INTERNATIONAL TELECOMMUNICATION UNION

# ITU-T V.44

(11/2000)

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

SERIES V: DATA COMMUNICATION OVER THE TELEPHONE NETWORK

Error control

## Data compression procedures

ITU-T Recommendation V.44

(Formerly CCITT Recommendation)

ITU-T V-SERIES RECOMMENDATIONS

**DATA COMMUNICATION OVER THE TELEPHONE NETWORK**

| | |
|---|---|
| General | V.1–V.9 |
| Interfaces and voiceband modems | V.10–V.34 |
| Wideband modems | V.35–V.39 |
| **Error control** | **V.40–V.49** |
| Transmission quality and maintenance | V.50–V.59 |
| Simultaneous transmission of data and other signals | V.60–V.99 |
| Interworking with other networks | V.100–V.199 |
| Interface layer specifications for data communication | V.200–V.249 |
| Control procedures | V.250–V.299 |
| Modems on digital circuits | V.300–V.399 |

*For further details, please refer to the list of ITU-T Recommendations.*

**ITU-T Recommendation V.44**

# Date compression procedures


## Summary

This Recommendation describes a data compression algorithm for use in DCEs. It achieves a better performance than V.42 *bis* on many types of data. In addition to the normal stream method, the algorithm has a method which can compress data already contained in packets in an efficient way.


## Source

ITU-T Recommendation V.44 was prepared by ITU-T Study Group 16 (2001-2004) and approved under the WTSA Resolution 1 procedure on 17 November 2000.

## FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

## NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

## INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

## CONTENTS

**Page**

**ITU-T Recommendation V.44**

# Data compression procedures

## 1 Scope

### 1.1 General

This Recommendation describes a lossless data compression procedure for use with V-series data circuit-terminating equipment (DCEs).

The principal characteristics of the data compression procedure are:

a) a compression procedure based on an algorithm that encodes strings of characters input from data terminal equipment (DTE) as binary codes of variable length;

b) a decoding procedure that recovers the strings of characters from received binary codes of variable length;

c) a string-building mechanism that rapidly extends existing strings;

d) an automatically invoked transparent mode of operation when uncompressible data are detected.

Annex B describes the implementation of this data compression procedure in packet networks.

A summary of the set of parameters used in this Recommendation is given in clause 8.

Clause I.7 summarizes the main differences between the algorithms of ITU-T V.44 and V.42 *bis*.

This Recommendation contains examples for illustrative purposes; in any case in which an example appears to contradict the normative text, the normative text shall take precedence.

### 1.2 Requirements for error-correcting procedures

For correct operation of the data compression function, it is necessary that an error-correcting procedure be implemented between the two entities using this Recommendation. In the case of V-series Recommendations, the Link Access Procedure for Modems (LAPM) error-correcting procedures defined in ITU-T V.42 or the error-correcting procedures in ITU-T V.76 or V.120 must be implemented.

NOTE – Undetected bit errors will cause mis-operation of the data compression function. Use of a 32-bit frame check sequence (FCS), as defined in ISO/IEC 13239, substantially reduces the possibility of such errors, and may be desirable in environments with severe impairments. This 32-bit FCS is an option in V.42 LAPM.

### 1.3 A DCE employing data compression

The data compression function may be used with an error-correcting DCE, as shown in Figure 1 and described in clause 5. The elements of an error-correcting V-series DCE are specified in ITU-T V.42.

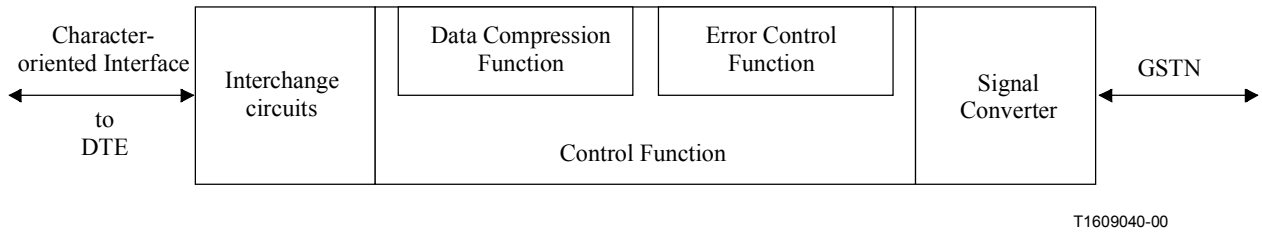


**Figure 1/V.44 – DCE employing data compression and error control**

## 2 References

The following ITU-T Recommendations contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published.

– ITU-T V.42 (1996), *Error-correcting procedures for DCEs using asynchronous-to-synchronous conversion*.

– ITU-T V.42 *bis* (1990), *Data compression procedures for data circuit-terminating equipment (DCE) using error correction procedures*.

– ITU-T V.120 (1996), *Support by an ISDN of data terminal equipment with V-series type interfaces with provision for statistical multiplexing*.

– ISO/IEC 13239:2000, *Information technology – Telecommunications and information exchange between systems – High-level data link control (HDLC) procedures*.

## 3 Definitions

This Recommendation defines the following terms:

**3.1 alphabet**: The set of all possible characters. In this Recommendation, the characters are contiguous from 0 to $N_4 - 1$ (see clause 8).

**3.2 append**: To create a new string using an unmatched character. If the unmatched character follows a string match, the new string consists of the string match with the unmatched character added to the end; if the unmatched character follows a character, the new string consists of both characters.

**3.3 character**: A single data element input from the DTE to the encoder using a predefined number of bits $N_3$ (see clause 8).

**3.4 code**: A sequence of bits output by the encoder that represents control or information. Defined code types are: control codes, ordinals, codewords, and string-extension lengths.

**3.5 code prefix**: One or two bits preceding a code that indicates the type of code that follows. These are defined in 6.6.3.

**3.6 codeword**: A binary number in the range 4 to $N_2 - 1$ that represents a string of consecutive characters. In the encoder, it corresponds to a particular string-segment. In the decoder, it corresponds to an entire string (see clause 8).

**3.7 compressed mode**: The mode of operation in which data from the DTE are transmitted as binary codes.

**3.8 compressed operation**: The state of the DCE in which the data compression function is active. Compressed operation has two modes: compressed mode and transparent mode. Transitions between these modes may be automatic, and based on the content of the data input from the DTE (see 6.5).

**3.9 control code**: A binary number in the range 0 to 3, reserved for use in DCE-to-DCE signalling of control information related to the compression function while in the compressed mode of operation (see 7.2).

**3.10 decoder**: The data compression subfunction that decompresses the output of an encoder.

**3.11 dictionary**: The data structure that represents the strings created while encoding or decoding. For the encoder, it consists of the root array, node-tree, and history; for the decoder, it consists of the string collection and the history.

**3.12 encoder**: The data compression subfunction that compresses data.

**3.13 ESCAPE**: In transparent mode, the indication of the beginning of a command sequence to the decoder.

**3.14 extend**: To create a new string, consisting of a string match with one or more characters added to the end.

**3.15 flush**: The encoder completes processing of characters input up to that point, transfers any codes that result, updates the dictionary as appropriate, and establishes octet alignment. The decoder subsequently establishes octet alignment.

**3.16 history**: The history buffer, herein called "the history", is the data structure that contains the characters input to the encoder (or equivalently, decoded by the decoder) since the most recent reinitialization of the data-structures. Characters are entered into the history in the order in which they are input (or decoded).

**3.17 multi-packet method**: The compression of data that are packetized (or framed) such that both ends of the data transmission can identify packet (or frame) boundaries, and several packets, or portions of packets, are processed as continuation. The multi-packet method is described in Annex B.

**3.18 node**: A node is a point in a tree-structure that represents a string-segment. It corresponds to a codeword in the encoder.

**3.19 node-tree**: The data structure in the encoder that represents the set of string-segments and their interrelationships. Combined with the root array, it represents the encoder tree-structures. A codeword corresponds to a particular string-segment and to a particular node.

**3.20 ordinal**: The ordinal of a character is the numerical equivalent of the character.

**3.21 packet method**: The compression of data that are packetized (or framed) such that both ends of the data transmission can identify packet (or frame) boundaries, and each packet is processed separately. The packet method is described in Annex B.

**3.22 parameter mode**: The mode of operation in which compression parameters are transferred between the data compression peer.

**3.23 receive**: In parameter negotiation, the direction corresponding to an entity's decoder.

**3.24 root**: A root is the point at the base of a tree-structure that represents, within the context of this Recommendation, the first character of a string (see Figure 2 and 6.2.1). Each character in the alphabet has a corresponding root.

**3.25 root array**: The data structure that contains the set of all encoder roots.

**3.26 stream method**: The compression of data that are transmitted continually over a link with guaranteed delivery. This method is the primary focus of ITU-T V.44.

**3.27 string**: An unbroken sequence of two or more characters. In the encoder dictionary, it consists of a first character (root) followed by one or more string-segments.

**3.28 string extension**: A sequence of one or more characters by which a string is extended to create a new and longer string.

**3.29 string-extension length**: A binary code indicating the number of characters by which the string corresponding to the immediately preceding codeword is extended.

**3.30 string-segment**: A section of a string that corresponds to a particular codeword in the encoder node-tree.

**3.31 string collection**: The data structure in the decoder that represents the decoded strings.

**3.32 transmit**: In parameter negotiation, the direction corresponding to an entity's encoder.

**3.33 transparent mode**: The mode of operation in which characters from the DTE are transmitted in uncompressed form. The data compression function is active, but does not influence the data transmission.

**3.34 tree-structure**: An abstract data structure in the encoder that is used in this Recommendation to represent a set of strings, all with the same initial character (see Figure 2 and 6.2.1).

**3.35 uncompressed operation**: The state of the DCE in which the data compression function is inactive.

**3.36 unmatched character**: A character that causes the termination of the string-matching or string-extension process because it does not match a specific character in the history. The unmatched character then becomes the first character of a possible new string match.

**3.37 { }**: {A} is the vector representation of a binary code. It has as many components as there are bits in the code. The component $A_1$ is the least significant bit of the binary code.

## 4 Abbreviations

This Recommendation uses the following abbreviations:

DCE Data Circuit-terminating Equipment

DTE Data Terminal Equipment

ECM Enter Compressed Mode: a transparent mode command defined in 7.2

EID ESCAPE In Data: a transparent mode command defined in 7.2

EPM Enter Parameter Mode: a transparent mode command defined in 7.2

ETM Enter Transparent Mode: a control code defined in 7.2

## 5 Functional description of a DCE

This clause describes a DCE employing a data compression function.

### 5.1 General

A DCE employing data compression, as depicted in Figure 1, contains the following components:

a) DTE/DCE interchange circuits;

b) a signal converter;

c) a control function;

d) an error-control function;

e) a data compression function.

### 5.2 DTE/DCE interchange circuits

This component shall have capabilities as described in ITU-T V.42.

### 5.3 Signal converter

This component shall have capabilities as described in ITU-T V.42.

### 5.4 Control function

This component shall have capabilities as described in 6.2/V.42. Additionally, it shall perform the following aspects of operation:

a) negotiation of modes of operation of the data compression function with the remote DCE, and negotiation of parameters associated with the operation of the data compression function;

b) instigation of the initialization or reinitialization of the data compression function;

c) coordination of the establishment of an error-controlled connection for use by the peer data compression functions;

d) coordination of the delivery of data between the DTE/DCE interface and the data compression function, in accordance with the procedures defined in 6.2/V.42 and 8.4/V.42, including the provision of the flow-control procedures defined therein;

e) coordination of the delivery of data between the data compression function and the error-control function;

f) action on detection of an exception condition.

### 5.5 Error-control function

This component shall have capabilities as described in ITU-T V.42.

### 5.6 Data compression function

The data compression function shall implement the procedures defined in this Recommendation in order to efficiently encode data prior to transmission over an error-controlled connection. It shall perform the following aspects of operation:

a) initialization and reinitialization of the encoder and decoder dictionaries;

b) data compression encoding and decoding;

c) switching between compressed and transparent modes of operation;

d) configuration of encoder and decoder in accordance with the parameters negotiated by the control function.

# 6 Procedures of the data compression function

## 6.1 Overview of the data compression function

The data compression function consists of an encoder and a decoder. A data connection, in general, supports data transmission in both directions, and thus can support data compression in both directions. Thus, each data connection peer can have an encoder and a decoder. The encoder transfers compressed data to its peer decoder at the other end of the connection, and the decoder decompresses the compressed data received from its peer encoder. The encoder-decoder pair for each direction must have coordinated data compression parameter values, established through negotiation; however at any one end, the encoder and decoder parameter values may differ, since they refer to different directions of transmission.

Characters from the DTE input to the encoder are matched against any previously identified strings of characters. If a string is matched, the codeword representing the string is transferred, and then an attempt is made to extend the matched string to encode additional characters and create a new and longer string. If no string match is found, the ordinal corresponding to the first character is transferred.

The encoder continually adds to its set of strings available for matching, by placing input characters into the history and by adding nodes to the node-tree. When either of these becomes full, the dictionary is reinitialized, and the encoding operation continues as before.

The decoder creates strings that replicate the strings created by its peer encoder, using the same assigned codewords. By using these strings, it decompresses the received compressed data stream. The decoder dictionary is reinitialized upon receipt of a REINIT control code.

## 6.2 Dictionary structure

To describe in detail the data compression algorithm, it is useful to define the following data structures.

