ATNP/WG2/21
IP/582

AERONAUTICAL TELECOMMUNICATIONS NETWORK PANEL

WG2/21

limerick, Irlande

11-14 July, 2000

Issue on the Deflate compressed PDU format

Prepared by Stéphane Tamalet

(France)

SUMMARY

The implementation of the draft 3rd edition enhancements to the mobile SNDCF (use of pre-stored dictionaries (ICS3_13) and maintenance of the Deflate history windows (ICS3_14)) led the ProATN A/G BIS team to deeply investigate the mechanisms of the deflate compression. This work allowed discovering that the ProATN A/G BIS did not comply with all the baseline SARPs requirements on the Deflate, and will not be able to interoperate with baseline SARPs compliant routeurs if deflate compression is used. This document describes the problem existing in the current ProATN A/G BIS implementation

STNA has the feeling that the ProATN A/G BIS may not be the only router concerned by the non-conformance issue exposed in this document. We would like to ask the other ATN router developers to tell us whether the implementation of the deflate compression in their software presents the same non-conformity, and to discuss on the possible ways to deal with this issue.

This document is also proposed as an Information Paper to the next WG2 meeting. In the event where all currently existing implementations are defective, the WG2 is invited to discuss on the possibility to deal with this issue via a modification to the SARPS.

TABLE OF CONTENTS

1 INTRODUCTION 2

2 DEFLATE COMPRESSED DATA FORMAT 2

3 ERROR MADE BY THE CURRENT PROATN A/G BIS IMPLEMENTATION 4

3.1 Description of the problem 4

3.2 Discussion of potential solutions 7

3.3 Conclusion 9

4 REQUEST FOR COMMENTS 10

1 Introduction

The implementation of the draft 3rd edition enhancements to the mobile SNDCF (use of pre-stored dictionaries (ICS3_13) and maintenance of the Deflate history windows (ICS3_14)) led the ProATN A/G BIS team to deeply investigate the mechanisms of the deflate compression. This work allowed discovering that the ProATN A/G BIS did not comply with all the baseline SARPs requirements on the Deflate, and will not be able to interoperate with SARPs compliant routeurs if deflate compression is used.

This document describes the problem existing in the current ProATN A/G BIS implementation, It is organized as follows:

- In order to ease understanding the issues, section 2 includes a brief reminder of the Deflate compressed data format.
- Section 3 explains the error made by the current ProATN A/G BIS implementation.
- Section 4 includes a request for comments to other ATN router developers and to the ATNP WG2 experts.

2 Deflate compressed data format

A Deflate compressed packet exchanged over a mobile subnetwork comprises:

- a variable length, octet aligned, encoded data stream
- followed by a two-octet Frame Check Sum (FCS)

The Encoded Data Stream consists of a series of "Deflate Data Blocks" of arbitrary length, as illustrated by the figure below. Note that the last Deflate Data block may be followed by padding zeros until the next octet boundary is reached, before the FCS.

Each Deflate Data block comprises a 3-bit header (H), that indicates the compression type applied to the data in the block, and a stream of self-delimited compressed data. This is illustrated by the figure below:

Note that a Deflate data block does not necessarily occupy an integral number of bytes. As a consequence, the header bits of a Deflate data block do not necessarily begin on a byte boundary.

There are 3 different types of Deflate Data blocks:

1. The Uncompressed Data blocks. In block of such a type, the data is not compressed at all. The data are simply copied from the original uncompressed PDU.

   The format of an uncompressed data block is represented herebelow:

H 3 bits	Padding 0	LEN	NLEN	LEN bytes of literal data

   → The 3-bit header is right padded with zeroes to the next octet boundary.

   → LEN (2 octets) gives the number of octets of literal data in the block (the number of octets that have been copied from the original uncompressed PDU)

   → NLEN (2 octets) is the ones complement to the value of the LEN field

   → The end of the block comprises the LEN bytes that have been copied from the original uncompressed PDU

   An uncompressed Data Block always ends at a byte boundary.

