buffer, for example with highly compressible data. If accelerator produces more target data than was allocated in the CRB (the target DDE bytecount), the accelerator returns CC=13 target space exhausted (*Table 6-8* on page 66). The job cannot be resumed. The job needs to be replayed from the beginning, with either more target space allocated or less source given. In general it may not be possible to know the expansion factor, i.e. how much source to give to the accelerator to limit target data size. Several solutions exist.

In the first approach, the source data size may be recursively cut in half and job replayed, possibly multiple times, until the CC=13 condition goes away. For the remainder of the same input stream, use the same source data size to mostly avoid this recursive divide and submit.

In the second approach, the CC=13 condition may be avoided by allocating a dedicated overflow buffer appended to every job's target DDE list; that is create a larger target buffer transparent to the user code. For example if the user target buffer size is N bytes, then allocate a K byte overflow buffer, as part of compressor state and link it as the last element of the DDE list, therefore creating an N+K byte target buffer. If the target data overflows from the user buffer (ubuf) into the overflow buffer (obuf) return the ubuf portion to the user as is. However, you need to remember that decompressed data is waiting in the overflow buffer. When the user calls inflate() the next time, copy the overflow amount to the user's target buffer first (e.g. using the wrap function code), then resume decompression after that point in ubuf.

Note that the CC=13 condition may still occur in the second approach because the combined buffer size may not be large enough especially with highly compressible data. Then, use the first approach too i.e. successively halving input data.

A third approach may be used if indirect DDEs are not allowed. In earlier DD1 versions of the P9 chip the indirect DDEs were not supported. The compression library may allocate a dedicated buffer larger than the user buffer and do this once as part of the initialization to avoid allocation overheads. Data is decompressed in to this buffer possibly using the successive halving approach. Once the decompression is successful, the data is copied back to the user buffer preferably using the wrap function code.

Using Zlib as an example for the second approach: the state of the decompression operations are kept in the z_strm data structure. Zlib allocates an overflow buffer obuf and keeps a pointer to it in the z_strm data structure. User calls inflate(). Inflate submits an accelerator job with a DDE list of [ubuf,obuf]. Accelerator decompresses source and reports the decompressed data amount in TPBC field of CSB (*Figure 6-7* on page 54).

(a) If TPBC is less than or equal to sizeof(ubuf), then there is no overflow. Inflate() returns and reports the decompressed bytes via z_strm.avail_out (=sizeof(ubuf)-TPBC). The user code consumes ubuf as needed, for example by writing the data to a file.

(b) If TPBC is greater than sizeof(ubuf), then the accelerator has written the excess to obuf. Inflate() returns with z_strm.avail_out=0 since ubuf is full. The user code consumes ubuf as needed, for example by writing the data to a file.

User code calls inflate() again. In scenario (b) z_strm shows that obuf has data from the previous call waiting to be consumed. Zlib copies obuf to ubuf. Inflate() returns again reporting the decompressed data amount via z_strm.avail_out. The memcpy library or accelerator wrap function may be used.

4.6 Inflating compressed streams

4.6.1 Detecting the end of a compressed stream, how to inflate()?

The end is encoded in the compressed stream. The last Deflate block in the stream will have BFINAL=1 in the block header (*Section 2.5.2* on page 14). Upon detecting the EOB character of the last block, NX should return a CC=0 or CC=3 (*Table 5-3* on page 43).

NX sets SFBT=0000 when it detected the end of the stream. SFBT != 0000 indicates that more input data must be supplied to resume the stream. In other words, NX ran out of data before detecting the end of the input stream.

SUBC indicates the number of bits following the EOB character. SUBC value will depend on the formatting of the Deflate stream. A raw deflate stream has no trailers therefore CC=0 and SUBC=0 to 7 is expected (*Table 5-3* on page 43). A gzip or zlib formatted stream have a checksum and additionally gzip has an ISIZE field at the end. Therefore, NX may report CC=3, SFBT=0000 and SUBC >7, specifically SUBC=32 to 39 in the Zlib case, and SUBC=64 to 71 in the Gzip case.

Note that calling inflate(strm,flush=Z_FINISH) does not tell zlib to end the stream. As we indicated above, end of the stream is encoded in the stream. Z_FINISH flag has to do with how to buffer of the input data as explained in the zlib.h. The library developer must check CC and SFBT fields and then accordingly either return a Z_STREAM_END (finished) or Z_OK (need more input).

4.7 Getting start with software development

4.7.1 How to initiate the gzip accelerator access?

Request from O/S a VAS window (the virtual accelerator switchboard). This will be an O/S specific interface. Look for the VAS subject line in the Linux kernel mailing list or H_ALLOCATE_VAS_WINDOW hypervisor call in the AIX documentation (*Section 7.1* on page 72). IBM LTC will provide a device driver for enabling user mode software to allocate a VAS window.

The VAS window is a 64 bit address of the hardware managed gzip job queue to which a Coprocessor Job Request (CRB) is sent. The *Copy* and *Paste* machine instructions are used for sending a CRB. See PowerISA version 3.0B document's Section 4.4. Also see the vas_copy() and vas_paste() function prototypes shown below.

Note that these are user mode instructions. Once the VAS window is open, the user software will communicate directly with the hardware which will eliminate any kernel or hypervisor calling overheads.

User code creates a CRB in the memory describing the gzip job request. Then Copy copies the CRB to a system managed memory location called a copy_buffer. User code follows up with a Paste which transfers the CRB from the copy_buffer to the gzip job queue.

When paste returns with rc=1 it means that the CRB is in the gzip job queue. Paste will not wait for the job completion. Job completion status will be reported elsewhere.

```
diff --git a/arch/powerpc/platforms/powernv/copy-paste.h...

+static inline int vas_copy(void *crb, int offset, int first)
+{
+ WARN_ON_ONCE(!first);
+
+ __asm__ __volatile(stringify_in_c(COPY(%0, %1, %2))";"
+  :
+  : "b" (offset), "b" (crb), "i" (1)
+  : "memory");
+
+ return 0;
+}
+
+static inline int vas_paste(void *paste_address, int offset, int last)
+{
+ unsigned long long cr;
+
+ WARN_ON_ONCE(!last);
+
+ cr = 0;
```

```
+ __asm__ __volatile(stringify_in_c(PASTE(%1, %2, 1, 1))";"
+  "mfocrf %0," CR0_FXM ";"
+  : "=r" (cr)
+  : "b" (paste_address), "b" (offset)
+  : "memory");
+
+ return cr;
+}
```

4.7.2 Submitting your first job to the accelerator

We propose that you to start with the simplest function code, Wrap to get familiar with the accelerator interface and to learn how to create a CRB. Wrap FC copies data from a source to a target buffer in memory (*Table 6-2* on page 59).

Create a CRB with the C=0 format (*Figure 6-3* on page 50). Place the source and target buffer addresses in the DDE fields of the CRB (*Section 6.4* on page 52).

Initially, consider setting the DDEcount field==0. DDEad is a pointer to the single data buffer, and the DDEbc is the length of the buffer in bytes. This type of DDE is called a direct DDE. Later, for memory scatter gather type operations consider using indirect DDEs.

Point the CSB field of CRB to the Coprocessor Status Block (CSB) in memory (*Figure 6-7* on page 54). Submit the job CRB with Copy/Paste.

Start polling for the CSB.V=1 bit waiting for the job completion. Once the job is complete the CSB.CC field indicates the completion status. Non-zero CC codes are detailed in *Section 6.12* on page 65).

For extended information, the gzip accelerator may write an optional Coprocessor Parameter Block (CPB) following the CSB address when CSB.F=0. See *Figure 6-7* on page 54 and *Figure 6-8* on page 56. CPB may be present or not depending on the accelerator function code.

4.7.3 Compress or decompress something

The simplest compress function code is 00000X Compression with fixed Huffman table. It’s like Wrap except the Target buffer will contain compressed data. Consider timing this operation to see how fast accelerator runs; larger data will result in higher throughput. You can decompress the target buffer using the 01000X Decompression function code.

If you try to decompress the target buffer with the gunzip command line utility, it will probably report a format error because the accelerator output is a raw Deflate formatted data. Gunzip looks for a gzip header that contains the file name, time stamp etc. Here is a dummy header to get started. Use xxd -r to convert its text to binary and then prepend it before the compressed data.

```
[~]$ xxd dummy_gz_header
00000000: 1f8b 0808 5d9a 6372 0003 6475 6d6d 7900  ....].cr..dummy.
[~]$ xxd -p dummy_gz_header
1f8b08085d9a6372000364756d6d7900
```

Next you need to append the CRC32 checksum to the compressed data tail, otherwise gunzip will complain. The accelerator has computed and written that to the CPB output CRC field. Gunzip also wants the ISIZE field appended after CRC32. It's simply the source data size modulo 2^32. Now gunzip the file and see if you can get the original file back. The uncompressed file name will be ‘dummy’ because that is in the header.

Next task may be compressing with a dynamic Huffman code to achieve better compression ratio. *Section 2.5.9* on page 17 details various ways to implement this task.

4.7.4 Multiple NX engines

One NX-gzip engine per chip exists. In a 2 socket system, there will be two NX-gzip engines. Linux kernel will implement a single device such as /dev/nxgzip (name to be determined) for managing all NX gzip engines. If the application passes -1 as the VAS ID, the kernel will open the VAS window on the socket that the current process is running (see *Section 7.2* on page 72 for VAS). Instead of -1, to run on a particular NX gzip engine the application may pass a particular VAS ID (which can be obtained from the /device-tree).

If the Linux kernel migrates the process to another socket due to NUMA balancing, the process will use continue to use the NX gzip engine on the first socket.

If an application opened VAS windows to multiple NX gzip engines then it is up to the application to distribute requests to multiple engines.

5. P9 Gzip Accelerator

5.1 Overview

The P9 gzip accelerator compresses and decompresses source data according to the DEFLATE standard. The DEFLATE RFC1951 compressed data format does not define a header or trailer. The GZIP RFC1952 is a wrapper of DEFLATE and it defines a header describing the deflate contents such as file modification time, filename, and OS type. The GZIP trailer contains a CRC32 checksum and the ISIZE sum of the uncompressed data. Likewise, ZLIB RFC1950 is also a wrapper of DEFLATE. The ZLIB trailer contains an Adler32 checksum instead of CRC32.

The accelerator does not process GZIP or ZLIB headers or trailers. Software needs to process the headers and provide just the DEFLATE stream. The accelerator computes both the CRC32 and Adler32 sums of the uncompressed data for both the compress and decompress functions. These can be used by software to construct the appropriate trailer.

An additional wrap function is included to transfer input data unchanged to the output. It also calculates both the CRC32 and Adler32 checksum.

5.1.1 Compression

The accelerator is designed to compress source data and produce a single deflate block that is written as target. This can either be a type “01” (fixed Huffman coding) or type “10” block (dynamic Huffman coding). CPB input is used to provide a compressed dynamic Huffman table to generate the type “10” block. The BFINAL bit in the block header is set to 1 by the accelerator. It is up to software to change that bit if necessary. No provisions are provided to adjust the starting bit for resuming a previous compression job. It is up to software to add a sync flush block to the suspended job’s target. That will allow the resumed job’s target to be appended to the first job since it will be byte aligned.

Along with the compressed block in target memory, helper fields are written to CPB output. These include the Adler-32 and CRC32 checksums that are used in the zlib and gzip formats. An indicator of the last valid bit of target (Target Ending Bit Count) is written. This can be used by software to add a sync flush block for byte alignment. An additional feature of the accelerator is the ability to count LZ77 symbols. A 24 bit counter is used for each of the 286 LL and 30 D symbols. The counts are written to CPB output and can be used to optimize dynamic Huffman coding.

The accelerator also has the capability to resume a previously suspended job. In order to do this, the partial Adler-32 and CRC32 checksums need to be provided in CPB input. In addition, the history buffer can be seeded to allow better compression results. History buffer seeding is accomplished by reading a user specified number of source bytes (0-32768) and not producing compressed results. The source is only used to seed the hash table and history buffer. Once the history has been seeded, compression begins and target is written. The length of the history is given in CPB input.

5.1.2 Decompression

The accelerator supports all deflate block types. It is up to software to strip off any file headers and trailing information. The accelerator does not handle those. The result of decompression is uncompressed data written as target. Both Adler-32 and CRC32 checksums are generated on the uncompressed data and written

as CPB output. It is up to software to compare these with the values in the trailing input data that was stripped off. A successful decompression operation will end with source containing a final block that has an end of block Huffman code. Bits of source following this are ignored.

If source data ends before the final block, the number of unprocessed bits is written to CPB output. This allows the job to be resumed. If the suspend point is within a block, the block type will be indicated. Also, if the block were a type “10” block, the compressed dynamic Huffman table is written to CPB output. There’s also the special case where the suspend point is within a block header. This can either be insufficient header bits or within a dynamic Huffman table. This is flagged for the resume operation.

The accelerator also has the capability to resume a previously suspended job. The information on the suspend type is passed in (within block type or beginning of block). The partial Adler-32 and CRC32 checksums need to be provided in CPB input. In addition, the history buffer has to be seeded for correct results. History buffer seeding is accomplished by reading a user specified number of source bytes (0-32768) and not decompressing them. These bytes are actually the uncompressed data produced by the suspended job and allow for references back in history. Once the history has been seeded, decompression begins and target is written. The length of the history is given in CPB input. On a resume, it is up to software to byte align the source based on the unprocessed bits from the suspended job. The first byte of source for the resume will follow the last completely processed byte of source from the suspend. So this first byte will have between 1 and 8 unprocessed bits.

5.2 Software Interface

5.2.1 CRB FC field

The Gzip accelerator requests input parameter data from dma after being started. The first 16 byte quadword contains the 6 bit function code field, FC(0:5), from the Coprocessor Request Block (CRB) in bits 26:31. Definition of the FC may be found in *Section 6.11.1 Gzip Function Codes* on page 59.

