Network Working Group
Request for Comments: 3309
Updates: 2960
Category: Standards

J. Stone
Stanford
R. Stewart
Cisco Systems
D. Otis
SANlight

September 2002

Stream Control Transmission Protocol (SCTP) Checksum Change

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Abstract

Stream Control Transmission Protocol (SCTP) currently uses an Adler-32 checksum. For small packets Adler-32 provides weak detection of errors. This document changes that checksum and updates SCTP to use a 32 bit CRC checksum.

Table of Contents

1 Introduction

A fundamental weakness has been detected in SCTP's current Adler-32 checksum algorithm [STONE]. This document updates and replaces the Adler-32 checksum definition in [RFC 2960]. Note that there is no graceful transition mechanism for migrating to the new checksum. Implementations are expected to immediately switch to the new algorithm; use of the old algorithm is deprecated.

One requirement of an effective checksum is that it evenly and smoothly spreads its input packets over the available check bits.

From an email from Jonathan Stone, who analyzed the Adler-32 as part of his doctoral thesis:

"Briefly, the problem is that, for very short packets, Adler32 is guaranteed to give poor coverage of the available bits. Don't take my word for it, ask Mark Adler. :-)

Adler-32 uses two 16-bit counters, s1 and s2. s1 is the sum of the input, taken as 8-bit bytes. s2 is a running sum of each value of s1. Both s1 and s2 are computed mod-65521 (the largest prime less than 2^16). Consider a packet of 128 bytes. The *most* that each byte can be is 255. There are only 128 bytes of input, so the greatest value which the s1 accumulator can have is 255 * 128 = 32640. So, for 128-byte packets, s1 never wraps. That is critical. Why?

The key is to consider the distribution of the s1 values, over some distribution of the values of the individual input bytes in each packet. Because s1 never wraps, s1 is simply the sum of the individual input bytes. (Even Doug's trick of adding 0x5555 doesn't help here, and an even larger value doesn't really help: we can get at most one mod-65521 reduction.)

Given the further assumption that the input bytes are drawn independently from some distribution (they probably aren't: for file system data, it's even worse than that!), the Central Limit Theorem tells us that that s1 will tend to have a normal distribution. That's bad: it tells us that the value of s1 will have hot-spots at around 128 times the mean of the input distribution: around 16k, assuming a uniform distribution. That's bad. We want the accumulator to wrap as many times as possible, so that the resulting sum has as close to a uniform distribution as possible. (I call this "fairness".)

So, for short packets, the Adler-32 s1 sum is guaranteed to be unfair. Why is that bad? It’s bad because the space of valid packets -- input data, plus checksum values -- is also small. If all packets have checksum values very close to 32640, then the likelihood of even a ’small’ error leaving a damaged packet with a valid checksum is higher than if all checksum values are equally likely."

Due to this inherent weakness, exacerbated by the fact that SCTP will first be used as a signaling transport protocol where signaling messages are usually less than 128 bytes, a new checksum algorithm is specified by this document, replacing the current Adler-32 algorithm with CRC-32c.

1.1 Conventions

The keywords MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD,SHOULD NOT, RECOMMENDED, NOT RECOMMENDED, MAY, and OPTIONAL, when they appear in this document, are to be interpreted as described in [RFC2119].

Bit number order is defined in [RFC1700].

2 Checksum Procedures

The procedures described in section 2.1 of this document MUST be followed, replacing the current checksum defined in [RFC2960].

Furthermore any references within [RFC2960] to Adler-32 MUST be treated as a reference to CRC-32c. Section 2.1 of this document describes the new calculation and verification procedures that MUST be followed.

2.1 Checksum Calculation

When sending an SCTP packet, the endpoint MUST strengthen the data integrity of the transmission by including the CRC-32c checksum value calculated on the packet, as described below.

After the packet is constructed (containing the SCTP common header and one or more control or DATA chunks), the transmitter shall:

1) Fill in the proper Verification Tag in the SCTP common header and initialize the Checksum field to 0’s.

2) Calculate the CRC-32c of the whole packet, including the SCTP common header and all the chunks.

3) Put the resulting value into the Checksum field in the common header, and leave the rest of the bits unchanged.