2. The blocks compressed with fixed Huffman codes.

   The format of such a block is represented in the next figure:

H	Sequence of fixed Huffman codes	End of Block code 7 bits to 0

   The block consists of the 3-bit header, followed by sequences of fixed (pre-determined) Huffman codes representing either literal bytes, or <length, backward distance pairs>, and terminated by the "End-of-Block' Code (7 bits to 0).

   Thanks to the 'end of block code', the block is self delimiting without requiring an explicit length indicator.

   It must be noted that a block of such a type does not necessarily ends at byte boundary.

3. The blocks compressed with dynamic Huffman codes

   The format of such a block is represented in the next figure:

H	HLIT 5 bits	HDIST 5 bits	HCLEN 4 bits	Code lengths alphabet	Literal/length alphabet	Distance alphabet	Sequence of dynamic Huffman codes	End of Block code

   Following the 3-bit header, the second to the seventh field are used to convey the set of dynamically determined Huffman code Tables. This is followed by sequences of dynamic Huffman codes representing either literal bytes, or <length, backward distance pairs>, and terminated by the "End-of-Block' Code. The block is self-delimiting without requiring an explicit length indicator.

   It must be noted that a block of such a type does not necessarily ends at byte boundary.

3 Error made by the current ProATN A/G BIS implementation

3.1 Description of the problem

The ProATN A/G BIS bases its support of the deflate compression on the use of the public 'zlib' code. The 'zlib' is the reference implementation of the deflate compression. It is a well proven, but tightly written and complex piece of code.

The ProATN A/G BIS developers have tried to avoid, as much as possible, modifying the standard zlib code. This allows simplifying the upgrade of the zlib in the router when a new version becomes available. This also avoids introducing software errors in the standard compression/decompression procedures. Therefore, whenever this is possible, the zlib is used via its standard software interface, and internal procedures are left unmodified.

The 'zlib' library provides in-memory deflate compression and decompression functions. For compression, the application must provide the deflate() function with an input buffer containing the data to be compressed and with an output buffer where the compressed data is to be stored. For decompression, the application must provide the inflate() function with an input buffer containing the compressed data and with an output buffer where the uncompressed data is to be stored.

Compression/decompression can be done in a single step if the input (resp. output) are large enough, or can be done by repeated calls. In the later case, the zlib consider the new input (resp output) buffer as the logical continuation of the preceding input buffer (i.e. the first bit of the new input (resp. output) buffer is considered to follow directly the last bit of the preceding input buffer). In that case, the zlib user must provide more input and/or consume the output (providing more output space) before each call.

By default, the zlib compression procedure directly copies the compressed data in the output buffer. However, there are 2 exceptions:

→ When the output buffer is full, the compression procedure can accept further input data to process, however, the resulting compressed data is kept in an internal buffer of the zlib, and not directly accessible to the zlib user. In that case, in order for the user to get the compressed data, the compression function must be recalled providing more output space (e.g. providing a new (void) output buffer).

→ When the resulting compressed data does not end at an octet boundary, the compression procedure stores in the output buffer all integral bytes of compressed data but keeps the last bits of the last octet in an internal bit buffer. These last bits will be concatenated with the next bits of compressed data, when the compression procedure is recalled to process further input data. We will see that these last bits can be flushed out of the zlib internal buffer if there is no further input data to process.

The compression procedure manages the creation of Deflate Data block in a way totally opaque to the zlib user. The zlib compression procedure may decide at any time to terminate a deflate data block and to open a new one, depending on the best way to achieve optimal compression. There, it must be noted that the zlib compression procedure does not necessarily terminate a Deflate Data block, when it has finished to process an input buffer. By default, the compression procedure keeps the current output deflate data block open, and waits for further input data to compress in the context of this block. Hence, a Deflate Data Block can span over several subsequent output buffers.

Considering the above, when the zlib is used to compress a CLNP PDU the following occurs by default:

→ If the compressed data does not fit in an integral number of octet, the last residual bits remain stored in the zlib internal bit buffer and are not accessible to the zlib user, until a new PDU is given to the compressor.