5.2.2 CPB Input Parameters

Depending on the function requested, the accelerator may request additional CPB input parameters. These are defined in *Section 6.11.2 Gzip CPB Input Parameters* on page 59.

5.2.3 CPB Output Parameters

The Gzip accelerator writes a variety of output parameters, depending on the operation requested. These are defined in *Section 6.11.3 Gzip CPB Output Parameters* on page 60.

5.2.4 CSB Completion Codes

The Gzip accelerator provides a non-zero completion code (CC) to dma that can be written to CSB output. These codes are written in response to errors in operation. Errors and completion codes associated with gzip are shown in *Table 6-8 CSB Non-zero CC Reported Error Types* on page 66. When an error is detected while processing source, the accelerator will stop making input requests and accept data from pending requests. In most cases, the input data will not be processed.

In addition, the accelerator will signal one of two terminate types back to dma. This signaling is made after all source has been received and all target sent. If the error is detected early enough, the timing of terminate is coincident with the last output request.

Full terminate indicates a fatal error where the output is suspect, and the job can't be resumed. After signaling terminate, the accelerator provides a completion code, but in most cases no CPB output. The exception to this is when an array parity error is detected on DHT data being written to CPB output. In this case, the completion code is written after CPB output is finished.

Partial terminate indicates that the output is valid, and the job may be resumed. This is the suspend case. The accelerator will finish normally, processing as much input as possible, and producing valid output. CPB output will be written followed by the completion code. Partial terminate is also used by dma to set the Completion Extension (CE) bits in CSB output.

The Gzip accelerator detects a variety of software exceptions or hardware errors. These are listed in *Table 5-1*.

Table 5-1. Gzip Error Detection

CC	Name	Operation	Response	Description
3	Insufficient source	decompression, decompression resume	partial teminate	Source ended before the final block was processed. Can also occur due to use of decompress single block FC. CPB output contains information indicating type of block being processed (SFBT) and number of bits of source not processed (SUBC). Can be resumed by software. Partial terminate signaled after all source received, used by dma to correctly set CSB[CE].
	Too much source	decompression, decompression resume	partial terminate	More than 7 bits of source received after the final block was processed. Could be due to gzip file trailer following last block. Accelerator stops making source requests. Pending requests will be accepted but not processed. All target sent. CPB output sent containing information indicating number of extra bits of source (SUBC). Partial terminate signaled to dma, stops additional source from being sent.
	Insufficient history	compression resume, decompression resume	terminate	Amount of source sent less than or equal to HistLen field in CPB input. Terminate signaled after last source received.

Table 5-1. Gzip Error Detection

CC	Name	Operation	Response	Description
10	DHT decode scratch array parity error	compression	terminate	Scratch array parity error while decoding DHT in CPB. Accelerator stops making source requests. Pending requests will be accepted but not processed. Partial target (type '10' block header) may be produced. No CPB output sent. Terminate not signaled until all source received and partial target sent.
		decompression resume	terminate	Scratch array parity error while decoding DHT in CPB. accelerator stops making source requests. Pending requests will be accepted but not processed. Target and CPB output (DHT) will not be produced. Terminate not signaled until all source received
		decompression, decompression resume	terminate	Scratch array parity error while decoding DHT in source. Accelerator stops making source requests. Pending requests will be accepted but not processed. No CPB output sent. Terminate not signaled until all source received and target sent.
	DHT translation array parity error	compression	terminate	Parity error while reading DHT translation array. LZ77 symbol will be mapped using bad data, likely producing incorrect code in target. Accelerator stops making source requests. Pending requests will be accepted but not processed. Terminate not signaled until all target sent. No CPB sent.
		decompression, decompression resume	terminate	Parity error while reading DHT translation array. The Huffman code could not be translated to an LZ77 symbol. Previous symbol becomes last one producing target. Accelerator stops making source requests. Pending requests will be accepted but not processed. Terminate not signaled until all source received. No CPB sent.
	DHT storage array parity error	decompression, decompression resume	terminate	Parity error on read from array holding DHT for CPB output during suspend. Terminate sent after all CPB output sent. Completion code written after CPB.
	History array UE	compression	terminate	An uncorrectable ECC error was detected while reading the history array. Data ignored as a potential match so target will be correct. Accelerator stops making source requests. Pending requests will be accepted and processed. Terminate not signaled until all target sent. No CPB output sent.
		decompression, decompression resume	terminate	An uncorrectable ECC error was detected while reading the history array. The corrected data, which is garbage due to the UE being "corrected", is used to generate target. Accelerator stops making source requests. Pending requests will be accepted and processed. Terminate not signaled until all target sent. No CPB output sent.
	Shadow register overflow	decompression, decompression resume	terminate	More source was requested to be rewound than the shadow registers hold (maximum 15 dwords). Rewinding source happens when a compressed block ends and the Accelerator needs to back up to the beginning of the next block. Should be impossible but checker added as insurance. Accelerator stops processing and making source requests. Pending requests will be accepted and ignored. Terminate not signaled until all target sent. No CPB output sent.

Table 5-1. Gzip Error Detection

CC	Name	Operation	Response	Description
66	Huffman code not found	compression	terminate	An LZ77 symbol was generated that doesn't have an associated code in the dynamic Huffman table given as CPB input. No code generated for this symbol. Accelerator stops making source requests. Pending requests will be accepted and processed. Terminate not signaled until all target sent. No CPB output sent.
	Undefined Huffman code	decompression, decompression resume	terminate	A Huffman code was found in compressed source that has no corresponding LZ77 symbol. Possible causes: Fixed Huffman code for unused symbols LL286-287. Requires 8 valid bits of source to detect. Fixed Huffman code for unused symbols D30-31. Can be detected with just 4 valid bits of source even though code is 5 bits. Dynamic Huffman code that is not in the Dynamic Huffman Table provided as CPB input or within the compressed source. Requires 15 valid bits of source to detect. This is the longest dynamic Huffman code. Code will not be mapped. Accelerator stops making source requests. Pending requests will be accepted but not processed. Terminate not signaled until all source received and target sent. No CPB output sent.
	Invalid Block Header	decompression, decompression resume	terminate	An invalid DEFLATE block header was found in compressed source. Could either be the unused header code '11' or a type '00' literal block with inconsistent length field (LEN /= NLEN). Accelerator stops making source requests. Pending requests will be accepted but not processed. Terminate not signaled until all source received and target sent. No CPB output sent
67	Invalid Distance	decompression, decompression resume	terminate	A Huffman code in compressed source translates to an LZ77 length, distance symbol that references data beyond the beginning of the history. Access will not be allowed, this symbol and all subsequent ones will be ignored. Accelerator stops making source requests. Pending requests will be accepted but not processed. Terminate not signaled until all source received and target sent. No CPB output sent.
68	Invalid DHT	compression	terminate	Error decoding DHT in CPB. Many causes: HLIT or HDIST greater than 29. Not allowing undefined symbols in DHT. Code Length, LL, DIST code table underflow or overflow Repeat previous length code for first LL symbol Undefined LL, DIST length code Repeat previous/zero length code repeats beyond last DIST symbol Not enough CPB input for DHT DHTlen field in CPB doesn't match number of DHT bits processed. Accelerator requests all CPB. CPB[DHTlen] field indicates how many CPB[DHT] bits get forwarded to target as header DHT. Accelerator stops making source requests and pending requests will be accepted but not processed. Terminate not signaled until all source received and target sent. No CPB output sent.
		decompression resume	terminate	Error decoding DHT in CPB. Causes are the same as compression above. Accelerator requests all CPB. Accelerator stops making source requests. Pending requests will be accepted but not processed. Terminate not signaled until all source received. No target or CPB output sent.
		decompression	terminate	Error decoding DHT in compressed source. Same causes as above, with exception of not enough input. That gives CC=3. Accelerator stops making source requests. Pending requests will be accepted but not processed. Target will only contain previously processed source. If the error is found in the first block, no target will be produced. No CPB output sent.

5.2.5 Software Usage of Gzip Accelerator

This section describes software techniques for using the gzip accelerator.

5.2.5.1 Compression Final Block

The accelerator will write to target a single DEFLATE type “01” or “10” block with BFINAL=’0’, indicating that additional blocks follow this one. If this is to be the final block, software has two options.

Software can change BFINAL=’1’ in this block’s header to indicate this is the last block. The bit is least significant bit 7 in the first byte of target produced. The byte following the last byte of target will be the starting point for the ZLIB or GZIP trailer that software can append.

Alternatively, software can add a sync flush block with BFINAL=’1’, described in next section.

5.2.5.2 Compression Sync Flush Block

Target produced by compression is not guaranteed to end on a byte boundary. If this is required, as for resume compression, software must add the required bits.

Software will read the Target Ending Bit Count (TEBC) field from CPB output. TEBC equal to 0 means all bits in the last byte are valid and target ends on a byte boundary. TEBC not equal to zero indicates that software needs to append a sync flush block to the target data. A sync flush block is a type 0 literal block with a length of zero. As part of its definition it forces the block to end on a byte boundary.

The following table illustrates how software may append a sync flush to the stream. The last byte of target contains some number of valid bits, indicated by “D”. Software creates a sync flush block by adding the BFINAL bit, the block type, pad bits, and the LEN/NLEN fields.

Pad bits are indicated with “P”. These can be any value, normally cleared to 0. The BFINAL bit, “F”, is set to ‘0’ on a resume, indicating that the sync flush block is not the end of the compressed output. The block type bits, “T”, are set to “00” for a type 0 literal block. Finally, software appends the 4 byte length field, 0x0000FFFF, which indicate a block length of 0. The first two bytes are the length (LEN) followed by its complement (NLEN).

Table 5-2. Gzip Compress Sync Flush

TEBC	Final Target Byte	Software added Bytes	Description
0	DDDDDDDD	-	No additional bytes needed
1	PPPPTTFD	00000000 00000000 11111111 11111111	
2	PPPTTFDD	00000000 00000000 11111111 11111111	
3	PPTTFDDD	00000000 00000000 11111111 11111111	
4	PTTFDDDD	00000000 00000000 11111111 11111111	
5	TTFDDDDD	00000000 00000000 11111111 11111111	
6	TFDDDDDD	PPPPPPPT 00000000 00000000 11111111 11111111	
7	FDDDDDDD	PPPPPPTT 00000000 00000000 11111111 11111111	

5.2.5.3 Compression Resume

Here are the steps that software will take to resume a previous job. Any compress job can be resumed. But if the DMA controller suspended the job due to a byte count limit, it must be resumed. A suspended job will return CC=3.

The resume job's CPB input quadword 0(0:63) needs the CRC32/Adler32 checksums from the suspended job's CPB output quadword 24(0:63). If the suspended job was using a DHT, it needs **input** quadword 0(116:127) and quadwords 1-18 copied over as well. For resume DHT compression, may be easier to first copy all of the suspend job's CPB input quadwords 0-18, then overwrite the checksum and HistLen fields.

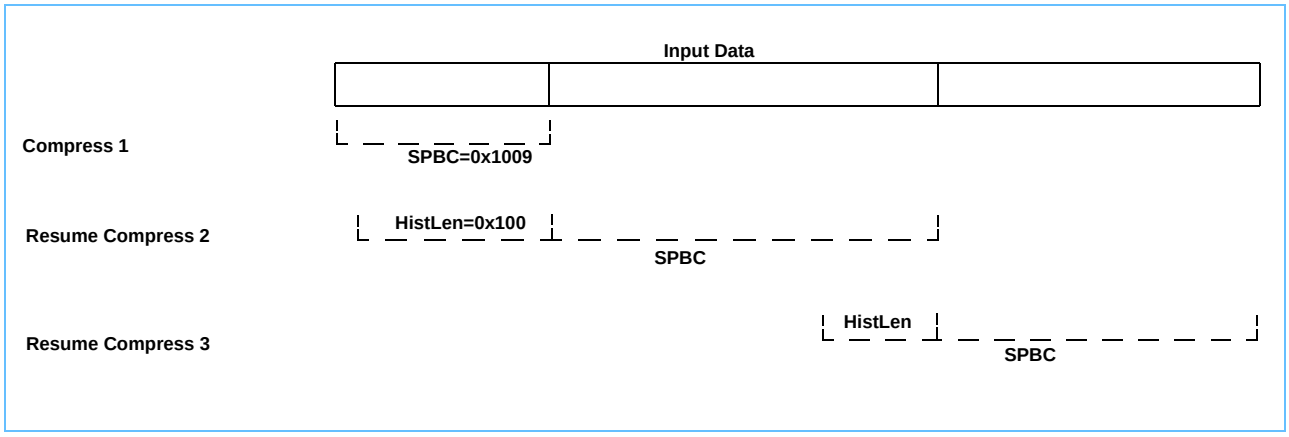
Resume has the option of seeding the history to allow for better compression ratio. Software writes the HistLen field in CPB input to tell the accelerator how many quadwords of source will be used to seed the hash tables and history fifo, and not compressed. After HistLen quadwords of source are read, subsequent source will be compressed. If there are less than 32Kbytes of history, only full quadwords of source can be used.

Source data for the resume job begins with the byte following the last source byte processed by the suspend job. The amount of source processed by the previous job is given by the Source Processed Byte Count (SPBC).

For resume to work correctly, the target from the suspend job needs to be byte aligned. If the previous job's output had nonzero TEBC, software needs to append a sync flush block to its target data. See *Section 5.2.5.2 Compression Sync Flush Block*. The byte following the sync flush block is the first target location of the resume job.

The figure below shows how input data can be broken up and compressed in pieces. Compress job 1 processes the first part of the input, with SPBC indicating how much was compressed. The job may be suspended because it hit a hardware byte count limit, or it could be software breaking up the input. Resume compress job 2 picks up where the first job left off. Since it's seeding history, it's told to start HistLen quadwords sooner. Note that HistLen quadwords of history doesn't include the beginning of source. This is because the number of byte available for history is not a multiple of 16. Upon completion, SPBC includes both the history and the input data that was compressed. Resume compress job 3 is much like the previous job but finishes up the remaining input. Software will need to adjust the source/target DDE addresses and DDE byte counts for each job.