### 6.2.1 Encoder dictionary

The encoder uses a dictionary consisting of three parts:

1) *root array*: This contains an entry for each character in the alphabet. The size of the alphabet is $N_4$, and is determined by the size of the characters input from the control function: for an alphabet of 8-bit characters, there are 256 entries. Each entry is indexed by its character and serves as the root for a tree-structure; it contains a down-index to a node in the node-tree. This index is the starting point for climbing down the tree-structure.

2) *node-tree*: This contains the codewords created during the operation of the encoder, and the string-segments to which they correspond. The nodes in the node-tree define a set of tree-structures, and the corresponding string-segments. Each node contains a codeword, the position in the history of the first character of the string-segment, the number of characters in the string-segment, and two indices for linking to other nodes. The "down-index", if valid, points to a node that represents a string-segment that follows this string-segment. The "side-index", if valid, points to a node that represents a string-segment at the same level as this string-segment (i.e. that also follows the root or string-segment that precedes this string segment). The number of codewords cannot exceed $N_{2T}$.

3) *history*: This contains all the characters input to the encoder, in order, since the most recent dictionary reinitialization. String-segments in the node-tree are referenced by the position of their first character in the history and by their length. The number of characters in the history cannot exceed $N_{8T}$.

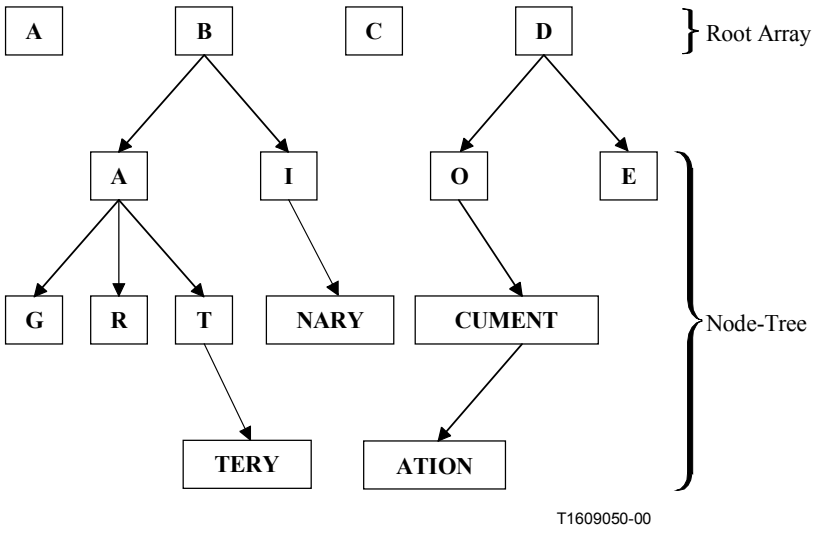


**Figure 2/V.44 – Example Tree-structures**

This figure shows only the tree-structures starting from roots A, B, C and D.

The tree-structures of Figure 2 show the string-segments, each represented by a node:

– A and I, following root character B;

– G, R, and T following string-segment A;

– TERY following string-segment T;

– NARY following string segment I;

– O and E following root character D;

– CUMENT following string-segment O;

– ATION following string-segment CUMENT.

The strings that are represented: BA, BAG, BAR, BAT, BATTERY, BI, BINARY, DO, DOCUMENT, DOCUMENTATION, DE. A complete string is matched by starting at the root using the next character to be encoded and matching successive string-segments down the tree using subsequently input characters.

### 6.2.2 Decoder dictionary

The decoder uses a dictionary consisting of two parts:

1) *string collection*: This defines the strings created during the decoding process that correspond to the string-segments created by the encoding process. Each entry has a codeword, the position in the history of the last character of the string, and the total length of the string. The number of codewords cannot exceed $N_{2R}$.

2) *history*: This contains all the characters decoded, in order, since the most recent reinitialization of the dictionary. It is identical to the encoder's history; however, it is generated using the characters resulting from the decoding process. The number of characters in the history cannot exceed $N_{8R}$.

## 6.3 Encoding

The encoding algorithm attempts to find a string of previously processed characters that matches the next characters to process; if this is found, the codeword representing the string is transferred. Subsequently, the decoder, upon receiving the codeword, can correctly regenerate this string. Data compression is successful if the encoded string requires fewer bits to transfer than the original uncompressed characters.

Input characters are placed into the next available locations in the history and processed by the encoder as follows:

a) Using the input character and the next character(s), the encoder attempts to find the longest matching string in the dictionary.

b) If a string match is not found, the encoder transfers the ordinal corresponding to the input character, and returns to step a) using the next character.

c) If a string match is found, the encoder transfers the codeword assigned to the last completely matched string-segment of the longest string match.

d) Subsequently, the encoder attempts to extend the longest string match, up to the maximum string length. The encoder attempts to match the next character(s) to the character(s) in the history immediately following the last character of the last completely matched string-segment of the longest string match.

e) If the longest string match is successfully extended by one or more characters, the encoder transfers a string-extension length indicating the number of characters by which the string has been extended.

Using the unmatched character (i.e. the first character that was not part of the string match or the extension) as the next character, the encoder returns to step a).

### 6.3.1 String-matching procedure

A string in the encoder dictionary consists of a first character followed by one or more string-segments of the node-tree; for example, in Figure 2, the string "BATTERY" has first character "B", followed by string-segment "A", followed by string-segment "T", followed by string-segment "TERY". The total length of the string cannot exceed $N_{7T}$.

The string-matching procedure uses the dictionary structure described in 6.2 to find a longest string match as follows:

– Each root in the root array has one down-index: if it is set to a valid index, the corresponding node provides the first string-segment for comparison.

– Each node in the node-tree has one side-index used by the encoder to search for an another string-segment that also branches from the same previous node or root (another node at the same level).

– Each node in the node-tree has one down-index used by the encoder to search for a string-segment for continued comparison (a node at the next level).

In searching for a string match, the encoder examines the root corresponding to the first character input, and compares subsequently input characters to the string-segments corresponding to nodes directly branching from that root. If the subsequently input characters exactly match any complete string-segment (partial matches are not valid), the encoder compares additional input characters to the string-segments of the nodes branching from that node. Thus, the encoder continues down the tree-structure until it cannot find any further matches or until there are no more branching nodes.

The string-matching procedure fails if the root corresponding to the first character has no valid down-index; or if no second-level node matches the second character. In this case, the ordinal corresponding to the first character is transferred, and the string-matching procedure begins again with the next (unmatched) character.

Otherwise, the string-matching procedure finds a node that corresponds to the longest string in the dictionary that matches the input characters. The encoder transfers the codeword for this node. (Note that the codeword actually corresponds directly to the last completely matched string-segment of the longest string match, as the encoder does not explicitly associate codewords with complete strings.)

The encoder can transfer any codeword it has created. If the codeword value is equal to the decoder's value of $C_1$, this indicates a codeword not yet created in the decoder; nonetheless, the decoder is required to correctly interpret all codeword values up through $C_1$. See 6.4.1.

The string-matching procedure can also be terminated for the following reasons: the history is full; or a FLUSH request is received. The processing is similar to that when the next input character does not match any character in the current string-segment: the codeword for the last completely matched string-segment (if any) is transferred; then if there is a partially matched string-segment, the successfully matched characters are transferred as a string-extension length, and a corresponding node-tree entry is created.

NOTE 1 – In order for a string-segment to match the input characters, all characters must match: partial matches are not valid. However, a partial match of a string-segment can be used to extend the string, if no complete string-segment match is found at the same level. In this case, the string-matching procedure has already done the comparisons needed to find the longest possible string extension specified in 6.3.2, and this number is precisely the number of matched characters of the partially matched string-segment. Therefore, it is not necessary to initiate the string-extension procedure explicitly, as the encoder can simply transfer the string-extension length corresponding to the number of matched characters of the partially matched string-segment.

NOTE 2 – It is advantageous to order the nodes at one level that branch off a particular node, alphabetically by first character, to allow more efficient search for matching string-segments. There can be multiple string-segments, each with the same first character, branching off a node.

### 6.3.2 String-extension procedure

If the string-matching procedure is successful (i.e. a longest string match is found), and if the total length of the string is less than $N_{7T}$, the encoder initiates the string-extension procedure. The encoder attempts to extend the longest string match by comparing the next character(s) to the character(s) in the history that immediately follow the last character of the last completely matched string-segment.

If a C-FLUSH is received from the control function immediately following a successful string match and before the string-extension procedure has processed a character, the string-extension procedure is terminated. The character immediately following the C-FLUSH, when received, starts a new string match and the dictionary is updated as if that character were unmatched.

For instance, suppose that the string "BATTERY" is the longest string match. The encoder compares the next character to the character in the history following the "TERY" string-segment. There are two possibilities:

- The next character does not match the character in the history immediately following the "Y" of the "TERY" string-segment. The string-extension procedure terminates unsuccessfully, and this next character is appended to the "TERY" to create a new and longer string, and also becomes the first character of a possible new string match.

- The next character does match the character in the history immediately following the "Y" of the "TERY" string-segment. The string-extension procedure continues by comparing the subsequent character input to the second character in the history following the "Y" of the "TERY" string-segment; and the next character input to the third character in the history following the "Y"; and so forth. The number of characters successfully matched is transferred as a string-extension length. The encoder creates a new string-segment in the node-tree, consisting of the string extension (the sequence of characters matched after the longest string match). This string-segment is assigned the next available codeword, and has string-segment length equal to the length of the extension. Its history index references the position in the history of the first of the most recent input characters used to extend the string.

The string-extension procedure can also be terminated for the following reasons: the history is full; a FLUSH request; the length of the total string has reached $N_{7T}$. The processing is handled in a way similar to that when the next input character does not match the next character in the history: any characters that have been successfully matched to extend the string are transferred as a string-extension length, and a corresponding node-tree entry is created.

### 6.3.3 Creating string-segments

The encoder continually creates new strings for possible future string matches. The two methods for creating new string-segments are:

**Append**

*String-matching failure*:

– If the string match fails because the first character has no valid down-index in the root array, a 1-character string-segment is created, linking the next character to the root of the first character.

– If the string match fails because the next character does not match any existing nodes branching from the first character's root, a 1-character string-segment is created using the unmatched character, branching from the root of the first character.

*String-extension failure*:

– A 1-character string-segment is created linking the unmatched character to the last completely matched string-segment of the longest string match.

**Extend**

*String-extension success*:

A string-segment is created extending the longest string match by one or more characters. The string-segment thus created has length equal to the number of characters of the extension, and is linked to the last completely matched string-segment.

### 6.3.4 Encoding summary

**Table 1/V.44 – Encoding procedures**

| Procedure | Result | Code transferred | New string-segment | Use of unmatched character |
|---|---|---|---|---|
| String-Matching | Failure | Ordinal corresponding to first character | Append unmatched character to root of first character | As first character for String-Matching |
| String-Matching | Success | Codeword for last completely matched string-segment | None | For String-Extension |
| String-Extension | Failure | Nothing | Append unmatched character to last completely matched string-segment | As first character for String-Matching |
| String-Extension | Success | String-extension length | All extension characters, linked to last completely matched string-segment | As first character for String-Matching |

Table 1 summarizes the encoding procedures. Appendix II illustrates the operation of the algorithm explicitly in terms of the dictionary structure.

## 6.4 Decoding

A string in the decoder is defined by the position in history of the last character of the string, and by the total length of the string. It corresponds to a particular codeword that, by construction, corresponds in the encoder to the final string-segment of that string.

In compressed mode, the binary codes received by the decoder are parsed into control codes, ordinals, codewords and string-extension lengths, by examining the code prefixes. The decoder must remain synchronized with the strings and codewords that are created by the encoder, updating the string collection and history based upon codes received. The decoder is simpler than the encoder, because the encoder must search for string matches and create new and longer strings, whereas the decoder needs only to keep track of the strings it has created.

The decoder shall be capable of operation in both transparent and compressed modes.

### 6.4.1 Processing codes

In compressed mode, the decoding function shall operate as follows:

1) On receipt of an ordinal, the decoder shall place the corresponding character into the decompressed output and into the history.

2) On receipt of a codeword less than $C_1$, the decoder shall copy the string represented by the codeword into the decompressed output and into the history.

3) On receipt of a codeword equal to $C_1$, if the previous code received was a codeword, the decoder shall copy the string represented by that previous codeword into the decompressed output and into the history; and shall then place the first character of the string represented by that previous codeword into the decompressed output and into the history.

4) On receipt of a codeword equal to $C_1$, if the previous code received was an ordinal, the decoder shall process the corresponding character according to step 3) above, as if it were a string of length one.

5) On receipt of a string-extension length, the decoder shall use the preceding codeword to access the characters in the history immediately following the string represented by that codeword. The number of characters indicated by the string-extension length are copied into the decompressed output and into the history.

6) Receipt of a codeword greater than $C_1$ constitutes a procedural error (see 7.15).

