Network Working Group
INTERNET-DRAFT
Expires: July 2002

Richard Price, Siemens/Roke Manor
Jonathan Rosenberg, dynamicsoft
Carsten Bormann, TZI/Uni Bremen
H. Hannu, Ericsson
Z. Liu, Nokia

28 January, 2002

Universal Decompressor Virtual Machine (UDVM)
<draft-ietf-rohc-sigcomp-udvm-00.txt>

Status of this Memo

This document is a submission to the IETF ROHC WG. Comments should be directed to the mailing list of ROHC, rohc@cdt.luth.se.

Abstract

This draft defines a "Universal Decompressor Virtual Machine" optimized for the task of running decompression algorithms. The UDVM can be configured to understand the output of many well-known compressors such as [DEFLATE].

Revision history

Changes from <draft-ietf-rohc-sigcomp-algorithm-00.txt>

State creation mechanism modified to use MD5 hash for improved security

Support added for streaming compressed data over TCP

Memory format modified to allow compilation of UDVM code

Additional instructions added for bit manipulation etc.

Feedback mechanism added for bidirectional UDVM operation

Table of contents

1. Introduction....3
2. Terminology....3
3. Description of the UDVM architecture....5
3.1. UDVM architecture....5
3.2. Requirements on application....7
3.3. Requirements on transport mechanism....9
3.4. Requirements on compressor....10
3.5. Application-defined parameters....11
4. Overview of the UDVM....14
4.1. UDVM memory allocation....14
4.2. Well-known variables....15
4.3. Instruction parameters....15
4.4. Byte copying....16
5. Decompressing a compressed message....17
5.1. Invoking the UDVM....17
5.2. Successful decompression....19
5.3. Decompression failure....20
6. UDVM instruction set....21
6.1. Bit manipulation instructions....22
6.2. Arithmetic instructions....23
6.3. Memory management instructions....23
6.4. Program flow instructions....25
6.5. I/O instructions....27
7. Feedback information....31
7.1. UDVM version....33
7.2. Memory size and CPU cycles....33
7.3. State identifiers....34
8. Security considerations....34
9. Acknowledgements....36
10. References....36
11. Authors' addresses....36
Appendix A. Mnemonic language....38

Appendix B. Example application-defined parameters.....40
Appendix C. Example decompression algorithms.....42

1. Introduction

This draft defines a "Universal Decompressor Virtual Machine" (UDVM). The UDVM is a virtual machine much like the Java Virtual Machine but with a key difference: it is designed solely for the purpose of running decompression algorithms.

The motivation for creating the UDVM is to provide unlimited flexibility when choosing how to compress a given item of data. Rather than picking one of a small number of pre-negotiated compression algorithms, the implementer has the freedom to select an algorithm of their choice. The compressed data is then combined with a set of UDVM instructions that allow the original data to be extracted, and the result is outputted as UDVM bytecode.

Since the UDVM is optimized specifically for running decompression algorithms, the code size of a typical algorithm is small (often sub 100 bytes). Moreover the UDVM approach does not add significant extra processing or memory requirements compared to running a fixed pre-programmed decompression algorithm.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC-2119].

Virtual machine

A machine architecture designed to be implemented in software (although silicon implementations are of course possible).

Universal Decompressor Virtual Machine (UDVM)

The virtual machine described in this draft. The UDVM is designed specifically for the task of running decompression algorithms.

Bytecode

Machine code that can be executed by a virtual machine. UDVM bytecode is a combination of UDVM instructions and compressed data.

Application

Entity which invokes the UDVM. The application is also responsible for supplying the compressed data to the UDVM and making use of the uncompressed data.

Transport mechanism

Mechanism for passing data between two instances of an application. The UDVM is designed to work in conjunction with a wide range of transport mechanisms including TCP, UDP and [SCTP].

Message-oriented transport mechanism

A transport mechanism that carries data as a set of distinct, bounded messages.

Stream-oriented transport mechanism

A transport mechanism that carries data as a continuous stream with no message boundaries. In this case, the UDVM reserves a specific character to delimit messages in the compressed stream.

Compressor

Entity which converts application data into compressed data that can be reconstructed by the UDVM.

Application-defined parameters

Parameters that must be agreed upon by the application invoking the compressor and the application invoking the UDVM. Depending on the application these parameters might be fixed a-priori or negotiated.

Per-message compression

Compression that does not reference data from previous messages. The UDVM can decompress a message of this type using only the application-defined parameters and the data in the message itself.

Dynamic compression

Compression relative to messages sent prior to the current compressed message. The UDVM stores and retrieves this data using the secure state reference mechanism.

State

Information which is saved by the UDVM and retrieved for the decompression of subsequent messages. For security reasons, state can only be saved with the permission of the application and can only be retrieved using an [MD5] hash of the state.

State identifier

A 16-byte value used to access an item of stored state information. (for security it is the first n bytes of an [MD5] hash of the state to be accessed). The minimum acceptable value of n is fixed for security purposes, but implementers can choose higher values of n.

CPU cycles

A measure of the amount of "CPU power" required to execute a UDVM instruction (the simplest UDVM instructions require a single CPU cycle). An upper limit is placed on the number of cycles that can be used to decompress each bit in a compressed message.

3. Description of the UDVM architecture

This chapter describes the overall UDVM architecture including the interfaces between the UDVM and its environment. The requirements on the entities external to the UDVM are also given.

In the architecture the UDVM is considered to provide a decompression service for a certain application. The application invokes the UDVM, and is responsible for supplying compressed data to the UDVM and making use of the corresponding uncompressed data.

In general the UDVM can offer a decompression service to a wide range of applications. The principal motivation for developing the UDVM has been the compression of application-layer protocols, in particular text-based signaling protocols such as [SIP]. The UDVM architecture is designed to operate securely and to provide a high compression ratio for this case.

Note however that the UDVM can be used in any situation provided that the requirements detailed in this chapter are satisfied by the application and the transport mechanism.

The following sections describe the overall UDVM architecture and the requirements on entities external to the UDVM such as the transport mechanism and the compressor.

3.1. UDVM architecture

The UDVM architecture includes the following basic entities, each of which is defined in subsequent sections of the document:

UDVM
Application (including state)
Transport mechanism
Compressor

Two variants of the architecture are available, depending on whether the transport mechanism offers unidirectional or bidirectional data transport. The unidirectional architecture can be considered to be a special case of the bidirectional version.

Note that the UDVM itself does not need to know which architecture has been chosen, because its operation is identical for both cases.

If bidirectional data transport is unavailable or undesirable for any other reason, then the UDVM architecture is illustrated in Figure 1.

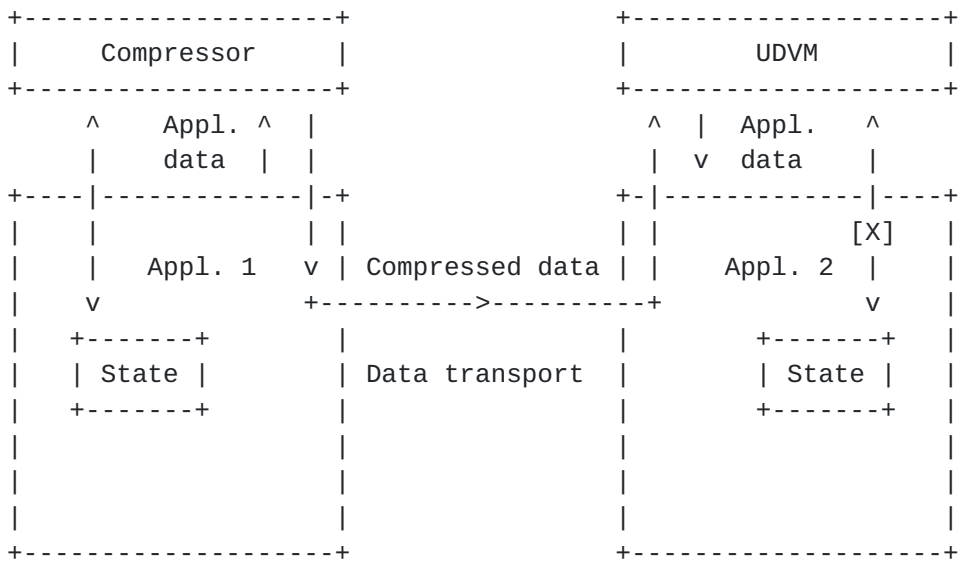


Figure 1: UDVM architecture for unidirectional data transport

In the unidirectional case the UDVM has two 2-way interfaces to the application. The first interface passes compressed data from the application to the UDVM, and provides the corresponding uncompressed data in return. The second interface allows the UDVM to request the creation of state (information that may improve the compression ratio of subsequent messages), and to access previously stored state.

Note that both of these interfaces can be provided as extensions to an existing application (e.g. a SIP client) or as a "shim" layer between a compression-unaware client and the UDVM. In the latter case, the term "application" refers to the combined client and shim layer.

The [X] symbol denotes that the application has a veto over the corresponding interface. In this case the application has veto over the state interface and can refuse state creation requests if it considers them to be inappropriate. See Section 3.2.2 for further details.

Note that although the UDVM architecture only shows one compression entity, it is possible for the UDVM to decompress messages from multiple compressors at different physical locations in a network. The UDVM architecture is designed to prevent data from one compressor interfering with data from a different compressor. A consequence of this design choice is that it is difficult for a malicious user to disrupt UDVM operation by inserting false compressed messages on the transport mechanism.