Figure 5-1. Gzip compress suspend/resume



5.2.5.4 Decompression Resume

Only decompress jobs with CC=3 and SFBT!=0000 can be resumed with this method. See *Table 5-3 Gzip Decompress SFBT, SUBC combinations* for an explanation.

accelerator the suspended job's CPB output quadwords 24-42 to the resume job's CPB input quadwords 0-18. No need to look at any fields. We set up bit assignments so this was possible. Even the extra bits of SUBC don't matter. To save time, only need to accelerator over DHT quadwords 25-42 if SFBT=1100, 1101. The only field missing is HistLen, explained below.

The source for the resume job needs to include up to 32KB of history, which is uncompressed target from all previously suspended jobs. History is given as quadwords, so padding may be needed for less than 32KB based on alignment. Giving the resume job history is probably easiest with indirect DDEs.

The starting point of the resume job's compressed source is adjusted by the amount of unprocessed source from the suspended job. The suspend job's CPB output SUBC(0:15) gives unprocessed bits. SUBC=0x0000 requires no adjustment. SUBC=0x0001-0x0008 mean 1 byte was partially or completely unprocessed, 0x0009-0x0010 mean 2 bytes were unprocessed, and so on. Probably a clever programming way to do this, maybe (SUBC(0:15) + 7) >> 3

The target of the resume job follows the last byte of target from the suspended job. The suspended job's Target Processed Byte Count (TPBC) field is used to determine this.

Here's an example.

Job 1 decompress
Source : DDEaddr = 0x01000, DDEbytecnt = 0x05000
Target : DDEaddr = 0x10000, DDEbytecnt = 0x10000
Suspended CC=3. SFBT=1010 (inside FHT block), SUBC(0:15)=0x0009, SPBC=0x1000, TPBC=0x0798

Job 2 decompress resume

accelerator job 1 CPBout QW24-42 to job 2 CPBin QW0-18. Since not a DHT block, could get by only accelerratoring QW24 to QW0.

The history in the source comes from job 1 target, which produced 1944 (0x0798) bytes. Since less than 32K, history must contain all previous target. Also need to prepend 8 bytes to the history to get quadwords. This pad data can be anything since the compressed data shouldn't reference it. A dummy DDE is one way to do this. With padding there's now 0x07A0 bytes of history, so CPBin QW0[HistLen] field is set to 0x07A quadwords.

Job 1 SUBC=0x0009 means 2 bytes of source were unprocessed. The start of job 2 compressed source is calculated as job 1 DDEaddr + SPBC - Unprocessed bytes = 0x01000 + 0x1000 - 2 = 0x01FFE

Source : indirect DDE with the following
DDEaddr = 0x10000, DDEbytecnt = 0x00008 Dummy, values don't matter. Using history for convenience.
DDEaddr = 0x10000, DDEbytecnt = 0x00798 Actual history, all the target produced by job 1
DDEaddr = 0x01FFE, DDEbytecnt = 0x04002 Compressed input

Target

DDEaddr = 0x10798, DDEbytecnt = 0x0F868 Adjusted based on job 1 address, TPBC

5.2.5.5 Decompression SFBT, SUBC Results

Decompress job output includes information about how source processing ended. This tells software how much source was consumed and how to resume the job, if necessary. The Source Final Block Type (SFBT) and Source Unprocessed Bit Count (SUBC) provide most of the information. *Table 5-3 Gzip Decompress SFBT, SUBC combinations* shows possible output values from decompression. These apply to either decompress or resume decompress. The row marked in bold face characters only applies to the decompress single block function. All other rows apply when either decompressing a single block or all blocks.

Table 5-3. Gzip Decompress SFBT, SUBC combinations

CC	Exception	SFBT	SUBC	Description
0	None	0000	0-7	Final block processed. No extra bytes of source.
3	DMA byte count limit	0000	0-7	Final block processed. But dma signaled suspend due to a byte count limit, and just happened to truncate source at the end of the final block. What are the odds?
3	Too much source	0000	>7	Final block processed. Extra bytes of source
3	Not enough source	1000	0	In middle of type "00" literal block with BFINAL=0
3	Not enough source	1001	0	In middle of type "00" literal block with BFINAL=1
3	Not enough source	1010	0-30	In middle of type "01" fixed Huffman block with BFINAL=0. Unprocessed bits are partial huffman code Max SUBC = 8b LEN + 5b Xtra + 5b Dist + 13b Xtra -1b missing
3	Not enough source	1011	0-30	In middle of type "01" fixed Huffman block with BFINAL=1. Unprocessed bits are partial huffman code.
3	Not enough source	1100	0-47	In middle of type "10" dynamic Huffman block with BFINAL=0. Unprocessed bits are partial huffman code Max SUBC = 15b LEN + 5b Xtra + 15b DIST + 13b Xtra -1b missing
3	Not enough source	1101	0-47	In middle of type "10" dynamic Huffman block with BFINAL=1.
3	Not enough source	1110	0	Previous block with BFINAL=0 ends on byte boundary. No bits available for next 3 bit block code.
3	Not enough/too much source	1110	>0	Previous block with BFINAL=0 ends. **FC indicates decompress single block, accelerator driven suspend.** Additional source may be requested
3	Not enough source	1110	1-2	In middle of 3 bit block code. BFINAL=0 (first bit of block code)
3	Not enough source	1110	3-34	In middle of type "00" literal block header with BFINAL=0. Max SUBC = 3b block header + 7b max padding + 32b len/nlen - 8b missing (always byte aligned).
3	Not enough source	1110	3-2285	In middle of type "10" dynamic Huffman block header with BFINAL=0. Max SUBC = 3b block header + 2283b max DHT size - 1b missing
3	Not enough source	1111	1-2	In middle of 3 bit block code. BFINAL=1 (first bit of block code)
3	Not enough source	1111	3-34	In middle of type "00" literal block header with BFINAL=1
3	Not enough source	1111	3-2285	In middle of type "10" dynamic Huffman block header with BFINAL=1

5.2.5.6 Decompression Resume Forward Progress

In certain suspend cases (CC=3), the accelerator may not have processed any of the source given. Resuming this job without additional source would give the same result, so software should check for this.

For a non-resume decompress operation, need to look at CPB output fields: Source Unprocessed Bit Count (SUBC), Source Processed Byte Count (SPBC). No progress has been made if SUBC = (SPBC * 8).

Resume decompress progress checking is a little more complicated. Need to look at CPB input fields SUBC and History Length (HistLen). As well as output fields SPBC and SUBC.

Accounting for history seeding, the total number of source **bits** available to be decompressed.

InputBits = (SPBC_out * 8) - (HistLen_in * 16)

From SUBC_input(0:2), know how many bits in the first byte have already been processed.

StartBits = (8 - SUBC_input) modulo 8. Maps as follows: 0=000, 7=001, ... , 2=110, 1=111

If the total unprocessed bit count is equal to the total source available to be decompressed, no progress has been made.

StartBits + SUBC_out == InputBits

5.2.5.7 Compression Literal Block

Using the wrap function, software can create type 00 literal blocks which contain uncompressed data. In some cases this results in a better compression ratio.

Case 1. First block.

Software writes first byte of target with header: PPPPPTTF = 0x00. P=pad bit (any value), T= block type “00”, F=final bit. Set F=1 if this is the last block.

Size is specified using 2 bytes, values 0-65535. Assume size LEN = 0xWWXX and the complement of size is NLEN=0xYYZZ. Software writes next 4 bytes of target with LEN followed by 2 byte complement NLEN. Note that the DEFLATE standard has byte reversal within each 16 bits. So the bytes written are 0xXXWW_ZZYY.

Use Wrap FC=01111x to accelerator WWXX bytes of source to target following the header. Checksums in CPB output can be used as input for subsequent blocks.

Case 2. Subsequent block.

The wrap function has no way to take previous checksums as input. One way to handle this is first run resume compress FHT or DHT. A sync flush may be needed for this. If the compressed data is significantly larger than the input, then a literal block can replace it. The checksums from the resume compress FHT or DHT need to be used as output from the literal block operation.

If a sync flush was added before the resume compress FHT or DHT, the literal block header already exists. Just need to replace the LEN/NLEN fields using the method in case 1 above. Then the wrap function can be used to accelerator the data following the literal block (former sync flush) header.

If a sync flush was not needed, the literal block including header is built using the method in case 1.

6. NX Accelerator Software Interface

In the sequel, the terms “accelerator”, “coprocessor”, “engine”, “algorithm engine” are used interchangeably. Additional terms are defined in the following table.

Table 6-1. Terms used software interface section

Term	Description
cache line	128 bytes, size aligned.
CD	Coprocessor Directive. CD(0:15) = CL(0:1) || CI(0:*x*) || FC(0:*y*), *x* + *y* = 10
CE	Correctable error.
CI	Coprocessor Instance. Each CT in the SMP shall have a unique non-zero CT. (*arcane*: P8)
CL	Function Class of coprocessor request. (*arcane*: P8)
CRB	Coprocessor Request Block. 128B cacheline of information that describes a job to be executed on a CT.
CSB	Coprocessor Status Block. DMA controller writes V = 1 when job is complete and CC to completion code value, 0 being completion without error. Other fields may also be written — see *Section 6.6 Coprocessor Status Block (CSB) Details* on page 54
CT	Coprocessor Type. NX has three CTs: 842 compression/decompression, Gzip compression/decompression, symmetric cryptography. RNG is not considered a CT.
DDE	Data Descriptor Element. The DDE in a CRB points to either some non-zero length of data at an address (direct DDE) or to a list of DDEs (indirect DDE).
DDL	DDE List. A contiguous list of up to 255 direct DDEs.
DW	Doubleword, 8 bytes.
EA	Effective Address. Requires translation to RA.
EFT	842 compression/decompression accelerator.
ERAT	Effective to Real Address Table.
FC	Function code. Defines the operation to be performed by a CRB, is specific to the CT.
FIR	Fault Isolation Registers. Captures various error conditions detected by the hardware.
FS	Fault Status. Encoded field indicating the specific type of fault received during the translation process for an address.
FSA	Faulting Storage Address. Byte address of the storage location that missed in the ERAT causing a checkout request, and received a fault in response. The specific type of fault is given by the FS. Generally, non-zero FS indicates a fault.
GZIP	Gzip (DEFLATE RFC 1951 compliant) compression/decompression accelerator.
job	Used synonymously with CRB. A unit of work for an accelerator to perform.
NMMU	Nest Memory Management Unit. Performs address translation services on behalf of nest clients including NX.
OW	Octword, 32 bytes.
PTE	Page table entry.
QW	Quadword, 16 bytes.
RA	Real Address. RAs are used exclusively on the SMP interconnect.
Receive FIFO	Circular buffer of 128B messages in memory. Written by VAS, read by NX. In this context a Receive FIFO is bound to a CT and is either high or normal priority. Size is programmable.
RTE	Receive Table Entry. Contains, among other things, the base address, size, write address, and empty entries (credits) of a Receive FIFO in memory. One or more STE may map to a single RTE. All of this may be referred to as Received Window Context, which has a unique identifier in the system.

Table 6-1. Terms used software interface section

Term	Description
STE	Send Table Entry. Contains, among other things, send widow credits, a pointer to an RTE, and address translation information, all of which may be referred to as Send Window Context, which has a unique identifier in the system.
SPBC	Source processed byte count. Amount of source data bytes an accelerator has consumed in processing this CRB. SPBC includes the amount of history bytes consumed.
SYM	Symmetric cryptographic functions (AES, SHA, MD5) accelerator.
STE	Send Window Table Entry. STE holds, among other things, user identification information and number of credits indicating how many operations are allowed at a time. In the An STE points to an RTE.
TLB	Translation lookaside buffer. Caches recently used PTEs.
TPBC	Target processed byte count. Amount of target data bytes an accelerator has written in processing this CRB.
UE	Uncorrectable error.
UMAC	User Mode Access Control logic. NX logic between the SIU and DMA controller that is responsive to notifications from VAS that a CRB has been queued in a receive FIFO, reads CRBs from Receive FIFOs and dispatches them to the DMA Controller.
VAS	Virtual Accelerator Switchboard. Contains send and receive window contexts, snoops ***paste[_last].*** s on the SMP interconnect, and notifies NX when a CRB has been written to a receive FIFO in memory.
XIVE	External Interrupt Virtualization Engine. XIVE is the on-chip entity on the SMP interconnect to which interrupt sources, such as NX, direct cache-inhibited writes to signal an interrupt.

6.1 Accelerator Initiation

Accelerator initiation makes uses of the *Copy/Paste Facility* described in Power ISA AS Version 3.0.

Assume the hypervisor has allocated hypervisor memory for each of these circular buffers:

GZIP High Priority Receive FIFO
GZIP Normal Priority Receive FIFO

and for each, has programmed NX to be notified by VAS when a CRB is written to either receive FIFO.

A user application issues a *syscall*() (OS call) to register to use the Gzip accelerator at high or normal priority. The OS makes the necessary hypervisor calls to enable the application to use the accelerator. The hypervisor sets up a Send Window Table Entry (STE). A send window can be unique per process, or can be shared by an OS among its many user processes. When the hypervisor establishes the send window, it assigns it a quantity of credits indicating how many operations are allowed at one time, and returns an address handle to the operating system. Then the operating system returns an effective address to the user.

The application may now request a job be performed by the Gzip accelerator at the requested priority by means of the copy/paste facility. The application populates a CRB in memory with fields appropriate to the job. Along with the CRB are other control structures including CSB (mandatory), CPB (optional), DDE list with direct DDEs (optional). (These structures will be defined in detail in subsequent sections.)

The application performs a ***copy*** to copy the CRB from its memory location to the copy buffer. The application then performs a ***paste_last.*** using the EA provided by hypervisor. The execution of the ***paste_last.*** causes the EA to be translated to an RA, and a paste command to be issued on the SMP interconnect along with the RA.

