Internet Research Task Force (IRTF)
Request for Comments: 6256
Category: Informational
ISSN: 2070-1721

W. Eddy
MTI Systems
E. Davies
Folly Consulting

May 2011

Using Self-Delimiting Numeric Values in Protocols

Abstract

Self-Delimiting Numeric Values (SDNVs) have recently been introduced as a field type in proposed Delay-Tolerant Networking protocols. SDNVs encode an arbitrary-length non-negative integer or arbitrary-length bitstring with minimum overhead. They are intended to provide protocol flexibility without sacrificing economy and to assist in future-proofing protocols under development. This document describes formats and algorithms for SDNV encoding and decoding, along with notes on implementation and usage. This document is a product of the Delay-Tolerant Networking Research Group and has been reviewed by that group. No objections to its publication as an RFC were raised.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Research Task Force (IRTF). The IRTF publishes the results of Internet-related research and development activities. These results might not be suitable for deployment. This RFC represents the consensus of the Delay-Tolerant Networking Research Group of the Internet Research Task Force (IRTF). Documents approved for publication by the IRSG are not a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6256.

Table of Contents

1. Introduction

This document is a product of the Internet Research Task Force (IRTF) Delay-Tolerant Networking (DTN) Research Group (DTNRG). The document has received review and support within the DTNRG, as discussed in the Acknowledgements section of this document.

This document begins by describing the drawbacks of using fixed-width protocol fields. It then provides some background on the Self-Delimiting Numeric Values (SDNVs) proposed for use in DTN protocols, and motivates their potential applicability in other networking protocols. The DTNRG has created SDNVs to meet the challenges it attempts to solve, and it has been noted that SDNVs closely resemble certain constructs within ASN.1 and even older ITU protocols, so the problems are not new or unique to DTN. SDNVs focus strictly on numeric values or bitstrings, while other mechanisms have been developed for encoding more complex data structures, such as ASN.1

encoding rules and Haverty's Message Services Data Transmission Protocol (MSDTP) [RFC0713]. Because of this focus, SDNVs can be quickly implemented with only a small amount of code.

SDNVs are tersely defined in both the Bundle Protocol [RFC5050] and Licklider Transmission Protocol (LTP) [RFC5326] specifications, due to the flow of document production in the DTNRG. This document clarifies and further explains the motivations and engineering decisions behind SDNVs.

1.1. Problems with Fixed-Value Fields

Protocol designers commonly face an optimization problem in determining the proper size for header fields. There is a strong desire to keep fields as small as possible, in order to reduce the protocol's overhead and also allow for fast processing. Since protocols can be used for many years (even decades) after they are designed, and networking technology has tended to change rapidly, it is not uncommon for the use, deployment, or performance of a particular protocol to be limited or infringed upon by the length of some header field being too short. Two well-known examples of this phenomenon are the TCP-advertised receive window and the IPv4 address length.

TCP segments contain an advertised receive window field that is fixed at 16 bits [RFC0793], encoding a maximum value of around 65 kilobytes. The purpose of this value is to provide flow control, by allowing a receiver to specify how many sent bytes its peer can have outstanding (unacknowledged) at any time, thus allowing the receiver to limit its buffer size. As network speeds have grown by several orders of magnitude since TCP's inception, the combination of the 65 kilobyte maximum advertised window and long round-trip times prevented TCP senders from being able to achieve the high throughput that the underlying network supported. This limitation was remedied through the use of the Window Scale option [RFC1323], which provides a multiplier for the advertised window field. However, the Window Scale multiplier is fixed for the duration of the connection, requires support from each end of a TCP connection, and limits the precision of the advertised receive window, so this is certainly a less-than-ideal solution. Because of the field width limit in the original design however, the Window Scale is necessary for TCP to reach high sending rates.