When an SCTP packet is received, the receiver MUST first check the CRC-32c checksum:

1) Store the received CRC-32c value,

2) Replace the 32 bits of the Checksum field in the received SCTP packet with all ’0’s and calculate a CRC-32c value of the whole received packet. And,

3) Verify that the calculated CRC-32c value is the same as the received CRC-32c value. If not, the receiver MUST treat the packet as an invalid SCTP packet.

The default procedure for handling invalid SCTP packets is to silently discard them.

Any hardware implementation SHOULD be done in a way that is verifiable by the software.

We define a ’reflected value’ as one that is the opposite of the normal bit order of the machine. The 32 bit CRC is calculated as described for CRC-32c and uses the polynomial code 0x11EDC6F41 (Castagnoli93) or x^32+x^28+x^27+x^26+x^25 +x^23+x^22+x^20+x^19+x^18+x^14+x^13+x^11+x^10+x^9+x^8+x^6+x^0. The CRC is computed using a procedure similar to ETHERNET CRC [ITU32], modified to reflect transport level usage.

CRC computation uses polynomial division. A message bit-string M is transformed to a polynomial, M(X), and the CRC is calculated from M(X) using polynomial arithmetic [Peterson 72].

When CRCs are used at the link layer, the polynomial is derived from on-the-wire bit ordering: the first bit ’on the wire’ is the high-order coefficient. Since SCTP is a transport-level protocol, it cannot know the actual serial-media bit ordering. Moreover, different links in the path between SCTP endpoints may use different link-level bit orders.

A convention must therefore be established for mapping SCTP transport messages to polynomials for purposes of CRC computation. The bit-ordering for mapping SCTP messages to polynomials is that bytes are taken most-significant first; but within each byte, bits are taken least-significant first. The first byte of the message provides the eight highest coefficients. Within each byte, the least-significant SCTP bit gives the most significant polynomial coefficient within that byte, and the most-significant SCTP bit is the least significant polynomial coefficient in that byte. (This bit ordering is sometimes called 'mirrored' or 'reflected' [Williams93].) CRC polynomials are to be transformed back into SCTP transport-level byte values, using a consistent mapping.

The SCTP transport-level CRC value should be calculated as follows:

- CRC input data are assigned to a byte stream, numbered from 0 to N-1.

- the transport-level byte-stream is mapped to a polynomial value. An N-byte PDU with j bytes numbered 0 to N-1, is considered as coefficients of a polynomial M(x) of order 8N-1, with bit 0 of byte j being coefficient x^(8(N-j)-8), bit 7 of byte j being coefficient x^(8(N-j)-1).

- the CRC remainder register is initialized with all 1s and the CRC is computed with an algorithm that simultaneously multiplies by x^32 and divides by the CRC polynomial.

- the polynomial is multiplied by x^32 and divided by G(x), the generator polynomial, producing a remainder R(x) of degree less than or equal to 31.

- the coefficients of R(x) are considered a 32 bit sequence.

- the bit sequence is complemented. The result is the CRC polynomial.

- The CRC polynomial is mapped back into SCTP transport-level bytes. Coefficient of x^31 gives the value of bit 7 of SCTP byte 0, the coefficient of x^24 gives the value of bit 0 of byte 0. The coefficient of x^7 gives bit 7 of byte 3 and the coefficient of x^0 gives bit 0 of byte 3. The resulting four-byte transport-level sequence is the 32-bit SCTP checksum value.

IMPLEMENTATION NOTE: Standards documents, textbooks, and vendor literature on CRCs often follow an alternative formulation, in which the register used to hold the remainder of the long-division algorithm is initialized to zero rather than all-1s, and instead the first 32 bits of the message are complemented. The long-division algorithm used in our formulation is specified, such that the the initial multiplication by 2^32 and the long-division are combined into one simultaneous operation. For such algorithms, and for messages longer than 64 bits, the two specifications are precisely equivalent. That equivalence is the intent of this document.

Implementors of SCTP are warned that both specifications are to be found in the literature, sometimes with no restriction on the long-division algorithm. The choice of formulation in this document is to permit non-SCTP usage, where the same CRC algorithm may be used to protect messages shorter than 64 bits.