The VAS unit on the SMP interconnect snoops the paste command, using the RA operand of the paste command to identify this user's STE, confirm that there is at least one credit available to send the CRB, and that the receive FIFO indicated by the send window has at least one credit available to receive the CRB. If both credits are available, the VAS acknowledges the paste command and waits for the operand to arrive, which is the contents of the copy buffer which holds the CRB. The ***paste_last.*** completes with CR0=0b001 || XER_{SO} "Data transfer successful". If sufficient credits are not available, the ***paste_last.*** receives a CR0=0b000 || XER_{SO} "Data transfer failed" and the user may either retry the operation at a later time or perform the gzip operation in software.

Assuming the data transfer was successful, the CRB arrives at VAS and VAS then writes it to the Receive FIFO in memory, stamping it with user's send window information for use by NX. Then VAS notifies NX that a CRB has been written in this Receive FIFO. UMAC logic in the NX receives the notification, reads the CRB from the indicated Receive FIFO, and also reads user's send window context information, which contains, among other things, address translation context information for this user. The CRB and send window context information are placed in storage facilities in NX from where they await dispatch to the DMA Controller and from there to the GZIP accelerator. NX returns a credit for this receive FIFO.

High and normal priority Receive FIFOs are defined for GZIP. UMAC logic will fetch jobs preferentially from the high priority Receive FIFO.

The DMA Controller has two queue positions for jobs in each channel controller: prefetch and current. Current is the job currently executing on the accelerator, and prefetch is awaiting execution. The DMA Controller will begin prefetching CPB and source data for the job in prefetch when the current job has completed fetching source data. A CRB for a given CT will dispatch from UMAC to the DMA Controller when the prefetch queue position is empty in the channel controller assigned to that CT and then move to current when the current CRB has completed execution.

When a CRB completes execution, in general the CSB(V) bit is written to a 1, the CSB(CC) field is updated to reflect CRB completion status, and other CSB fields may be updated, an interrupt may be signaled if the CRB has requested it, and a send window credit for the user is returned. More details are given in *Section 6.9 Coprocessor Job Completion*,

6.2 NX Control Data Structures

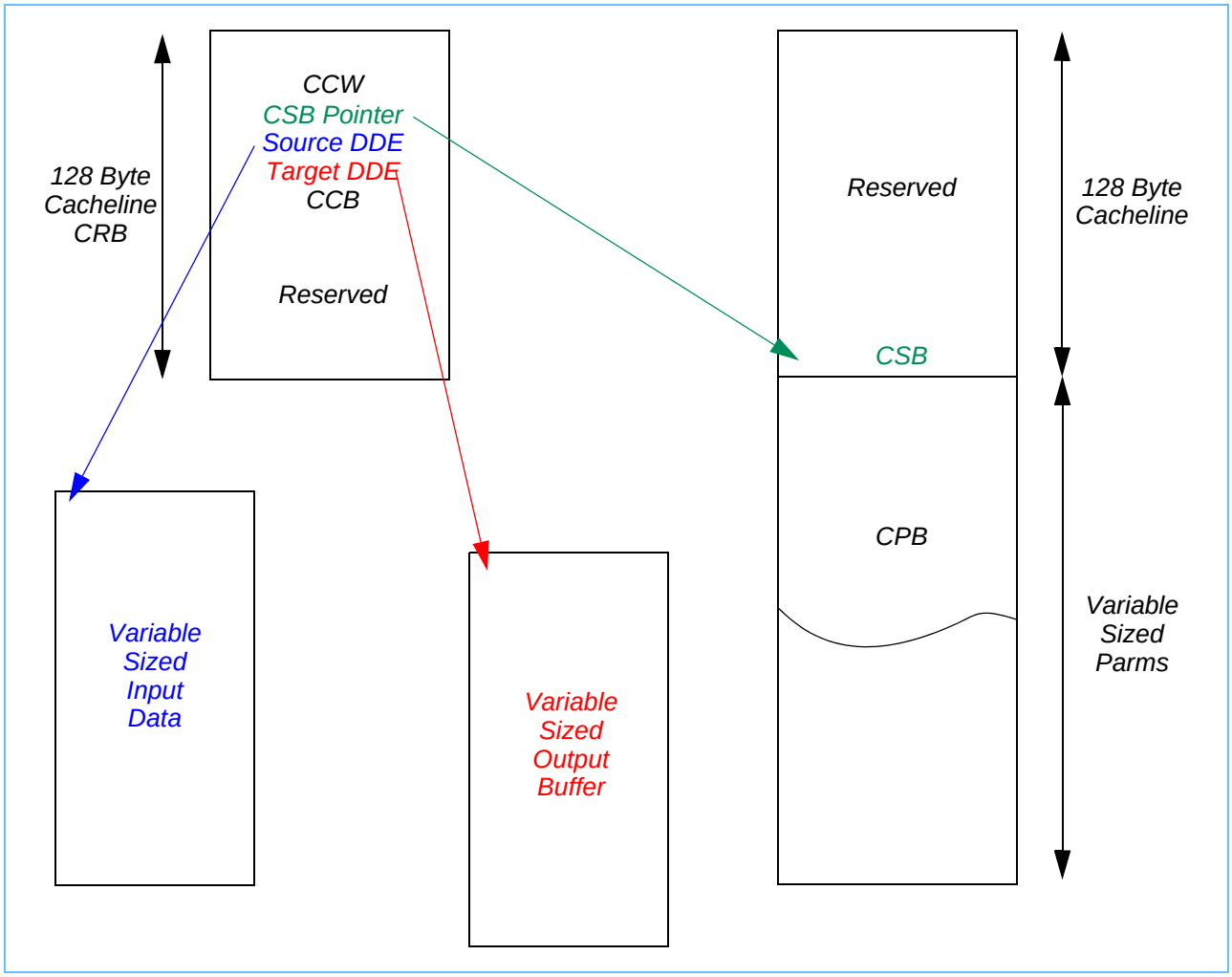
NX control data structures are shown in *Figure 6-1*. NX uses a 128 byte CRB, a 16-byte size-aligned CSB, a variable length CPB that starts at the next QW after the CSB, and a CCB embedded inside the CRB.

Programming Note

The preferred location of a CSB is the last QW of a 128-byte cacheline as shown in the figure. This allows for a performance optimization that has the consequence of overwriting the entire cacheline. Details are in *Section 6.6 Coprocessor Status Block (CSB) Details* on page 54.

The CRB contains all the information needed to perform the requested coprocessor function. It contains all the control information and pointers needed to allow the accelerator to access the input parameters, input data and know where to store the output data and finishing status.

Figure 6-1. Control structures used by the NX Accelerator



6.2.1 Endianness of NX control and data structures

All NX control structures are big endian, regardless of coprocessor type.

NX data structures, i.e., input (source) and output (target) data buffers, for 842 compression/decompression, DEFLATE (gzip) compression/decompression, and symmetric cryptography are treated as bit streams and are therefore endianness agnostic.

6.3 NX Coprocessor Request Block (CRB) Details

The CRB is shown generically in *Figure 6-2 CRB overview* on page 49. Detailed CRBs will be shown in subsequent figures.

Figure 6-2. CRB overview

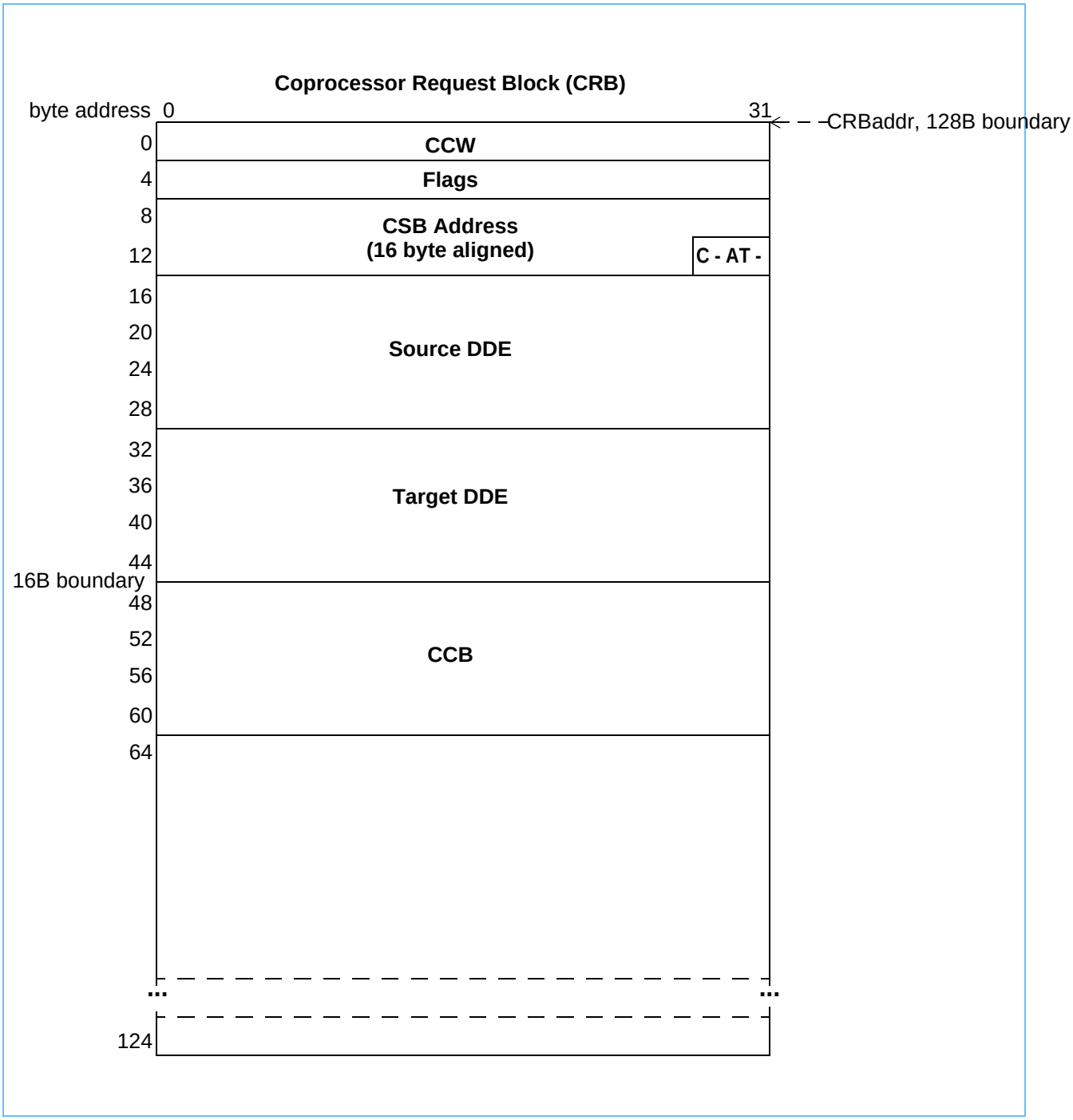
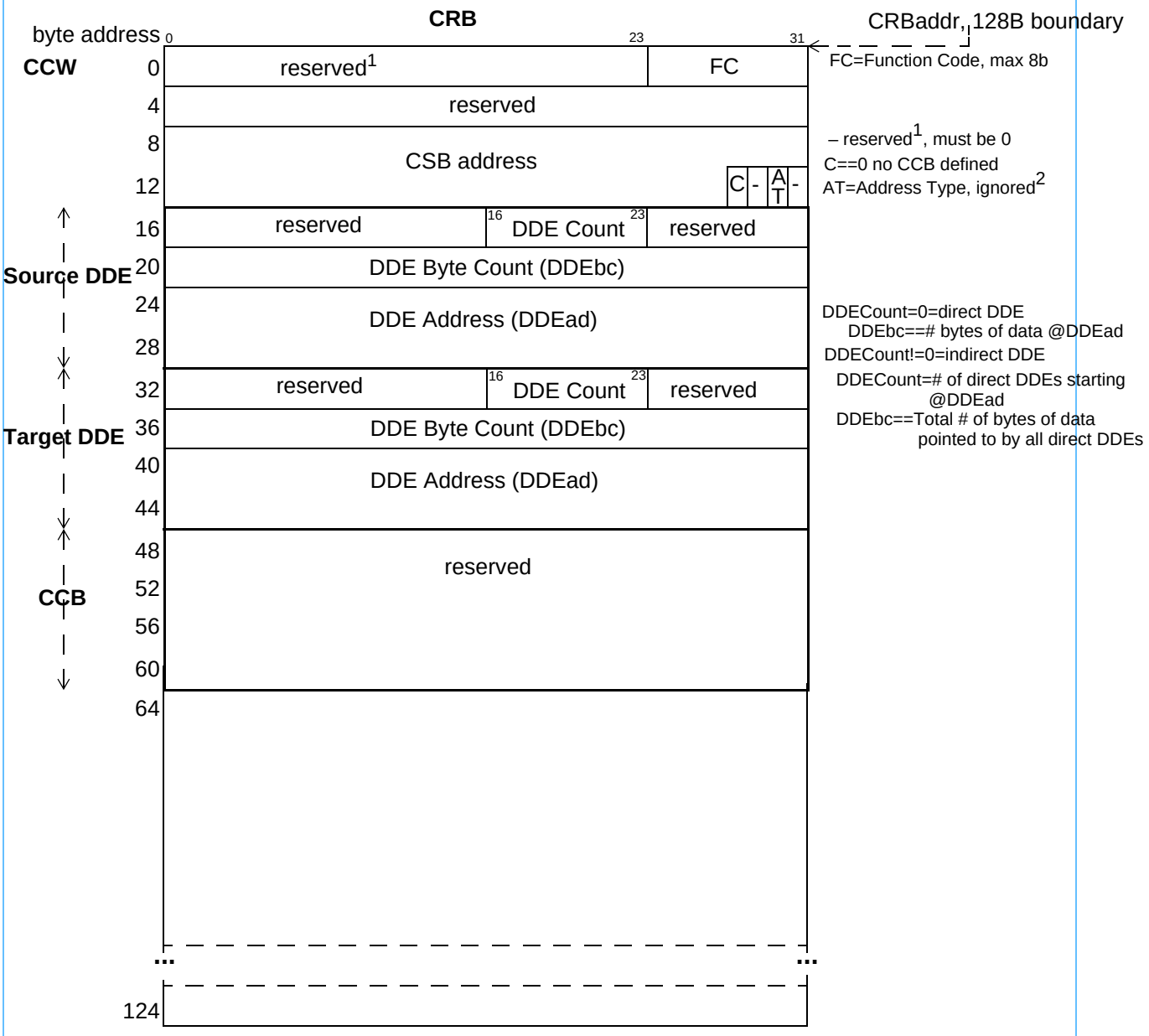