### 6.4.2 Creating new strings

The rules for creating new strings are as follows:

**Table 2/V.44 – Rules for string creation**

| Current code | Previous code | New string created |
|---|---|---|
| Ordinal | Ordinal | The corresponding character is appended to the previous character to create a 2-character string. |
| Ordinal | Codeword | The corresponding character is appended to the previous string to create a longer string. (Note) |
| Ordinal | String-extension length | The corresponding character is not appended to the extended string. |
| Ordinal | FLUSH control code | The corresponding character is treated as if the FLUSH had not occurred. Action is taken as if the code preceding the FLUSH were the "previous code." |
| Ordinal | REINIT, ECM or initialization | None: the dictionary is empty. |
| Codeword | Ordinal | The first character of the codeword's string is appended to the previous character to create a 2-character string. |
| Codeword | Codeword | The first character of the codeword's string is appended to the previous string to create a longer string. (Note) |
| Codeword | String-extension length | The first character of the codeword's string is not appended to the extended string. |
| Codeword | FLUSH control code | The first character of the codeword's string is treated as if the FLUSH had not occurred. Action is taken as if the code preceding the FLUSH were the "preceding code." |
| String-extension length | Codeword | The previous string is extended to create a new and longer string. (Note) |
| NOTE – Strings of length greater than $N_{7R}$ shall not be created. | | |

An intervening STEPUP control code does not affect string creation.

## 6.5 Transparent mode

The overhead of the compressed mode operation can sometimes exceed the savings from compression. When such a situation occurs, the encoder can enter the transparent mode of operation to transfer the input characters un-encoded, without requiring a new connection.

While in transparent mode, the encoding activity may continue, in order to determine when re-entry to compressed mode would be advantageous; but the output of the encoder is not transferred. Instead, the characters input to the encoder are transferred without modification, octet-aligned. The encoder shall reinitialize the dictionary upon leaving transparent mode and re-entering compressed mode.

In transparent mode, the decoder shall process the transparent characters and commands but not update the history or string collection. The decoder dictionary shall be brought into a state compatible with the peer encoder dictionary, by re-initializing upon leaving transparent mode.

### 6.5.1 Transition from compressed mode to transparent mode

In compressed mode, if the encoder determines that the data stream is not compressible, it shall:

a) ensure that all characters input up to that point are transferred according to normal encoding procedures;

b) transfer the ETM (enter transparent mode) control code to indicate a transition to transparent mode to the peer decoder;

c) transmit enough 0 bits to establish octet-alignment (see Table 6);

d) enter transparent mode.

In compressed mode, upon receipt of an ETM control code, the decoder shall enter transparent mode.

### 6.5.2 Transition from transparent mode to compressed mode

In transparent mode, if the encoder determines that the data stream is compressible, it shall:

a) transfer all characters input up to that point, in transparent mode;

b) reinitialize the encoder dictionary;

c) transfer the current value of the ESCAPE to indicate a command sequence;

d) transfer the ECM command;

e) enter compressed mode.

In transparent mode, upon receipt of an ESCAPE immediately followed by an ECM command, the decoder shall reinitialize the dictionary and enter compressed mode.

## 6.6 Transfer

In transparent mode, characters shall be transferred to the control function for transmission in octet-aligned form, using a C-TRANSFER_indication. They may be transferred individually during the string-matching or string-extension procedures, or as a sequence following completion of the string-matching or string-extension procedure.

In compressed mode, the binary codes output by the encoder (6.3) shall be transferred to the control unction. The least significant bit of the binary code prefix shall immediately follow the most significant bit of the preceding binary code. Each code is preceded by a code prefix.

After transition from transparent to compressed mode, the least significant bit of the code prefix for the first binary code to be transferred shall be bit 1 of the first compressed octet.

In transition from compressed to transparent mode, enough 0 bits shall be transferred to ensure that the next transmitted character is octet-aligned.

Following a FLUSH control code, enough 0 bits shall be transferred to ensure that the next transmitted code is octet-aligned.

### 6.6.1 Transfer of control codes, ordinals, and codewords

Control codes are transferred using the number of bits defined by the current value of $C_2$.

Ordinals are transferred using the number of bits defined by the current value of $C_5$.

Codewords are transferred using the number of bits defined by the current value of $C_2$.

### 6.6.2 Transfer of string-extension length

A string-extension length is a code representing the length by which the string represented by the previous codeword is to be extended. A string-extension length consists of one or more subfields, each of which shall be individually parsed by the decoder. The number of subfields in a string-extension length is dependent on the number of characters (i.e. the length) of the string extension. The length of a string extension ranges from 1 to the lesser of 253 or ($N_{7T} – 2$). The encoding of the subfields for string extensions with length 1 through 12 is shown in Table 3; for clarity, the order of transmission of bits is also shown.

**Table 3/V.44 – String-extension length: lengths 1 through 12**

| Length | Subfields | Order of transmission of bits (left-to-right) |
|---|---|---|
| 1 | "1" | 1 |
| 2 | "0" "01" | 0 10 |
| 3 | "0" "10" | 0 01 |
| 4 | "0" "11" | 0 11 |
| 5 | "0" "00" "0" "000" | 0 00 0 000 |
| 6 | "0" "00" "0" "001" | 0 00 0 100 |
| 7 | "0" "00" "0" "010" | 0 00 0 010 |
| 8 | "0" "00" "0" "011" | 0 00 0 110 |
| 9 | "0" "00" "0" "100" | 0 00 0 001 |
| 10 | "0" "00" "0" "101" | 0 00 0 101 |
| 11 | "0" "00" "0" "110" | 0 00 0 011 |
| 12 | "0" "00" "0" "111" | 0 00 0 111 |

For string-extension lengths greater than 12, the encoding depends on the value of the maximum string length $N_{7T}$, as shown in Table 4. The value of $N_{7T}$ determines the number of bits in the last subfield, while the string-extension length determines the value of the last subfield:

$$N = \text{string-extension length} - 13$$

and:

$$N_x = \text{x-th binary digit of } N$$

**Table 4/V.44 – String-extension length: lengths 13 through 253**

| Maximum string length parameter $N_{7T}$ | Range of lengths | Subfields | Order of transmission of bits (left-to-right) |
|---|---|---|---|
| $32 \le N_{7T} \le 46$ | 13-44 | "0" "00" "1" "$N_4N_3N_2N_1N_0$" | 0 00 1 $N_0N_1N_2N_3N_4$ |
| $46 < N_{7T} \le 78$ | 13-76 | "0" "00" "1" "$N_5N_4N_3N_2N_1N_0$" | 0 00 1 $N_0N_1N_2N_3N_4N_5$ |
| $78 < N_{7T} \le 142$ | 13-140 | "0" "00" "1" "$N_6N_5N_4N_3N_2N_1N_0$" | 0 00 1 $N_0N_1N_2N_3N_4N_5N_6$ |
| $142 < N_{7T} \le 255$ | 13-253 | "0" "00" "1" "$N_7N_6N_5N_4N_3N_2N_1N_0$" | 0 00 1 $N_0N_1N_2N_3N_4N_5N_6N_7$ |
| NOTE – The maximum length of a string extension is ($N_{7T} - 2$), because the minimum length of a string is 2 characters. | | | |

### 6.6.3 Code prefixes

Code prefixes are transferred immediately before their codes. The code prefixes are shown in Table 5.

**Table 5/V.44 – Code prefix usage**

| Code type | **Prefix: immediately after a codeword** | | **Prefix: otherwise** (Note) | |
|---|---|---|---|---|
| | *Subfields* | *Order of transmission of bits (left-to-right)* | *Subfields* | *Order of transmission of bits (left-to-right)* |
| Control code | "1" | 1 | "1" | 1 |
| Ordinal | "0" "0" | 0 0 | "0" | 0 |
| Codeword | "1" | 1 | "1" | 1 |
| String-extension length | "0" "1" | 0 1 | n/a | n/a |
| NOTE – Immediately after a control code, ordinal, string-extension length; or after reinitialization. | | | | |

### 6.6.4 Example of Transfer

Table 6 illustrates a sequence of compressed octets transferred by the encoder. In this example, several codes are transferred in compressed mode, and then the encoder makes a transition to transparent mode. The example assumes that the value of the current codeword size $C_2$ is 11 and the value of the current ordinal size $C_5$ is 8 (see Table 12). The codes are:

- codeword{A}with prefix "1";
- ordinal{B}with prefix "0" "0" (because it immediately follows a codeword);
- ordinal {C} with prefix "0";
- codeword {D}with prefix "1";
- String-extension length 8: prefix "0" "1", subfields "0" "00" "0" "011";
- ordinal{E}with prefix "0";
- ETM control code with prefix "1". The value of ETM is "00000000" which is extended to 11 bits;
- padding: five 0's to establish alignment in octet i + 9;
- character{F}in transparent mode, octet-aligned.

**Table 6/V.44 – Octet mapping for an example data stream**

| Bit No. | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | Octet | Mode |
|---|---|---|---|---|---|---|---|---|---|---|
| | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | ***1*** | ... | ... | i | compressed |
| | ***0*** | ***0*** | $A_{11}$ | $A_{10}$ | $A_9$ | $A_8$ | $A_7$ | $A_6$ | i + 1 | compressed |
| | $B_8$ | $B_7$ | $B_6$ | $B_5$ | $B_4$ | $B_3$ | $B_2$ | $B_1$ | i + 2 | compressed |
| | $C_7$ | $C_6$ | $C_5$ | $C_4$ | $C_3$ | $C_2$ | $C_1$ | ***0*** | i + 3 | compressed |
| | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | ***1*** | $C_8$ | i + 4 | compressed |
| | 0 | ***1*** | ***0*** | $D_{11}$ | $D_{10}$ | $D_9$ | $D_8$ | $D_7$ | i + 5 | compressed |
| | $E_1$ | ***0*** | 0 | 1 | 1 | 0 | 0 | 0 | i + 6 | compressed |
| | ***1*** | $E_8$ | $E_7$ | $E_6$ | $E_5$ | $E_4$ | $E_3$ | $E_2$ | i + 7 | compressed |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | i + 8 | compressed |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | i + 9 | compressed |
| | $F_8$ | $F_7$ | $F_6$ | $F_5$ | $F_4$ | $F_3$ | $F_2$ | $F_1$ | i + 10 | transparent |
| Code prefix bits are in ***bold italics***, padding bits are underlined. | | | | | | | | | | |

# 7 Operations of data compression

## 7.1 Communication between the control and data compression functions

Communication between the control function and the data compression function is modelled as a set of abstract primitives of the form X-NAME_type that represent the logical exchange of information and control to accomplish a task or service. In the context of this Recommendation, the control function is viewed as the "service user" while the data compression function is viewed as the "service provider". The types of primitive are: request, indication, response, and confirm. The services used in this Recommendation are listed in Table 7.

**Table 7/V.44 – Services expected by the control function**

| Service | Primitive | Reference |
|---|---|---|
| Initialize the data compression function | C-INIT | 7.5 |
| Transfer negotiation parameters to/from the data compression function | C-PARM | 7.4.2 |
| Indicate an error to the control function | C-ERROR | 7.15 |
| Transfer uncompressed data to/from the data compression function | C-DATA | 7.7 |
| Transfer compressed data to/from the data compression function | C-TRANSFER | 7.9 |
| Flush remaining un-transferred data from the encoder | C-FLUSH | 7.13 |

## 7.2 Communications between peer data compression functions

The control codes and commands used for communication between peer data compression functions are in Tables 8 and 9.

**Table 8/V.44 – Control codes used in compressed mode**

| Code | Name | Description |
|---|---|---|
| 0 | ETM | Enter Transparent Mode |
| 1 | FLUSH | Flush data |
| 2 | STEPUP | Increase codeword size or ordinal size by one |
| 3 | REINIT | Force reinitialization of dictionaries |

**Table 9/V.44 – Commands used in transparent mode**

| Value | Name | Description |
|---|---|---|
| 0 | ECM | Enter Compressed Mode |
| 1 | EID | ESCAPE In Data |
| 2 | EPM | Enter Parameter Mode |
| Variable: see 7.14 | ESCAPE | Initiates a command sequence |

## 7.3 Negotiation of V.44 capability

Prior to initiating the data compression function, the modem peers must determine if they have V.44 capability. V.44 capability shall be determined at link establishment through a protocol (for example, the XID procedure defined in ITU-T V.42), and shall remain in-effect for the duration of the error-corrected connection. In particular, when using this Recommendation with V.42 error control, the XID negotiation procedure shall be used (see ISO/IEC 13239 and 7.7/V.42, 8.10/V.42 and clause 10/V.42). Parameters within the user data subfield, in addition to those defined in ITU-T V.42, shall be used for this purpose. The user data subfield shall appear in the XID frame immediately before the FCS and shall be encoded as in Table A.1.

The presence of the "V.44 capability" parameter within an XID message (or other protocol message) indicates that the sending entity has the capability described in this Recommendation. The format of the V.44 capability parameter for an XID message when using V.42 error-corrected procedures is described in Annex A. It should be noted that certain bits within this parameter should be ignored while negotiating over modem connections, as they specify V.44 capabilities for use in packet networks, as described in Annex B.

NOTE – During the protocol establishment phase, the presence of parameter type 0x40 with a Parameter Set Identifier of "V.44" in the User Data Subfield of the XID frame shall indicate a request for V.44 data compression. The responder shall include parameters for at most one compression algorithm (V.42 *bis* or V.44) in the response XID.

## 7.4 Negotiation of data compression parameters

Once V.44 capability has been determined, the values of the data compression parameters can be negotiated between the data compression peers. These parameters are defined in clause 8, Table 10: the "P" parameters are the values proposed by either side of the negotiation, and the "N" parameters are the values finally agreed.

The parameters for data compression may differ for the two directions of transmission. During negotiation, either side of the connection may propose values for any of these parameters: the subscript "T" indicates that the parameter refers to the transmit direction (encoder) of the entity sending the message; the subscript "R" indicates that the parameter refers to the receive direction (decoder) of the entity sending the message.

Thus, in negotiating the directions in which data compression will operate, the complementary response to one entity's $P_0$ value of 01 (proposing operation only in the direction of the sending entity's encoder and the responding entity's decoder) is a $P_0$ value of 10 (agreeing to operation only in the direction of the responding entity's decoder and the (original) sending entity's encoder). A response of 01 or 11 would inappropriately propose data compression in a direction not originally requested; a response of 00 will result in data compression in neither direction.

In negotiating parameter values, the lesser value of two proposals is used. For example, the proposed $P_{2T}$ from one entity is compared with the proposed $P_{2R}$ from the other entity; if both values are valid, the lesser value shall be used; any attempt to specify a value less than the minimum is a procedural error and shall result in disconnection. The final agreed value is set into $N_{7T}$ by the entity proposing $P_{2T}$ and into $N_{7R}$ by the entity proposing $P_{2R}$.

Appendix I provides guidance on the implications of the values of these parameters for data compression performance.

The negotiation of these values may proceed in one of two ways, as indicated in the V.44 capability parameter value: either through the XID capability determination itself, or through negotiation after link establishment. XID negotiation is the default. A modem receiving an XID requesting XID parameter negotiation must respond with an XID with V.44 parameters; therefore a modem which has requested Parameter Mode negotiation in the XID sequence, but which has received an XID negotiation request, must further respond with an XID with V.44 parameters.

### 7.4.1 Negotiation through XID

In the User Data Subfield of the XID, the V.44 capability parameter indicates how the data compression parameters are to be negotiated; if it indicates that XID procedures are to be used, the values proposed by the sending entity are included later in the same subfield.

### 7.4.2 Negotiation after link establishment

If the V.44 capability parameter indicates that parameter negotiation is to take place after the error-corrected link is established, the control function performs this after link establishment and anytime thereafter, as it deems necessary. The control function shall insure that data transfer between the data compression peers is not in-progress, and that flow control is in-effect, prior to initiating parameter negotiation.

The control function uses C-PARM primitives to pass parameters to/from the data compression function, as follows:

- Upon receipt of a C-PARM_request from the control function, the encoder shall:
  - transition from compressed mode to transparent mode using the procedures of 6.5.1; or else remain in transparent mode;
  - transition to parameter mode and transfer the EPM (Enter Parameter Mode) command to indicate this transition to the peer decoder;
  - transfer the parameters passed with the C-PARM_request to the peer decoder;
  - transfer the End parameter (see Table 10) to indicate a transition to transparent mode to the peer decoder;

- issue a C-PARM_confirm to the control function;
- transition to transparent mode.

• Upon receipt of a C-PARM_confirm from the data compression function, the control function shall:
  - if the parameters just transferred to the peer decoder successfully complete parameter negotiation, the control function shall reinitialize the data compression function according to the procedures defined in 7.5;
  - otherwise, the control function shall continue to maintain flow control pending receipt of parameters from the data compression peer.

• Upon receipt of an EPM command from the peer encoder while in transparent mode, the decoder shall:
  - transition to parameter mode;
  - receive the parameters;
  - transition to transparent mode when the End parameter is received;
  - issue a C-PARM_indication to the control function passing to it the received parameters.

• Upon receipt of a C-PARM_indication from the data compression function, the control function shall:
  - if the parameters passed with the C-PARM_indication successfully complete parameter negotiation, the control function shall reinitialize the data compression function according to the procedures defined in 7.5;
  - otherwise, the control function shall issue a C-PARM_request to transfer responding parameters to its data compression peer.

### 7.5 Initialization of the data compression function

After successful negotiation of V.44 capability, the control function shall issue the primitive C-INIT_request to the data compression function. If the V.44 data compression parameters have been negotiated within the XID message exchange, the C-INIT primitive shall indicate the negotiated value of the parameters; otherwise, the C-INIT primitive shall indicate the default value of the parameters as defined in Table 10. The data compression function shall initialize the encoder dictionary to the state defined by 7.5.1, and the decoder dictionary to the state defined by 7.5.2. The data compression function shall be set to compressed mode, and the ESCAPE is assigned the value 0.

A data compression function reinitialization can be invoked by the control function. The control function shall issue a C-INIT_request to the data compression function on the following conditions:

- receipt of L-ESTABLISH_indication or L-ESTABLISH_confirm;
- receipt of L-SIGNAL_indication or L-SIGNAL_confirm, where the primitive indicates a destructive form. See Table 4/V.42;
- receipt of C-PARM_indication which successfully completes parameter negotiation initiated by this entity. The C-INIT primitive shall indicate the just-negotiated value of the parameters; or
- receipt of C-PARM_confirm which successfully completes parameter negotiation initiated by the data compression peer. The C-INIT primitive shall indicate the just-negotiated value of the parameters.

It is the responsibility of the control function to ensure that C-INIT_request primitives are issued only when no data are in transit between the data compression functions (e.g. in the error-control functions), to ensure synchronization between the encoders and decoders.

### 7.5.1 Initial state of the encoder dictionary

The initial values for the encoder variables are:

– The "next codeword" $C_1 = N_5$.

– The "current codeword size" $C_2 = 6$.

– The "threshold for changing the codeword size" $C_3 = 64$.

– The "current history position" $C_4 = 0$.

– The "current ordinal size" $C_5 = 7$.

– All down-indices in the root array are set to non-valid values.

### 7.5.2 Initial state of the decoder dictionary

The initial values for the decoder variables are:

– The "next codeword" $C_1 = N_5$.

– The "current codeword size" $C_2 = 6$.

– The "current history position" $C_4 = 0$.

– The "current ordinal size" $C_5 = 7$.

## 7.6 Establishment of error-controlled connection

On receipt of the primitive C-INIT_confirm from the data compression function, the control function shall indicate to the DTE that data transfer may commence.

## 7.7 Transfer of data between the DTE/DCE interface and the data compression function

On completion of connection establishment, the control function shall request encoding of data input to the DTE/DCE interface. To encode data, the control function shall issue the primitive C-DATA_request to the data compression function. The primitive shall indicate the data to be encoded.

On receipt of the primitive C-DATA_indication from the data compression function, the control function shall deliver the decoded data to the DTE/DCE interface.

Flow-control procedures will be needed to avoid potential loss of data due to buffer overflow. When the procedures defined in this Recommendation are used in conjunction with those defined in ITU-T V.42, the flow-control procedures defined in 7.3.1/ V.42 and 8.4.2/V.42 shall be applied.

## 7.8 Encoding

Characters input from the control function shall be encoded using the procedures described in 6.3.

## 7.9 Transfer of data between the data compression function and the error-control function

On receipt of the primitive C-TRANSFER_indication from the data compression function, the control function shall issue the primitive L-DATA_request to the error-control function.

On receipt of the primitive L-DATA_indication from the error-control function, the control function shall issue the primitive C-TRANSFER_request to the data compression function.

The data compression function will use the procedures described in 6.6.

## 7.10 Decoding

Encoded data received from the control function shall be decoded using the procedures of 6.4.

### 7.11 Autonomous adjustments

During the course of operation, certain variables are autonomously changed by the data compression function. The mode of operation may also change autonomously from compressed mode to transparent mode.

#### 7.11.1 Ordinal size and STEPUP

At dictionary reinitialization, the encoder and its peer decoder initialize for the transfer of an ordinal of 7 bits ($C_5 = 7$). When the encoder is about to transfer the first ordinal with numerical value greater than $127_{10}$, it shall set $C_5$ to 8. If it is in compressed mode, it shall also transfer the STEPUP control code immediately before transferring the prefix and ordinal.

#### 7.11.2 Codeword size and STEPUP

The codewords and control codes are transferred using the number of bits defined by $C_2$. At dictionary reinitialization, $C_2$ is set to the value of 6 and $C_3$ is set to the value of 64. When the encoder is about to transfer a codeword with numerical value greater than or equal to $C_3$:

a) the encoder shall transfer the STEPUP control code, using the current codeword size, $C_2$;

b) the codeword size $C_2$ shall be increased by one;

c) $C_3$ shall be doubled;

d) if the codeword to be transferred is still numerically greater than or equal to $C_3$, steps a) through c) shall be repeated.

Then the prefix and codeword are transferred.

#### 7.11.3 Node-tree full

When the encoder is unable to create a new codeword because the node-tree is full ($C_1 = N_{2T}$), it shall reinitialize the dictionary, as described in 7.12.

#### 7.11.4 History full

When the history is full ($C_4 = N_{8T}$), the encoder shall terminate any string-matching or string-extension activity in progress, transfer the code or codes that result, and reinitialize the dictionary, as described in 7.12. The next input character shall be placed in the first position of the history.

#### 7.11.5 Data compressibility monitoring

The encoder shall periodically apply a test to determine the compressibility of the data. The nature of the test is not specified in this Recommendation; however, it would consist of a comparison of the number of bits required to represent a segment of the data stream before and after compression.

The monitoring of the data stream for compressibility continues in both compressed and transparent modes of operation.

In compressed mode, if the encoder determines that the data stream is not compressible, it shall transition to transparent mode, as described in 6.5.1.

In transparent mode, if the encoder determines that the data stream is compressible, it shall transition to compressed mode, as described in 6.5.2.

### 7.12 Dictionary reinitialization

The encoder dictionary is reinitialized by being set to the state defined by 7.5.1. If the data compression function is in compressed mode, the encoder also transfers the REINIT control code; and the decoder, upon the receipt of the REINIT, sets the decoder dictionary to the state defined by 7.5.2.

### 7.13 Expedited data transfer and FLUSH

Under certain circumstances, it may be necessary that string-matching or string-extension be terminated and any partially encoded data be transferred immediately. The specification of such conditions is outside the scope of this Recommendation, but an example would be if the error-control function were in idle condition. The control function shall issue a C-FLUSH_request primitive to the data compression function, and shall then transfer the remaining data in accordance with 7.9.

If the data compression function is in compressed mode, upon receipt of a C-FLUSH_request from the control function, the encoder shall:

a) terminate string-matching or string-extension procedures, as described in 6.3.1 and 6.3.2;

b) transfer the remaining code(s) in accordance with 6.6;

c) update the encoder dictionary as appropriate;

d) transfer the FLUSH control code;

e) if necessary, transfer enough 0 bits to establish octet-alignment.

The encoder dictionary is not reinitialized. The dictionary is updated, as appropriate, by appending the character following the flush.

Upon receipt of a FLUSH control code, the decoder shall establish octet alignment. The decoder dictionary is not reinitialized, and the receipt of the FLUSH control code does not affect the creation of new strings: the decoder shall process the code received after the FLUSH in continuity with the code received just before the FLUSH, in accordance with the decoder string creation process defined in 6.4.2.

If the data compression function is in transparent mode, upon receipt of a C-FLUSH_request from the control function, the encoder shall transfer all data input up to that point.

### 7.14 ESCAPE command sequence

A transparent mode command sequence shall consist of the ESCAPE, followed by one of the commands listed in Table 9 above, except ESCAPE.

The value of the ESCAPE is variable. At initialization of the data compression function, it is assigned the value 0. During transparent mode, if the current value of the ESCAPE is detected within the data stream from the DTE, the detected ESCAPE shall be transferred and followed immediately by the EID command. The current value of the ESCAPE is then changed by adding to it $51_{10}$, the addition performed modulo 256. Note that while in compressed or parameter mode, the current value of the ESCAPE is NOT modified if detected within the data stream from the DTE.

### 7.15 Action on detection of C-ERROR

The C-ERROR_indication is used to inform the control function that an exception (for example, a procedural error or loss of synchronization) has been detected by the data compression function. The control function shall take appropriate recovery action, including re-establishment of the error-corrected connection.

The following conditions recognized by the decoder result in the generation of a C-ERROR_indication primitive:

– receipt of a STEPUP control code that would cause the value of $C_2$ to exceed $N_1$;

– receipt of a STEPUP control code that would cause the value of $C_5$ to exceed 8; or

– receipt of a codeword greater than $C_1$.

## 8 Parameters

Table 10 defines the parameters negotiated between peer data compression functions. The "P" parameters are the values proposed by either side of the negotiation, and the "N" parameters are the values finally agreed. Table 11 defines all data compression operating parameters. (MSB indicates most significant byte, LSB indicates least significant byte.)