→ In the compressed data, the last Deflate Data Block is not terminated.

This could be a problem, to build a compressed data PDU that is compliant to the ATN SARPs. Hopefully, the zlib compression procedure provides the user with options that allows forcing the termination of the current compression block, and flushing that compression block to the output buffer so that the user can get all the compressed data available so far.

The zlib option currently used by the ProATN router (and possibly the TAR) to terminate the block and flush the data is the so-called Z_PARTIAL_FLUSH. This option appeared to be appropriate after having tested the exchange of compressed PDUs between 2 routeurs: it was observed that PDUs compressed at one end of the connection were always successfully uncompressed by the peer routeur.

However, we discovered recently that the compressed PDUs resulting from the use of the Z_PARTIAL_FLUSH option are not always structured according to the format specified in the SARPs.

It has been observed that with the Z_PARTIAL_FLUSH option,

1. The compression procedure terminates the current data block (for instance by appending an 'End Of Block' Symbol at the end of the compressed data),

2. Next, the procedure appends one or two empty[1] Deflate Data Block(s) of the type "compressed with fixed Huffman code". Such empty Data blocks are ten-bits long and have the following form:

H 3bits (010)	End of Block code 7 bits to 0

3. Finally, the procedure copies the compressed data into the output buffer, with the exception of the last trailing bits remaining stored in the internal zlib bit buffer if the compressed data does not fit in an integral number of octets.

The consequences of this behaviour are the following:

1. **The compressed PDUs terminate by an empty data block compressed with fixed Huffman code, the 'End of Block' code of which is truncated at the last byte boundary.**

2. **The remaining part (if any) of this truncated 'End of Block' Symbol will be pre-pended to the next compressed PDU.**

The resulting format of a compressed PDU. exchanged between 2 ProATN routers is typically as illustrated by the next figure (assuming that the PDU has been compressed into one single Deflate Data block compressed with fixed Huffman code (this is generally the case)).

---

[1] An empty Deflate data block will be totally transparent to the decompressor: its presence has no effect on the result of decompression procedure. (A compressed PDU containing such a block will give the same result once uncompressed as the one that would be obtained from the compressed PDU without that block).

Remaining bits of the last End of-block Symbol of the previous compressed PDU (if any)	H	Compressed data (Sequences of Huffman codes)	End of Block Symbol for that block	H	First part of the truncated End of Block code of that empty block	FCS

<u>These compressed PDUs are therefore not compliant to the format that is specified in the Sub-Volume V</u>. They do not start systematically with a 3-bit Header. Compliant routers will likely be unable to decode such packets, and interoperate with the ProATN A/G BIS.

It is anecdotal to note that the same "error" was made originally when the Deflate compression method was evaluated and validated before incorporation in the SubVolume V. The following figure is an extract of the figure contained in the "Data Link Compression Evaluation Report" that was presented at the ATNP/WG2/12 (IP407).

Frame	**Time**	**Source**	**M Bit**	**Bytes**	**Data**
1	10:22:21.6709	DTE	0	34	6a5262646061906360107167506f740d0d627475766400112ec18c0cc79898220002
2	10:22:31.6725	DTE	0	6	a889a00a8000
3	10:22:41.6749	DTE	0	4	22ac0220
4	10:22:51.6764	DTE	0	5	8008ab0008
5	10:23:01.6796	DTE	0	5	20c22a0002
6	10:23:11.6812	DTE	0	5	88b00a8000
7	10:23:21.6828	DTE	0	4	22ac0220
8	10:23:31.6844	DTE	0	5	8008ab0008
9	10:23:41.6876	DTE	0	5	20c22a0002
10	10:23:51.6892	DTE	0	5	88b00a8000
11	10:24:01.6907	DTE	0	4	22ac0220

Figure 3-1 ISH PDU Convergence for Deflate

On this figure, the compressed data are always compressed within one single Deflate Data Block of the type ’compressed with fixed Huffman codes". It can be observed that these compressed PDUs do not all begin with the expected 3bit header set to 010 that normally indicates a PDU compressed with fixed Huffman code. (Note: the first transmitted bit is the least significant bit of the first byte). The following table indicates for each of the above frame, the effective location of the Deflate Data Block header (Header bits are represented in bold).