If the transport mechanism exchanges data in both directions then the architecture of Figure 2 can also be used. In this case, two

instances of the application communicate using a bidirectional transport mechanism. Both instances of the application invoke a compressor to compress their data, and a UDVM to retrieve the uncompressed data sent by the remote application.

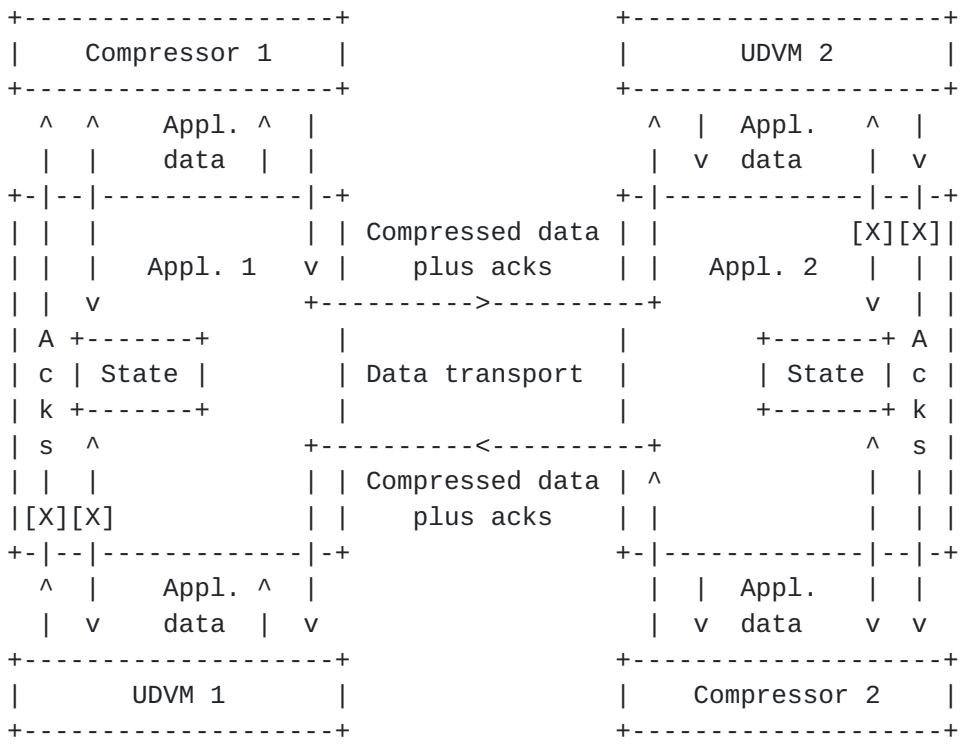


Figure 2: UDVM architecture for bidirectional data transport

For a bidirectional transport mechanism an additional interface is provided from the UDVM, via the application, to the compressor on the reverse transport channel. This interface can be used to send feedback information from an application to the remote compressor. The path taken by feedback data between Application 2 and Compressor 1 is as follows:

Appl. 2 --> Compressor 2 --> UDVM 1 --> Appl. 1 --> Compressor 1

This feedback information monitors the behavior of the UDVM, including whether data has been successfully decompressed, the amount of available memory etc. The compressor can make use of this information to improve the overall compression ratio.

Note that it is an implementation decision whether to use the feedback channel or not, and compressors must operate successfully even if no feedback information is received.

3.2. Requirements on application

The application is the entity responsible for invoking the UDVM, supplying the UDVM with compressed data, and making use of the corresponding uncompressed data.

Note that in order to use the UDVM decompression service an

application (e.g. a SIP client) will require interfaces to the UDVM. These interfaces can be provided by extending the original client, or by providing a "shim" layer between a compression-unaware client and the UDVM.

In addition, two instances of an application MUST agree on how to invoke the UDVM decompression service, and MUST fix or negotiate a common set of application-defined parameters (e.g. maximum_compressed_size) as per Section 3.5.

Certain application-defined parameters can be modified on the fly using the state creation mechanism and the feedback mechanism. The application SHOULD additionally provide some external means of resetting or renegotiating these parameters (possibly by terminating the decompression service offered by the UDVM).

The UDVM has a total of three interfaces to the environment: a two-way interface for exchanging compressed and uncompressed data, a two-way interface for storing and receiving state, and a one-way interface for forwarding feedback data. To protect against the malicious establishment of false state or false feedback data all of the UDVM interfaces pass through the application, and requests for state creation and feedback can be rejected if they are not accompanied by a valid uncompressed message.

Each of the three interfaces is described in greater detail below:

3.2.1. Compressed and uncompressed data

The first interface supplies compressed data to the UDVM and retrieves the corresponding uncompressed data. Note that when the UDVM is invoked it does not receive any compressed data by default, but instead requests new data explicitly using a specific instruction. This means that the first part of a message can be decompressed without waiting for the entire message to arrive, which is especially useful over a stream-oriented transport such as TCP.

Uncompressed data is also passed to the application using a specific instruction. It is an application decision whether to make use of the data immediately or to buffer and wait for a complete message to be successfully decompressed.

3.2.2. Storing and retrieving state

To provide security against the malicious insertion of false compressed data, the contents of the UDVM memory are reinitialized after each compressed message. This ensures that damaged compressed messages do not prevent the successful decompression of subsequent valid messages.

Note however that the overall compression ratio is often significantly higher if messages can be compressed relative to the information stored in previous messages. For this reason it is possible for the UDVM to create "state" information for access when a later message is being decompressed.

Both the creation and access of state are designed to be secure against malicious tampering with the compressed data. State can only be created when a complete message has been successfully

decompressed, and the application can veto a state creation request based on the contents of the decompressed message. This is especially useful if the application has an authentication mechanism that can be applied to determine whether the uncompressed data is legitimate.

Furthermore, the UDVM can only access previously created state information by providing an [MD5] hash of the state to be accessed. The advantage of using a secure hash to access state information is that it is very difficult to guess the correct hash value without complete knowledge of the state being accessed.

Also note that state is not deleted when it is accessed. So even if a malicious user manages to access state information, subsequent messages compressed relative to this state can still be successfully decompressed. Instead, the application is responsible for deleting state information once it determines that the state will no longer be needed.

3.2.3. Feedback information

The final interface is only used when the transport mechanism is bidirectional. It provides feedback information from the UDVM to the compressor on the reverse channel, and can be used to improve the overall compression ratio.

Note that the feedback information is forwarded via the application. Just as for the state interface above, the application can veto feedback information if it considers the corresponding decompressed message to be invalid.

If the transport mechanism only provides one-way data transport then the feedback interface is considered to be null: any feedback information sent across the interface is simply discarded by the application.

3.3. Requirements on transport mechanism

The transport mechanism is the entity that passes data between two instances of an application. Since the motivation for developing the UDVM has been the compression of signaling protocols such as [SIP], the UDVM is designed to operate successfully over both stream-oriented protocols such as TCP and message-oriented protocols such as UDP.

Note that the UDVM is not given direct access to the underlying transport mechanism; instead the compressed data is considered to first pass through the application. It is an application decision whether to pass all data from the transport mechanism directly to the UDVM or whether to mix compressed and uncompressed data (e.g. by restricting compressed data to a certain port).

If the transport mechanism is message-oriented then the UDVM converts each compressed message into a corresponding uncompressed message. It

is not possible for one compressed message to reconstruct multiple uncompressed messages.

If the transport mechanism is stream-oriented then the UDVM simply converts a stream of compressed data into a stream of uncompressed data. However, when running over a stream-oriented transport such as TCP, applications often insert their own internal message delimiters into the data stream. As the message is compressed, it will not be possible to detect these delimiters in the compressed data stream. Therefore the UDVM provides a similar character that can be inserted into the compressed data stream to delimit messages (see Section 3.4 for further details).

No assumption is made about the reliability of the transport mechanism. The UDVM can operate successfully over unreliable transport mechanisms such as UDP as well as reliable transport mechanisms such as TCP.

No assumption is made about the security of the transport mechanism. It may be possible for a malicious user to insert or modify data on the path between the compressor and the UDVM. In this case, the design goal of the UDVM is to avoid presenting additional security risks compared to simply transporting the application data uncompressed.

3.4. Requirements on compressor

An important feature of the UDVM is that it can decompress data generated by arbitrary compression algorithms. In particular this means that it is not necessary to standardize a compression algorithm for use with the UDVM; instead the choice can be left to the implementer.

The overall requirement placed on the compressor is that of transparency, i.e. the compressor MUST NOT send instructions which cause the UDVM to incorrectly decompress a given message.

The following more specific requirements are also placed on the compressor (they can be considered particular instances of the transparency requirement):

* Since feedback information is purely optional, the compressor MUST be able to operate successfully even if it receives no feedback data.

* It is RECOMMENDED that the compressor supply a CRC over the uncompressed message to ensure that successful decompression has occurred. A UDVM instruction is provided to verify this CRC.

* If the transport mechanism is message-oriented then the

compressor MUST preserve the boundaries between messages.

* If the transport mechanism is stream-oriented but the application defines its own internal message boundaries, then

the compressor SHOULD preserve the boundaries between messages by using the "end-of-message" character 0xFFFF reserved in UDVM bytecode.

The reason for preserving the message boundaries over a stream-oriented transport is that damage to one compressed message does not affect the decompression of subsequent messages. Moreover, the application typically vetoes state creation and feedback requests on a per-message basis.