If SCTP could follow link level CRC use, the CRC would be computed over the link-level bit-stream. The first bit on the link mapping to the highest-order coefficient, and so on, down to the last link-level bit as the lowest-order coefficient. The CRC value would be transmitted immediately after the input message as a link-level 'trailer'. The resulting link-level bit-stream would be (M(X)x) * x^32) + (M(X)*x^32))/ G(x), which is divisible by G(X). There would thus be a constant CRC remainder for 'good' packets. However, given that implementations of RFC 2960 have already proliferated, the IETF discussions considered that the benefit of a 'trailer' CRC did not outweigh the cost of making a very large change in the protocol processing. Further, packets accepted by the SCTP 'header' CRC are in one-to-one correspondence with packets accepted by a modified procedure using a 'trailer' CRC value, and where the SCTP common checksum header is set to zero on transmission and is received as zero.

There may be a computational advantage in validating the Association against the Verification Tag, prior to performing a checksum, as invalid tags will result in the same action as a bad checksum in most cases. The exceptions for this technique would be INIT and some SHUTDOWN-COMPLETE exchanges, as well as a stale COOKIE-ECHO. These special case exchanges must represent small packets and will minimize the effect of the checksum calculation.

3 Security Considerations

In general, the security considerations of RFC 2960 apply to the protocol with the new checksum as well.

4 IANA Considerations

There are no IANA considerations required in this document.

5 Acknowledgments

The authors would like to thank the following people that have provided comments and input on the checksum issue:

Mark Adler, Ran Atkinson, Stephen Bailey, David Black, Scott Bradner, Mikael Degermark, Laurent Glaude, Klaus Gradischnig, Alf Heidermark, Jacob Heitz, Gareth Kiely, David Lehmann, Allision Mankin, Lyndon Ong, Craig Partridge, Vern Paxson, Kacheong Poon, Michael Ramalho, David Reed, Ian Rytina, Hanns Juergen Schwarzbauer, Chip Sharp, Bill Sommerfeld, Michael Tuexen, Jim Williams, Jim Wendt, Michael Welzl, Jonathan Wood, Lloyd Wood, Qiaobing Xie, La Monte Yarroll.

Special thanks to Dafna Scheinwald, Julian Satran, Pat Thaler, Matt Wakeley, and Vince Cavanna, for selection criteria of polynomials and examination of CRC polynomials, particularly CRC-32c [Castagnoli93].

Special thanks to Mr. Ross Williams and his document [Williams93]. This non-formal perspective on software aspects of CRCs furthered understanding of authors previously unfamiliar with CRC computation. More formal treatments of [Blahut 94] or [Peterson 72], was also essential.

6 References

[Castagnoli93] G. Castagnoli, S. Braeuer and M. Herrman, "Optimization of Cyclic Redundancy-Check Codes with 24 and 32 Parity Bits", IEEE Transactions on Communications, Vol. 41, No. 6, June 1993

[McKee75] H. McKee, "Improved {CRC} techniques detects erroneous leading and trailing 0’s in transmitted data blocks", Computer Design Volume 14 Number 10 Pages 102-4,106, October 1975

[RFC1700] Reynolds, J. and J. Postel, "ASSIGNED NUMBERS", RFC 1700, October 1994.

[RFC2026] Bradner, S., "The Internet Standards Process -- Revision 3", BCP 9, RFC 2026, October 1996.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2960] Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L. and V. Paxson, "Stream Control Transmission Protocol," RFC 2960, October 2000.

[ITU32] ITU-T Recommendation V.42, "Error-correcting procedures for DCEs using asynchronous-to-synchronous conversion", section 8.1.1.6.2, October 1996.

7.1 Informative References

[STONE] Stone, J., "Checksums in the Internet", Doctoral dissertation - August 2001.

[Williams93] Williams, R., "A PAINLESS GUIDE TO CRC ERROR DETECTION ALGORITHMS" - Internet publication, August 1993, http://www.geocities.com/SiliconValley/Pines/8659/crc.htm.

[Blahut 1994] R.E. Blahut, Theory and Practice of Error Control Codes, Addison-Wesley, 1994.

[Easics 2001] http://www.easics.be/webtools/crctool. Online tools for synthesis of CRC Verilog and VHDL.

[Feldmeier 95] David C. Feldmeier, Fast software implementation of error detection codes, IEEE Transactions on Networking, vol 3 no 6, pp 640-651, December, 1995.

[Glaise 1997] R. J. Glaise, A two-step computation of cyclic redundancy code CRC-32 for ATM networks, IBM Journal of Research and Development} vol 41 no 6, 1997. http://www.research.ibm.com/journal/rd/416/glaise.html.