Frame number	First octet of the frame (hexadecimal representation)	First octet (reverse binary representation - least significant bit on the left)	Comment
1.	6A	**010**10110	The 3bit Header is at the front of the PDU
2.	A8	00**010**101	There are 2 zeroes appended before the 3bits Header
3.	22	**010**00100	The 3bit Header is at the front of the PDU
4.	80	00000**010**	There are 5 zeroes appended before the 3bits Header
5.	20	0000**010**0	There are 4 zeroes appended before the 3bits Header
6.	88	0**010**0010	There is one zero appended before the 3bits Header

According to the SARPs, the true value of the compressed PDUs resulting from this validation exercize should have been:

Frame	Bytes	Data
1	33	6a5262646061906360107167506f740d0d627475766400112ec18c0cc798982200 (not sure about the last 00)
2	5	6a22a80200
3	4	22ac0200
4	4	22ac0200
5	4	22ac0200
6	4	22ac0200
7	4	22ac0200
8	4	22ac0200
9	4	22ac0200
10	4	22ac0200
11	4	22ac0200

3.2 Discussion of potential solutions

A potential solution to this problem was suggested by M. Duncan Roe in a mail received on the WG2_SDM mailing list, last year. (This mail is attached in annex of this document).

In his mail, Duncan recommends the use of another flush option provided by the zlib; the so-called Z_SYNC_FLUSH.

The Z_SYNC_FLUSH option also allows forcing the termination of the current Deflate Data Block and flushing that compression block to the output buffer. However the way it does so differs from the method of the Z_PARTIAL_FLUSH.

Before flushing the compressed data to the output buffer, the Z_SYNC_FLUSH terminates the current deflate data block, and appends an extra empty "uncompressed" deflate data block.

An empty "uncompressed" block has the following form:

H (000)	Padding 0s	LEN (00 00)	NLEN (FF FF)

The benefit of this method is that the insertion of empty "uncompressed" data block has the double particularity to:

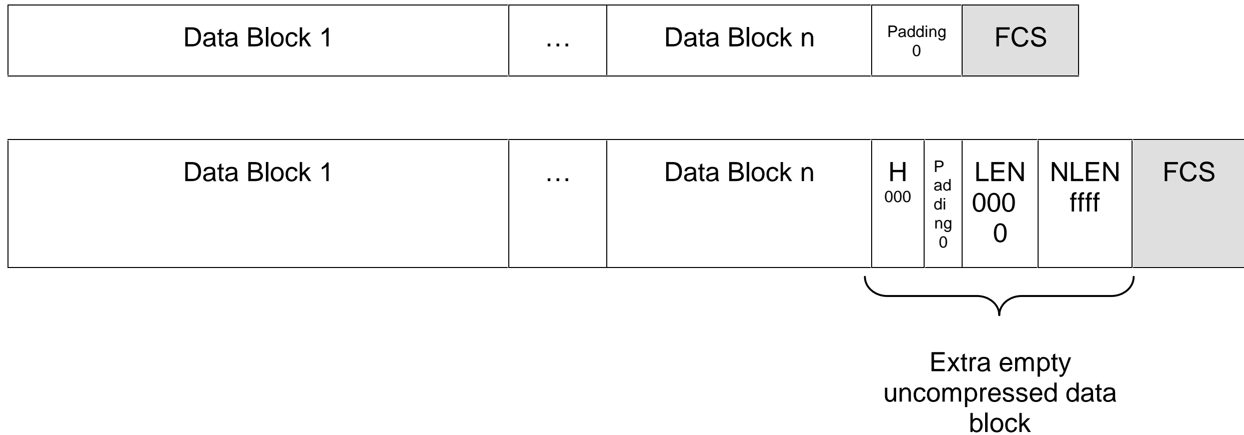
1. Be totally transparent to the decompressor: its presence has no effect on the result of decompression procedure. (A compressed PDU containing such a block will give the same result once uncompressed as the one that would be obtained from the compressed PDU without that block).