An IPv4 address is fixed at 32 bits [RFC0791] (as a historical note, an early version of the IP header format specification in [IEN21] used variable-length addresses in multiples of 8 bits up to 120 bits). Due to the way that subnetting and assignment of address blocks was performed, the number of IPv4 addresses has been seen as a

limit to the growth of the Internet [Hain05]. Two divergent paths to solve this problem have been the use of Network Address Translators (NATs) and the development of IPv6. NATs have caused a number of other issues and problems [RFC2993], leading to increased complexity and fragility, as well as forcing workarounds to be engineered for many other protocols to function within a NATed environment. The IPv6 solution's transitional work has been underway for several years, but has still only just begun to have visible impact on the global Internet.

Of course, in both the case of the TCP receive window and IPv4 address length, the field size chosen by the designers seemed like a good idea at the time. The fields were more than big enough for the originally perceived usage of the protocols, and yet were small enough to allow the headers to remain compact and relatively easy and efficient to parse on machines of the time. The fixed sizes that were defined represented a trade-off between the scalability of the protocol versus the overhead and efficiency of processing. In both cases, these engineering decisions turned out to be painfully restrictive in the longer term.

1.2. SDNVs for DTN Protocols

In specifications for the DTN Bundle Protocol (BP) [RFC5050] and Licklider Transmission Protocol (LTP) [RFC5326], SDNVs have been used for several fields including identifiers, payload/header lengths, and serial (sequence) numbers. SDNVs were developed for use in these types of fields, to avoid sending more bytes than needed, as well as avoiding fixed sizes that may not end up being appropriate. For example, since LTP is intended primarily for use in long-delay interplanetary communications [RFC5325], where links may be fairly low in capacity, it is desirable to avoid the header overhead of routinely sending a 64-bit field where a 16-bit field would suffice. Since many of the nodes implementing LTP are expected to be beyond the current range of human spaceflight, upgrading their on-board LTP implementations to use longer values if the defined fields are found to be too short would also be problematic. Furthermore, extensions similar in mechanism to TCP's Window Scale option are unsuitable for use in DTN protocols since, due to high delays, DTN protocols must avoid handshaking and configuration parameter negotiation to the greatest extent possible. All of these reasons make the choice of SDNVs for use in DTN protocols attractive.

1.3. SDNV Usage

In short, an SDNV is simply a way of representing non-negative integers (both positive integers of arbitrary magnitude and 0), without expending much unnecessary space. This definition allows SDNVs to represent many common protocol header fields, such as:

- Random identification fields as used in the IPsec Security Parameters Index or in IP headers for fragment reassembly (Note: the 16-bit IP ID field for fragment reassembly was recently found to be too short in some environments [RFC4963]).
- Sequence numbers as in TCP or the Stream Control Transmission Protocol (SCTP).
- Values used in cryptographic algorithms such as RSA keys, Diffie-Hellman key agreement, or coordinates of points on elliptic curves.
- Message lengths as used in file transfer protocols.
- Nonces and cookies.

As any bitfield can be interpreted as an unsigned integer, SDNVs can also encode arbitrary-length bitfields, including bitfields representing signed integers or other data types; however, this document assumes SDNV encoding and decoding in terms of unsigned integers. Implementations may differ in the interface that they provide to SDNV encoding and decoding functions, in terms of whether the values are numeric, bitfields, etc.; this detail does not alter the representation or algorithms described in this document.

The use of SDNVs rather than fixed-length fields gives protocol designers the ability to ameliorate the consequences of making difficult-to-reverse field-sizing decisions, as the SDNV format grows and shrinks depending on the particular value encoded. SDNVs do not necessarily provide optimal encodings for values of any particular length; however, they allow protocol designers to avoid potential blunders in assigning fixed lengths and remove the complexity involved with either negotiating field lengths or constructing protocol extensions. However, if SDNVs are used to encode bitfields, it is essential that the sender and receiver have a consistent interpretation of the decoded value. This is discussed further in Section 2.

To our knowledge, at this time, no IETF transport or network-layer protocol designed for use outside of the DTN domain has proposed to use SDNVs; however, there is no inherent reason not to use SDNVs more

broadly in the future. The two examples cited here, of fields that have proven too small in general Internet protocols, are only a small sampling of the much larger set of similar instances that the authors can think of. Outside the Internet protocols, within ASN.1 and previous ITU protocols, constructs very similar to SDNVs have been used for many years due to engineering concerns very similar to those facing the DTNRG.