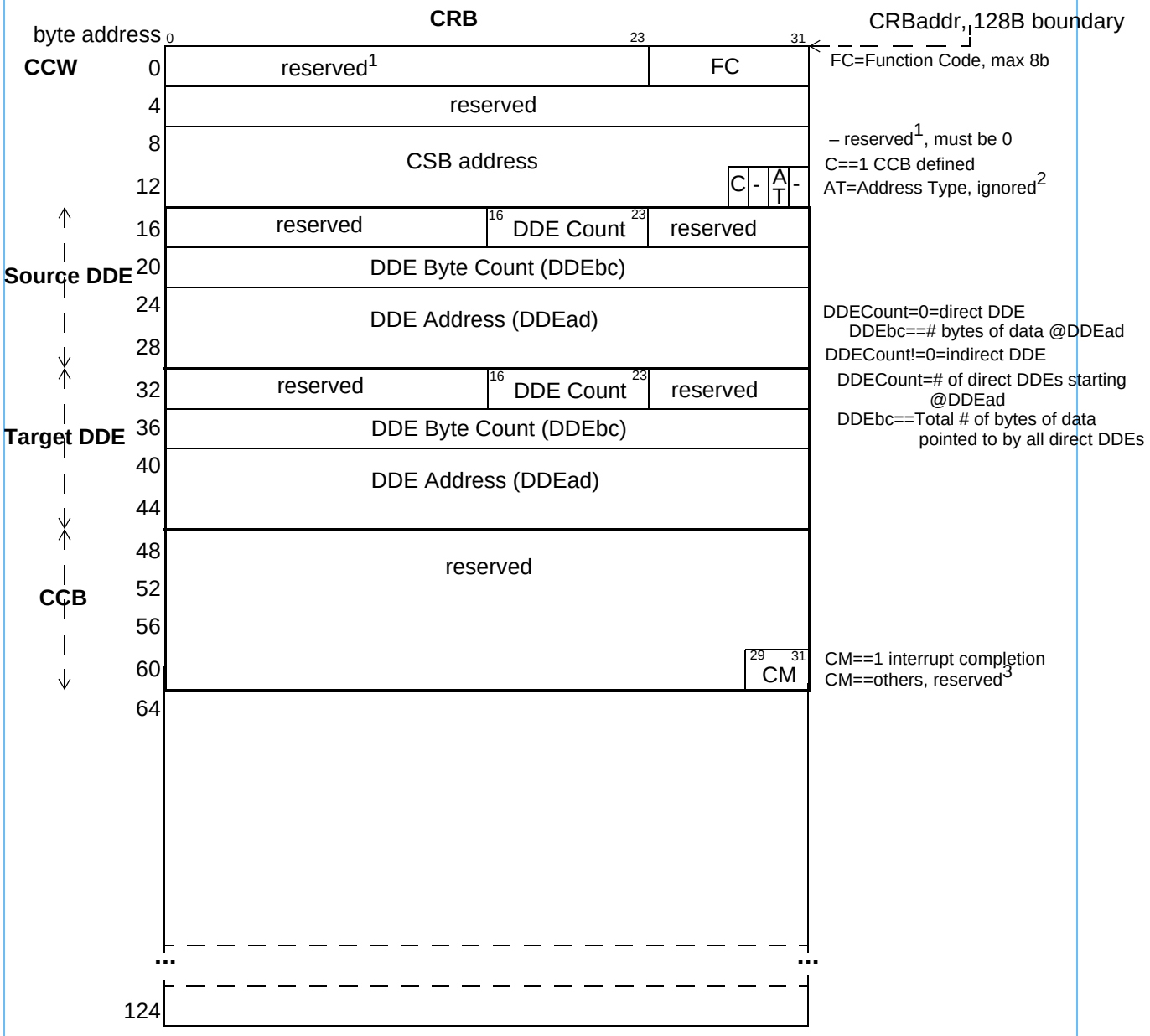
Figure 6-3. CRB C=0



Notes:

1. Reserved bits must be 0. Non-zero terminates the CRB with “CRB Format Error”.
2. AT=0, all addresses EA, assumed.
3. Contents of bytes 64 - 127 are ignored by NX.

Figure 6-4. CRB C=1



Notes:

1. Reserved bits must be 0. Non-zero terminates the CRB with “CRB Format Error”.
2. AT=0, all addresses EA, assumed.
3. Specifying a reserved code point terminates the CRB with “CRB Format Error”.
4. Contents of bytes 64 - 127 are ignored by NX.

6.4 DDE Description

This section describes the details of Data Descriptor Elements (DDE). These CRB structures point to source and target data either directly or indirectly using DDE Lists (DDL). With reference to *Figure 6-3* and *Figure 6-4*:

When the DDEcount field==0, the DDEad is a pointer to the single data buffer, and the DDEbc is the length of the buffer in bytes. This type of DDE is called a direct DDE.

When the DDEcount field >=1, the DDE is an indirect DDE. In a indirect DDE, the DDEad is a pointer to the first DDE of a contiguous set of DDEcount direct DDE(s) called a DDL, which together point to the data. The format of a DDE in a DDL is shown in *Figure 6-5*. In an indirect DDE, the DDEbc is the total length of all the data pointed to by all the direct DDE(s) in the DDL, in bytes.

For a source indirect DDE: If indirect DDEbc is <= the sum of all the direct DDEbc values, the accelerator will process only indirect DDEbc bytes, and no error has occurred. If indirect DDEbc is > the sum of all the direct DDEbc values, the accelerator will terminate with a “DDE Overflow” error, and the accelerator will process only the sum of all the direct DDEbc values.

For a target indirect DDE: If the amount of target data generated is > the sum of all the direct DDEbc values, the accelerator will terminate with a “Target Space Exhausted” error.

When a target indirect DDE is not being used for a CT that does not produce target data, then the target DDEcount field must be zeros. If the DDEcount field is not zeros, then an attempt will be made to prefetch the first direct DDE, which in turn may lead to undesired effects. For a CT that does not generate target data as may be the case when all output data written to the CPB, if the target DDEcount field is non-zero, the CRB will terminate with an “Excessive DDE Count error”, since use of indirect DDE is never allowed for this CT.

Restrictions when using DDEs:

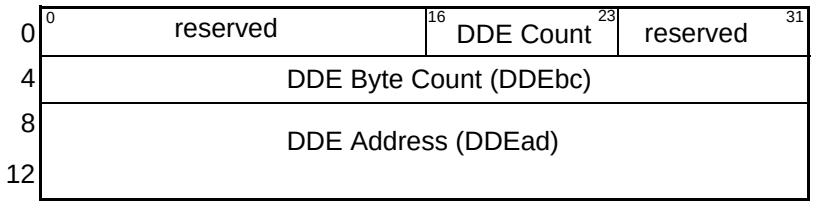
The contiguous set of DDEs pointed to by an indirect DDE must themselves all be direct DDEs. If any of them are indirect DDEs, the accelerator will terminate with an error. This is a programming error, and will be reported via the CSB CC Segmented-DDL Exception code.

Reserved bits in a DDE must be zero otherwise the job is terminated with a non-zero CC.

Source and target address overlap is not detected. If source and target addresses overlap, execution result is boundedly undefined.

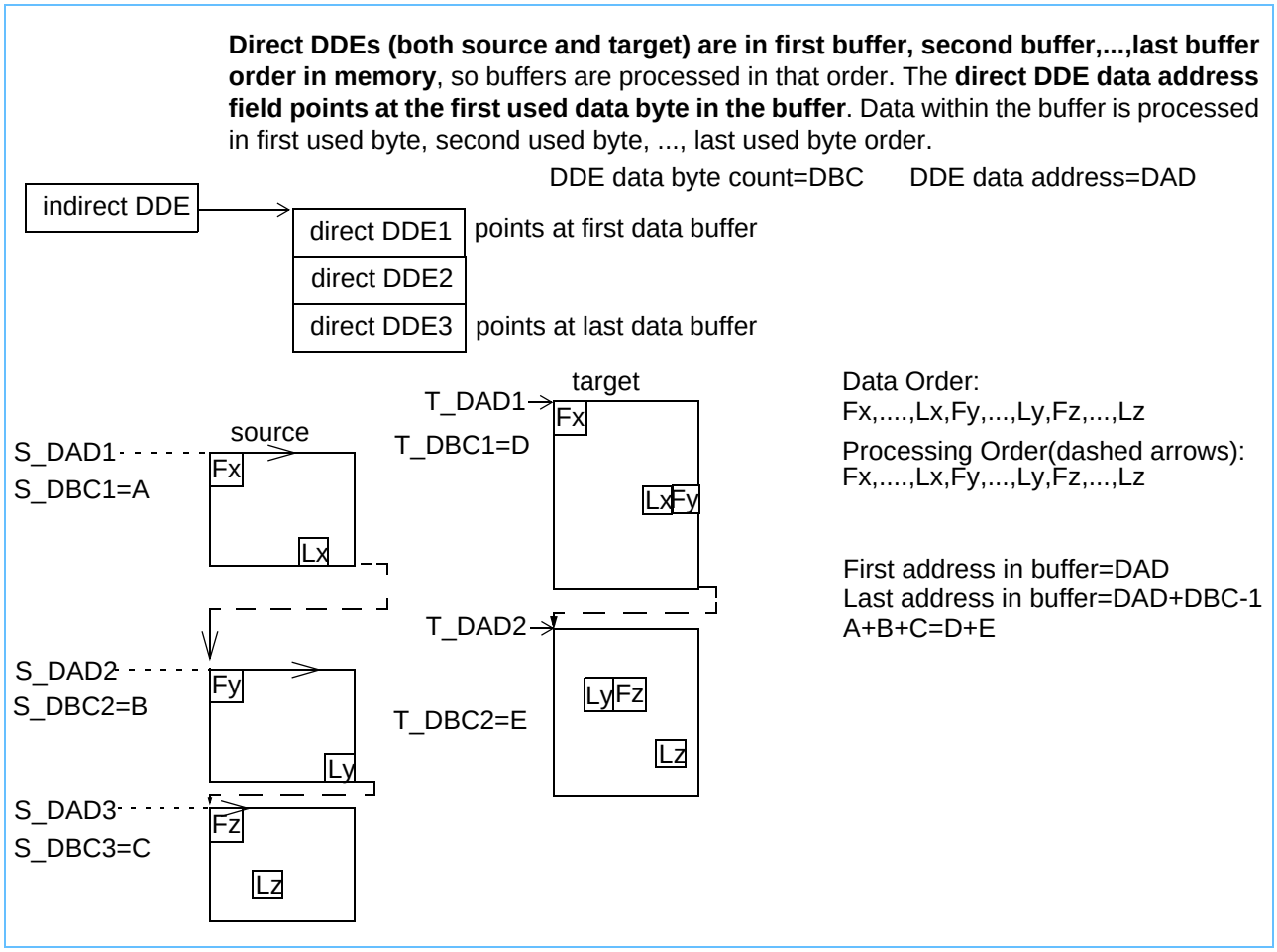
Target and CPB or CSB address overlap is not detected. If such overlap occurs, execution result is boundedly undefined.

Figure 6-5. DDE format in DDL



DDE processing is described in further detail in *Figure 6-6 DDE processing* on page 53.

Figure 6-6. DDE processing



6.5 Job Length Limits

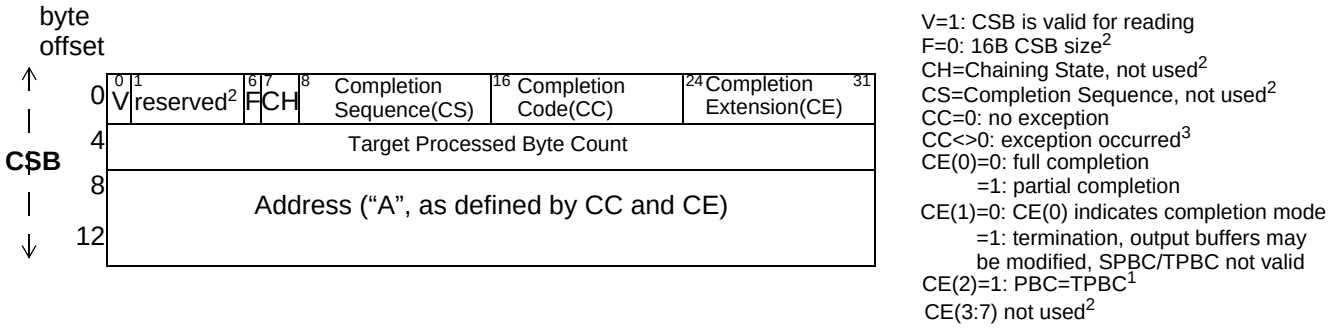
Limits may be specified in *GZIP Maximum Byte Count Register* and *GZIP Maximum Byte Count Register* and are applied to either source or target byte counts as defined in the registers. Separate limits are provided for high- and normal-priority CRBs. A CRB so rejected is terminated by the DMA Controller with the CC indicated in *Table 6-8 CSB Non-zero CC Reported Error Types* on page 66. A CRB so-terminated never begins execution.

The Gzip accelerator has the ability to suspend a job when a maximum byte count has been reached. This is described in the *GZIP Maximum Byte Count Register*, *Section 6.11.5 Autonomic Gzip job length limits* on page 64, and *Section 5 P9 Gzip Accelerator* on page 35.