2. Re-align the compressed data on an octet boundary: this is because the 3-bit Header of such a block is followed by padding zeroes until the next byte boundary, and because the LEN and NLEN fields have a fixed length of 2 bytes.

With a Z_SYNC_FLUSH, the zlib compression function does not need to keep trailing bits in its internal bit buffer (the bit buffer is cleared) and, consequently, extra (remaining) bits are never prepended at the head of the subsequent compressed packets.

Hence, with the Z_SYNC_FLUSH, the compressed PDUs are always formatted in way compliant to the ATN SARPs (they always start at a Deflate Data block Boundary).

The downside of this option is that 4 or 5 extra octets are systematically appended to the end of the compressed data. This is shown on the next figure which represents the expected format of a SARPs-compliant compressed PDU beside the format of the same compressed PDU obtained as a result of the use of the Z_SYNC_FLUSH option:



Then, the following solution was investigated as a method to obtain a compressed PDU formatted in the way expected by the SARPS (and without modifying the zlib internal procedures):

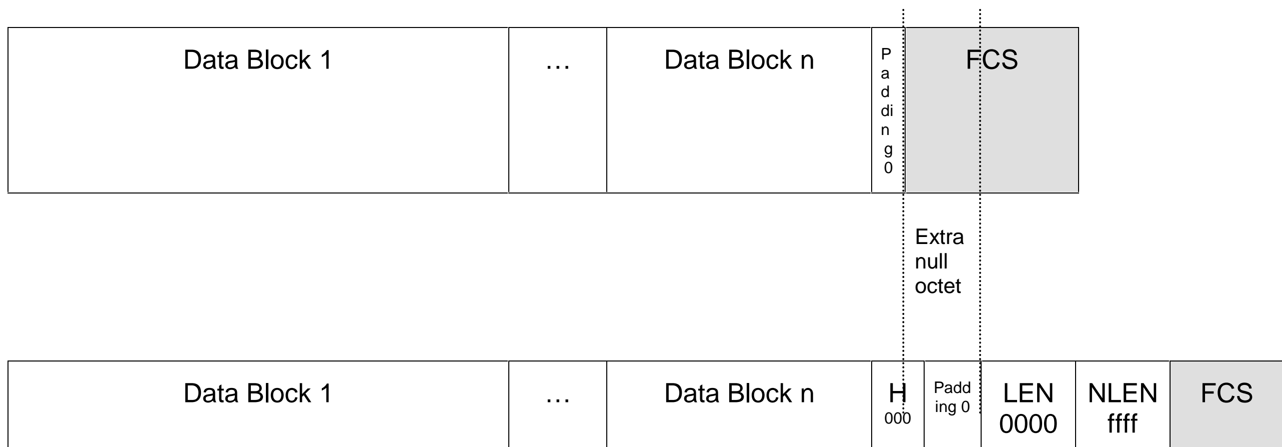
1. Produce a compressed PDU using the Z_SYNC_FLUSH option

2. Remove from that PDU the 4 bytes "0000FFFF" corresponding to LEN and NLEN fields of the Extra empty uncompressed data block.

The reverse procedure would be applied to decompress a received PDU (append the 4 bytes "0000FFFF" to the PDU and submit the PDU to the zlib decompression function).

This works in most of the cases, but there are exceptions which invalidate this method.