Many protocols use a Type-Length-Value method for encoding variable-length fields (e.g., TCP's options format or many of the fields in the Internet Key Exchange Protocol version 2 (IKEv2)). An SDNV is equivalent to combining the length and value portions of this type of field, with the overhead of the length portion amortized out over the bytes of the value. The penalty paid for this in an SDNV may be several extra bytes for long values (e.g., 1024-bit RSA keys). See Section 4 for further discussion and a comparison.

As is shown in later sections, for large values, the current SDNV scheme is fairly inefficient in terms of space (1/8 of the bits are overhead) and not particularly easy to encode/decode in comparison to alternatives. The best use of SDNVs may often be to define the Length field of a TLV structure to be an SDNV whose value is the length of the TLV's Value field. In this way, one can avoid forcing large numbers from being directly encoded as an SDNV, yet retain the extensibility that using SDNVs grants.

2. Definition of SDNVs

Early in the work of the DTNRG, it was agreed that the properties of an SDNV were useful for DTN protocols. The exact SDNV format used by the DTNRG evolved somewhat over time before the publication of the initial RFCs on LTP and BP. An earlier version (see the initial version of LTP Internet Draft [BRF04]) bore a resemblance to the ASN.1 [ASN1] Basic Encoding Rules (BER) [X.690] for lengths (Section 8.1.3 of X.690). The current SDNV format is the one used by ASN.1 BER for encoding tag identifiers greater than or equal to 31 (Section 8.1.2.4.2 of X.690). A comparison between the current SDNV format and the early SDNV format is made in Section 4.

The format currently used is very simple. Before encoding, an integer is represented as a left-to-right bitstring beginning with its most significant bit and ending with its least significant bit. If the bitstring's length is not a multiple of 7, then the string is left-padded with zeros. When transmitted, the bits are encoded into a series of bytes. The low-order 7 bits of each byte in the encoded format are taken left-to-right from the integer's bitstring

representation. The most significant bit of each byte specifies whether it is the final byte of the encoded value (when it holds a 0), or not (when it holds a 1).

For example:

- 1 (decimal) is represented by the bitstring "0000001" and encoded as the single byte 0x01 (in hexadecimal).
- 128 is represented by the bitstring "10000001 00000000" and encoded as the bytes 0x81 followed by 0x00.
- Other values can be found in the test vectors of the source code in Appendix A.

To be perfectly clear, and avoid potential interoperability issues (as have occurred with ASN.1 BER time values), we explicitly state two considerations regarding zero-padding. (1) When encoding SDNVs, any leading (most significant) zero bits in the input number might be discarded by the SDNV encoder. Protocols that use SDNVs should not rely on leading-zeros being retained after encoding and decoding operations. (2) When decoding SDNVs, the relevant number of leading zeros required to pad up to a machine word or other natural data unit might be added. These are put in the most significant positions in order to not change the value of the number. Protocols using SDNVs should consider situations where lost zero-padding may be problematic.

The issues of zero-padding are particularly relevant where an SDNV is being used to represent a bitfield to be transmitted by a protocol. The specification of the protocol and any associated IANA registry should specify the allocation and usage of bit positions within the unencoded field. Unassigned and reserved bits in the unencoded field will be treated as zeros by the SDNV encoding prior to transmission. Assuming the bit positions are numbered starting from 0 at the least significant bit position in the integer representation, then if higher-numbered positions in the field contain all zeros, the encoding process may not transmit these bits explicitly (e.g., if all the bit positions numbered 7 or higher are zeros, then the transmitted SDNV can consist of just one octet). On reception, the decoding process will treat any untransmitted higher-numbered bits as zeros. To ensure correct operation of the protocol, the sender and receiver must have a consistent interpretation of the width of the bitfield. This can be achieved in various ways:

- the bitfield width is implicitly defined by the version of the protocol in use in the sender and receiver,

- sending the width of the bitfield explicitly in a separate item,
- the higher-numbered bits can be safely ignored by the receiver (e.g., because they represent optimizations), or
- marking the highest-numbered bit by prepending a '1' bit to the bitfield.