[Prange 1957] E. Prange, Cyclic Error-Correcting codes in two symbols, Technical report AFCRC-TN-57-103, Air Force Cambridge Research Center, Cambridge, Mass. 1957.

[Peterson 1972] W. W. Peterson and E.J Weldon, Error Correcting Codes, 2nd. edition, MIT Press, Cambridge, Massachusetts.

[Shie2001] Ming-Der Shieh et. al, A Systematic Approach for Parallel CRC Computations. Journal of Information Science and Engineering, Vol.17 No.3, pp.445-461

[Sprachman2001] Michael Sprachman, Automatic Generation of Parallel CRC Circuits, IEEE Design & Test May-June 2001

Appendix

This appendix is for information only and is NOT part of the standard.

The anticipated deployment of SCTP ranges over several orders of magnitude of link speed: from cellular-power telephony devices at tens of kilobits, to local links at tens of gigabits. Implementors of SCTP should consider their link speed and choose, from the wide range of CRC implementations, one which matches their own design point for size, cost, and throughput. Many techniques for computing CRCs are known. This Appendix surveys just a few, to give a feel for the range of techniques available.

CRCs are derived from early work by Prange in the 1950s [Prange 57]. The theory underlying CRCs and choice of generator polynomial can be introduced by either the theory of Galois fields [Blahut 94] or as ideals of an algebra over cyclic codes [cite Peterson 72].

One of the simplest techniques is direct bit-serial hardware implementations, using the generator polynomial as the taps of a linear feedback shift register (LSFR). LSFR computation follows directly from the mathematics, and is generally attributed to Prange. Tools exist which, a CRC generator polynomial, will produce synthesizable Verilog code for CRC hardware [Easics 2001].

Since LSFRs do not scale well in speed, a variety of other techniques have been explored. One technique exploits the fact that the divisor of the polynomial long-division, G, is known in advance. It is thus possible to pre-compute lookup tables giving the polynomial remainder of multiple input bits --- typically 2, 4, or 8 bits of input at a time. This technique can be used either in software or in hardware. Software to compute lookup tables yielding 2, 4, or 8 bits of result is freely available. [Williams93]

For multi-gigabit links, the above techniques may still not be fast enough. One technique for computing CRCS at OC-48 rates is ’two-stage’ CRC computation [Glaise 1997]. Here, some multiple of G(x), G(x)H(x), is chosen so as to minimize the number of nonzero coefficients, or weight, of the product G(x)H(x). The low weight of the product polynomial makes it susceptible to efficient hardware divide-by-constant implementations. This first stage gives M(x)/(G(x)H(x)), as its result. The second stage then divides the result of the first stage by H(x), yielding (M(x)/(G(x)H(x)))/H(x). If H(x) is also relatively prime to G(x), this gives M(x)/G(x). Further developments on this approach can be found in [Shie2001] and [Sprachman2001].

The literature also includes a variety of software CRC implementations. One approach is to use a carefully-tuned assembly code for direct polynomial division. [Feldmeier 95] reports that for low-weight polynomials, tuned polynomial arithmetic gives higher throughput than table-lookup algorithms. Even within table-lookup algorithms, the size of the table can be tuned, either for total cache footprint, or (for space-restricted environments) to minimize total size.

Implementors should keep in mind, the bit ordering described in Section 2: the ordering of bits within bytes for computing CRCs in SCTP is the least significant bit of each byte is the most-significant polynomial coefficient(and vice-versa). This ’reflected’ SCTP CRC bit ordering matches on-the-wire bit order for Ethernet and other serial media, but is the reverse of traditional Internet bit ordering.

One technique to accommodate this bit-reversal can be explained as follows: sketch out a hardware implementation, assuming the bits are in CRC bit order; then perform a left-to-right inversion (mirror image) on the entire algorithm. (We defer, for a moment, the issue of byte order within words.) Then compute that "mirror image" in software. The CRC from the "mirror image" algorithm will be the bit-reversal of a correct hardware implementation. When the link-level media sends each byte, the byte is sent in the reverse of the host CPU bit-order. Serialization of each byte of the "reflected" CRC value re-reverses the bit order, so in the end, each byte will be transmitted on-the-wire in the specified bit order.

The following non-normative sample code is taken from an open-source CRC generator [Williams93], using the "mirroring" technique and yielding a lookup table for SCTP CRC32-c with 256 entries, each 32 bits wide. While neither especially slow nor especially fast, as software table-lookup CRCs go, it has the advantage of working on both big-endian and little-endian CPUs, using the same (host-order) lookup tables, and using only the pre-defined ntohl() and htonl() operations. The code is somewhat modified from [Williams93], to ensure portability between big-endian and little-endian architectures. (Note that if the byte endian-ness of the target architecture is known to be little-endian the final bit-reversal and byte-reversal steps can be folded into a single operation.)

```
/*************************************************************/
/* Note Definition for Ross Williams table generator would   */
/* be: TB_WIDTH=4, TB_POLLY=0x1EDC6F41, TB_REVER=TRUE        */
/* For Mr. Williams direct calculation code use the settings */
/* cm_width=32, cm_poly=0x1EDC6F41, cm_init=0xFFFFFFFF,      */
/* cm_refin=TRUE, cm_refot=TRUE, cm_xorort=0x00000000        */
/*************************************************************/

/* Example of the crc table file */
#ifndef __crc32cr_table_h__
#define __crc32cr_table_h__

#define CRC32C_POLY 0x1EDC6F41
#define CRC32C(c,d) (c=(c>>8)^crc_c[(c^(d))&0xFF])

unsigned long  crc_c[256] =
{
0x00000000L, 0xF26B8303L, 0xE13B70F7L, 0x1350F3F4L,
0xC79A971FL, 0x35F1141CL, 0x26A1E7E8L, 0xD4CA64EBL,
0x8AD958CFL, 0x78B2DBCCL, 0x6BE22838L, 0x9989AB3BL,
0x4D43CFD0L, 0xBF284CD3L, 0xAC78BF27L, 0x5E133C24L,
0x105EC76FL, 0xE235446CL, 0xF165B798L, 0x030E349BL,
0xD7C45070L, 0x25AFD373L, 0x36FF2087L, 0xC494A384L,
0x9A879FA0L, 0x68EC1CA3L, 0x7BBCEF57L, 0x89D76C54L,
0x5D1D08BFL, 0xAF768BBCL, 0xBC267848L, 0x4E4DFB4BL,
0x20BD8EDEL, 0xD2D60DDDL, 0xC186FE29L, 0x33ED7D2AL,
0xE72719C1L, 0x154C9AC2L, 0x061C6936L, 0xF477EA35L,
0xAA64D611L, 0x580F5512L, 0x4B5FA6E6L, 0xB93425E5L,
0x6DFE410EL, 0x9F95C20DL, 0x8CC531F9L, 0x7EAEB2FAL,
0x30E349B1L, 0xC288CAB2L, 0xD1D83946L, 0x23B3BA45L,
0xF779DEAEL, 0x05125DADL, 0x1642AE59L, 0xE4292D5AL,
0xBA3A117EL, 0x4851927DL, 0x5B016189L, 0xA96AE28AL,
0x7DA08661L, 0x8FCB0562L, 0x9C9BF696L, 0x6EF07595L,
0x417B1DBCL, 0xB3109EBFL, 0xA0406D4BL, 0x522BEE48L,
0x86E18AA3L, 0x748A09A0L, 0x67DAFA54L, 0x95B17957L,
0xCBA24573L, 0x39C9C670L, 0x2A993584L, 0xD8F2B687L,
0x0C38D26CL, 0xFE53516FL, 0xED03A29BL, 0x1F682198L,
0x5125DAD3L, 0xA34E59D0L, 0xB01EAA24L, 0x42752927L,
0x96BF4DCCL, 0x64D4CECFL, 0x77843D3BL, 0x85EFBE38L,
0xDBFC821CL, 0x2997011FL, 0x3AC7F2EBL, 0xC8AC71E8L,
0x1C661503L, 0xEE0D9600L, 0xFD5D65F4L, 0x0F36E6F7L,
0x61C69362L, 0x93AD1061L, 0x80FDE395L, 0x72966096L,
0xA65C047DL, 0x5437877EL, 0x4767748AL, 0xB50CF789L,
0xEB1FCBADL, 0x197448AEL, 0x0A24BB5AL, 0xF84F3859L,
0x2C855CB2L, 0xDEEEDFB1L, 0xCDBE2C45L, 0x3FD5AF46L,
0x7198540DL, 0x83F3D70EL, 0x90A324FAL, 0x62C8A7F9L,
0xB602C312L, 0x44694011L, 0x5739B3E5L, 0xA55230E6L,
0xFB410CC2L, 0x092A8FC1L, 0x1A7A7C35L, 0xE811FF36L,
0x3CDB9BDDL, 0xCEB018DEL, 0xDDE0EB2AL, 0x2F8B6829L,
0x82F63B78L, 0x709DB87BL, 0x63CD4B8FL, 0x91A6C88CL,
0x456CAC67L, 0xB7072F64L, 0xA457DC90L, 0x563C5F93L,
0x082F63B7L, 0xFA44E0B4L, 0xE9141340L, 0x1B7F9043L,
0xCFB5F4A8L, 0x3DDE77ABL, 0x2E8E845FL, 0xDCE5075CL,
0x92A8FC17L, 0x60C37F14L, 0x73938CE0L, 0x81F80FE3L,
0x55326B08L, 0xA759E80BL, 0xB4091BFFL, 0x466298FCL,
0x1871A4D8L, 0xEA1A27DBL, 0xF94AD42FL, 0x0B21572CL,
0xDFEB33C7L, 0x2D80B0C4L, 0x3ED04330L, 0xCCBBC033L,
0xA24BB5A6L, 0x502036A5L, 0x4370C551L, 0xB11B4652L,
0x65D122B9L, 0x97BAA1BAL, 0x84EA524EL, 0x7681D14DL,
0x2892ED69L, 0xDAF96E6AL, 0xC9A99D9EL, 0x3BC21E9DL,
0xEF087A76L, 0x1D63F975L, 0x0E330A81L, 0xFC588982L,
0xB21572C9L, 0x407EF1CAL, 0x532E023EL, 0xA145813DL,
0x758FE5D6L, 0x87E466D5L, 0x94B49521L, 0x66DF1622L,
0x38CC2A06L, 0xCAA7A905L, 0xD9F75AF1L, 0x2B9CD9F2L,
0xFF56BD19L, 0x0D3D3E1AL, 0x1E6DCDEEL, 0xEC064EEDL,
0xC38D26C4L, 0x31E6A5C7L, 0x22B65633L, 0xD0DDD530L,
0x0417B1DBL, 0xF67C32D8L, 0xE52CC12CL, 0x1747422FL,
0x49547E0BL, 0xBB3FFD08L, 0xA86F0EFCL, 0x5A048DFFL,
0x8ECEE914L, 0x7CA56A17L, 0x6FF599E3L, 0x9D9E1AE0L,
0xD3D3E1ABL, 0x21B862A8L, 0x32E8915CL, 0xC083125FL,
0x144976B4L, 0xE622F5B7L, 0xF5720643L, 0x07198540L,
0x590AB964L, 0xAB613A67L, 0xB831C993L, 0x4A5A4A90L,
0x9E902E7BL, 0x6CFBAD78L, 0x7FAB5E8CL, 0x8DC0DD8FL,
0xE330A81AL, 0x115B2B19L, 0x020BD8EDL, 0xF0605BEEL,
0x24AA3F05L, 0xD6C1BC06L, 0xC5914FF2L, 0x37FACCF1L,
0x69E9F0D5L, 0x9B8273D6L, 0x88D28022L, 0x7AB90321L,
0xAE7367CAL, 0x5C18E4C9L, 0x4F48173DL, 0xBD23943EL,
0xF36E6F75L, 0x0105EC76L, 0x12551F82L, 0xE03E9C81L,
0x34F4F86AL, 0xC69F7B69L, 0xD5CF889DL, 0x27A40B9EL,
0x79B737BAL, 0x8BDCB4B9L, 0x988C474DL, 0x6AE7C44EL,
0xBE2DA0A5L, 0x4C4623A6L, 0x5F16D052L, 0xAD7D5351L,
};

#endif

 /* Example of table build routine */



#include <stdio.h>
#include <stdlib.h>

#define OUTPUT_FILE   "crc32cr.h"
#define CRC32C_POLY    0x1EDC6F41L
FILE *tf;
```

```
unsigned long
reflect_32 (unsigned long b)
{
  int i;
  unsigned long rw = 0L;

  for (i = 0; i < 32; i++){
      if (b & 1)
        rw |= 1 << (31 - i);
      b >>= 1;
  }
  return (rw);
}

unsigned long
build_crc_table (int index)
{
  int i;
  unsigned long rb;

  rb = reflect_32 (index);

  for (i = 0; i < 8; i++){
      if (rb & 0x80000000L)
       rb = (rb << 1) ^ CRC32C_POLY;
      else
       rb <<= 1;
  }
  return (reflect_32 (rb));
}

main ()
{
  int i;

  printf ("\nGenerating CRC-32c table file <%s>\n", OUTPUT_FILE);
  if ((tf = fopen (OUTPUT_FILE, "w")) == NULL){
      printf ("Unable to open %s\n", OUTPUT_FILE);
      exit (1);
  }
  fprintf (tf, "#ifndef __crc32cr_table_h__\n");
  fprintf (tf, "#define __crc32cr_table_h__\n\n");
  fprintf (tf, "#define CRC32C_POLY 0x%08lX\n", CRC32C_POLY);
  fprintf (tf, "#define CRC32C(c,d) (c=(c>>8)^crc_c[(c^(d))&0xFF])\n");
  fprintf (tf, "\nunsigned long  crc_c[256] =\n{\n");
  for (i = 0; i < 256; i++){
      fprintf (tf, "0x%08lXL, ", build_crc_table (i));
      if ((i & 3) == 3)
```

```
        fprintf (tf, "\n");
  }
   fprintf (tf, "};\n\n#endif\n");

  if (fclose (tf) != 0)
    printf ("Unable to close <%s>." OUTPUT_FILE);
  else
    printf ("\nThe CRC-32c table has been written to <%s>.\n",
      OUTPUT_FILE);
}



/* Example of crc insertion */

#include "crc32cr.h"

unsigned long
generate_crc32c(unsigned char *buffer, unsigned int length)
{
  unsigned int i;
  unsigned long crc32 = ~0L;
  unsigned long result;
  unsigned char byte0,byte1,byte2,byte3;

  for (i = 0; i < length; i++){
      CRC32C(crc32, buffer[i]);
  }
  result = ~crc32;

  /*  result  now holds the negated polynomial remainder;
   *  since the table and algorithm is "reflected" [williams95].
   *  That is,  result has the same value as if we mapped the message
   *  to a polynomial, computed the host-bit-order polynomial
   *  remainder, performed final negation, then did an end-for-end
   *  bit-reversal.
   *  Note that a 32-bit bit-reversal is identical to four inplace
   *  8-bit reversals followed by an end-for-end byteswap.
   *  In other words, the bytes of each bit are in the right order,
   *  but the bytes have been byteswapped.  So we now do an explicit
   *  byteswap.  On a little-endian machine, this byteswap and
   *  the final ntohl cancel out and could be elided.
   */

  byte0 = result & 0xff;
  byte1 = (result>>8) & 0xff;
  byte2 = (result>>16) & 0xff;
  byte3 = (result>>24) & 0xff;
  crc32 = ((byte0 << 24) |
           (byte1 << 16) |
           (byte2 << 8)  |
           byte3);
  return ( crc32 );
}

int
insert_crc32(unsigned char *buffer, unsigned int length)
{
  SCTP_message *message;
  unsigned long crc32;
  message = (SCTP_message *) buffer;
  message->common_header.checksum = 0L;
  crc32 = generate_crc32c(buffer,length);
  /* and insert it into the message */
  message->common_header.checksum = htonl(crc32);
  return 1;
}

int
validate_crc32(unsigned char *buffer, unsigned int length)
{
  SCTP_message *message;
  unsigned int i;
  unsigned long original_crc32;
  unsigned long crc32 = ~0L;

  /* save and zero checksum */
  message = (SCTP_message *) buffer;
  original_crc32 = ntohl(message->common_header.checksum);
  message->common_header.checksum = 0L;
  crc32 = generate_crc32c(buffer,length);
  return ((original_crc32 == crc32)? 1 : -1);
}
```

Authors’ Addresses

Jonathan Stone
Room 446, Mail code 9040
Gates building 4A
Stanford, Ca 94305

EMail: jonathan@dsg.stanford.edu

Randall R. Stewart
24 Burning Bush Trail.
Crystal Lake, IL 60012
USA

EMail: rrs@cisco.com

Douglas Otis
800 E. Middlefield
Mountain View, CA 94043
USA

EMail: dotis@sanlight.net

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.