The exception cases are when it remains less than 3 bits between the end of the compressed data (last bit of Deflate Data Block n in the figure) and the next octet boundary (i.e. when 0, 1 or 2 padding zeroes would be sufficient). In that cases, there is not enough space to insert the 3 -bit Header of an empty uncompressed data block). Then, if an extra empty uncompressed data block is appended to the compressed data, an extra null octets is inserted between the end of the compressed data end the LEN and NLEN fields to be removed. This is illustrated by the next figure:



Hence, the suggested solution is not SARPs compliant: the use of the Z-SYNC-FLUSH option followed by the removal of the LEN and NLEN field may produce a compressed PDU that is one (null) octet longer that what is expected.

Mr Duncan Roe, in his mail, suggest another solution which consists, at the compression stage, in:

1. Producing a compressed PDU using the Z_SYNC_FLUSH option
2. Removing from that PDU the 4 bytes "0000FFFF" corresponding to LEN and NLEN fields of the Extra empty uncompressed data block, and finally,
3. Removing all the last trailing null octets.

According to Duncan, it is possible to inverse the process at the decompression stage, by feeding the decompressor with the compressed PDU and with additional null octets until we get some indication that the decompressor is resynchonized and waits for the value of the LEN and NLEN field.

This solution gives a better compression ratio than the one obtained with the preceding solution. However, like the preceding solution, it does not allow producing compressed PDUs which format is compliant to the SARPs. Furthermore, this solution is not possible without modifying some parts of the zlib code and this is what we tried to avoid.

3.3 Conclusion

After having considered the potential solutions, we came to the conclusion that implementing the deflate compression in the ProATN Router in a way totally compliant to the SARPs is not possible without making modifications to the standard zlib code.

The ProATN A/G BIS team decided to make the best of a bad job, and is trying to modify the zlib code so that to make the ProATN A/G BIS compliant to the SARPs.

As far as the compression procedure is concerned, the ProATN team quickly succeeded in upgrading the zlib code with a new flush option (the Z_ATN_FLUSH option) that allows, as requested by the SARPs,:

1. Terminating the current Deflate Data Block,
2. Padding this block with zero bits until the next octet boundary is reached, and
3. Flushing out all the resulting compressed data, in a way that leaves the compressor ready to process a new PDU.

On the other hand, the upgrade of the decompression procedure poses a lot of problems. After some days of hard work on the zlib code, (and at the time this document is produced), the ProATN team did not yet succeed in implementing an ATN specific decompression procedure that could inflate a SARPs compliant deflate compressed PDU and then leave the decompressor ready to process a new PDU.

The ProATN team thinks that the upgrade is feasible. However, it is questioning whether such a work is worth the effort, considering that some slight changes in the SARPs could outweigh the complexity for implementations when modifying the standard zlib code. Also, the ProATN team is very concerned about the fact that it will be difficult to ascertain that no error has been introduced in the standard code, and hence, that the upgraded compression and decompression procedures works correctly in all cases.

4 Request for comments

STNA has the feeling that the ProATN A/G BIS may not be the only router concerned by the non-conformance issue exposed in this document. We would like to ask the other ATN router developers to tell us whether the implementation of the deflate compression in their software presents the same non-conformity, and to discuss on the possible ways to deal with this issue.

This document is also proposed as an Information Paper to the next WG2 meeting. In the event where all currently existing implementations are defective, the WG2 is invited to discuss on the possibility to deal with this issue via a modification to the SARPS.

ANNEX

(Mail received from M. Duncan Roe)

Date: 6/21/99 5:55 PM
Sender: Duncan Roe <dunc@dimstar.cvsi.com>
To: Klaus-Peter Graf <klaus.graf@unibw-muenchen.de>
cc: Tony Whyman <whyman@mwassocs.demon.co.uk>; atnp_wg2 <atnp_wg2@cenatoulouse.dgac.fr>
bcc: Ronnie Jones
Priority: Normal
Subject: Air/Ground Data Compression

Hi Klaus-Peter,

This mail doesn’t contain a formal PDR: I seek your feedback as
to whether that would be appropriate.

As part of enhancing the EURATN Airborne ATN router package for
AIRSYS ATM here in Australia, I implemented DEFLATE compression exactly as
per the ICS SARPs including Section 5.7.6.5.4.2.5.3. I was not aware of
P1DR 98100004, so implemented "trailing zero-octet removal when the last
compression block uses static trees".