The protocol specification must record how the consistent interpretation is achieved.

The SDNV encoding technique is also known as Variable Byte Encoding (see Section 5.3.1 of [Manning09]) and is equivalent to Base-128 Elias Gamma Encoding (see Section 5.3.2 of [Manning09] and Section 3.5 of [Sayood02]). However, the primary motivation for SDNVs is to provide an extensible protocol framework rather than optimal data compression, which is the motivation behind the other uses of the technique. [Manning09] points out that the key feature of this encoding is that it is "prefix free" meaning that no code is a prefix of any other, which is an alternative way of expressing the self-delimiting property.

3. Basic Algorithms

This section describes some simple algorithms for creating and parsing SDNV fields. These may not be the most efficient algorithms possible, however, they are easy to read, understand, and implement. Appendix A contains Python source code implementing the routines described here. The algorithms presented here are convenient for converting between an internal data block and serialized data stream associated with a transmission device. Other approaches are possible with different efficiencies and trade-offs.

3.1. Encoding Algorithm

There is a very simple algorithm for the encoding operation that converts a non-negative integer (value n, of length 1+floor(log n) bits) into an SDNV. This algorithm takes n as its only argument and returns a string of bytes:

- (Initial Step) Set a variable X to a byte sharing the least significant 7 bits of n, and with 0 in the most significant bit, and a variable Y to n, right-shifted by 7 bits.
- (Recursion Step) If Y == 0, return X. Otherwise, set Z to the bitwise-or of 0x80 with the 7 least significant bits of Y, and append Z to X. Right-shift Y by 7 bits and repeat the Recursion Step.

This encoding algorithm has a time complexity of O(log n), since it takes a number of steps equal to ceil(n/7), and no additional space beyond the size of the result (8/7 log n) is required. One aspect of this algorithm is that it assumes strings can be efficiently appended to new bytes. One way to implement this is to allocate a buffer for the expected length of the result and fill that buffer one byte at a time from the right end.

If, for some reason, an implementation requires an encoded SDNV to be some specific length (possibly related to a machine word), any leftmost zero-padding included needs to properly set the high-order bit in each byte of padding.

3.2. Decoding Algorithm

Decoding SDNVs is a more difficult operation than encoding them, due to the fact that no bound on the resulting value is known until the SDNV is parsed, at which point the value itself is already known. This means that if space is allocated in advance to hold the value that results from decoding an SDNV, in general, it is not known whether this space will be large enough until it is 7 bits away from being overflowed. However, as specified in Section 3.3, protocols using SDNVs must specify the largest number of bits that an implementation is expected to handle, which mitigates this problem.

- (Initial Step) Set the result to 0. Set an index to the first byte of the encoded SDNV.

- (Recursion Step) Shift the result left 7 bits. Add the low-order 7 bits of the value at the index to the result. If the high-order bit under the pointer is a 1, advance the index by one byte within the encoded SDNV and repeat the Recursion Step, otherwise return the current value of the result.

This decoding algorithm takes no more additional space than what is required for the result (7/8 the length of the SDNV) and the pointer. The complication is that before the result can be left-shifted in the Recursion Step, an implementation needs to first make sure that this will not cause any bits to be lost, and re-allocate a larger piece of memory for the result, if required. The pure time complexity is the same as for the encoding algorithm given, but if re-allocation is needed due to the inability to predict the size of the result, decoding may be slower.

These decoding steps include removal of any leftmost zero-padding that might be used by an encoder to create encodings of a certain length.

3.3. Limitations of Implementations

Because of efficiency considerations or convenience of internal representation of decoded integers, implementations may choose to limit the number of bits in SDNVs that they will handle. To avoid interoperability problems, any protocol that uses SDNVs must specify the largest number of bits in an SDNV that an implementation of that protocol is expected to handle.