Note that the UDVM also reserves the character 0xFF00 over a stream-oriented transport mechanism, and replaces every instance of 0xFF00 with 0xFF before decompressing the data. This ensures that arbitrary compression algorithms can be used over a stream-oriented transport, provided that every instance of 0xFF in the compressed data stream is identified and replaced with 0xFF00. This "byte-stuffing" scheme prevents the compression algorithm from inserting a message delimiter into the data stream where one is not required.

3.4.1. Types of compression algorithm

Any of the following classes of compression algorithm may be useful depending on the type of application:

* Generic compressor (for example [DEFLATE] or a similar algorithm).

* Protocol-aware compressor offering excellent performance for one particular type of data (for example the text messages generated by [SIP]).

* Hybrid compressor with similar performance to [DEFLATE] for generic data and superior performance for certain types of data.

Provided that the uncompressed data can be reconstructed at the UDVM using the available memory and CPU cycles, implementers have freedom to use a compression algorithm of their choice.

Note that when using an "off-the-shelf" compression algorithm, bytecode for the corresponding decompressor will need to be made available at the UDVM. In general the decompressor bytecode is placed at the front of the first compressed message, unless the application offers the ability to download UDVM bytecode offline (in which case the UDVM memory will be initialized already containing a copy of the decompression algorithm).

3.5. Application-defined parameters

When an application invokes an instance of the UDVM, a number of parameters are provided by the application to control the UDVM memory

size, maximum number of CPU cycles etc. The application invoking the UDVM and the application invoking the compressor MUST initially agree on a common set of values for these parameters.

Note that if the transport mechanism is bidirectional then the application invoking the UDVM can use the reverse channel to indicate that additional memory or CPU cycles are available (compared to the values initially agreed by the application invoking the compressor). The compressor can then make use of these extra resources to improve the compression ratio.

The feedback mechanism can also advertise that an upgraded version of the UDVM is available (e.g. offering additional UDVM instructions), provided that the upgraded version is backwards compatible with the basic version described in this document. See Chapter 7 for further details.

Each parameter is described in greater detail below; example values for the parameters are listed in Appendix B.

UDVM_version

The UDVM_version parameter specifies the level of functionality available at the UDVM. The basic version of the UDVM (Version 0) is defined in this document.

maximum_compressed_size

The maximum_compressed_size parameter limits the size of one compressed message. Decompression failure occurs if a message larger than the specified value is provided.

maximum_uncompressed_size

The maximum_uncompressed_size parameter limits the size of one uncompressed message. Decompression failure occurs if a message larger than the specified value is provided.

minimum_hash_size

The minimum_hash_size parameter specifies the minimum size of the state identifier that can be used to reference state. This value needs to be sufficiently large to prevent malicious users from guessing a state identifier by brute force.

overall_memory_size

The overall_memory_size parameter specifies the total number of bytes in the UDVM memory.

working_memory_start

The working_memory_start parameter specifies the start of the UDVM memory area that can be modified. Memory addresses below this

value are considered read-only by the UDVM.

working_memory_end

The working_memory_end parameter specifies the end of the UDVM memory area that can be modified. Memory addresses above this value are considered read-only by the UDVM.

cycles_per_bit

The cycles_per_bit parameter specifies the number of "CPU cycles" that can be used to decompress a single bit of data. One CPU cycle typically corresponds to a single UDVM instruction, although some of the high-level instructions may require additional cycles.

cycles_per_message

The cycles_per_message parameter specifies the number of additional CPU cycles made available at the start of a compressed message. These cycles can be useful when decompressing algorithms that download additional data on a per-message basis, for example a new set of Huffman codes as with [DEFLATE].

The total number of "CPU cycles" available for each compressed message is specified by the following formula:

total_cycles = message_size * cycles_per_bit + cycles_per_message

first_instruction

The first_instruction parameter specifies the memory address of the first instruction to be executed when the UDVM is initialized.

Initial memory contents

For each new compressed message the UDVM memory is reinitialized with contents defined by the application. For example, the application may be able to download UDVM bytecode for a decompression algorithm before the first compressed message arrives. In this case, for each new compressed message the UDVM memory is initialized already containing a copy of the decompression algorithm.

Initial state

As well as deciding the initial contents of the UDVM memory, the application can also store useful information in the form of state. This predefined state will typically contain optional data that can be used to improve the overall compression ratio, for example a well-known decompression algorithm or a dictionary of commonly used [SIP] phrases. Note that unlike state created on the fly by the UDVM, there is no need for the application-defined state to use an

[MD5] hash as the state identifier.

4. Overview of the UDVM

This chapter describes some basic features of the UDVM, including the memory allocation, well-known variables and instruction parameters.

4.1. UDVM memory allocation

The memory available to the UDVM is partitioned into a number of sections, providing space for program code, variables and miscellaneous data:

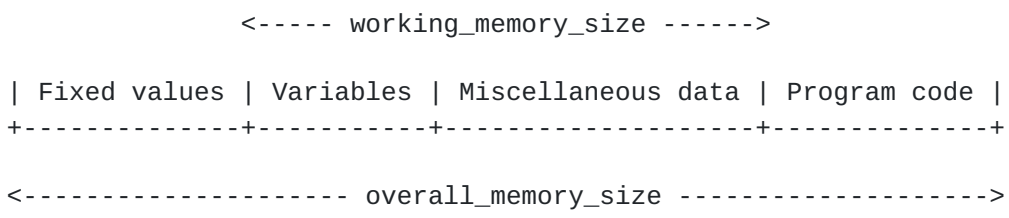


Figure 3: Memory allocation in the UDVM

Recall that the amount of memory available to the UDVM is defined by the application-specific parameters overall_memory_size, working_memory_start and working_memory_end. Note that all of these parameters are initialized by the application, but can be renegotiated on the fly using the feedback mechanism of Chapter 7.

The memory area from Address (working_memory_start) to Address (working_memory_end) inclusive can be used to store arbitrary data (variables, program code, Huffman codes etc.). UDVM instructions are allowed to read from or write to any address in this memory area.

The first part of this memory area is typically used to store a number of 2-byte variables. UDVM instructions can reference these variables using a special instruction parameter as described in Section 4.3.

The memory area from Address 0 to Address (working_memory_start - 1) and from Address (working_memory_end + 1) to Address (overall_memory_size - 1) inclusive is write-protected, so UDVM instructions can read from this memory area but cannot write to it. This memory area is intended for storing UDVM bytecode that can be compiled.

Any attempt to read memory addresses beyond the overall memory size or to write to memory addresses outside the working memory area MUST cause a decompression failure (see Section 5.3).

The first part of the write-protected UDVM memory is intended for storing variables whose values no longer need to be modified. The second part of the write-protected memory is intended for storing

program code including UDVM instructions and their associated parameters. Note that if an instruction references a variable that has been write-protected, the compiled version of the instruction

will typically run faster than if the referenced variable lies in the working memory area.

4.2. Well-known variables

The first few variables in the UDVM memory have special tasks, for example specifying the location of the stack used by the CALL and RETURN instructions. Each of these well-known variables is a 2-byte integer.

The following list gives the name of each well-known variable and the memory address at which the variable can be found:

Name:	Starting memory address:
byte_copy_left	0
byte_copy_right	2
stack_location	4

The MSBs of each variable are always stored before the LSBs. So, for example, the MSBs of stack_location are stored at Address 4 whilst the LSBs are stored at Address 5.

The use of each well-known variable is described in the following sections of the draft.

4.3. Instruction parameters

Each of the UDVM instructions is followed by 0 or more bytes containing the parameters required by the instruction.

To reduce the code size of a typical UDVM program, each parameter for a UDVM instruction is compressed using variable-length encoding. The aim is to store more common parameter values using fewer bits than rarely occurring values.

Three different types of parameter are available: the literal, the reference and the multitype. The parameter types that follow each UDVM instruction are specified in Chapter 6.

The UDVM bytecode for each parameter type is illustrated in Figure 4 to Figure 6, together with the integer values represented by the bytecode.

Note that the MSBs in the bytecode are illustrated as preceding the LSBs. Also, any string of bits marked with k consecutive "n"s is to be interpreted as an integer N from 0 to 2^k - 1 inclusive (with the MSBs of n illustrated as preceding the LSBs).

The decoded integer value of the bytecode can be interpreted in two

ways. In some cases it is taken to be the actual value of the parameter. In other cases it is taken to be a memory address at which the 2-byte parameter value can be found (MSBs found at the specified address, LSBs found at the following address). The latter case is

denoted by memory[X] where X is the address and memory[X] is the 2-byte value starting at Address X.

The simplest parameter type is the literal (#), which encodes a constant integer from 0 to 65535 inclusive. A literal parameter may require between 1 and 3 bytes depending on its value.

Bytecode:	Parameter value:	Range:
0nnnnnnn	N	0 - 127
10nnnnnn nnnnnnnn	N	0 - 16383
11000000 nnnnnnnn nnnnnnnn	N	0 - 65535

Figure 4: Bytecode for a literal (#) parameter

The second parameter type is the reference ($), which is always used to access a 2-byte value located elsewhere in the UDVM memory. The bytecode for a reference parameter is decoded to be a constant integer from 0 to 65535 inclusive, which is interpreted as the memory address containing the actual value of the parameter.

Bytecode:	Parameter value:	Range:
0nnnnnnn	memory[2 * N]	0 - 254
10nnnnnn nnnnnnnn	memory[2 * N]	0 - 32766
11000000 nnnnnnnn nnnnnnnn	memory[N]	0 - 65535

Figure 5: Bytecode for a reference ($) parameter

The third kind of parameter is the multitype (%), which can be used to encode both actual values and memory addresses. The multitype parameter also offers efficient encoding for small integer values (both positive and negative) and for powers of 2.

Bytecode:	Parameter value:	Range:
00nnnnnn	N	0 - 63
01nnnnnn	memory[2 * N]	0 - 126
1000011n	2 ^ (N + 6)	64 - 128
10001nnn	2 ^ (N + 8)	256 - 32768
111nnnnn	N + 65504	65504 - 65535
1001nnnn nnnnnnnn	N + 61440	61440 - 65535
101nnnnn nnnnnnnn	N	0 - 8191
110nnnnn nnnnnnnn	memory[N]	0 - 8191
10000000 nnnnnnnn nnnnnnnn	N	0 - 65535
10000001 nnnnnnnn nnnnnnnn	memory[N]	0 - 65535

Figure 6: Bytecode for a multitype (%) parameter

4.4. Byte copying

A number of UDVM instructions require a string of bytes to be copied to and from areas of the UDVM memory. This section defines how the byte copying operation should be performed.

In general, the string of bytes is copied in ascending order of memory address. So if a byte is copied from/to Address n then the next byte is copied from/to Address n + 1. As usual, if a byte is read from an address beyond the overall memory size or is written to an address outside the working memory area then decompression failure occurs.

Note however that if a byte is copied from/to the memory address specified in byte_copy_right, the byte copy operation continues by copying the next byte from/to the memory address specified in byte_copy_left. This is useful for setting up a "circular buffer" within the UDVM memory.

Note that the string of bytes is copied on a purely byte-by-byte basis. In particular, some of the later bytes to be copied may themselves have been written into the UDVM memory by the byte copying operation currently being performed.

Equally, it is possible for a byte copying operation to overwrite the instruction that called the byte copy. If this occurs then the byte copying operation MUST be completed as if the original instruction were still in place in the UDVM memory (this also applies if byte_copy_left or byte_copy_right are overwritten).

5. Decompressing a compressed message

This chapter lists the steps involved in the decompression of a single compressed message.

5.1. Invoking the UDVM

Whenever the application receives a message to be decompressed, it invokes a new instance of the UDVM. The overall_memory_size and initial contents of the UDVM memory are initialized using the corresponding application-defined parameters. The following steps are then taken:

1.) The number of remaining CPU cycles is set equal to the application-defined parameter cycles_per_message.

Notes:

The amount of compressed data available to the UDVM is exactly one compressed message. If the transport mechanism is stream-oriented then the UDVM uses the reserved byte string 0xFFFF to delimit the compressed messages: the UDVM takes the data between a pair of neighboring reserved byte strings to be a single compressed message. The reserved byte string itself is not considered to be part of the compressed message.

For a stream-oriented transport, the UDVM parses the compressed data stream for instances of 0xFF and takes the following actions:

Occurs in data stream:	Action:
0xFFFF	Delimit compressed message
0xFF00	Replace with 0xFF
0xFF01 - 0xFFFE	Decompression failure

The reserved character 0xFF00 is useful for byte stuffing (if a compression algorithm generates compressed data containing the character 0xFF then it should be replaced by the character 0xFF00 to avoid accidentally inserting a message delimiter into the compressed data stream).

The compressed data is not provided to the UDVM by default. Instead, the UDVM requests compressed data using the INPUT instructions (useful when running over a stream-oriented transport since there is no need to wait for the entire compressed message before decompression can begin). Note that in particular, this means that the application MUST define the initial contents of the UDVM memory to contain at least one INPUT instruction. See Appendix B for an example of how the application might initialize the UDVM memory.

The application MUST NOT make more than one compressed message available to a given instance of the UDVM. In particular, the application MUST NOT concatenate two messages to form a single compressed message. This is because compressed messages are typically padded with trailing zero bits so that they are a whole number of bytes long. Concatenating two messages would cause these padding bits to be incorrectly interpreted as compressed data.

2.) Next, the instructions contained within the UDVM memory are executed beginning at the address specified in first_instruction.

Notes:

The instructions are executed consecutively unless otherwise indicated (for example when the UDVM encounters a JUMP instruction).

If the next instruction to be executed lies outside the available memory then decompression failure occurs (see Section 5.3).

3.) Each time an instruction is executed the number of available CPU cycles is decreased by the amount specified in Chapter 6. Additionally, if the UDVM requests n bits of compressed data (using one of the INPUT instructions) then the number of available CPU cycles is increased by n * cycles_per_bit.

Notes:

This means that the total number of CPU cycles available for processing a compressed message is given by the formula:

```
total_cycles = cycles_per_message + message_size * cycles_per_bit
```

The reason that this total is not allocated to the UDVM when it is invoked is that the UDVM can begin to decompress a message that has only been partially received. So the total message size may not be known when the UDVM is initialized.

4.) The UDVM stops executing instructions when it encounters an END-MESSAGE instruction or if decompression failure occurs.

Notes:

The UDVM passes uncompressed data to the application using the OUTPUT instruction. The OUTPUT instruction can be used to output a partially decompressed message; it is an application decision whether to use the data immediately or whether to buffer and wait until the entire message has been decompressed.

The UDVM passes state creation and feedback requests to the application using the END-MESSAGE instruction. This means that it is only possible to make a state creation and a feedback request once the message has been decompressed, which is necessary since the application typically checks the validity of these requests based on the contents of the decompressed message.

5.2. Successful decompression

The END-MESSAGE instruction indicates that the compressed message has been successfully decompressed and passed to the application. Note that the actual uncompressed message is outputted beforehand using the OUTPUT instruction; this allows the UDVM to output each part of the message to the application as soon as it has been decompressed.

The END-MESSAGE instruction provides two additional pieces of information to the application: the state creation request and the feedback data. The state creation request mechanism is discussed below; feedback information is discussed separately in Chapter 7.

The UDVM may optionally save part of its memory for retrieval by later messages. However to prevent malicious storage of a large amount of unnecessary state information, the application MUST give permission before any state can be created. The application typically makes a decision on whether state can be created based on the contents of the decompressed message, particularly if the message contains authentication data that can verify whether or not the sender is legitimate.

The END-MESSAGE instruction requests the creation of state using the parameters state start and state length, which together denote a byte string state_value. Provided that the application gives permission, state_value is byte copied from the UDVM memory (obeying the rules of Section 4.4) and stored together with a 16-byte state identifier that

can be used to access the state by a later compressed message.

To provide security against malicious access, the identifier for any item of state created by the UDVM is derived from the [MD5] hash of

the state_value to be stored. The state identifier is constructed by taking the 16-byte [MD5] hash and replacing all but the first hash_length most significant bytes with zeroes. Note that if hash_length is 16 then the unmodified [MD5] hash is the state identifier. Decompression failure occurs if hash_length is less than the application-defined parameter minimum_hash_size or greater than 16.

Each item of state stores the following information (accessed by the state_identifier):

state_identifier
state start
state length
state_value
state_instruction

Note that state_start, state_length and state_instruction are all parameters from the END-MESSAGE instruction, whereas state_identifier and state_value are created as specified above.

If a state creation request is made with a state identifier that has been used by existing state, then the request fails automatically.

This state can subsequently be accessed by using the STATE-REFERENCE and STATE-EXECUTE instructions (by providing the correct state identifier).

5.3. Decompression failure

If a compressed message given to the UDVM is corrupted (either accidentally or maliciously) then the UDVM may terminate with a decompression failure.

Reasons for decompression failure include the following:

* A compressed or uncompressed message exceeds the maximum size defined by the application.

* The UDVM exceeds the available CPU cycles for decompressing a message.

* The UDVM attempts to read a memory address beyond the overall memory size, or to write into a memory address outside the working memory area.

* An unknown instruction type is encountered.

* An unknown parameter type is encountered.

* An instruction is encountered that cannot be processed successfully by the UDVM (for example a RETURN instruction when no CALL instruction has previously been encountered).

* The UDVM attempts to access non-existent state.

* A manual decompression failure is triggered using the DECOMPRESSION-FAILURE instruction.

If a decompression failure occurs when decompressing a message then the UDVM informs the application and takes no further action. It is the responsibility of the application to decide how to cope with the decompression failure. In general an application SHOULD discard the compressed message and any decompressed data that has been outputted.

6. UDVM instruction set

The UDVM currently understands 28 instructions, chosen to support the widest possible range of compression algorithms with the minimum possible overhead.

Figure 7 lists the different instructions and the bytecode values used to store the instructions at the UDVM. The cost of each instruction in CPU cycles is also given:

Instruction:	Bytecode value:	Cost in CPU cycles:
DECOMPRESSION-FAILURE	0	1
AND	1	1
OR	2	1
NOT	3	1
ADD	4	1
SUBTRACT	5	1
MULTIPLY	6	1
DIVIDE	7	1
LOAD	8	1
MULTILOAD	9	1 + n
WORKING-MEMORY	10	1
COPY	11	1 + length
COPY-LITERAL	12	1 + length
COPY-OFFSET	13	1 + length + offset
JUMP	14	1
COMPARE	15	1
CALL	16	1
RETURN	17	1
SWITCH	18	1 + n
CRC	19	1 + length
END-MESSAGE	20	1 + state length
OUTPUT	21	1 + output_length
NBO	22	1
INPUT-BYTECODE	23	1 + length
INPUT-FIXED	24	1
INPUT-HUFFMAN	25	1 + n

STATE-REFERENCE	26	1 + state_length
STATE-EXECUTE	27	1 + state length

Figure 7: UDVM instructions and corresponding bytecode values

Each UDVM instruction costs a minimum of 1 CPU cycle. Certain high-level instructions may cost additional cycles depending on the value of one of the instruction parameters.