I did have to modify zlib, but only to provide the caller with
extra information during compression & decompression. On building, zlib
still passed all its self-tests.

Now that the SARPs have been changed, if they are to be changed
again then I suggest they be changed to a specify simpler method, which
gives slightly better compression even than the original SARPs and involves
fewer changes to zlib. In fact the amended zlib builds identical binaries
to the original, on systems where that is ever possible.

I suggest a change to the SARPs for two reasons:-

1. The method I detail below gives better compression with little
code complication, and

2. The current ATN implementations are using Z_PARTIAL_FLUSH
which is deprecated as at zlib-1.1.3, see zlib.h therein:-

```
 #define Z_PARTIAL_FLUSH 1 /* will be removed, use Z_SYNC_FLUSH
instead */
```

You are welcome to my patch file for zlib-1.1.3 and some test
programs. Please mail me for a copy: if there is enough interest I’ll
submit it to the ATN FTP
archive.

```
 Simpler Method - compression
 ======= ====== = ===========
```

The compressor always calls deflate() with key Z_SYNC_FLUSH. This
terminates the block with an empty Non-Compressed block. This consists of a
3-bit header "000", zero padding to the next octet boundary, then 4 octets
of value
0x0 0x0 0xff 0xff.

After calling deflate(), the compressor discards the last 4
octets (perhaps after checking that their value is as above). It then
discards all trailing zero octets. (In testing, I have seen a maximumof 3
of these). The NPDU is now ready to send (checksums excepted).

```
(The original SARPs specified to only delete a trailing null from
a static trees compression block, but we don't now care what kind of block
it was nor how many nulls we remove).

 Simpler Method - decompression
 ======= ====== = =============

The decompressor needs to know how many zero octets to re-append.
The way it does this is to use include files to spy on the internal state
of zlib's inflate(). It appends zero octets until the block's bit buffer is
empty and it is waiting for the first length octet of a Non-Compressed
block.
I.e.:-

 z_stream ustrm;                   /* Decompression stream */

 while(((struct internal_state*)ustrm.state)->blocks->bitk||
   ((struct internal_state*)ustrm.state)->blocks->mode!=LENS) {
              --- present a null octet to inflate() ---
              }

Next, give inflate() the 4-octet {0x0,0x0,0xff,0xff} buffer. This
Will synchronise its state with that of the compresssor - the next
item will be a block header, octet-aligned.

 Nature of zlib changes
 ====== == ==== =======

The principal change is to move a bunch of declarations out of
inflate.c into the new header file infprivate.h. To avoid name clashes,
DONE BAD are renamed IDONE & IBAD in infprivate.h and inflate.c.

infutil.h is changed not to declare a dummy internal_state
structure if infprivate.h has been #include'd.

Makefile & Makefile.in are changed to reflect the new dependency
of inflate.c on infprivate.h.

 Evaluation Revisited
 ========== =========

I repeated Tony Whyman's ISH test from his original "Data Link
Compression Evaluation Report" dated 23rd June 1997.

Every message after the first 2 is 3 octets exactly (rather than
Cycling 4,5,5,5) and they are all identical:-

 13:55:59dr@bart:~/zlib-1.1.3/atn_patch$ ./wrish
 6a52626460619063601071675O6f740d0d627475766400112ec18c0cc7989822
 6a22a802
 22ac02
 22ac02
 22ac02
 22ac02
 22ac02
 22ac02
 22ac02
 22ac02

Compare this with how it was:-

 15:14:20dr@bart:~/zlib-1.1.3/atn_patch$ ./wrish
```

```
 6a5262646061906360107167506f740d0d627475766400112ec18c0cc79898220002
 a889a00a8000
 22ac0220
 8008ab0008
 20c22a0002
 88b00a8000
 22ac0220
 8008ab0008
 20c22a0002
 88b00a8000
```


Cheers ... Duncan.