For example, Section 4.1 of [RFC5050] specifies that implementations of the DTN Bundle Protocol are not required to handle SDNVs with more than 64 bits in their unencoded value. Accordingly, integer values transmitted in SDNVs have an upper limit and SDNV-encoded flag fields must be limited to 64 bit positions in any future revisions of the protocol unless the restriction is altered.

4. Comparison to Alternatives

This section compares three alternative ways of implementing the concept of SDNVs: (1) the TLV scheme commonly used in the Internet family, and many other families of protocols, (2) the old style of SDNVs (both the SDNV-8 and SDNV-16) defined in an early stage of LTP's development [BRF04], and (3) the current SDNV format.

The TLV method uses two fixed-length fields to hold the Type and Length elements that then imply the syntax and semantics of the Value element. This is only similar to an SDNV in that the value element can grow or shrink within the bounds capable of being conveyed by the Length field. Two fundamental differences between TLVs and SDNVs are that through the Type element, TLVs also contain some notion of what their contents are semantically, while SDNVs are simply generic non-negative integers, and protocol engineers still have to choose fixed-field lengths for the Type and Length fields in the TLV format.

Some protocols use TLVs where the value conveyed within the Length field needs to be decoded into the actual length of the Value field. This may be accomplished through simple multiplication, left-shifting, or a look-up table. In any case, this tactic limits the granularity of the possible Value lengths, and can contribute some degree of bloat if Values do not fit neatly within the available decoded Lengths.

In the SDNV format originally used by LTP, parsing the first byte of the SDNV told an implementation how much space was required to hold the contained value. There were two different types of SDNVs defined for different ranges of use. The SDNV-8 type could hold values up to 127 in a single byte, while the SDNV-16 type could hold values up to 32,767 in 2 bytes. Both formats could encode values requiring up to

N bytes in N+2 bytes, where N<127. The major difference between this old SDNV format and the current SDNV format is that the new format is not as easily decoded as the old format was, but the new format also has absolutely no limitation on its length.

The advantage in ease of parsing the old format manifests itself in two aspects: (1) the size of the value is determinable ahead of time, in a way equivalent to parsing a TLV, and (2) the actual value is directly encoded and decoded, without shifting and masking bits as is required in the new format. For these reasons, the old format requires less computational overhead to deal with, but is also very limited in that it can only hold a 1024-bit number, at maximum. Since according to IETF Best Current Practices, an asymmetric cryptography key needed to last for a long term requires using moduli of over 1228 bits [RFC3766], this could be seen as a severe limitation of the old style of SDNVs, from which the currently used style does not suffer.

Table 1 compares the maximum values that can be encoded into SDNVs of various lengths using the old SDNV-8/16 method and the current SDNV method. The only place in this table where SDNV-16 is used rather than SDNV-8 is in the 2-byte row. Starting with a single byte, the two methods are equivalent, but when using 2 bytes, the old method is a more compact encoding by one bit. From 3 to 7 bytes of length though, the current SDNV format is more compact, since it only requires one bit per byte of overhead, whereas the old format used a full byte. Thus, at 8 bytes, both schemes are equivalent in efficiency since they both use 8 bits of overhead. Up to 129 bytes, the old format is more compact than the current one, although after this, limit it becomes unusable.

Bytes	SDNV-8/16 Maximum Value	SDNV Maximum Value	SDNV-8/16 Overhead Bits	SDNV Overhead Bits
1	127	127	1	1
2	32,767	16,383	1	2
3	65,535	2,097,151	8	3
4	2^24 - 1	2^28 - 1	8	4
5	2^32 - 1	2^35 - 1	8	5
6	2^40 - 1	2^42 - 1	8	6
7	2^48 - 1	2^49 - 1	8	7
8	2^56 - 1	2^56 - 1	8	8
9	2^64 - 1	2^63 - 1	8	9
10	2^72 - 1	2^70 - 1	8	10
16	2^120 - 1	2^112 - 1	8	16
32	2^248 - 1	2^224 - 1	8	32
64	2^504 - 1	2^448 - 1	8	64
128	2^1016 - 1	2^896 - 1	8	128
129	2^1024 - 1	2^903 - 1	8	129
130	N/A	2^910 - 1	N/A	130
256	N/A	2^1792 - 1	N/A	256

Table 1

Suggested usages of the SDNV format that leverage its strengths and limit the effects of its weaknesses are discussed in Section 1.3.