6.6 Coprocessor Status Block (CSB) Details

NX supports a 16-byte, size-aligned Coprocessor Status Block (CSB). The format of the CSB with definitions of its fields is given in *Figure 6-7*.

Figure 6-7. NX CSB Format



Notes:

1. TPBC (Target Processed Byte Count) will reflect the count of target data that was *successfully* stored.
2. Written to 0 by DMA when it writes the CPB.
3. See *Section 6.12 Error and Status Reporting* on page 65 for non-zero CC definitions.

NX will always store the CSB using a partial cacheline inject, which provides lower latency for software polling CSB V=1. (If the line is not held as HPC in a cache, the memory controller will accept the store data.)

Programming Notes:

- SPBC may or may not be written in termination scenarios caused by errors. It is recommended to initialize SPBC to 0 to avoid confusion in these scenarios.
- In P9 hardware CE(2) = 1 always. The coprocessor architecture allows CE(2) = 0 to indicate that the PBC is SPBC, the count of source data that was successfully processed. In P9, if there is an SPBC, it is stored in the CPB output. In a partial completion or error scenario, SPBC may be less than, greater than, or equal to TPBC.

6.7 Coprocessor Completion Block (CCB) Details

The CCB is defined to exist within the CRB if C = 1 as shown in *Figure 6-4*. The CCB is defined to contain the 3-bit CM field of which only CM=1 is valid and indicates interrupt completion. NX signals an interrupt by performing a cache-inhibited write to the Interrupt Address. The Interrupt Address is the User Interrupt RA defined in the user’s send window context.

6.8 Coprocessor Parameter Block (CPB) Details

The optional CPB, if it exists for a given CRB, shall be contiguous with the CSB as shown in *Figure 6-8*. The preferred location of the CSB is the last QW in a cache line — in this case, the CPB begins in the next contiguous cache line as shown. The FC of an operation defines whether input and output parameters are present.

The CPB is typically unique for each CT. Definitions for particular CT CPBs are given in the following CT description sections.

The specification of the CPB is such that NX CRB idempotency is maintained, i.e., input parameters are input-only, never overwritten as part of CRB processing.

Reserved bits in CPB are not checked for ones. Writing reserved bits to one is bad programming practice and compatibility with future NX implementations can not be guaranteed.

6.8.1 CPB Over-fetching

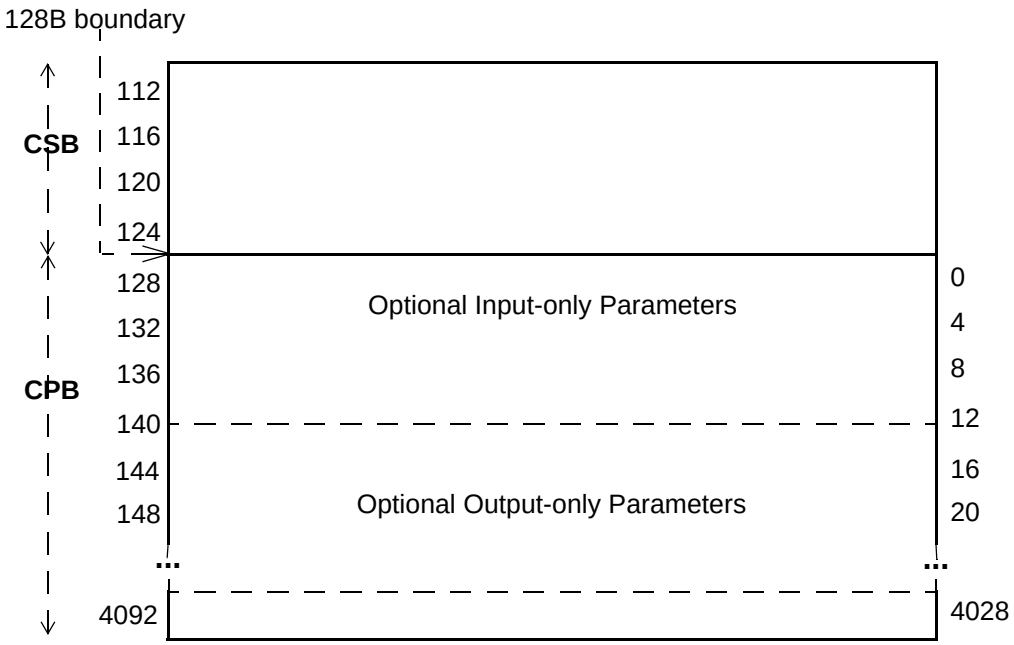
For the Gzip accelerator, the length of the CPB input regions varies depending on the FC. The DMA controller may fetch the maximum length input region per FC. These are

FC	Function	Maximum Input QWs
2, 3	compression DHT	19
6, 7	compression DHT with counts	19
10, 11	resume compression DHT	19
14, 15	resume compression DHT with counts	19
20 - 23	resume decompression block	19

A translation fault may be reported for a CPB input location even if the location is not used for the operation. Software should therefore ensure that translations exist for the maximum length input region for a given FC. See *Section 6.12 Error and Status Reporting* on page 65 and Gzip *Section 6.11.4 Gzip CPB Format* for details on the various CPBs.

An uncorrectable error (UE) detected on CPB over-fetched data may or may not be reported. If such a UE is reported, it is recoverable with a non-zero CC being returned in the CSB.

Figure 6-8. CPB details (preferred location shown)



6.9 Coprocessor Job Completion

When a CRB completes either with success (CC=0) or some exceptional condition (CC != 0), NX performs several steps:

1. Write CSB(V) = 1 and other CSB fields as appropriate.
2. Signal an interrupt if CRB(C) = 1 and CM = 1.
3. Return one send window credit to the VAS for the send window context associated with the CRB.

If the CRB encountered an address translation fault the CRB is not completed and none of the above steps are performed — see *Section 6.10*.

Programming Notes

It is best practice to place large timeout values on job submissions to NX. If software does not read CSB V=1 or receive a requested job completion interrupt notification within the timeout value, the user may assume that a catastrophic hardware upset has occurred.

Certain CRB format and hardware errors might preclude a completion interrupt. For example, if the user codes C = 1 and CM = 1 in the CRB but NX encounters a uncorrectable error on the CRB quadword containing CM, NX will write CSB(V) = 1 and CSB(CC) = 20 “Corrupted CRB” and complete the job without posting a completion interrupt.

6.10 Address Translation

All addresses in a CRB and DDEs are considered Effective Addresses and are therefore subject to translation. In P9, the NMMU provides address translation services for various EA-using agents including NX. These services comprise

- translation table walks
- TLB for caching PTEs
- hypervisor real mode “translation”
- translation invalidation filtering

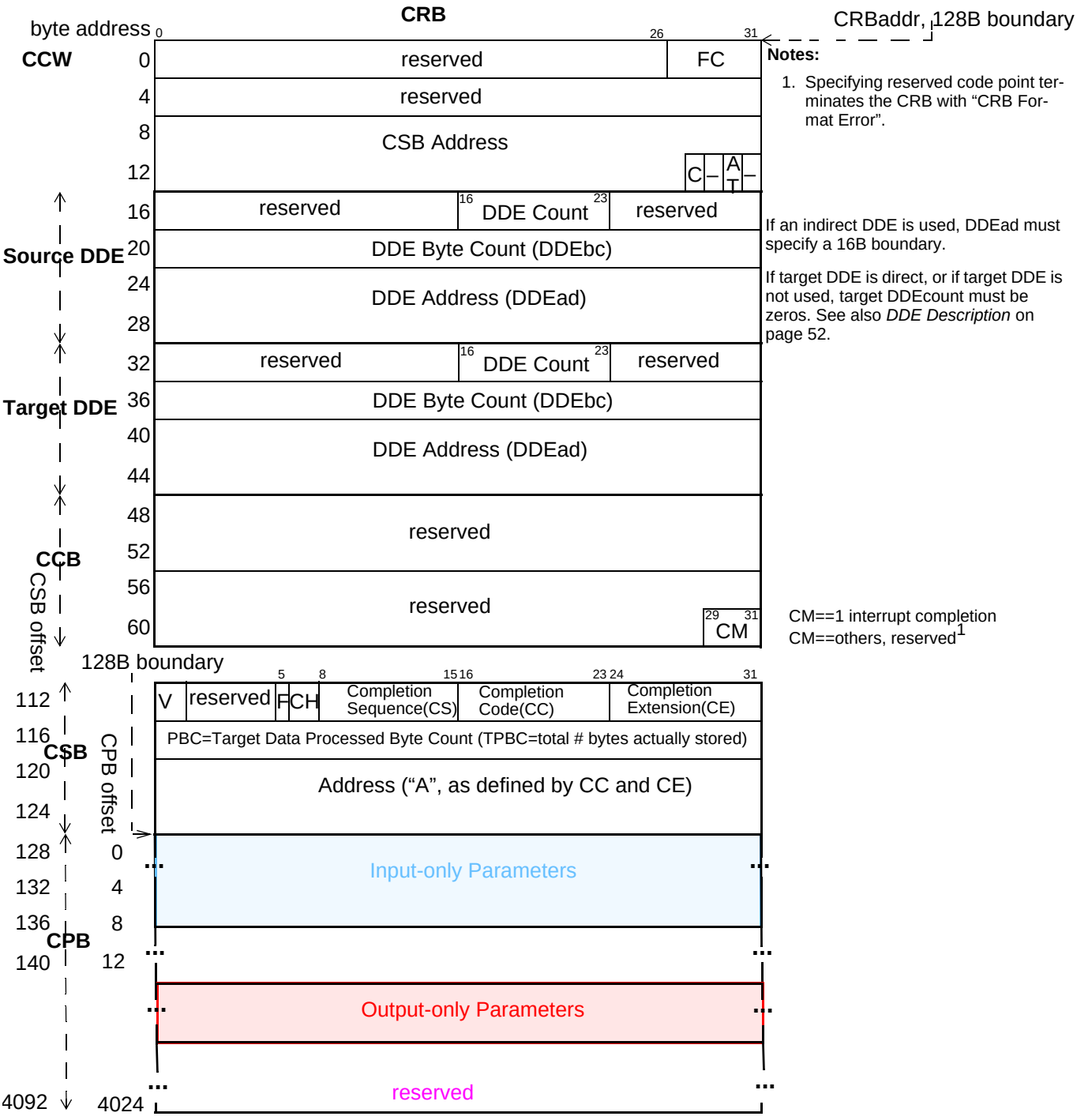
NX contains a 32-entry ERAT to cache recently used translations. Effective to real translations are requested from the NMMU and installed in the NX ERAT. The NMMU tracks ERAT entries in use for invalidation purposes.

An address translation attempt may fail — this is referred to as an address translation fault and is signaled to NX from NMMU by a non-zero translation checkout response status code. Fault handling is OS dependent: an OS may choose to repair the translation fault and replay the job on behalf of the user, or the OS may return the faulting CRB and storage address to the user. Any adjustment of send window credits for a faulting CRB is handled by system software.

Note that at the time of writing this document, IBM’s Linux kernel engineers were considering returning the faulting CRB to the user. User code may handle a fault by touching the faulting page or avoid faults by touching pages in advance.

6.11 Gzip Accelerator CRB/CCB/CSB/CPB Details

Figure 6-9. Gzip CRB Details



6.11.1 Gzip Function Codes

Gzip FC is a 6-bit field. Bits 0:4 indicate the function being performed by the accelerator. FC bit 5 is used by the DMA controller to select which byte count limit to use on that operation: if FC(5) = 0 *GZIP Maximum Byte Count Register[Max Byte Count High]* is used; if FC(5) = 1 *GZIP Maximum Byte Count Register[Max Byte Count Low]* is used. This is transparent to the gzip accelerator. The FC encodings are shown in *Table 6-2*.

Table 6-2. Gzip function codes

FC(0:5)	Function
00000X	Compression with fixed Huffman table.
00001X	Compression with dynamic Huffman table.
00010X	Compression with fixed Huffman table. Count LZ77 symbols.
00011X	Compression with dynamic Huffman table. Count LZ77 symbols.
00100X	Resume compression with fixed Huffman table.
00101X	Resume compression with dynamic Huffman table.
00110X	Resume compression with fixed Huffman table. Count LZ77 symbols.
00111X	Resume compression with dynamic Huffman table. Count LZ77 symbols.
01000X	Decompression.
01001X	Decompression. Process single block and suspend.
01010X	Resume decompression.
01011X	Resume decompression. Process single block and suspend.
01111X	Wrap. copy source data to target.

6.11.2 Gzip CPB Input Parameters

Gzip CPB input parameters and their definitions are shown in *Table 6-3*.

Table 6-3. Gzip CPB input parameter fields

Name	Description
Adler(0:31)	Adler32 checksum for zlib. During resume functions, the Adler32 checksum from a partial operation is used as a starting value.
CRC(0:31)	CRC32 checksum for gzip. During resume functions, the CRC32 checksum from a partial operation is used as a starting value. The least significant byte is in bits 0:7, and the most significant byte is in bits 24:31. This matches the gzip trailer byte ordering.
HistLen(0:11)	History Length. During resume functions, this is the number of 16 byte quadwords of source that are not processed but just used to load history. Any value is allowed. The maximum distance in the deflate standard is 32768 bytes, so setting HistLen greater than 2048 quadwords is wasting data. Only the last 32KB of history will be available for compression matching or decompression references.
SUBC(0:2)	Source Unprocessed Bit Count. During decompression resume functions, how many bits in the first source byte need to be processed. Value of “000” indicates all 8 bits. Note that the deflate standard has reversed bit ordering within each byte. So SUBC=”011” means bits 0:2 in the first source byte are unprocessed, bits 3:7 have already been processed.