The only exception when calculating the number of CPU cycles is that the STATE-EXECUTE instruction takes (1 + state_length) cycles even though it does not have a state_length parameter; instead the value of state length is provided by the application as part of the state being accessed.

All instructions are stored as a single byte to indicate the instruction type, followed by 0 or more bytes containing the parameters required by the instruction. The instruction specifies which of the three parameter types of Section 4.3 is used in each case. For example, the ADD instruction is followed by two parameters as shown below:

ADD ($parameter_1, %parameter_2)

When converted into bytecode the number of bytes required by the ADD instruction depends on the size of each parameter value, and whether the second (multitype) parameter contains the parameter value itself or a memory address where the actual value of the parameter can be found.

The instruction set available for the UDVM offers a mix of low-level and high-level instructions. The high-level instructions can all be emulated using the low-level instructions provided, but given a choice it is generally preferable to use a single instruction rather than a large number of general-purpose instructions. The resulting bytecode will be more compact (leading to a higher overall compression ratio) and decompression will typically be faster because the implementation of the compression-specific instructions can be optimized for the UDVM.

Each instruction is explained in more detail below:

6.1. Bit manipulation instructions

The AND, OR and NOT instructions provide simple bit manipulation on 2-byte words.

AND ($parameter_1, %parameter_2)
OR ($parameter_1, %parameter_2)
NOT ($parameter_1)

After the operation is complete, the value of the first parameter is overwritten with the result. Note that since this parameter is a reference, the memory address specified by the parameter is always overwritten and not the parameter itself.

6.2. Arithmetic instructions

The ADD, SUBTRACT, MULTIPLY and DIVIDE instructions perform arithmetic on 2-byte words.

ADD ($parameter_1, %parameter_2)
SUBTRACT ($parameter_1, %parameter_2)
MULTIPLY ($parameter_1, %parameter_2)
DIVIDE ($parameter_1, %parameter_2)

After the operation is complete, the first parameter is overwritten with the result.

Note that in all cases the arithmetic operation is performed modulo 2^16. So for example, subtracting 1 from 0 gives the result 65535.

For the SUBTRACT instruction the second parameter is subtracted from the first. Similarly, for the DIVIDE instruction the first parameter is divided by the second parameter. Note that if the second parameter does not divide exactly into the first parameter then the remainder is ignored.

6.3. Memory management instructions

The following instructions are used to manipulate the UDVM memory. Bytes can be copied from one area of memory to another, and areas of memory can be write-protected to make it easier for UDVM code to be compiled.

6.3.1. LOAD

The LOAD instruction sets a 2-byte variable to a certain specified value. The format of a LOAD instruction is as follows:

LOAD (%address, %value)

The first parameter specifies the starting address of the 2-byte variable, whilst the second parameter specifies the value to be loaded into this variable. As usual, MSBs are stored before LSBs in the UDVM memory.

6.3.2. MULTILOAD

The MULTILOAD instruction sets a contiguous block of 2-byte variables to specified values.

MULTILOAD (%address, #n, %value_0, ..., %value_n-1)

The first parameter specifies the starting address of the contiguous variables, whilst the parameters value_0 through to value_n-1 specify

the values to load into these variables (in the same order as they appear in the instruction).

6.3.3. WORKING-MEMORY

The WORKING-MEMORY instruction is used to prevent part of the UDVM memory from being modified. This can be very useful when offering UDVM code for compilation.
WORKING-MEMORY (%memory_start, %memory_end)

The parameters memory_start and memory_end specify the new working memory area for the UDVM. These parameters replace the application-defined parameters working_memory_start and working_memory_end, but only while the current message is being decompressed. When a new instance of the UDVM is invoked the working memory area is set by the original application-defined parameters.

If memory_end < memory_start, or if the parameters reference a memory address beyond the overall UDVM memory size, then decompression failure occurs.

After the WORKING-MEMORY instruction has been encountered, the only way to write into UDVM memory within the protected region is to cancel the protection using another WORKING-MEMORY instruction (or to invoke a new instance of the UDVM).

6.3.4. COPY

The COPY instruction is used to copy a string of bytes from one part of the UDVM memory to another.

COPY (%position, %length, %destination)

The position parameter specifies the memory address of the first byte in the string to be copied, and the length parameter specifies the number of bytes to be copied.

The destination parameter gives the address to which the first byte in the string will be copied.

Note that byte copying is performed as per the rules of Section 4.4.

6.3.5. COPY-LITERAL

A modified version of the COPY instruction is given below:

COPY-LITERAL (%position, %length, $destination)

The COPY-LITERAL instruction behaves as a COPY instruction except that after copying, the destination parameter is replaced with the memory address immediately following the address to which the final byte was copied. If the final byte was copied to the memory address specified in byte_copy_right, the destination parameter is set to the

memory address specified in byte_copy_left.

6.3.6. COPY-OFFSET

A further version of the COPY-LITERAL instruction is given below:

COPY-OFFSET (%offset, %length, $destination)

The COPY-OFFSET instruction behaves as a COPY-LITERAL instruction except that an offset parameter is given instead of a position parameter.

To derive a suitable position parameter, starting at the memory address specified by destination, the UDVM counts backwards a total of offset memory addresses. If the memory address specified in byte_copy_left is reached, the next memory address is taken to be byte_copy_right.

The COPY-OFFSET instruction then behaves as a COPY-LITERAL instruction, taking the position parameter to be the last memory address reached in the above step.

6.4. Program flow instructions

The following instructions alter the flow of UDVM code. Each instruction jumps to one of a number of memory addresses based on a certain specified criterion. Note that all of the instructions give the memory addresses in the form of deltas relative to the memory address of the instruction. The actual memory address is calculated as follows:

memory_address = (memory_address_of_instruction + delta) modulo 2^16

Note that certain I/O instructions (see Section 6.5) can also alter program flow.

6.4.1. JUMP

The JUMP instruction moves program execution to the specified memory address.

JUMP (%delta)

Note that if the address (specified as a delta from the address of the JUMP instruction) lies beyond the overall UDVM memory size then decompression failure occurs.

6.4.2. COMPARE

The COMPARE instruction compares two parameters and then jumps to one of three specified memory addresses depending on the result.

COMPARE (%parameter_1, %parameter_2, %delta_1, %delta_2, %delta_3)

If parameter_1 < parameter_2 then the UDVM continues instruction execution at the (relative) memory address specified by delta 1. If

parameter_1 = parameter_2 then it jumps to the address specified by delta_2. If parameter_1 > parameter_2 then it jumps to the address specified by delta_3.

6.4.3. CALL and RETURN

The CALL and RETURN instructions provide support for compression algorithms with a nested structure.

CALL (%delta)

RETURN

The CALL and RETURN instructions make use of a stack of 2-byte variables stored at the memory address specified by the well-known variable stack_location. The stack contains the following variables:

Name:	Starting memory address:
stack_free	stack_location
stack[0]	stack_location + 2
stack[1]	stack_location + 4
stack[2]	stack_location + 6
:	:

The MSBs of these variables are stored before the LSBs in the UDVM memory.

When the UDVM reaches a CALL instruction, it finds the memory address of the instruction immediately following the CALL instruction and copies this 2-byte value into stack[stack_free] ready for later retrieval. It then increases stack_free by 1 and continues instruction execution at the (relative) memory address specified by the parameter.

When the UDVM reaches a RETURN instruction it decreases stack_free by 1, and then continues instruction execution at the byte position stored in stack[stack_free].

If the variable stack_free is ever increased beyond 65535 or decreased below 0 then a bad compressed message has been received and decompression failure occurs (see Section 5.3).

Decompression failure also occurs if one of the above instructions is encountered and the value of stack_location is smaller than 6 (this prevents the stack from overwriting the well-known variables).

6.4.4. SWITCH

The SWITCH instruction performs a conditional jump based on the value

of one of its parameters.

SWITCH (#n, %j, %delta_0, %delta_1, ... , %delta_n-1)

When a SWITCH instruction is encountered the UDVM reads the value of j. It then continues instruction execution at the (relative) address specified by delta j.

If j specifies a value of n or more, a bad compressed message has been received and decompression failure occurs.

6.4.5. CRC

The CRC instruction verifies a string of bytes using a 2-byte CRC.

CRC (%value, %position, %length, %delta)

The actual CRC calculation is performed using the generator polynomial x^16 + x^12 + x^5 + 1, which coincides with the 2-byte Frame Check Sequence (FCS) of [RFC-1662].

The position and length parameters define the string of bytes over which the CRC is evaluated. Byte copying rules are enforced as per Section 4.4.

Important note: Since a CRC calculation is always performed over a bitstream, for interoperability it is necessary to define the order in which bits are supplied within each individual byte. In this case the MSBs of the byte MUST be supplied to the CRC calculation before the LSBs.

The value parameter contains the expected integer value of the 2-byte CRC. If the calculated CRC matches the expected value then the UDVM continues at the following instruction. Otherwise the UDVM jumps to the (relative) memory address specified by delta.

6.5. I/O instructions

The following instructions allow the UDVM to interface with its environment. Note that in the current UDVM architecture all of the interfaces pass through the application (which has a veto over any information supplied to or from the UDVM).

6.5.1. END-MESSAGE

The END-MESSAGE instruction successfully terminates the UDVM and passes feedback and state information to the application.

END-MESSAGE (%hash_length, %state_start, %state_length, %state_instruction, %feedback_location)

The actions taken by the UDVM upon encountering the END-MESSAGE instruction are described in Section 5.2.

6.5.2. DECOMPRESSION-FAILURE

The DECOMPRESSION-FAILURE instruction triggers a manual decompression failure. This is useful if the UDVM program discovers that it cannot

successfully decompress the message (e.g. by using the CRC instruction).

This instruction has no parameters.

6.5.3. OUTPUT

The OUTPUT instruction provides successfully decompressed data to the application.

OUTPUT (%output_start, %output_length)

The parameters define the starting memory address and length of the byte string to be provided to the application. Note that the OUTPUT instruction can be used to output a partially decompressed message; each time the instruction is encountered it appends a byte string to the end of the data previously passed to the application via the OUTPUT instruction.

The string of data is byte copied from the UDVM memory obeying the rules of Section 4.4.

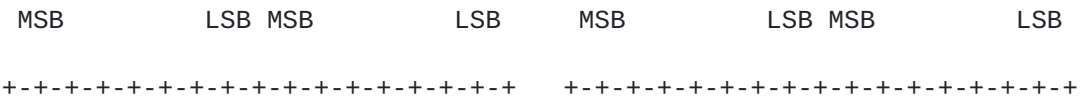
Decompression failure occurs if the cumulative number of bytes provided to the application exceeds the application-defined parameter maximum_uncompressed_size.

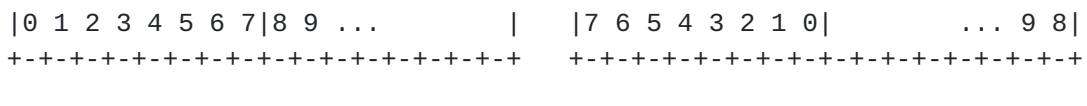
Since there is technically a difference between outputting a 0-byte decompressed message, and not outputting a decompressed message at all, the OUTPUT instruction needs to distinguish between the two cases. Thus, if the UDVM terminates before encountering an OUTPUT instruction it is considered not to have outputted a decompressed message. If it encounters one or more OUTPUT instructions, each of which provides 0 bytes of data to the application, then it is considered to have outputted a 0-byte decompressed message.

6.5.4. NBO

The NBO instruction modifies the order in which compressed bits are passed to the UDVM.

As the INPUT-FIXED and INPUT-HUFFMAN instructions read individual bits from within a byte, to avoid ambiguity it is necessary to define the order in which these bits are read. The default operation is to read the MSBs before the LSBs, but if the NBO instruction is encountered then the LSBs are read before the MSBs. Both cases are illustrated below:




|0 1 2 3 4 5 6 7|8 9 ... |
|7 6 5 4 3 2 1 0| ... 9 8|
Byte 0 Byte 1
Byte 0 Byte 1

Default operation After NBO instruction

The NBO instruction can only be used before bitwise compressed data is passed to the UDVM. Therefore, a decompression failure occurs if it is encountered after an INPUT-FIXED or an INPUT-HUFFMAN instruction has been used.

6.5.5. INPUT-BYTECODE

The INPUT-BYTECODE instruction requests a certain number of bytes of compressed data from the application.

INPUT-BYTECODE (%length, %destination, %delta)

The length parameter indicates the requested number of bytes of compressed data, and the destination parameter specifies the starting memory address to which they should be copied. Byte copying is performed as per the rules of Section 4.4.

If the instruction requests data that lies beyond the end of the compressed message, no data is returned. Instead the UDVM moves program execution to the memory address specified by the formula (memory_address_of_INPUT-BYTECODE_instruction + delta) modulo 2^16.

The INPUT-BYTECODE instruction can only be used before bitwise compressed data is passed to the UDVM. Therefore, a decompression failure occurs if it is encountered after an INPUT-FIXED or an INPUT-HUFFMAN instruction has been used.

6.5.6. INPUT-FIXED

The INPUT-FIXED instruction requests a certain number of bits of compressed data from the application.

INPUT-FIXED (%length, %destination, %delta)

The length parameter indicates the requested number of bits. If this parameter does not lie between 1 and 16 inclusive then a decompression failure occurs.

The destination parameter specifies the memory address to which the compressed data should be copied. Note that the requested bits are interpreted as a 2-byte integer ranging from 0 to 2^length - 1. Under default operation the MSBs of this integer are provided first, but if an NBO instruction has been executed then the LSBs are provided first.

If the instruction requests data that lies beyond the end of the compressed message, no data is returned. Instead the UDVM moves

program execution to the memory address specified by the formula (memory_address_of_INPUT-FIXED_instruction + delta) modulo 2^16.

6.5.7. INPUT-HUFFMAN

The INPUT-HUFFMAN instruction requests a variable number of bits of compressed data from the application. The instruction initially requests a small number of bits and compares the result against a certain criterion; if the criterion is not met then additional bits are requested until the criterion is achieved.

The INPUT-HUFFMAN instruction is followed by three mandatory parameters plus n additional sets of parameters. Every additional set contains four parameters as shown below:

INPUT-HUFFMAN (%destination, %delta, #n, %bits_1, %lower_bound_1, %upper_bound_1, %uncompressed_1, ... , %bits_n, %lower_bound_n, %upper_bound_n, %uncompressed_n)

Note that if n = 0 then the INPUT-HUFFMAN instruction is ignored by the UDVM. If bits_1 = 0 or (bits_1 + ... + bits_n) > 16 then decompression failure occurs.

In all other cases, the behavior of the INPUT-HUFFMAN instruction is defined below:

1.) Set j = 1.

2.) Request an additional bits_j compressed bits. Interpret the total (bits_1 + ... + bits_j) bits of compressed data requested so far as an integer H, with the first bit to be supplied as the MSB and the last bit to be supplied as the LSB (note that this is always the case, independently of whether the NBO instruction has been used).

3.) If data is requested that lies beyond the end of the compressed message, terminate the INPUT-HUFFMAN instruction and move program execution to the memory address specified by the formula (memory_address_of_INPUT-HUFFMAN_instruction + delta) modulo 2^16.

4.) If (H < lower_bound_j) or (H > upper_bound_j) then set j = j + 1. Then go back to Step 2, unless j > n in which case decompression failure occurs.

5.) Copy (H + uncompressed_j - lower_bound_j) modulo 2^16 to the memory address specified by the destination parameter.

6.5.8. STATE-REFERENCE

The STATE-REFERENCE instruction retrieves some previously stored state information.

STATE-REFERENCE (%id_start, %id_length, %state_start, %state_length, %state_destination)

The id_start and id_length parameters specify the location of the state identifier used to retrieve the state information. The state identifier is always 16 bytes long; if id_length is less than 16 then

the remaining least significant bytes of the identifier are padded with zeroes.

Decompression failure occurs if id_length is greater than 16. Decompression failure also occurs if no state information matching the state identifier can be found.

Note that when accessing state information that has been previously created by the UDVM, the state identifier is always taken from an [MD5] hash of the state to be retrieved. However this is not necessarily the case for application-defined state as per Section 3.5.

The state_start and state_length parameters define the starting byte and number of bytes to copy from the state_value contained in the identified item of state. If more state is requested than is actually available then decompression failure occurs.

The state_destination parameter contains a UDVM memory address. The requested state is byte copied to this memory address using the rules of Section 4.4.

6.5.9. STATE-EXECUTE

The STATE-EXECUTE instruction retrieves and runs some previously stored state information.

STATE-EXECUTE (%id_start, %id_length)

The id_start and id_length parameters function as per the STATE-REFERENCE instruction.

STATE-EXECUTE is similar to STATE-REQUEST except that it does not require the amount of state being requested or the proposed destination for the state to be specified explicitly. Instead, it simply puts the state back into the UDVM memory using the original parameters from the END-MESSAGE instruction that created the state.

The entire state_value (all state length bytes of it) is byte copied into the memory address specified by state start The UDVM then jumps to the (absolute) memory address specified by state_instruction.

Note that state start, state length and state_instruction are all stored together with state_value as part of an item of state information.

7. Feedback information

If the transport mechanism offers bidirectional data transport then the compression ratio can be improved by sending feedback

information. Since feedback data is optional, compressors must be able to function correctly even if no feedback information is provided.

In the bidirectional UDVM architecture, suppose that Application 2 wishes to send feedback information to Compressor 1. The path taken by the feedback information is as follows:

Appl. 2 --> Compressor 2 --> UDVM 1 --> Appl. 1 --> Compressor 1

The first hop along the path is between Application 2 and Compressor 2. If permitted by the application, Compressor 2 MAY be supplied with some or all of the following items of data:

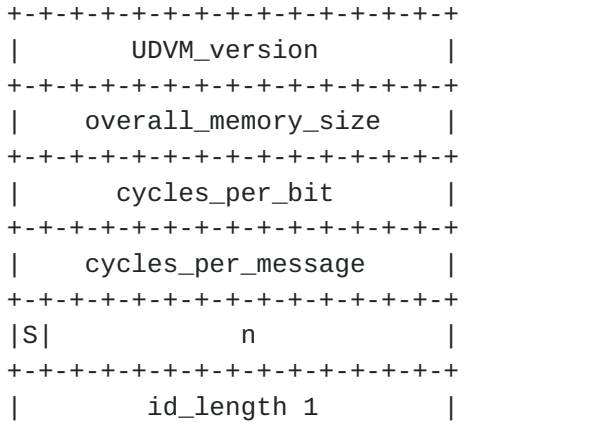
overall_memory_size
cycles_per_bit
cycles_per_message
id lengths and id values of successfully established state

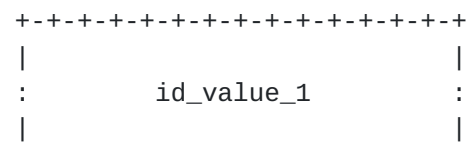
Since the design of each compressor is left as an implementation decision, there is no need to standardize the format in which this data is provided to Compressor 2.

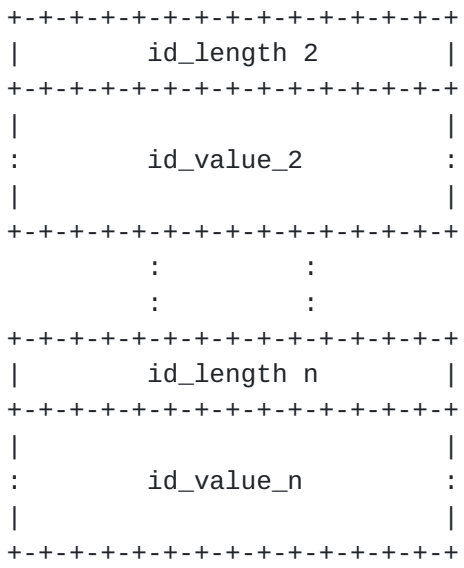
The second hop along the path is between Compressor 2 and UDVM 1. For this step Compressor 2 transmits the feedback information to UDVM 1 across the same transport mechanism used to carry compressed data. Typically this feedback information is piggybacked onto existing compressed messages (standalone feedback messages are generally vetoed by the application due to the lack of a corresponding decompressed message).

Note that Compressor 2 can send the feedback information compressed in order to reduce the total number of bits transmitted. Equally, Compressor 2 may opt not to send feedback information at all.

If Compressor 2 chooses not to send feedback information then it sets the feedback_location parameter in the END-MESSAGE instruction to 0. Otherwise, it copies the following block of data to the memory of UDVM 1 and places the starting memory address of this block in the feedback_location parameter:




id_value_1



Each of the items of data is explained in greater detail below:

7.1. UDVM version

The first 2 bytes of feedback data specify whether only the basic version of the UDVM is available, or whether an upgraded version of the UDVM is available offering additional instructions, feedback data etc.

The basic version of the UDVM is Version 0, which is the version described in this document. Upgraded versions MUST be backwards-compatible with the basic version in the following sense:

* If some UDVM bytecode reaches the END-MESSAGE or DECOMPRESSION-FAILURE instructions when running on Version 0 of the UDVM, then the upgraded version MUST run the bytecode in an identical manner.

This condition ensures that all bytecode that is valid for Version 0 of the UDVM will continue to be valid for upgraded versions of the UDVM. However, bytecode that is invalid on Version 0 of the UDVM (i.e. bytecode that produces a decompression failure that is not manually triggered) may become valid on upgraded versions.

Examples of how to upgrade the UDVM in a backwards-compatible manner include: adding new UDVM instructions, adding more items of feedback data etc.

7.2. Memory size and CPU cycles

The next 6 bytes of feedback data specify new values for the application-defined parameters overall_memory_size, cycles_per_bit and cycles_per_message. This allows Application 2 to inform

Compressor 1 that it has additional memory or processing power available that could be used to improve the overall compression ratio.

Note that the feedback data can only be used to increase the amount of resources available for Compressor 1 to use. If the feedback data specifies a parameter value that is smaller than the value already possessed by Compressor 1, the parameter keeps its original value (i.e. the feedback data for this parameter is simply ignored).

The reason for this behavior is that if UDVM 2 is initialized with more memory than expected by Compressor 1 then no problem arises, but if UDVM 2 is initialized with less memory that expected by Compressor 1 then decompression failure may occur. Therefore, only allowing the parameter values to increase means that the feedback mechanism is robust against message loss or reordering on the feedback channel.

The parameters can only be restored to their original values if reset or renegotiated by the application.

7.3. State identifiers

The variable n specifies the number of state identifiers to be acknowledged.

Each state identifier is usually the first few bytes from an [MD5] hash of the state being acknowledged. When a state identifier is placed in the feedback information of UDVM 1, it is known by Compressor 1 that the corresponding state has been successfully established and can be referenced in future by using a STATE-REFERENCE or a STATE-EXECUTE instruction. The feedback information includes the length and value of each hash to be acknowledged.

Note that the MSB of n has a special meaning; if set to 1 then it acknowledges the state that is currently being created by UDVM_1 via the END-MESSAGE instruction. This saves having to transmit the id_length and id_value explicitly on the feedback channel.

8. Security considerations

The following chapter identifies the potential security risks associated with the overall UDVM architecture, and details the proposed solution for each risk.

** Avoid snooping into state of other users

State can only be accessed using a state identifier, which is a (prefix of a) cryptographic hash of the state being referenced. This implies that the referencing packet already needs knowledge about the state. To enforce this, a minimum reference length of 48 bits is RECOMMENDED for applications running over an unsecure transport mechanism. This also minimises the probability of an accidental state collision.

Generally, ways to obtain knowledge about the state identifier (e.g., passive attacks) will also easily provide knowledge about the state referenced, so no new vulnerability results.

The application needs to handle state identifiers with the same care it would handle the state itself.

** Avoid DoS vulnerabilities

*** Use of the UDVM as a tool in a DoS attack to another target

The UDVM cannot easily be used as an amplifier in a reflection attack, as it only generates one decompressed message per incoming compressed message. This packet is then handed to the application; the utility as a reflection amplifier is therefore limited by the utility of the application.

However, it must be noted that the UDVM can be used to generate larger packets as input to the application than have to be sent from the malicious sender; this therefore can send smaller packets (at a lower bandwidth) than are delivered to the application. Depending on the reflection characteristics of the application, this can be considered a mild form of amplification. The application MUST limit the number of packets reflected to a potential target - even if the UDVM is used to generate a large amount of information from a small incoming attack packet.

*** Attacking the UDVM as the DoS target by filling it with state

Excessive state can only be installed by a malicious sender (or a set of malicious senders) with the consent of the application. The system consisting of UDVM and application is thus approximately as vulnerable as the application itself, unless it allows the installation of state from a message where it would not have installed state itself.

If this is desirable to increase the compression ratio, the effect can be mitigated by adding feedback at the application level that indicates whether the state was actually installed - this allows a system under attack to gracefully degrade by no longer installing compressor state that is not matched by application state.

*** Attacking the UDVM by faking state or making unauthorized changes to state

State cannot be destroyed or changed by a malicious sender - it can only add new state.

*** Attacking the UDVM by sending it looping code

The application sets an upper limit to the number of "CPU cycles" that can be used per compressed message and per input bit in the compressed message. The damage inflicted by sending packets with looping code is therefore limited, although this may still be

substantial if a large number of CPU cycles are offered by the UDVM. However, this would be true for any decompressor that can receive packets from anywhere.

9. Acknowledgements

Individual compression algorithms such as [DEFLATE] have been important sources of ideas and knowledge.

Thanks to

Abigail Surtees (abigail.surtees@roke.co.uk)
Mark A West (mark.a.west@roke.co.uk)
Lawrence Conroy (lwc@roke.co.uk)
Christian Schmidt (christian.schmidt@icn.siemens.de)
Max Riegel (maximilian.riegel@icn.siemens.de)
Jan Christoffersson (jan.christoffersson@epl.ericsson.se)
Stefan Forsgren (stefan.forsgren@epl.ericsson.se)
Krister Svanbro (krister.svanbro@epl.ericsson.se)
Christopher Clanton (christopher.clanton@nokia.com)
Khiem Le (khiem.le@nokia.com)
Ka Cheong Leung (kacheong.leung@nokia.com)

for valuable input and review.

10. References

[DEFLATE] "DEFLATE Compressed Data Format Specification version 1.3", P. Deutsch, RFC 1951, Internet Engineering Task Force, May 1996

[SCTP] "Stream Control Transmission Protocol", Stewart et al, RFC 2960, Internet Engineering Task Force, October 2000

[SIP] "SIP: Session Initiation Protocol", Handley et al, RFC 2543, Internet Engineering Task Force, March 1999

[MD5] "The MD5 Message-Digest Algorithm", R. Rivest, RFC 1321, Internet Engineering Task Force, April 1992

[RFC-1662] "PPP in HDLC-like Framing", Simpson et al, Internet Engineering Task Force, July 1994

[RFC-2026] "The Internet Standards Process - Revision 3", Scott Bradner, Internet Engineering Task Force, October 1996

[RFC-2119] "Key words for use in RFCs to Indicate Requirement Levels", Scott Bradner, Internet Engineering Task Force, March 1997

11. Authors' addresses

Richard Price Tel: +44 1794 833681

Email: richard.price@roke.co.uk

Roke Manor Research Ltd
Romsey, Hants, SO51 0ZN

United Kingdom

Jonathan Rosenberg
Email: jdrosen@dynamicsoft.com

dynamicsoft
72 Eagle Rock Avenue
First Floor
East Hanover, NJ 07936

Carsten Bormann Tel: +49 421 218 7024
Email: cabo@tzi.org

Universitaet Bremen TZI
Postfach 330440
D-28334 Bremen, Germany

Hans Hannu Tel: +46 920 20 21 84
Email: hans.hannu@epl.ericsson.se

Box 920
Ericsson Erisoft AB
SE-971 28 Lulea, Sweden

Zhigang Liu Tel: +1 972 894-5935
Email: zhigang.liu@nokia.com

Nokia Research Center
6000 Connection Drive
Irving, TX 75039
USA

Appendix A. Mnemonic language

Writing UDVM programs directly in bytecode would be a daunting task, so a simple mnemonic language is provided to facilitate the creation of new decompression algorithms. Most importantly, the language allows the parameters of an instruction to be specified as text names rather than as integer values.

If an instruction parameter is given as a text name, it should correspond to exactly one instance of a label, a reserved memory address or an externally defined keyword. A label is simply a text name preceded by a colon, for example:

```
:loop
JUMP (loop)
```

For any parameters corresponding to a label, the integer value of the parameter is calculated by the following formula:

```
 parameter_value = (instruction_address - label_address) modulo 2^16
```

Note that the "label address" is simply the memory address of the instruction immediately following the label. In particular, the above example can be rewritten as JUMP (0).

A reserved memory address is specified using the "reserve" keyword followed by a text_name and (optionally) an integer value. For example:

```
reserve apples
reserve pears (8)
reserve bananas
LOAD (bananas, 5)
```

For any parameters corresponding to a reserved memory address, the integer value of the parameter is the next free memory address that has not yet been reserved. Starting at this address, the specified number of bytes of memory are then reserved (if no value is given then a total of 2 bytes is reserved).

The first instance of a "reserve" keyword begins reserving memory at Address 6 (to avoid overwriting the three well-known variables of Section 4.2). So the above example can be rewritten as LOAD (16, 5).

An externally defined keyword is specified outside of the mnemonic language. All of the application-defined parameters are considered to be externally defined keywords and can be referenced in the mnemonic code (useful for adapting the code based on the available memory or CPU cycles). The following additional keywords can also be used:

Keyword:	Corresponding value:
byte_copy_left	0
byte_copy_right	2

stack_location	4
reserved_end	See below
bytecode_length	See below
total_length	See below

The keyword reserved_end specifies the highest reserved memory address for the entire mnemonic code (taking into account all the occasions where memory is reserved).

The keyword bytecode_length specifies the total size of the bytecode corresponding to the mnemonic code. Any instances of bytecode_length are initially replaced with 3 bytes of zeroes, and then are filled in after the remainder of the bytecode has been generated.

Similarly, the keyword total_length specifies the total amount of memory required at the UDVM including bytecode and reserved memory addresses.

A complete description of the mnemonic language and how it should be translated into bytecode is given below:

Instructions:	Instruction names are given in capitals. Replace each name with the corresponding 1-byte value as per Chapter 6.
$:	When appended to the front of an instruction parameter then the parameter is a memory address rather than a direct value. This symbol is mandatory for reference parameters, optional for multitype parameters and disallowed for literals.
Integers:	Instruction parameters can be given in the form of decimal integers. They are converted into the shortest bytecode capable of representing the integer by the rules of Section 4.3.
Text references:	Instruction parameters can also be given in the form of lowercase names. These names should match exactly one label, reserved memory address or externally defined keyword as described above.
Labels:	Label names are given as a colon followed by lowercase text. They are deleted when converting the mnemonics to bytecode.
Reserved memory:	Memory addresses are reserved using the "reserve" keyword. The line containing the reserve keyword is deleted when converting to bytecode.
.LSB:	When appended to the end of a text name, the

integer value corresponding to the name is increased by 1. This is useful for addressing the LSBs of a 2-byte variable.

0b, 0d:	Bytecode values can be specified directly in binary or decimal via the appropriate prefix. The direct bytecode continues until a character occurs that is not an integer or whitespace.
Whitespace:	All whitespace (plus brackets and commas) just delimit the instructions. Delete.
Comments:	These are indicated by a semicolon and continue to the end of the line. Delete.

Once the mnemonic code has been converted into bytecode, it can be executed by copying the bytecode into the UDVM memory beginning at the first memory address that has not been reserved by an instance of the "reserve" keyword. Program execution is assumed to begin at this address.

Note that further to the rules outlined above, well-written mnemonic code will also have the following properties:

* Any instance of a memory address will be specified as a text reference rather than an integer value. This ensures that the mnemonic code is portable.

* The mnemonic code will not write to any memory address except those reserved by the "reserve" keyword. This ensures that the code can be compiled.

Appendix B. Example application-defined parameters

This appendix gives some example values for each of the application-defined parameters. These values are geared towards the compression of a text-based protocol running over UDP or TCP, for example a signaling protocol such as [SIP].

Note that all of the proposed values are fixed and not negotiated between the two instances of the application invoking the compressor and the UDVM. This is because it is possible for the application invoking the UDVM to receive compressed messages from several different applications, and it is difficult to determine which message corresponds to which application. [SIP] does this using "From:" and "To:" fields in the message itself, but these are not visible until the message has been decompressed. It is simpler just to fix a set of parameter for every instance of the application.

UDVM_version	0
maximum_compressed_size	65535
maximum_uncompressed_size	65535
minimum_hash_size	6
overall_memory_size	8192

```
   working_memory_start                0
   working_memory_end                  8191
   cycles_per_bit                      20
   cycles_per_message                  2000
```

first_instruction 26

Note that the parameters overall_memory_size, cycles_per_bit and cycles_per_message can be increased on the fly using the feedback mechanism of Chapter 7. This mechanism is designed to be function correctly even when the application invoking the UDVM is sent compressed messages from several different applications.

The initial contents of the UDVM memory also need to be defined. It is not enough simply to initialize the memory containing all zeroes, as the UDVM would be unable to input any compressed data. Instead, for each new compressed message the memory should be initialized containing a simple decompressor capable of extracting the first few bytes of compressed data. These bytes can then be interpreted as UDVM instructions for a more powerful decompression algorithm, a state reference to retrieve a previously stored algorithm etc.

As an example, the following mnemonic code can be converted to bytecode and pasted into the UDVM memory beginning at Address 26:

```
reserve length
reserve destination
reserve hash (16)

INPUT-BYTECODE (1, length, fail)
COMPARE (length, 16, retrieve_state, retrieve_state, new_code)

:retrieve_state

INPUT-BYTECODE ($length, hash, fail)
STATE-EXECUTE (hash, $length)

:new_code

INPUT-BYTECODE (2, destination, fail)
INPUT-BYTECODE ($length, $destination, fail)
SUBTRACT ($destination, execute_new_code)

:execute_new_code

JUMP ($destination)

:fail

DECOMPRESSION-FAILURE
```

The mnemonic code requests a single byte of compressed data, which is considered to be a length from 0 to 255. Lengths from 0 to 16 inclusive are interpreted as the length of a hash value that is used to retrieve and run bytecode previously stored as state. Lengths from 17 to 255 are interpreted as an amount of new UDVM bytecode to be

extracted from the start of the compressed data.

Finally, the application can define initial state that is available to the UDVM. Examples of application-defined state include common decompression algorithms, dictionaries of common text phrases etc.

Appendix C. Example decompression algorithms

This appendix gives examples of decompression algorithms which can be downloaded to the UDVM in the form of bytecode.

C.1. Example UDVM code for simple LZ77 decompression

The first example gives the code required to decompress data from a very simple LZ77-based algorithm. The UDVM is instructed to interpret a compressed message as a set of 4-byte characters, where each character contains a 2-byte position integer followed by a 2-byte length integer. Taken together these integers point to a previously received text string in the UDVM memory, which is then copied to the end of the uncompressed message.

Since the compressor can only send references to strings already present in the UDVM memory, before the first message is decompressed the memory must be initialized with a static dictionary containing the 256 ASCII characters.

The algorithm write-protects the memory containing the UDVM instructions used to decompress each character, so that they can easily be compiled to improve the speed of decompression.

A 2-byte CRC over the uncompressed message is appended to the end of the compressed message, to verify that correct decompression has occurred. The algorithm also requests that the contents of the UDVM memory be saved using the state request mechanism, so that it can be retrieved by sending the appropriate 6-byte hash.

```
reserve byte_copy_left
reserve byte_copy_right
reserve uncompressed_start
reserve uncompressed_end
reserve uncompressed_length
reserve position
reserve length
reserve static_dictionary (256)
reserve circular_buffer (2048)

WORKING-MEMORY (uncompressed_start, reserved_end)
MULTILOAD (0, 7, circular_buffer, reserved_end, static_dictionary,
circular_buffer, 0, 0, 0)

:unpack_static_dictionary
```

```
   ; The following instructions initialize the static dictionary.

   COPY-LITERAL (position.LSB, 1, $uncompressed_start)
   ADD ($position, 1)
```

```
   COMPARE ($position, 256, unpack_static_dictionary, next_character, 0)

   :next_character

   INPUT-FIXED (16, position, fail)
   INPUT-FIXED (16, length, end_of_message)
   COPY-LITERAL ($position, $length, $uncompressed_end)
   ADD ($uncompressed_length, $length)
   JUMP (next_character)

   :fail

   DECOMPRESSION-FAILURE

   :end_of_message

   CRC ($position, $uncompressed_start, $uncompressed_length, fail)
   OUTPUT ($uncompressed_start, $uncompressed_length)
   END-MESSAGE (6, 0, total_length, next_character, 0)
```