Another aspect of the comparison between SDNVs and alternatives using fixed-length fields is the result of errors in transmission. Bit-errors in an SDNV can result in either errors in the decoded value,

or parsing errors in subsequent fields of the protocol. In fixed-length fields, bit errors always result in errors to the decoded value rather than parsing errors in subsequent fields. If the decoded values from either type of field encoding (SDNV or fixed-length) are used as indexes, offsets, or lengths of further fields in the protocol, similar failures result.

5. Security Considerations

The only security considerations with regard to SDNVs are that code that parses SDNVs should have bounds-checking logic and be capable of handling cases where an SDNV's value is beyond the code's ability to parse. These precautions can prevent potential exploits involving SDNV decoding routines.

Stephen Farrell noted that very early definitions of SDNVs also allowed negative integers. This was considered a potential security hole, since it could expose implementations to underflow attacks during SDNV decoding. There is a precedent in that many existing TLV decoders map the Length field to a signed integer and are vulnerable in this way. An SDNV decoder should be based on unsigned types and not have this issue.

6. Acknowledgements

Scott Burleigh, Manikantan Ramadas, Michael Demmer, Stephen Farrell, and other members of the IRTF DTN Research Group contributed to the development and usage of SDNVs in DTN protocols. George Jones and Keith Scott from Mitre, Lloyd Wood, Gerardo Izquierdo, Joel Halpern, Peter TB Brett, Kevin Fall, and Elwyn Davies also contributed useful comments on and criticisms of this document. DTNRG last call comments on the document were sent to the mailing list by Lloyd Wood, Will Ivancic, Jim Wyllie, William Edwards, Hans Kruse, Janico Greifenberg, Teemu Karkkainen, Stephen Farrell, and Scott Burleigh. Further constructive comments from Dave Crocker, Lachlan Andrew, and Michael Welzl were incorporated.

Work on this document was performed at NASA's Glenn Research Center, in support of the NASA Space Communications Architecture Working Group (SCAWG), NASA's Earth Science Technology Office (ESTO), and the FAA/Eurocontrol Future Communications Study (FCS) in the 2005-2007 time frame, while the editor was an employee of Verizon Federal Network Systems.

7. Informative References

[ASN1] ITU-T Rec. X.680, "Abstract Syntax Notation One (ASN.1). Specification of Basic Notation", ISO/IEC 8824-1:2002, 2002.

[BRF04] Burleigh, S., Ramadas, M., and S. Farrell, "Licklider Transmission Protocol", Work in Progress, May 2004.

[Hain05] Hain, T., "A Pragmatic Report on IPv4 Address Space Consumption", Internet Protocol Journal Vol. 8, No. 3, September 2005.

[IEN21] Cerf, V. and J. Postel, "Specification of Internetwork Transmission Control Program: TCP Version 3", Internet Experimental Note 21, January 1978.

[Manning09] Manning, c., Raghavan, P., and H. Schuetze, "Introduction to Information Retrieval", Cambridge University Press ISBN-13: 978-0521865715, 2009, <http://informationretrieval.org/>.

[RFC0713] Haverty, J., "MSDTP-Message Services Data Transmission Protocol", RFC 713, April 1976.

[RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC1323] Jacobson, V., Braden, B., and D. Borman, "TCP Extensions for High Performance", RFC 1323, May 1992.

[RFC2993] Hain, T., "Architectural Implications of NAT", RFC 2993, November 2000.

[RFC3766] Orman, H. and P. Hoffman, "Determining Strengths For Public Keys Used For Exchanging Symmetric Keys", BCP 86, RFC 3766, April 2004.