Table 6-3. Gzip CPB input parameter fields

Name	Description
SFBT(0:3)	Source Final Block Type. From suspended decompression operation. Used during decompression resume. Indicates type of compressed data beginning with first unprocessed bit in first byte of source. “0XXX” : invalid. Accelerator will treat as block header with BFINAL=0. “1000” : type “00” literal block with BFINAL=0 “1001” : type “00” literal block with BFINAL=1 “1010” : type “01” fixed Huffman table block with BFINAL=0 “1011” : type “01” fixed Huffman table block with BFINAL=1 “1100” : type “10” dynamic Huffman table block with BFINAL=0 “1101” : type “10” dynamic Huffman table block with BFINAL=1 “1110” : block header with BFINAL=0. Since BFINAL is included in unprocessed bits, encoding is superfluous. “1111” : block header with BFINAL=1. Since BFINAL is included in unprocessed bits, encoding is superfluous.
RemByteCnt(0:15)	Remaining Byte Count. Used for resuming decompression of a type 00 literal block. Since literal blocks don’t contain an end of block indicator, a byte count needs to be passed in during a resume operation. The remaining byte count is given when the literal block is suspended, and is passed along on a resume.
DHTlen(0:11)	Dynamic Huffman Table length. Number of bits in compressed dynamic Huffman table for compression and decompression resume operations. Valid tables for the accelerator contain between 42 and 2283 bits. Accelerator uses value to determine how much CPB input is required. Values less than 128 bits will result in a single quadword of CPB being fetched, since that’s the finest granularity the dma channel supports. Values greater than 2304 bits (18 quadwords) will result in only 18 quadwords of CPB being fetched.
DHT	Dynamic Huffman Table. Compressed dynamic Huffman table. Same format used in deflate type “10” dynamic Huffman block header. Variable length between 42-2283 bits. Maximum of 18 quadwords. Sufficient quadwords will be fetched by the accelerator to cover the amount given by DHTlen.

6.11.3 Gzip CPB Output Parameters

The Gzip accelerator writes a variety of output parameters, depending on the operation requested. These are defined in *Table 6-4*.

Table 6-4. Gzip CPB output parameter fields

Name	Description
Adler(0:31)	Adler32 checksum for zlib. Produced by all compression, decompression functions.
CRC(0:31)	CRC32 checksum for gzip. Produced by all compression, decompression functions. The least significant byte is in bits 0:7, and the most significant byte is in bits 24:31. This matches the gzip trailer byte ordering.
TEBC(0:2)	Target Ending Bit Count. Produced by compression. Indicates number of valid bits in last byte of target. A helper field that tells software the position of the last bit of target for compression resume. A sync flush block (type “00” literal block with length of 0) may be required for compression resume since all compression target begins byte aligned. Note that the deflate standard has reversed bit ordering within each byte. So TEBC=3 (“011”) means bits 5:7 in the last target byte are valid, bits 0:4 are empty.
SUBC(0:15)	Source Unprocessed Bit Count. Produced when a decompression function stops receiving source before the end of the last block (also called suspend). Or if additional source is received after the final end-of-block code is seen, e.g. a trailer. This field indicates how many bits of source could not be processed. Worst case is when the source interruption occurs just after the beginning of a type “10” header containing a compressed dynamic Huffman table. The largest table is 2283 bits. Other interruption points will yield much less unprocessed source. This field is also used by software to figure out the first partial or full byte that wasn’t processed in preparation for resuming the decompression. The low order 3 bits are fed back to the accelerator as part of the resume function, along with the adjusted source. Note that the deflate standard has reversed bit ordering within each byte. So SUBC(13:15)=5 (”101”) means bits 0:4 in the first unprocessed source byte were unprocessed, bits 5:7 have been processed. ZLIB and GZIP trailers (checksum and ISIZE fields) if present in the stream will typically effect the SUBC count. For ZLIB and GZIP these values are 32 and 64 bits respectively. Since the trailer is byte aligned up to 7 unused bits may be present between the Deflate block and the trailer, and therefore the values may range from 32 to 39, and 64 to 71 bits respectively.

Table 6-4. Gzip CPB output parameter fields

Name	Description
SFBT(0:3)	Source Final Block Type. Produced when decompression stops receiving source, either normally or as a result of a suspend operation. Indicates type of compressed data being processed. If the source was suspended, this field is forwarded to the subsequent resume operation as SFBT. “0000” : final end-of-block (EOB) code received. Any source after this is counted in SUBC. “1000” : data within a type “00” literal block with BFINAL=0 “1001” : data within a type “00” literal block with BFINAL=1 “1010” : data within a type “01” fixed Huffman table block with BFINAL=0 “1011” : data within a type “01” fixed Huffman table block with BFINAL=1 “1100” : data within a type “10” dynamic Huffman table block with BFINAL=0 “1101” : data within a type “10” dynamic Huffman table block with BFINAL=1 “1110” : within a block header with BFINAL=0. Also given if source data exactly ends (SUBC=0) with EOB code with BFINAL=0. Means the next byte will contain a block header. “1111” : within a block header with BFINAL=1.
RemByteCnt(0:15)	Remaining Byte Count. Used for resuming decompression of a type 00 literal block.. Since literal blocks don’t contain an end of block indicator, a byte count needs to be passed in during a resume operation. The remaining byte count is given when the literal block is suspended, and is passed along on a resume.
DHTlen(0:11)	Dynamic Huffman Table length. Produced when a decompression function stops receiving source within a type “10” dynamic Huffman block. Number of valid bits in compressed dynamic Huffman table, which will also be written to allow the job to be resumed. Valid values from 42 to 2283 bits. Accelerator uses value to determine how much CPB output is written, always as 16B quadwords.
DHT	Dynamic Huffman Table. Compressed dynamic Huffman table produced when a decompression function stops receiving source within a type “10” dynamic Huffman block. Same format used in deflate type “10” dynamic Huffman block. Variable length between 42-2283 bits. Sufficient quadwords will be written by the accelerator to cover the amount given by DHTlen.
LZcount	LZ77 symbol counts. Compression has 24 bit saturating counters for each of the 286 LL symbols and 30 D symbols. Each count value is written right-aligned into 4 bytes of CPB. The count values are contained in 1264 bytes or 79 quadwords in sequential order, LL0-LL285, D0-D29. These counts can be used to generate a dynamic Huffman table to improve compression ratio.
SPBC(0:31)	Source Processed Byte Count. Appended by dma. Indicates how many bytes of source were given to the accelerator including history bytes.

6.11.4 Gzip CPB Format

Each Gzip FC has its own parameter requirements. This section describes where each field is placed within CPB input and output for each FC. In the following tables, each cell indicates the CPB quadword offset with bits in parentheses. For example, 0(3:5) in a cell means that quadword 0, the first quadword in CPB space, contains the field in bits 3:5. Empty cells means the field doesn’t apply to that function. Undefined CPB bits for a particular function can be set to any value. The cell headings are the parameter fields defined in *Table 6-3* and *Table 6-4*.

Table 6-5. CPB Field Assignments : Compression and Wrap

		Input					Output				
FC(0:5)	Function	Adler	CRC	HistLen	DHTlen	DHT	Adler	CRC	TEBC	LZcount	SPBC
00000X	Compression FHT						24 (0:31)	24 (32:63)	24 (77:79)		25 (0:31)
00001X	Compression DHT				0 (116:127)	1-18[1]	24 (0:31)	24 (32:63)	24 (77:79)		25 (0:31)
00010X	Compression FHT with counts						24 (0:31)	24 (32:63)	24 (77:79)	25-103	104 (0:31)
00011X	Compression DHT with counts				0 (116:127)	1-18[1]	24 (0:31)	24 (32:63)	24 (77:79)	25-103	104 (0:31)
00100X	Resume FHT compression	0 (0:31)	0 (32:63)	0 (64:75)			24 (0:31)	24 (32:63)	24 (77:79)		25 (0:31)
00101X	Resume DHT compression	0 (0:31)	0 (32:63)	0 (64:75)	0 (116:127)	1-18[1]	24 (0:31)	24 (32:63)	24 (77:79)		25 (0:31)
00110X	Resume FHT compression with counts	0 (0:31)	0 (32:63)	0 (64:75)			24 (0:31)	24 (32:63)	24 (77:79)	25-103	104 (0:31)
00111X	Resume DHT compression with counts	0 (0:31)	0 (32:63)	0 (64:75)	0 (116:127)	1-18[1]	24 (0:31)	24 (32:63)	24 (77:79)	25-103	104 (0:31)
01111X	Wrap						24(0:31)	24(32:63)			25 (0:31)

1. DHT is a variable length field, between 1 and 18 quadwords long. DHTlen indicates number of valid bits in DHT.

Table 6-6. CPB Field Assignments : Decompression

		Input								Output							
FC(0:5)	Function	Adler	CRC	His-tLen	SUBC	SFBT	Rem-ByteCnt	DHT-len	DHT	Adler	CRC	SUBC	SFBT	Rem-ByteCnt	DHT-len	DHT	SPBC
01000X 01001X	**Decompression** Source ends with final block. Source ends inside header. Source ends inside FHT block.									24 (0:31)	24 (32:63)	24 (80:95)	24 (108:111)				43 (0:31)
	Source ends inside literal block									24 (0:31)	24 (32:63)	24 (80:95)	24 (108:111)	24 (112:127)			43 (0:31)
	Source ends inside DHT block									24 (0:31)	24 (32:63)	24 (80:95)	24 (108:111)		24 (116:127)	25-42[1]	43 (0:31)
01010X 01011X	**Resume decompression header or FHT block** Source ends with final block. Source ends inside header. Source ends inside FHT block.	0 (0:31)	0 (32:63)	0 (64:75)	0 (93:95)	0 (108:111)				24 (0:31)	24 (32:63)	24 (80:95)	24 (108:111)				43 (0:31)
	Source ends inside literal block.	0 (0:31)	0 (32:63)	0 (64:75)	0 (93:95)	0 (108:111)				24 (0:31)	24 (32:63)	24 (80:95)	24 (108:111)	24 (112:127)			43 (0:31)
	Source ends inside DHT block.	0 (0:31)	0 (32:63)	0 (64:75)	0 (93:95)	0 (108:111)				24 (0:31)	24 (32:63)	24 (80:95)	24 (108:111)		24 (116:127)	25-42[1]	43 (0:31)
01010X 01011X	**Resume decompression literal block** Source ends with final block. Source ends inside header. Source ends inside FHT block.	0 (0:31)	0 (32:63)	0 (64:75)	0 (93:95)	0 (108:111)	0 (112:127)			24 (0:31)	24 (32:63)	24 (80:95)	24 (108:111)				43 (0:31)
	Source ends inside literal block.	0 (0:31)	0 (32:63)	0 (64:75)	0 (93:95)	0 (108:111)	0 (112:127)			24 (0:31)	24 (32:63)	24 (80:95)	24 (108:111)	24 (112:127)			43 (0:31)
	Source ends inside DHT block.	0 (0:31)	0 (32:63)	0 (64:75)	0 (93:95)	0 (108:111)	0 (112:127)			24 (0:31)	24 (32:63)	24 (80:95)	24 (108:111)		24 (116:127)	25-42[1]	43 (0:31)
01010X 01011X	**Resume decompression DHT block** Source ends with final block. Source ends inside header. Source ends inside FHT block.	0 (0:31)	0 (32:63)	0 (64:75)	0 (93:95)	0 (108:111)		0 (116:127)	1-18[1]	24 (0:31)	24 (32:63)	24 (80:95)	24 (108:111)				43 (0:31)
	Source ends inside literal block.	0 (0:31)	0 (32:63)	0 (64:75)	0 (93:95)	0 (108:111)		0 (116:127)	1-18[1]	24 (0:31)	24 (32:63)	24 (80:95)	24 (108:111)	24 (112:127)			43 (0:31)
	Source ends inside DHT block.	0 (0:31)	0 (32:63)	0 (64:75)	0 (93:95)	0 (108:111)		0 (116:127)	1-18[1]	24 (0:31)	24 (32:63)	24 (80:95)	24 (108:111)		24 (116:127)	25-42[1]	43 (0:31)

1. DHT is a variable length field, between 1 and 18 quadwords long. DHTlen indicates number of valid bits in DHT.

6.11.5 Autonomic Gzip job length limits

Capability is provided in the DMA current slot to use either a high or low maximum byte count limit depending on whether or not a high priority CRB is queued behind the current CRB in either the DMA pending slot, UMAC CRB dispatch slot, or Gzip High Priority Receive FIFO. This capability is enabled by *GZIP Maximum Byte Count Register[Threshold Mode]* = 1. When this bit is 1 and Gzip FC(5) = 0 (see below) *Table 6-7* lists which maximum byte count is applied by the DMA controller. Note that the priority of the CRB in DMA current does not matter: if a high priority job is running in the DMA current slot, it is subject to the low limit if another high priority job is queued behind it.

Table 6-7. Application of Gzip job length limits when Threshold Mode = 1 and FC(5) = 0

High priority CRB queued in UMAC or receive FIFO	CRB priority DMA pending	CRB priority DMA current	Use Max Byte Count Low on DMA current CRB?
0	Normal	X	N
0	High	X	Y
>0	X	X	Y

If FC(5) = 1 *GZIP Maximum Byte Count Register[Max Byte Count Low]* is used and *Table 6-7* does not apply. (see *Section 5.2.1 CRB FC field*). This capability is provided to allow an OS to force the lower limit in instances where memory pressure is high and thus page faults are running high and replaying a page faulting CRB in smaller pieces will allow the job to complete faster and more efficiently than replaying the job at the high limit for every page fault.

6.12 Error and Status Reporting

6.12.1 CSB Non-zero Error Type Summary

The table below lists errors that are detected and reported as non-zero CSB CC. The columns are defined as follows:

CC	Completion code in decimal.
Error Type	Descriptive name of error encountered.
FSA	Failing storage address. Address where error occurred. When available, written to A field of CSB.
SW reporting (CSB) used?	If “Y” then this error is reported by writing the given CC to the CSB. If “N” or some conditional statement is listed then the CSB may not be written.
HW error reporting (FIR) used?	If “Y” then this error sets a FIR bit; otherwise a FIR bit is not set.
subunit(s) detecting	Subunit(s) detecting the error: DMA controller, 842, sym, gzip.
CE(0)	If = 0 a full completion is indicated; if = 1 a partial completion is indicated. See RFC02130 for definition.
CE(1)	If = 0 then CE(0) applies; if = 1 then the job was terminated. See RFC02130 for definition.
Possible CT types	CT that can encounter the error.

Note: Whenever NX is writing status to the CSB, and the values of any of the output buffers (target data, CPB, CSB) have been modified, and the values of SPBC, TPBC or any other output buffer field may not be valid (e.g., the output buffers may not contain valid data, or the SPBC or TPBC do not correctly reflect the number of locations that have been modified in the output buffers, etc.), then the terminate bit (CE[1]) is asserted.

All error codes in the range 1 - 255 that are not listed in the table are reserved.

Hardware error reporting structures, including the FIR, are beyond the scope of this document.

Table 6-8. CSB Non-zero CC Reported Error Types

CC	Error Type	[3]FSA required?	SW reporting (CSB) used?	HW error report-ing used?	subunit(s) detecting (dma owns write to CSB)	CE(0)	CE(1)	Possible CT types	Condition/Comment
1	Invalid Alignment	Y	Y	N	dma	0	1	sym, gzip, 842	Indirect DDE address not on 16B boundary 842: direct DDE address not on 128B boundary

Table 6-8. CSB Non-zero CC Reported Error Types

CC	Error Type	[3]FSA required?	SW reporting (CSB) used?	HW error reporting used?	subunit(s) detecting (dma owns write to CSB)	CE(0)	CE(1)	Possible CT types	Condition/Comment
3	Data Length	N	Y	N	sym (invalid source data size), 842 (data size), dma, gzip	0	1	sym,gzip, 842	sym: incomplete last block for first or middle partial message, incomplete last block if padding is not specified. gzip compression, data move: accelerator detected history length field in CPB input greater than or equal to received source. See also *Table 5-1 Gzip Error Detection*. gzip decompression resume: accelerator detected amount of received source less than history length field in CPB input. See also *Table 5-1 Gzip Error Detection* and *Table 5-3 Gzip Decompress SFBT, SUBC combinations*. 842: non 8 byte multiple input; too much input data for decomp (end template seen, data still coming in); too little input data (reached end of data, no end template seen) for decomp.
	Data Length	N	Y	N	dma	1	0	gzip	gzip compression, data move: DMA detected source processed byte count exceeded byte count limit; job suspended. Check CPB output for more information. gzip decompression: DMA detected target processed byte count exceeded byte count limit; job suspended. Check CPB output for more information. If, after DMA suspends the job, gzip reports a non-zero CC prior to going idle, that CC is reported. If all source data has been sent to the accelerator and then target processed byte count exceeds byte count limit, the job completes normally. See also *Table 5-1 Gzip Error Detection* and *Table 5-3 Gzip Decompress SFBT, SUBC combinations*.
	Data Length	N	Y	N	gzip	1	0	gzip	Decompression: accelerator detected insufficient source. Source ended before the final block was processed. Check CPB output for more information. Decompression: accelerator detected too much source. A final block was processed and more source is available. Check CPB output for more information. See also *Table 5-3 Gzip Decompress SFBT, SUBC combinations*.

Table 6-8. CSB Non-zero CC Reported Error Types

CC	Error Type	[3]FSA required?	SW reporting (CSB) used?	HW error reporting used?	subunit(s) detecting (dma owns write to CSB)	CE(0)	CE(1)	Possible CT types	Condition/Comment
7	External Uncorrectable Error on Read	Y	Y	Y	dma	0	1	sym,gzip, 842	ECC UE on DDE data. The FSA is provided, which is guaranteed for source and target DDE addresses.
8	Invalid Operand Exception	N	Y	N	dma (invalid KeySize, invalid digest size, invalid input parm(s), invalid Mode)	0	1	sym	An input operand or input parameter is an invalid value. Invalid key size applies to AES, undefined digest size applies to SHA. *Appendix B Available Symmetric Cryptography Operations* defines valid input parameters for symmetric operations. If *DMA Configuration Register*[Disable AES/SHA CPB Checking]=0, and one or more input parameters are invalid, this CC is returned.
10	Internal Uncorrectable Error (state error)	N	N	Y	dma, sym, gzip, 842	0	1	sym, gzip, 842	Internal invalid state error found, state machines hold in error state. For gzip details see *Section 5.2.4 CSB Completion Codes*.
	Internal Uncorrectable Error (non-state error)	[1]N	Y, unless CSB addr is UE	Y	ECC UE: gzip, 842, dma	0	1	sym,gzip, 842	ECC UE. For gzip details see *Section 5.2.4 CSB Completion Codes*.
13	Target Space Exhausted	N	Y	N	dma	0	1	sym,gzip, 842	The total amount of target data generated by the algorithm is greater than the total available target space. This is caused by any target DDE (direct or indirect) with zero byte count, or the sum of all direct target DDE byte count(s) is less than the actual amount of target data requested to be moved.[11]
14	Excessive DDE Count	N	Y	N	dma	0	1	sym,gzip, 842	A source DDE (direct or indirect) with zero byte count was found (AES, AES-SHA combo, gzip, 842). Source/target DDE Count in CRB exceeds limit defined in *GZIP Maximum Byte Count Register*, *GZIP Maximum Byte Count Register*, *GZIP Maximum Byte Count Register*.
17	Unrecognized Subfunction	N	Y	N	dma (invalid FC, accelerator or CT disabled)	0	1	sym,gzip, 842	Invalid subfunction code detected. The accelerator has received a CRB that is requesting a function which the accelerator can not provide. This is because either the accelerator contains no such function, or the function has been completely disabled. The invalid function may be a request to one or more disabled accelerators, a disabled CT, or an unsupported FC.

Table 6-8. CSB Non-zero CC Reported Error Types

CC	Error Type	[3]FSA required?	SW reporting (CSB) used?	HW error reporting used?	subunit(s) detecting (dma owns write to CSB)	CE(0)	CE(1)	Possible CT types	Condition/Comment
19	Byte Count > Maximum Specified	N	Y	N	dma	0	1	sym, 842	CRB specifies a source (sym, 842 comp) or target (842 decomp) > the maximum allowed. See *Section 6.5 Job Length Limits*.
20	Corrupted CRB	N	Y, unless CSB add is UE	Y	ECC UE: dma	0	1	all	Detection of an ECC UE on a CRB. See also *FIR Error Reporting and Handling* on page 76.
21	Invalid CRB	N	Y	N	dma	0	1	all	CRB format errors: Non-zero detected in reserved field; C=1 and CM != 1
30	Invalid DDE	N	Y	N	dma	0	1	all	Non-zero bit detected in reserved field in DDE in DDL.
31	Segmented DDL	N	Y	N	dma	0	1	sym, gzip, 842	Any of the DDEs pointed to by an indirect DDE is itself an indirect DDE.
33	DDE Overflow	N	Y	N	dma	0	1	sym,gzip, 842	Indirect DDE DDEbc is > the sum of all the direct DDEbc values.[11]
64	TPBC > SPBC	N	Y	N	dma	0	0	842, gzip	Compression: If the TPBC > SPBC the CC value will be 64 instead of 0 if no other errors were found.
	AtE decrypt Buffer Overflow	N	Y	N	sym	0	1	sym	[7]AtE decryption buffer overflow, MAC is not properly generated, target data (Plaintext & MAC & CipherPad) is properly generated. This error may be recovered from fully as described in the footnote.
65	CRC-32 Mismatch	N	Y	N	842	0	0[12]	842	Decompression: If the CRC-32 of the output data does not match the imbedded CRC-32 sent with the input data. Accelerator does not send terminate thus CE(1)=0.
66	Decomp Illegal Template	N	Y	N	842	0	1	842	Decompression: if the input data contains an illegal template field.
	DHT code not found	N	Y	N	gzip	0	1	gzip	Compression: An LZ77 symbol was generated that doesn't have an associated code in the dynamic Huffman table. Due to an incomplete DHT from CPB input.
	Undefined Huffman code	N	Y	N	gzip	0	1	gzip	Decompression: An undefined huffman code was found in source data.
	Invalid block header	N	Y	N	gzip	0	1	gzip	Decompression: BTYPE=00 AND LEN != !NLEN; or BTYPE=11
									See also *Section 5.2.4 CSB Completion Codes*.

Table 6-8. CSB Non-zero CC Reported Error Types

CC	Error Type	[3]FSA required?	SW reporting (CSB) used?	HW error reporting used?	subunit(s) detecting (dma owns write to CSB)	CE(0)	CE(1)	Possible CT types	Condition/Comment
67	Decomp Pointer Out of Range	N	Y	N	842	0	1	842	Decompression: If the Template indicates data beyond the end of the input stream.
	Invalid Distance	N	Y	N	gzip	0	1	gzip	Decompression: A length, distance symbol references data beyond the beginning of the history.
									See also *Section 5.2.4 CSB Completion Codes*.
68	Invalid DHT	N	Y	N	gzip	0	1	gzip	Invalid DHT detected. See *Section 5.2.4* for details.
90	External Uncorrectable Error on Read	N	Y	Y	dma	0	1	sym,gzip, 842	Error occurred on source data read or CPB data read on GBIF.
Error codes 224 - 239 (0xE0 - 0xEF) are reserved for hardware errors.									
224	Watchdog Timer Expired	N	Y	Y	dma	0	1	sym, gzip, 842	Job did not finish in the maximum time allowed; job terminated.
Error codes 240 - 255 (0xF0 - 0xFF) are reserved for system software use and are not signaled by the hardware. These CCs are written in the CSB by system software to signal the indicated conditions to the entity originating the associated CRB.									
240 - 249									reserved for future system software use
250									Address translation fault occurred.
251									reserved for future system software use
252									No valid interrupt server.
253									UE occurred.
254		N	Y	N	N/A			all	No functional hardware available to complete operation.

Table 6-8. CSB Non-zero CC Reported Error Types

CC	Error Type	[3]FSA required?	SW reporting (CSB) used?	HW error report-ing used?	subunit(s) detecting (dma owns write to CSB)	CE(0)	CE(1)	Possible CT types	Condition/Comment
255		N	Y	N	N/A			all	Hung async operation detected and killed.
All error codes not listed in this table that are in the range 1 - 255 are reserved.									

Notes:

1. N/A
2. N/A
3. The Failing Storage Address(FSA), when present, will be located in the “A” field of the CSB.
4. N/A
5. N/A
6. N/A
7. Note that this error may be fully recovered from as follows: Software strips the MAC and the CipherPad from the data in the AtE decrypt CRB’s target buffer. The remaining data is used as the source data for a SHA-HMAC CRB that uses the same digest size as the AtE decrypt CRB. This SHA-HMAC CRB will generate a MAC correctly in all cases.
8. N/A
9. N/A
10. N/A.
11. In scenarios where target DDE in the CRB is indirect, whether CC=13 (Target Space Exhausted), CC=33 (DDE Overflow), or CC=0 is presented in a given scenario depends on the ordering of certain events between DMA Controller and accelerator. There are six cases. In the cases, “undiscovered” means the potential for a non-zero CC completion exists, but the accelerator stops writing target data before the DMA Controller encounters the error condition, whereas “discovered” means the DMA Controller does encounter the error condition; targetIndirectByteCount is the target indirect byte count in the CRB; sum(DDE(targetByteCount)) is the sum of the byte counts in the target DDEs from the DDL processed by the DMA Controller.

    Case 1, CC=0: targetIndirectByteCount > sum(DDE(targetByteCount)) undiscovered - accelerator stops writing target before sum(DDE(targetByteCount)) has been decremented to 0 by DMA Controller.

    Case 2, CC=33: targetIndirectByteCount > sum(DDE(targetByteCount)) discovered - accelerator requests to write at least one byte of target and sum(DDE(targetByteCount)) has been decremented to 0.

    Case 3, CC=0: targetIndirectByteCount < sum(DDE(targetByteCount)) undiscovered - accelerator stops writing target before targetIndirectByteCount has been decremented to 0.

    Case 4, CC=13: targetIndirectByteCount < sum(DDE(targetByteCount)) discovered - accelerator requests to write at least one byte of target and targetIndirectByteCount has been decremented to 0.

    Case 5, CC=0: targetIndirectByteCount = sum(DDE(targetByteCount)) - accelerator stops writing target before targetIndirectByteCount has been decremented to 0

    Case 6, CC=13: targetIndirectByteCount = sum(DDE(targetByteCount)) - accelerator requests to write at least one byte of target and targetIndirectByteCount has been decremented to 0.
12. May be 1 in the rare case of concurrent watchdog timer expiration and CRC error detected (CC=65).

Summary of errors detected in algorithm engines from table above:

- Unrecognized Subfunction: sym
- Data Length Error: sym
- Invalid Operand: sym
- Internal Correctable (ECC CE): 842, gzip
- Internal Uncorrectable (ECC UE or state error): sym, 842, gzip

7. References

7.1 P9 Gzip user-mode support

At the time of writing this document, Linux system software supporting P9 Gzip was being discussed in the linuxppc-embedded mailing list. Search for the keyword VAS and nx842 in the kernel archives.

Copy/Paste Facility described in Power ISA AS Version 3.0.

VAS code sample teaches how to open a VAS window and do a Copy/Paste to it

For most up to date information, documentation and code samples visit

https://github.com/abalib/power-gzip

7.2 The NX-842 compression accelerator references

NX-842 is a proprietary compression accelerator that is being ported from P8 to P9. The P9 Gzip system software may share many of the interfaces with NX-842.

AIX kernel calls for 842 (EFT)

Linux calls for 842 (EFT) on PowerVM

nx-842.c is top level file which calls nx-842-powernv.c (powerNV) or nx-842-pseries.c (powerVM)

Revision Log

Revision Date	Modification
May 23, 2017	Version 0.5. Initial draft.
June 2, 2017	Version 0.6. Target buffer full, *Section 4.5* on page 30 added
Aug 1, 2017	Version 0.7. Target buffer full minor enhancements. Remove crypto and rng from overview and NX block diagram
Dec 15, 2017	Version 0.8. Accelerator sets BFINAL=1 by default not 0.