**Table 10/V.44 – Negotiation parameters for data compression function**

| Parameter | Identifier 8……1 | Length (octets) | Value 8……1 | Meaning | Range | Default | Agreed value |
|---|---|---|---|---|---|---|---|
| | 00000000 to 01000000 | | | Not Used | | | |
| $C_0$ | 01000001 | 00000001 | PM00000N | V.44 capability:<br>Value of N:<br>0 parameter negotiation via protocol (i.e. XID)<br>1 parameter negotiation after link establishment<br>Bits P and M are not used by modem connections: see Annex B. | | | |
| $P_0$ | 01000010 | 00000001 | 000000RT | Data compression request:<br>Value of RT:<br>00 neither direction<br>01 only in transmit direction<br>10 only in receive direction<br>11 both directions | 00-11 | 11 | |
| $P_{1T}$ | 01000011 | 00000010 | (MSB)<br>(LSB) | Proposed number of codewords (including control codes) in the transmit direction | 256-65535 | 1024 | $N_{2T}$ |
| $P_{1R}$ | 01000100 | 00000010 | (MSB)<br>(LSB) | Proposed number of codewords (including control codes) in the receive direction | 256-65535 | 1024 | $N_{2R}$ |
| $P_{2T}$ | 01000101 | 00000001 | | Proposed maximum string length for the transmit direction | 32-255 | 255 | $N_{7T}$ |
| $P_{2R}$ | 01000110 | 00000001 | | Proposed maximum string length for the receive direction | 32-255 | 255 | $N_{7R}$ |

**Table 10/V.44 – Negotiation parameters for data compression function *(concluded)***

| Parameter | Identifier 8……1 | Length (octets) | Value 8……1 | Meaning | Range | Default | Agreed value |
|---|---|---|---|---|---|---|---|
| $P_{3T}$ | 01000111 | 00000010 | (MSB)<br>(LSB) | Proposed length of history for the transmit direction | ≥512 | $3 \times P_{1T}$ | $N_{8T}$ |
| $P_{3R}$ | 01001000 | 00000010 | (MSB)<br>(LSB) | Proposed length of history for the receive direction | ≥512 | $3 \times P_{1R}$ | $N_{8R}$ |
| | 01001001<br>to<br>11111110 | | | Reserved for future use | | | |
| End | 11111111 | | | End of Parameters: exit parameter mode | | | |

**Table 11/V.44 – Operating parameters for data compression function**

| Parameter | Meaning | Value |
|---|---|---|
| $N_{1T}$, $N_{1R}$ | Maximum codeword size (in bits) | Derived from $N_{2T}$, $N_{2R}$ |
| $N_{2T}$, $N_{2R}$ | Maximum number of codewords (including control codes) | See Table 10 |
| $N_3$ | Input character size, in bits | 8 |
| $N_4$ | Number of characters in alphabet | 256 |
| $N_5$ | Number of control codes & first available codeword | 4 |
| $N_6$ | Reserved | |
| $N_{7T}$, $N_{7R}$ | Maximum string length | See Table 10 |
| $N_{8T}$, $N_{8R}$ | History size | See Table 10 |

Table 12 defines variables used in the operation of the data compression function. A separate set of these variables must be maintained by the encoder and decoder.

**Table 12/V.44 – Operating variables for data compression function**

| Parameter | Meaning |
|---|---|
| $C_1$ | Codeword value of next available entry of node-tree (in encoder) or string collection (in decoder) |
| $C_2$ | Current codeword size, in bits |
| $C_3$ | Threshold for changing the codeword size; equals 2 to the power $C_2$. Only used by the encoder. |
| $C_4$ | Current position in history |
| $C_5$ | Ordinal size, in bits |

ANNEX A

## XID information field for negotiating V.44 capability when used with V.42

Refer to 7.3 and 7.4.1 for XID negotiation.

**Table A.1/V.44 – XID User Data Subfield for negotiating V.44 data compression parameters**

| MSB: most significant byte<br>LSB: least significant byte | Bit<br>8.....1 | |
|---|---|---|
| Group identifier | 1 1 1 1 1 1 1 1 | User Data Subfield |
| Parameter identifier | 0 1 0 0 0 0 0 0 | ITU-T V.44 – Parameter Set Identifier |
| Parameter length | 0 0 0 0 0 0 1 1 | Length of string, in octets: 3 |
| Parameter value | 0 1 0 1 0 1 1 0<br>0 0 1 1 0 1 0 0<br>0 0 1 1 0 1 0 0 | V<br>4<br>4 |
| Parameter identifier | 0 1 0 0 0 0 0 1 | ITU-T V.44 capability ($C_0$) |
| Parameter length | 0 0 0 0 0 0 0 1 | Length of field, in octets: 1 |
| Parameter value | PM 0 0 0 0 0 N | Value for PM:<br>00 Neither packet method nor multi-packet method supported: for modem connections only.<br>01 Not valid.<br>10 Packet method supported.<br>11 Packet and multi-packet methods supported.<br>Bit N is not used in packet networks.<br>Value for N:<br>0 Parameter negotiation using XID exchange and parameters below.<br>1 Parameter negotiation after link establishment.<br>Bits P and M are ignored for modem connections. |
| Parameter identifier | 0 1 0 0 0 0 1 0 | ITU-T V.44 – Data Compression request ($P_0$) |
| Parameter length | 0 0 0 0 0 0 0 1 | Length of field, in octets: 1 |
| Parameter value | 0 0 0 0 0 0 RT | Request for direction of compression is signalled by value of RT:<br>0 neither direction<br>01 only in transmit direction<br>10 only in receive direction<br>11 both directions |
| Parameter identifier | 0 1 0 0 0 0 1 1 | ITU-T V.44 – Number of codewords for transmit direction ($P_{1T}$) |
| Parameter length | 0 0 0 0 0 0 1 0 | Length of field, in octets: 2 |
| Parameter value | NNNNNNNN (MSB)<br>NNNNNNNN (LSB) | Value of parameter $P_{1T}$ |

**Table A.1/V.44 – XID User Data Subfield for negotiating V.44 data compression parameters *(concluded)***

| | | |
|---|---|---|
| Parameter identifier | 0 1 0 0 0 1 0 0 | ITU-T V.44 – Number of codewords for receive direction ($P_{1R}$) |
| Parameter length | 0 0 0 0 0 0 1 0 | Length of field, in octets: 2 |
| Parameter value | NNNNNNNN (MSB)<br>NNNNNNNN (LSB) | Value of parameter $P_{1R}$ |
| Parameter identifier | 0 1 0 0 0 1 0 1 | ITU-T V.44 – Maximum string length for transmit direction ($P_{2T}$) |
| Parameter length | 0 0 0 0 0 0 0 1 | Length of field, in octets: 1 |
| Parameter value | NNNNNNNN | Value of parameter $P_{2T}$ |
| Parameter identifier | 0 1 0 0 0 1 1 0 | ITU-T V.44 – Maximum string length for receive direction ($P_{2R}$) |
| Parameter length | 0 0 0 0 0 0 0 1 | Length of field, in octets: 1 |
| Parameter value | NNNNNNNN | Value of parameter $P_{2R}$ |
| Parameter identifier | 0 1 0 0 0 1 1 1 | ITU-T V.44 – Length of history for transmit direction ($P_{3T}$) |
| Parameter length | 0 0 0 0 0 0 1 0 | Length of field, in octets: 2 |
| Parameter value | NNNNNNNN (MSB)<br>NNNNNNNN (LSB) | Value of parameter $P_{3T}$ |
| Parameter identifier | 0 1 0 0 1 0 0 0 | ITU-T V.44 – Length of history for receive direction ($P_{3R}$) |
| Parameter length | 0 0 0 0 0 0 1 0 | Length of field, in octets: 2 |
| Parameter value | NNNNNNNN (MSB)<br>NNNNNNNN (LSB) | Value of parameter $P_{3R}$ |

ANNEX B

## Operation of V.44 in Packet Networks

The operation of V.44 in packet networks differs from its operation in modems, primarily because a network terminal has knowledge of packet boundaries, including the delineation between header and data. A network terminal can identify a complete transport-layer packet and process it as a unit, whereas a modem processes a continuous stream of unframed data.

This annex describes two methods of compressing packetized data: the packet method, in which each packet is processed separately; and a multi-packet method, in which several packets, or portions of packets, are processed as continuation. The multi-packet method requires the guaranteed delivery of all packets. The primary method of operation of V.44, described in the main body of this Recommendation, is referred to as the "stream method".

In this annex, the term "packet" is used to refer to a transport-layer packet, IP packet, frame, block, N-PDU, etc. The term "packet network" is used to describe any network that handles packets; examples are IP, Frame Relay, etc. The term "segment" is used to describe a portion of a packet that has been segmented to allow for transmission over a link.

Important features of data compression in packet networks include:

- Packets are compressed and transmitted as a unit. Thus, a flush is performed, using a FLUSH control code, after the last data byte of the packet has passed into the encoder to complete compression and transmit the entire packet.
- Negotiation of data compression parameters is not required, as the default V.44 parameter set for packet networks is defined in B.1. If necessary, negotiation of data compression and related parameters can be done outside of V.44, prior to data transfer. This negotiation can use XID or another protocol message exchange.
- If V.44 is negotiated between data compression peers (i.e. using XID), the P and M bits of the V.44 capability parameter are used to indicate support of V.44 packet method and V.44 multi-packet method. A peer supporting V.44 multi-packet method shall also support packet method. If it supports both but the other peer only supports packet method, packet method is selected. See Table A.1.
- Transparent mode is not supported in either packet or multi-packet method. An indicator is used to signal the other end as to whether or not the packet is compressed. Thus, the smaller of the original packet and the result of the data compression operation is transmitted. The indicator is either a bit (or other indication) in the packet or protocol header; or the ETM control code (preceded by its code prefix of "1") in the first byte of the packet, followed by the original data, octet-aligned. (In this context, the ETM code is simply an indicator, and does not imply the behaviour of 6.5.)

NOTE – In packet networks, execution time may be improved if, at the time of stepping up the codeword size (e.g. from 8 to 9 bits), a check is made as to whether the data compression has been successful up to that point. If not, the encoding may be aborted and the original packet transmitted.

## B.1 Packet method operation of V.44

The packet method is the default method of V.44 operation in packet networks. Using this method, negotiation between data compression peers is not needed.

### B.1.1 General description

Packet method operation must be used in networks in which packets or segments are transmitted using unacknowledged mode, without guaranteed delivery. In such a case, since a previous packet may not have been received, the packet method does not rely on information in previous packets for compression and decompression of any individual packet. The packet method may also be preferred even if delivery is guaranteed, because of its simplicity compared to the multi-packet method. The packet method allows a network hub or packet switch supporting numerous links to use a single instance of V.44 encoder and decoder dictionaries and context to support all links, since each packet is processed separately.

If V.44 is negotiated between data compression peers, support of packet method is indicated in the V.44 capability parameter by setting the P bit to "1" and the M bit to "0".

Specific differences between V.44 packet method and V.44 stream method include:

- Packets are compressed and transmitted as a unit; the encoder transfers a FLUSH control to end the packet.
- Between packets, the encoder and decoder dictionaries are reinitialized. Reinitialization is performed by the control function or is built into the algorithm itself. The encoder does not transfer a REINIT control code.
- Encoder and decoder histories are not required. The uncompressed data in the packet being encoded serve as the encoder history; and the decompressed data placed into an output buffer by the decoder serve as the decoder history.

– When the last codeword has been created (the node-tree filled), the string-matching and string-extension activities continue as normal, but no additional codewords are created. Thus, string-extension lengths are transferred, but the string extensions are not used to define additional string-segments in the node-tree or new strings in the string collection.

– Default values of the data compression parameters are listed in B.1.2. Other values can be set by negotiation.

### B.1.2 Default values of data compression parameters for packet method

If no parameter negotiation is done between devices within a packet network, V.44 operates in packet method with the default parameter values as shown below:

– $P_0$ – 11, both directions;

– $P_{1T}$ – 1525 codewords;

– $P_{1R}$ – 1525 codewords;

– $P_{2T}$ – 255 maximum string length;

– $P_{2R}$ – 255 maximum string length;

– $P_{3T}$ – not applicable;

– $P_{3R}$ – not applicable.

## B.2 Multi-packet method of operation of V.44

At the cost of additional complexity and memory, the multi-packet method can provide better compression performance than packet method. It can only be used in a packet network in which packets or segments are transmitted using acknowledged mode with guaranteed delivery. Multi-packet method operation places a burden of complexity and memory on a network hub or packet switch that supports numerous point-to-point links with terminals or other user devices, since each point-to-point link using multi-packet method requires an individual instance of encoder and decoder dictionaries and context.

The multi-packet method of operation requires a superset of the procedures and structures used in the packet method. Devices which support multi-packet method must also support operation in the packet method.

### B.2.1 General description

If V.44 is negotiated between data compression peers, support of the multi-packet method is indicated in the V.44 capability parameter. If one of the peers indicates multi-packet method and the other indicates packet method, packet method is used.

Specific differences between V.44 multi-packet method and V.44 stream method are minor, and include:

– Packets are compressed and transmitted as a unit; the encoder transfers a FLUSH control to end the packet.

– The history size must be larger than the maximum packet length. The history length is 3072 by default, unless otherwise negotiated or pre-set. If the dictionary sizes are changed from default values through negotiation, the history size should be changed accordingly.

– Both the encoder and decoder shall reinitialize the dictionary after processing a packet in which compression was unsuccessful (i.e. a packet in which the original data input was smaller than the result of the compression operation) and the original packet has been therefore transmitted.

– Prior to or during encoding, the data to be compressed are copied from the packet into the next available positions of the encoder history, up to the limit of the maximum size of the history. If the history becomes full in the middle of a packet, after the character in the last position of the history is encoded, the dictionary is reinitialized, a REINIT control code is transferred, and the remainder of the data from the packet are copied to the first positions of the history, and compressed.
– When the last codeword has been created (the node-tree is full), the encoder dictionary is reinitialized, a REINIT control code is transferred (even in the middle of a packet) and unprocessed characters in the history are moved to the first positions of the history.

### B.2.2 Default values of data compression parameters for multi-packet method

In multi-packet method operation, the default values are the same as used in stream method. Refer to Table 10.

Note that, for proper operation of multi-packet method, the number of usable codewords must be greater than the maximum length packet expected. Thus, for a network with a maximum length packet of 1518 bytes, the number of codewords must be at least 1522, to account for the 4 reserved codewords, and should be considerably greater for best compression.

## APPENDIX I

## Notes on Implementation

The following notes provide information on the implementation of the data compression algorithm and on the selection of parameters. By varying the number of codewords and the history length, the V.44 algorithm can be scaled to take advantage of available memory.

### I.1 Selection of $N_2$: the total number of codewords

The dictionary size is equal to $N_2$ (assuming that entries are provided for the reserved control codes). In general selecting a large value for $N_2$ provides better compression at the cost of more memory for dictionaries. The selection of a large value for $N_2$ means that the number of strings available is increased, but also that the value of $N_1$ is correspondingly increased. In some instances the gain in performance obtained from the selection of a larger dictionary may be offset by the larger codeword size required. Indeed, for certain types of data, better performance may be obtained by using a smaller dictionary. In general, the value 2048 for $N_2$ provides good compression performance across a wide range of data types.

To accommodate hardware platforms with limited memory, $N_2$ may be different for the two directions. This could allow, for instance, a larger number of codewords in the server-to-client direction than in the client-to-server direction, to improve the compression in the web-download direction while minimizing the total memory required for the encoder and decoder.

The value of $N_2$ need not be a power of 2. Since the dictionary is reinitialized as soon as the node-tree entry corresponding to codeword ($N_2 - 1$) is created, in V.44 there is no permanent penalty for stepping up to a larger value of $N_1$ to obtain additional codewords. This is an advantage over V.42 *bis*, which rarely reinitializes the dictionary, and thus maintains $N_1$ at its maximum value, once it has been reached.

## I.2 Selection of $N_7$: maximum string length

For the data compression algorithm described in this Recommendation, unlike that of V.42 *bis*, larger maximum string length provides much better compression. Therefore, it is suggested that the maximum value, 255, be used.

## I.3 Selection of $N_8$: data structures and length of history

The history contains all characters input to the encoder, or decoded by the decoder, since the most recent reinitialization of the dictionary. To optimize performance, the dictionary should be reinitialized as infrequently as possible. The dictionary is reinitialized whenever either the history or the node-tree is full; however, for data that are more compressible, performance is generally better if the node-tree fills up before the history does. For example, for a compression ratio of ~ 10:1, a node-tree of 2044 (corresponding to a value of 2048 for $N_{2T}$) codewords performs best with a history of about 15 000 characters (although good performance can be obtained with 4096).

For satisfactory operation of V.44, it is suggested that the history length ($N_8$) always be greater than or equal to $2 \times N_2$. However, to accommodate hardware platforms with limited memory, $N_8$ may differ for the two directions. This could allow, for example, a larger history in the server-to-client direction than in the client-to-server direction, to improve compression in the downstream (web page downloading) direction within a fixed total memory requirement for the encoder and decoder. For instance, instead of using the default value of $3 \times N_2$ for both directions one might use $4 \times N_2$ for the server-to-client direction and only $2 \times N_2$ in the client-to-server direction.

Another method to maximize the utilization of the encoder's memory resources is to aggregate the memory allocated for the history and the node-tree, and to allow the history to grow over its nominal limit into un-utilized memory set aside for the node-tree's codewords. Then if the data are more compressible than usual, and the node-tree grows more slowly than usual, the encoder will have a longer run before reinitialization, because the history can make use of the remaining memory planned for the node-tree.

The encoder must not use more history than the decoder has negotiated, to avoid over-run. However, in some useful applications (e.g. decoder in the client, encoder in the server), the decoder's memory resources may be much greater than the encoder's. If the encoder negotiates a size for the decoder's history larger than it allocates for its own history, it has margin to grow past its own allocation; as long as it reinitializes before the decoder's history is full.

As an example, we assume that the server has 6000 bytes budgeted for the encoder's history (6000 characters) and 14 308 bytes budgeted for the encoder's node-tree (2 044 codewords, with 7 bytes per codeword entry: each codeword entry uses one byte for string-segment length and two bytes each for history index, down-index, and side-index); the aggregate memory for the encoder is 20 308 bytes. However, in negotiation with the client, the server's encoder proposes a larger value for the history than 6000, such as 15 000. The client may accept this number or agree to a smaller number; nonetheless the agreed value, denoted as $N_{8T}$ by the encoder, will likely be greater than the 6000 budgeted by the encoder for history. Let us assume that $N_{8T}$ is 15 000.

In the server's encoder, the history and the node-tree will share a common memory array such that:

- the first history position is at the *first* byte of the memory array;
- the current history index is incremented for each new character placed into the history;
- the structure for each node-tree entry (all data fields for a given codeword) are in 7 consecutive bytes of memory;
- the structure for the first codeword (4) is located in the *last* 7 bytes of the memory array;

– the structure for the last codeword (2047) is located in the 7 bytes immediately *after* the last byte of the history; and

– the next available codeword-structure pointer is decremented for each codeword created.

In operation, the history is filled from the first byte of the array and increments. Node-tree entries are created from the last 7 bytes of the array and decrement. The encoder reinitializes the dictionary whenever any of the following occurs:

a) the last codeword is created ($C_1 = N_{2T} – 1 = 2047$);

b) the current history position reaches the $N_{8T}$ ($C_4 = N_{8T} = 15\ 000$); or

c) the current history position is within $N_{7T}$ of the next available codeword structure.

Condition a) is normal, indicating that the encoder's node-tree is full. Condition b) prevents the *decoder's* history from being over-run. Condition c) ensures that the encoder's history does not over-write codeword entries, while it is expanding into the memory set aside for the codeword structures.

The result is that if the data compress in the range of 10:1, the node-tree grows slowly compared to the history, and the history can expand into the memory originally allocated for the node-tree, potentially increasing the usable history by up to 9000 (= 15 000 – 6000) bytes.

## I.4 Efficient compression of Unicode data

Unicode is an international standard 16-bit character coding system designed to support the interchange, processing and display of many modern written languages. It consists of numeric representations for nearly all the characters found in the principal written languages of Europe, the Americas, the Middle East, India, Africa, Asia, and Pacifica. It is an implementation of the ISO/IEC 10646 standard, restricted to 2-octets, and is also known as UCS-2.

The data compression algorithm defined in this Recommendation does not specifically address 16-bit characters, but will efficiently compress Unicode data as 8-bit characters. For applications in which the protocol layers above V.42 are aware that UCS-2 is a substantial portion of the traffic, it is recommended that the compression scheme specified in Unicode Technical Report #6 be performed on the data prior to passing it to V.44, as this may improve overall performance.

## I.5 Applicability of transparent mode

Transparent mode is required for operation over a modem connection. In packet networks, the transparent mode is not needed because the frame header is heeded. In all other cases, there would be some basis for using it, as has been ceded.

## I.6 Calculation of compression performance

The calculation of compression performance may be expressed as the ratio of the number of characters received by the encoder, to the number of octets transferred to the control function. The count of characters and octets should be set to zero on initialization of the data compression function, and also between tests.

## I.7 Differences between V.44 and V.42 *bis*

Following is a list of the major differences between the data compression algorithm described in this Recommendation and that described in ITU-T V.42 *bis*:

a) for most file types encountered in Internet browsing, it achieves superior compression ratios;

b) the encoder and the decoder both use a history buffer;

c) the decoder dictionary structure is different from the encoder dictionary structure;

d) the dictionary sizes, etc., for the transmit and receive directions are negotiated independently;

e) the nodes in the encoder dictionary may define string-segments, instead of single characters;

f) the dictionary nodes use indices to the position in history of a character, instead of the characters themselves;

g) reinitializes the dictionary when it becomes full, instead of recovering codewords;

h) does not reserve the first 256 codewords for all possible 8-bit characters;

i) uses a string-extension procedure to rapidly encode a string encountered for the second time;

j) transfers codes for characters and string-extension lengths, as well as for strings and control;

k) can use immediately all codewords created by the encoder, even the most recent one, a codeword that the decoder may not yet have created;

l) does not modify the value of ESCAPE when detected in compressed mode;

m) does not update decoder dictionary when in transparent mode;

n) reinitializes dictionaries in the transition from transparent to compressed mode;

o) the best response to a parameter $P_0$ proposal is inversion: for example, the complementary response to a proposal, from entity A, for compression in the *transmit direction* (of entity A) *only*, is an agreement, from entity B, for compression in the *receive direction* (of entity B) *only*.

## APPENDIX II

## Illustration of operation of V.44 algorithm

The examples in this appendix assume an encoder implementation that appends characters and updates the node-tree before the next character has actually been received. Thus, at certain steps in these examples, the contents of the node-tree may differ from those for an implementation that appends characters and updates the node-tree only after the next character has been received.

### II.1 Compression and decompression of "ABCDEXABCDEYABCDE $FF_H$ AC"

This clause is intended to illustrate the operation of the V.44 algorithm in compression mode, by showing the processing of a specific example. The processing is demonstrated by showing, at different stages:

i) the new characters, as input to the encoder;

ii) the analysis of the input stream by the algorithm, for encoding;

iii) the revised dictionary structure of the encoder;

iv) the information transferred by the encoder, and the method by which this is encoded;

v) the analysis of the compressed data by the algorithm, for decoding; and

vi) the revised dictionary structure of the decoder.

In this example, it is assumed that the dictionary structure has just been reinitialized, so there is no previous history, and $C_5$ is set to 7 and $C_2$ is set to 6. The characters are input one-by-one, but this presentation will show stages, consolidating some stages where the processing is obvious. In the root array, roots before the "A" are not shown.

Uncompressed characters are placed into the encoder history as they arrive; the processing by the encoder produces binary codes, which are transferred; after the decoder has received the compressed data, the decoded characters are placed into the decoder history.

Stage 0: no previous data.

The state of the encoder is:

Encoder History:

| Position | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | | | | | | | | | | | | | | | | | | | |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | .. | $FF_H$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | - | - | - | - | - | - | - | - | - | - |

Node-Tree:

| Codeword | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| First-character Position | | | | | | | | | | | | |
| Segment Length | | | | | | | | | | | | |
| Down-Index | | | | | | | | | | | | |
| Side-Index | | | | | | | | | | | | |

The state of the decoder is:

String Collection:

| Codeword | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Last-character Position | | | | | | | | | | | | |
| String Length | | | | | | | | | | | | |

Decoder History:

| Position | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | | | | | | | | | | | | | | | | | | | |

Stage 1:

New characters: A

Analysis by Encoder: There is no down-index for "A" in the root array, so it is transferred as an ordinal. The first available node-tree entry (codeword 4) is used to create a string-segment: it has first-character position 1 (the position following "A"), segment length 1 (it is a 1-character segment), and has no valid down-index or side-index: essentially appending the next character, sight-unseen, creating a 2-character string. The root array is also updated with a down-index for "A", pointing to this codeword 4.

Encoder History:

| Position | 0 | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | | | | | | | | | | | | | | | | | | |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | .. | $FF_H$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | 4 | - | - | - | - | - | - | - | - | - |

Node-Tree:

| Codeword | 4 | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| First-character Position | 1 | | | | | | | | | | | | |
| Segment Length | 1 | | | | | | | | | | | | |
| Down-Index | - | | | | | | | | | | | | |
| Side-Index | - | | | | | | | | | | | | |

Transferred: ordinal for "A": $41_H$ = "1000001"

Analysis by Decoder: The ordinal for "A" is expanded to the 8-bit character and placed into the history. At dictionary reinitialization, the decoder is initialized to not create a codeword when the very first ordinal is received.

String Collection:

| Codeword | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Last-character Position | | | | | | | | | | | | |
| String Length | | | | | | | | | | | | |

Decoder History:

| Position | 0 | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | | | | | | | | | | | | | | | | | | |

Stage 2:

New characters: B

Analysis by Encoder: There is no down-index for "B" in the root array, so it is transferred as an ordinal. The next available node-tree entry (codeword 5) is used to create a string-segment: it has first-character position 2 (the position following "B"), segment length 1 (it is a 1-character segment), and has no valid down-index or side-index. The root dictionary is also updated with a down-index for "B", pointing to codeword 5.

Encoder History:

| Position | 0 | 1 | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | | | | | | | | | | | | | | | | | |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | .. | $FF_H$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | 4 | 5 | - | - | - | - | - | - | - | - |

Node-Tree:

| Codeword | 4 | 5 | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| First-character Position | 1 | 2 | | | | | | | | | | |
| Segment Length | 1 | 1 | | | | | | | | | | |
| Down-Index | - | - | | | | | | | | | | |
| Side-Index | - | - | | | | | | | | | | |

Transferred: ordinal for "B": $42_H$ = "1000010"

Analysis by Decoder: The character "B" is placed into the history. The string collection is updated by creating a string ending with "B" (at position 1), of length 2 (representing the string "AB") and using the first available codeword, 4.

String Collection:

| Codeword | 4 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Last-character Position | 1 | | | | | | | | | | | |
| String Length | 2 | | | | | | | | | | | |

Decoder History:

| Position | 0 | 1 | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | | | | | | | | | | | | | | | | | |

Stages 3, 4, 5, 6:

New characters: C D E X

Analysis by Encoder: For these characters, the processing is the same as for "B": each of them is transferred as an ordinal, and the node-tree and root array are updated accordingly.

New Data: C D E X

Encoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | C | D | E | X | | | | | | | | | | | | | |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | .. | $FF_H$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | 4 | 5 | 6 | 7 | 8 | - | 9 | - | - | - |

Node-Tree:

| Codeword | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| First-character Position | 1 | 2 | 3 | 4 | 5 | 6 | | | | | | |
| Segment Length | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | |
| Down-Index | - | - | - | - | - | | | | | | | |
| Side-Index | - | - | - | - | - | | | | | | | |

Transferred: ordinals for "C", "D", "E", "X": $43_H$ $44_H$ $45_H$ $58_H$ = "1000011" "1000100" "1000101" "1011000"

Analysis by Decoder: The characters "C", "D", "E", "X" are individually placed into the history. 2-character strings ending with C, D, E and X are created in the string collection.

String Collection:

| Codeword | 4 | 5 | 6 | 7 | 8 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Last-character Position | 1 | 2 | 3 | 4 | 5 | | | | | | | |
| String Length | 2 | 2 | 2 | 2 | 2 | | | | | | | |

Decoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | C | D | E | X | | | | | | | | | | | | | |

Stage 7:

New characters: A B

Analysis by Encoder: When the "A" is input, the encoder follows the down-index from root "A" to codeword 4 in the node-tree, corresponding to string-segment "B". This matches the next character "B"; since codeword 4 has no valid down-index, this is the longest string match. So codeword 4 is transferred, and the encoder initiates the string-extension procedure. A new string-segment is not created at this time.

Encoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | C | D | E | X | A | B | | | | | | | | | | | |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | .. | $FF_H$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | 4 | 5 | 6 | 7 | 8 | - | 9 | - | - | - |

Node-Tree:

| Codeword | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| First-character Position | 1 | 2 | 3 | 4 | 5 | 6 | | | | | | |
| Segment Length | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | |
| Down-Index | - | - | - | - | - | - | | | | | | |
| Side-Index | - | - | - | - | - | - | | | | | | |

Transferred: codeword 4: "000100"

Analysis by Decoder: The decoder accesses codeword 4 in the string collection. It copies the string "AB" from position 0 in the history to the next available positions in the history. It also creates the 2-character string ending with the first character of the string represented by codeword 4 (i.e. the "A"), using codeword 9.

String Collection:

| Codeword | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Last-character Position | 1 | 2 | 3 | 4 | 5 | 6 | | | | | | |
| String Length | 2 | 2 | 2 | 2 | 2 | 2 | | | | | | |

Decoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | C | D | E | X | A | B | | | | | | | | | | | |

Stage 8a:

New characters: C

Analysis by Encoder: The string-extension procedure finds that the new character "C" matches the character "C" in the history that immediately follows the string-segment "B" of codeword 4. Since the total length of the string for codeword 4 and the 1-character extension is less than the maximum string length, the string-extension procedure continues with the next character.

Encoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | C | D | E | X | A | B | C | | | | | | | | | | | |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | .. | $FF_H$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | 4 | 5 | 6 | 7 | 8 | - | 9 | - | - | - |

Node-Tree:

| Codeword | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| First-character Position | 1 | 2 | 3 | 4 | 5 | 6 | | | | | | |
| Segment Length | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | |
| Down-Index | - | - | - | - | - | - | | | | | | |
| Side-Index | - | - | - | - | - | - | | | | | | |

Transferred: nothing transferred at this substage.

Analysis by Decoder: nothing received at this substage.

String Collection:

| Codeword | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Last-character Position | 1 | 2 | 3 | 4 | 5 | 6 | | | | | | |
| String Length | 2 | 2 | 2 | 2 | 2 | 2 | | | | | | |

Decoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | C | D | E | X | A | B | | | | | | | | | | | | |

Stage 8b:

New characters: D E Y

Analysis by Encoder: In the string-extension procedure, the encoder finds that the next characters keep matching the history, until the "Y". A string-extension length of 3 is transferred, representing the "CDE" following the original "B" of codeword 4. The encoder creates a new string-segment, to extend the longest string match, with length 3 and a history position of 8, the position of the first extension character; it uses the next available codeword, 10. This ends the string-extension procedure: the encoder does NOT additionally append a 1-character string-segment.

The unmatched character "Y" becomes the root of a new possible string. However, since it has no valid down-index, it is transferred as an ordinal; codeword 11 is created to append the 1-character string-segment for the character following the "Y", and the down-index for the root entry of "Y" is updated with codeword 11.

Encoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | C | D | E | X | A | B | C | D | E | Y | | | | | | | | |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | .. | $FF_H$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | 4 | 5 | 6 | 7 | 8 | - | 9 | 11 | - | - |

Node-Tree:

| Codeword | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| First-character Position | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 12 | | | | |
| Segment Length | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 1 | | | | |
| Down-Index | 10 | - | - | - | - | - | - | - | | | | |
| Side-Index | - | - | - | - | - | - | - | - | | | | |

Transferred: string-extension length 3: "0" "10"; ordinal for "Y": $59_H$ = "1011001"

Analysis by Decoder: The decoder extends the previous codeword's string (codeword 4) by 3: it copies the 3 characters in the history that immediately follow the string represented by codeword 4 (using the last-character position plus 1 as the starting point) to the next available positions in the history. It then creates the string ending in "E" and assigns it codeword 10: its length is 5 ((the length of string for codeword 4) + (the length of string-extension)) = (2 + 3) = 5), and the last-character position is set to 10, the position of the last character copied to history.

The character Y is also placed into the history. A 2-character string is NOT created.

String Collection:

| Codeword | 4 | 5 | 6 | 7 | 8 | 9 | 10 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Last-character Position | 1 | 2 | 3 | 4 | 5 | 6 | 10 | | | | | |
| String Length | 2 | 2 | 2 | 2 | 2 | 2 | 5 | | | | | |

Decoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | C | D | E | X | A | B | C | D | E | Y | | | | | | | | |

Stage 9:

New characters: A B C D E

Analysis by Encoder: When the "A" is input, the encoder follows the down-index from root "A" to codeword 4 in the node-tree, corresponding to string-segment "B". This matches the next character, so the encoder continues the string-matching procedure: it follows the down-pointer to codeword 10, corresponding to string-segment "CDE". The next characters match completely this string-segment. Since codeword 10 has no valid down-index, this is the longest string match. So codeword 10 is transferred, and the encoder initiates the string-extension procedure.

Encoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | C | D | E | X | A | B | C | D | E | Y | A | B | C | D | E | | | |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | .. | $FF_H$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | 4 | 5 | 6 | 7 | 8 | - | 9 | 11 | - | - |

Node-Tree:

| Codeword | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| First-character Position | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 12 | | | | |
| Segment Length | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 1 | | | | |
| Down-Index | 10 | - | - | - | - | - | - | - | | | | |
| Side-Index | - | - | - | - | - | - | - | - | | | | |

Transferred: codeword 10: "001010"

Analysis by Decoder: The decoder accesses codeword 10 in the string collection. It copies this string of 5 characters into the next available positions in the history. It also appends "A" to "Y", and creates the 2-character string, using codeword 11.

String Collection:

| Codeword | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Last-character Position | 1 | 2 | 3 | 4 | 5 | 6 | 10 | 12 | | | | |
| String Length | 2 | 2 | 2 | 2 | 2 | 2 | 5 | 2 | | | | |

Decoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | C | D | E | X | A | B | C | D | E | Y | A | B | C | D | E | | | |

Stage 10:

New characters: $FF_H$

Analysis by Encoder: In the string-extension procedure, the encoder finds that the next character "$FF_H$" does not match the history, and the string-extension procedure is terminated. A new string-segment of length 1 (the "$FF_H$") is appended to the end of "ABCDE", and assigned codeword 12. The down-index for codeword 10 is set to codeword 12.

Then the "$FF_H$" is used as the root for a possible new string match: since it has no valid down-index, it is transferred as an ordinal. A 1-character string-segment is created, codeword 13, appending the next character to the "$FF_H$", and the down-index of the root entry of "$FF_H$" is set to codeword 13.

(Because $FF_H = 255_{10} > 127_{10}$, the ordinal requires 8 bits to transfer; $C_5$ is set to 8, and the STEPUP control code is transferred just before the ordinal.)

Encoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | C | D | E | X | A | B | C | D | E | Y | A | B | C | D | E | $FF_H$ | | |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | .. | $FF_H$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | 4 | 5 | 6 | 7 | 8 | - | 9 | 11 | - | 13 |

Node-Tree:

| Codeword | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| First-character Position | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 12 | 17 | 18 | | |
| Segment Length | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | | |
| Down-Index | 10 | - | - | - | - | - | 12 | - | - | - | | |
| Side-Index | - | - | - | - | - | - | - | - | - | - | | |

Transferred: STEPUP ("000010"), ordinal for "$FF_H$": $FF_H$ = "11111111"

Analysis by Decoder: Upon receipt of the STEPUP control code, the decoder sets $C_5$ to 8. The character "$FF_H$" is placed into the history. The decoder assigns codeword 12 to the string of length 6 ending with "$FF_H$".

String Collection:

| Codeword | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Last-character Position | 1 | 2 | 3 | 4 | 5 | 6 | 10 | 12 | 17 | | | |
| String Length | 2 | 2 | 2 | 2 | 2 | 2 | 5 | 2 | 6 | | | |

Decoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | C | D | E | X | A | B | C | D | E | Y | A | B | C | D | E | $FF_H$ | | |

Stage 11:

New characters: A C and *C-FLUSH_request*

Analysis by Encoder: When the "A" is input, the encoder follows the down-index from root "A" to codeword 4 in the node-tree, corresponding to string-segment "B". This does NOT match the next character "C"; since there is no side-index for another option for a match on "C", the "A" is transferred as an ordinal. The encoder creates a 1-character string-segment (codeword 14) appending the "C" to the "A"; codeword 14 is set as the side-index for codeword 4 (the earlier "AB" string-segment).

The "C" becomes the root for a new possible string match: however, since the encoder receives a FLUSH command from the control function, it must complete any string matching in progress. In this instance, it transfers the "C" as an ordinal, followed by the FLUSH control code. As usual, the encoder creates a 1-character string-segment appending the next character to the "C", assigned to codeword 15.

Encoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | C | D | E | X | A | B | C | D | E | Y | A | B | C | D | E | $FF_H$ | A | C |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | .. | $FF_H$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | 4 | 5 | 6 | 7 | 8 | - | 9 | 11 | - | 13 |

Node-Tree:

| Codeword | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| First-character Position | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 12 | 17 | 18 | 19 | 20 |
| Segment Length | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 1 |
| Down-Index | 10 | - | - | - | - | - | 12 | - | - | - | - | |
| Side-Index | 14 | - | - | - | - | - | - | - | - | - | - | |

Transferred: ordinals for "A" "C", control code FLUSH, padding of 0 bits: "01000001" "01000011" "000001" *0000*

Analysis by Decoder: The character "A" is placed into the history and the 2-character string ending with "A" is created and assigned to codeword 13. The character "C" is placed into history and the 2-character string ending with "C" is created and assigned to codeword 14. Receipt of the FLUSH control code indicates an end to encoded data for now. The next significant bit will be at the first bit-position of the next octet. The decoder does not create codeword 15 until the code after the FLUSH is received.

String Collection:

| Codeword | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Last-character Position | 1 | 2 | 3 | 4 | 5 | 6 | 10 | 12 | 17 | 18 | 19 | |
| String Length | 2 | 2 | 2 | 2 | 2 | 2 | 5 | 2 | 6 | 2 | 2 | |

Decoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | A | B | C | D | E | X | A | B | C | D | E | Y | A | B | C | D | E | $FF_H$ | A | C |

At this point, the encoder and decoder dictionaries are synchronized, except for codeword 15.

Figure II.1 shows the node-tree after step 11.

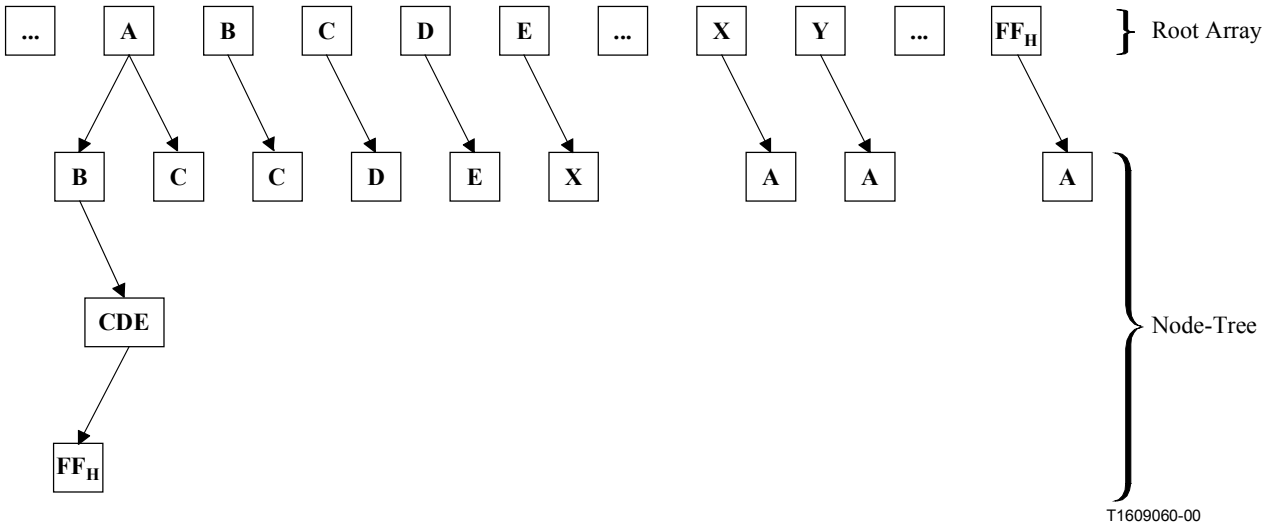


**Figure II.1/V.44 – Tree-structures after step 11**

Table II.1 shows the octet mapping for the example of II.1. Refer to Tables 3 and 5 for code prefix and string-extension length coding.

**Table II.1/V.44 – Octet mapping for example of II.1**

| Bit No. | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | Octet |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 0 | 0 | 0 | 0 | 1 | ***0*** | 1 |
| | 1 | 0 | 0 | 0 | 0 | 1 | 0 | ***0*** | 2 |
| | 1 | 0 | 0 | 0 | 0 | 1 | 1 | ***0*** | 3 |
| | 1 | 0 | 0 | 0 | 1 | 0 | 0 | ***0*** | 4 |
| | 1 | 0 | 0 | 0 | 1 | 0 | 1 | ***0*** | 5 |
| | 1 | 0 | 1 | 1 | 0 | 0 | 0 | ***0*** | 6 |
| | ***0*** | 0 | 0 | 0 | 1 | 0 | 0 | ***1*** | 7 |
| | 0 | 0 | 1 | ***0*** | 1 | 0 | 0 | ***1*** | 8 |
| | 0 | 1 | 0 | ***1*** | 1 | 0 | 1 | 1 | 9 |
| | 0 | 0 | 1 | 0 | ***1*** | 0 | 0 | 1 | 10 |
| | 1 | 1 | 1 | 1 | 1 | ***0*** | 0 | 0 | 11 |
| | 0 | 0 | 0 | 1 | ***0*** | 1 | 1 | 1 | 12 |
| | 0 | 1 | 1 | ***0*** | 0 | 1 | 0 | 0 | 13 |
| | 0 | 1 | ***1*** | 0 | 1 | 0 | 0 | 0 | 14 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 15 |
| Code prefix bits are in ***bold italics***, padding bits are underlined. | | | | | | | | | |

## II.2 Compression & decompression of "CCCCCCCCCCX"

This example in this clause is intended to illustrate a particular aspect of operation of the V.44 algorithm, which allows it to compress sequences of repeated characters very quickly. This aspect requires the decoder to interpret a codeword that it has not yet created.

In the general operation of the algorithm, codewords are continually being created on both sides, although in the encoder they correspond specifically with string-segments and to the decoder they correspond with the complete string that ends with the encoder's corresponding string-segment (in the context of the tree-structure). As demonstrated in the example of II.1, the encoder can be one step ahead of the decoder in creating codewords. In some special circumstances, the encoder will transfer a codeword that is, to the decoder, exactly the next available codeword (codeword equal to $C_1$).

The decoder handles this situation according to the rules of 6.4.1, in particular 3) and 4).

The processing of the example is demonstrated by showing, at different stages:

i) the new characters, as input to the encoder;

ii) the analysis of the input stream by the algorithm, for encoding;

iii) the revised dictionary structure of the encoder;

iv) the information transferred by the encoder, and the method by which this is encoded;

v) the analysis of the compressed data by the algorithm, for decoding; and

vi) the revised dictionary structure of the decoder.

In this example, it is assumed that the dictionary structure has just been reinitialized, so there is no previous history. The characters are input one-by-one, but this presentation will show stages, skipping some steps where the processing is obvious.

Uncompressed characters are placed into the encoder history as they arrive; the processing by the encoder produces binary codes, which are transferred; after the decoder has received the compressed data, the decoded characters are placed into the decoder history.

Stage 0: no previous data.

The state of the encoder is:

Encoder History:

| Position | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | | | | | | | | | | | | | | | | | | | |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | Z | .. |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | - | - | - | - | - | - | - | - | - | - |

Node-Tree:

| Codeword | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| First-character Position | | | | | | | | | | | | |
| Segment Length | | | | | | | | | | | | |
| Down-Index | | | | | | | | | | | | |
| Side-Index | | | | | | | | | | | | |

The state of the decoder is:

String Collection:

| Codeword | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Last-character Position | | | | | | | | | | | | |
| String Length | | | | | | | | | | | | |

Decoder History:

| Position | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | | | | | | | | | | | | | | | | | | | | |

Stage 1: New characters: C

Analysis by Encoder: There is no down-index for "C" in the root array, so it is transferred as an ordinal. The first available node-tree entry (codeword 4) is used to create a string-segment: it has first-character position 1 (the position following "C"), segment length 1 (it is a 1-character string-segment), and has no valid down-index or side-index: essentially appending the next character, sight-unseen, to create a 2-character string. The root array is also updated with a down-index for "C", pointing to codeword 4.

Encoder History:

| Position | 0 | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | C | | | | | | | | | | | | | | | | | | | |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | Z | .. |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | - | - | 4 | - | - | - | - | - | - | - |

Node-Tree:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Codeword | 4 | | | | | | | | | | | |
| First-character Position | 1 | | | | | | | | | | | |
| Segment Length | 1 | | | | | | | | | | | |
| Down-Index | - | | | | | | | | | | | |
| Side-Index | - | | | | | | | | | | | |

Transferred: ordinal "C"

Analysis by Decoder: The character "C" is placed into the history. At dictionary reinitialization, the decoder is initialized to not create a codeword when the very first character is received.

String Collection:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Codeword | | | | | | | | | | | | |
| Last-character Position | | | | | | | | | | | | |
| String Length | | | | | | | | | | | | |

Decoder History:

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Position | 0 | | | | | | | | | | | | | | | | | | | |
| Character | C | | | | | | | | | | | | | | | | | | | |

Stage 2: New characters: C C

Analysis by Encoder: When the first new C is received, the encoder follows the down-index from root "C" to codeword 4 in the node-tree, corresponding to string-segment "C" (i.e. the C at position 1). This matches the second new C; and since codeword 4 has no valid down-index, this is the longest string match. So codeword 4 is transferred, and the encoder initiates the string-extension procedure. A new string-segment is not created at this time.

Encoder History:

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Position | 0 | 1 | 2 | | | | | | | | | | | | | | | | | |
| Character | C | C | C | | | | | | | | | | | | | | | | | |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | Z | .. |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | - | - | 4 | - | - | - | - | - | - | - |

Node-Tree:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Codeword | 4 | | | | | | | | | | | |
| First-character Position | 1 | | | | | | | | | | | |
| Segment Length | 1 | | | | | | | | | | | |
| Down-Index | - | | | | | | | | | | | |
| Side-Index | - | | | | | | | | | | | |

Transferred: codeword 4

Analysis by Decoder: In accordance with 6.4.1, rule 4): since codeword 4 is the next codeword to be created ($C_1 = 4$), the decoder copies the previous 1-character string (the "C" from position 0) into the history (at position 1); and then places the first character of the copied string (the "C" from position 1) into the history (at position 2).

In accordance with Table 2: the first character of codeword 4's string ("CC") is appended to the previous character (the C at position 0) to create codeword 4: this is a 2-character string with last-character at position 1.

String Collection:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Codeword | 4 | | | | | | | | | | | |
| Last-character Position | 1 | | | | | | | | | | | |
| String Length | 2 | | | | | | | | | | | |

Decoder History:

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Position | 0 | 1 | 2 | | | | | | | | | | | | | | | | |
| Character | C | C | C | | | | | | | | | | | | | | | | |

Stage 3: New characters: C

Analysis by Encoder: The string-extension procedure finds that the new character C does match the character following the string-segment (for codeword 4) in history (the C at position 2). The string-extension procedure therefore continues (since the total length of the string corresponding to codeword 4 plus this 1-character extension is less than the maximum string length).

Encoder History:

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Position | 0 | 1 | 2 | 3 | | | | | | | | | | | | | | | |
| Character | C | C | C | C | | | | | | | | | | | | | | | |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | Z | .. |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | - | - | 4 | - | - | - | - | - | - | - |

Node-Tree:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Codeword | 4 | | | | | | | | | | | |
| First-character Position | 1 | | | | | | | | | | | |
| Segment Length | 1 | | | | | | | | | | | |
| Down-Index | - | | | | | | | | | | | |
| Side-Index | - | | | | | | | | | | | |

Transferred: nothing transferred at this substage.

Analysis by Decoder: nothing received at this substage.

String Collection:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Codeword | 4 | | | | | | | | | | | |
| Last-character Position | 1 | | | | | | | | | | | |
| String Length | 2 | | | | | | | | | | | |

Decoder History:

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Position | 0 | 1 | 2 | | | | | | | | | | | | | | | | |
| Character | C | C | C | | | | | | | | | | | | | | | | |

Stage 4: New characters: C C C C C C

Analysis by Encoder: In the string-extension procedure, each new character is successively compared with the characters in the history: in this case, since the extension begins with the character being placed at position 3 (which was compared with the character at position 2), the first new character is the one at position 4 (to be compared with that at position 3); and so on. Since the new characters consist of consecutive Cs, each matching the one before it, the string-extension continues to succeed through the 6th new C. The total string length ((length of codeword 4's string) + (number of extension characters so far) = 2 + 7 = 9) is still less than the maximum, so the string-extension procedure continues.

Encoder History:

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | | | | |
| Character | C | C | C | C | C | C | C | C | C | C | | | | | | | | | |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | Z | .. |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | - | - | 4 | - | - | - | - | - | - | - |

Node-Tree:

| Codeword | 4 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| First-character Position | 1 | | | | | | | | | | | |
| Segment Length | 1 | | | | | | | | | | | |
| Down-Index | - | | | | | | | | | | | |
| Side-Index | - | | | | | | | | | | | |

Transferred: nothing transferred at this substage.

Analysis by Decoder: nothing received at this substage.

String Collection:

| Codeword | 4 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Last-character Position | 1 | | | | | | | | | | | |
| String Length | 2 | | | | | | | | | | | |

Decoder History:

| Position | 0 | 1 | 2 | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | C | C | C | | | | | | | | | | | | | | | | |

Stage 5: New characters: X

Analysis by Encoder: The X terminates string-extension matching: the encoder finds that it does not match the character in position 9 of the history (the 10th C). The encoder creates a new string-segment (codeword 5) to extend the longest string match with an extension length of 7 (representing the "CCCCCCC" following the longest string match, codeword 4); the first-character position is 3. This ends the string-extension procedure; and the encoder does not append a 1-character string-segment.

The unmatched character "X" becomes the root of a new possible string. However, since it has no valid down-index, it is transferred as an ordinal; codeword 6 is created to append the 1-character string-segment for this character following the "X"; and the down-index for the root entry of "X" is updated with codeword 6.

Encoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | C | C | C | C | C | C | C | C | C | C | X | | | | | | | | | |

Root Array:

| Root | A | B | C | D | E | .. | X | Y | Z | .. |
|---|---|---|---|---|---|---|---|---|---|---|
| Down-Index | - | - | 4 | - | - | - | 6 | - | - | - |

Node-Tree:

| Codeword | 4 | 5 | 6 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| First-character Position | 1 | 3 | 11 | | | | | | | | | |
| Segment Length | 1 | 7 | 1 | | | | | | | | | |
| Down-Index | 5 | | | | | | | | | | | |
| Side-Index | - | | | | | | | | | | | |

Transferred: string-extension length 7, ordinal for "X"

Analysis by Decoder: The decoder extends the previous codeword's string by 7: it copies the 7 characters in the history that immediately follow the string represented by codeword 4, using the last-character position (position 1) plus 1 as its starting point. This is done one character at a time: the starting point for copying from is position 2 (to be copied to the first available position, which is position 3); the next character to be copied is the one at position 3 (to be copied to position 4); and so on. The decoder then creates the string ending with the 10th C, using codeword 5: its length is 9 ((length of string for codeword 4) + (length of string extension) = (2 + 7) = 9); and the position of the last character of the string is the position of the last character copied to history, 9.

The character "X" is subsequently placed into the history at position 10. A 2-character string is not created.

String Collection:

| Codeword | 4 | 5 | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Last-character Position | 1 | 9 | | | | | | | | | | |
| String Length | 2 | 9 | | | | | | | | | | |

Decoder History:

| Position | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character | C | C | C | C | C | C | C | C | C | C | X | | | | | | | | | |

## Bibliography


– ITU-T V.14 (1993), *Transmission of start-stop characters over synchronous bearer channels*.

– ITU-T V.76 (1996), *Generic multiplexer using V.42 LAPM-based procedures*.

– ISO/IEC 10646-1:2000, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane*.

– Unicode Technical Report No. 6, *A Standard Compression Scheme for Unicode*, Revision 3.1.

## SERIES OF ITU-T RECOMMENDATIONS

| | |
|---|---|
| Series A | Organization of the work of ITU-T |
| Series B | Means of expression: definitions, symbols, classification |
| Series C | General telecommunication statistics |
| Series D | General tariff principles |
| Series E | Overall network operation, telephone service, service operation and human factors |
| Series F | Non-telephone telecommunication services |
| Series G | Transmission systems and media, digital systems and networks |
| Series H | Audiovisual and multimedia systems |
| Series I | Integrated services digital network |
| Series J | Transmission of television, sound programme and other multimedia signals |
| Series K | Protection against interference |
| Series L | Construction, installation and protection of cables and other elements of outside plant |
| Series M | TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits |
| Series N | Maintenance: international sound programme and television transmission circuits |
| Series O | Specifications of measuring equipment |
| Series P | Telephone transmission quality, telephone installations, local line networks |
| Series Q | Switching and signalling |
| Series R | Telegraph transmission |
| Series S | Telegraph services terminal equipment |
| Series T | Terminals for telematic services |
| Series U | Telegraph switching |
| **Series V** | **Data communication over the telephone network** |
| Series X | Data networks and open system communications |
| Series Y | Global information infrastructure and Internet protocol aspects |
| Series Z | Languages and general software aspects for telecommunication systems |

Geneva, 2001