[RFC4963] Heffner, J., Mathis, M., and B. Chandler, "IPv4 Reassembly Errors at High Data Rates", RFC 4963, July 2007.

[RFC5050] Scott, K. and S. Burleigh, "Bundle Protocol Specification", RFC 5050, November 2007.

[RFC5325] Burleigh, S., Ramadas, M., and S. Farrell, "Licklider Transmission Protocol - Motivation", RFC 5325, September 2008.

[RFC5326] Ramadas, M., Burleigh, S., and S. Farrell, "Licklider Transmission Protocol - Specification", RFC 5326, September 2008.

[Sayood02] Sayood, K., "Lossless Data Compression", Academic Press ISBN-13: 978-0126208610, December 2002, <http://books.google.co.uk/books?id=LjQiGwyabVwC>.

[X.690] ITU-T Rec. X.690, "Abstract Syntax Notation One (ASN.1). Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8825-1:2002, 2002.

Appendix A. SDNV Python Source Code

```
#  This code may be freely copied.  Attribution would be appreciated.
#
# sdnv_decode() takes a string argument (s), which is assumed to be
#   an SDNV, and optionally a number (slen) for the maximum number of
#   bytes to parse from the string.  The function returns a pair of
#   the non-negative integer n that is the numeric value encoded in
#   the SDNV, and integer that is the distance parsed into the input
#   string.  If the slen argument is not given (or is not a non-zero
#   number) then, s is parsed up to the first byte whose high-order
#   bit is 0 -- the length of the SDNV portion of s does not have to
#   be pre-computed by calling code.  If the slen argument is given
#   as a non-zero value, then slen bytes of s are parsed.  The value
#   for n of -1 is returned for any type of parsing error.
#
# NOTE: In python, integers can be of arbitrary size.  In other
#   languages, such as C, SDNV-parsing routines should take
#   precautions to avoid overflow (e.g., by using the Gnu MP library,
#   or similar).
#
def sdnv_decode(s, slen=0):
  n = long(0)
  for i in range(0, len(s)):
    v = ord(s[i])
    n = n<<7
    n = n + (v & 0x7F)
    if v>>7 == 0:
      slen = i+1
      break
    elif i == len(s)-1 or (slen != 0 and i > slen):
      n = -1 # reached end of input without seeing end of SDNV
  return (n, slen)

# sdnv_encode() returns the SDNV-encoded string that represents n.
#   An empty string is returned if n is not a non-negative integer
def sdnv_encode(n):
  r = ""
  # validate input
  if n >= 0 and (type(n) in [type(int(1)), type(long(1))]):
    flag = 0
    done = False
    while not done:
      # encode lowest 7 bits from n
      newbits = n & 0x7F
      n = n>>7
      r = chr(newbits + flag) + r
      if flag == 0:
```

```
          flag = 0x80
        if n == 0:
          done = True
    return r


  # test cases from LTP and BP internet-drafts, only print failures
  def sdnv_test():
    tests = [(0xABC, chr(0x95) + chr(0x3C)),
             (0x1234, chr(0xA4) + chr (0x34)),
             (0x4234, chr(0x81) + chr(0x84) + chr(0x34)),
             (0x7F, chr(0x7F))]

    for tp in tests:
      # test encoding function
      if sdnv_encode(tp[0]) != tp[1]:
        print "sdnv_encode fails on input %s" % hex(tp[0])
      # test decoding function
      if sdnv_decode(tp[1])[0] != tp[0]:
        print "sdnv_decode fails on input %s, giving %s" % \
              (hex(tp[0]), sdnv_decode(tp[1]))
```

Authors' Addresses

Wesley M. Eddy
MTI Systems
NASA Glenn Research Center
MS 500-ASRC; 21000 Brookpark Rd
Cleveland, OH 44135

Phone: 216-433-6682
EMail: wes@mti-systems.com

Elwyn Davies
Folly Consulting
Soham
UK

Phone:
EMail: elwynd@folly.org.uk
URI: