Improving the Efficiency of the OSI Checksum Calculation.

Keith Sklower

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

ABSTRACT

It is known that using larger byte-sizes to access memory usually results in faster computations of checksum algorithms. This paper proposes two different ways to use larger byte-sizes to improve the performance of the OSI checksum. First, an algorithm is presented that computes the 8–bit checksum using 16–bit integers. It is shown that this algorithm yields a 5 to 20 percent performance improvement on many architectures. Second, the benefits of expanding the basic computation unit of the OSI checksum algorithm to 16–bits integers is considered. This change can yield an additional performance improvement of up to 50% and greatly extended error detection properties, although it is incompatible with the current standard. The measurements of these algorithms are compared with some taken of checksums in common use, such as IP and XNS†.

1. Introduction.

The computing community in the world at large is slowly ratifying agreements for international standards for the exchange of information over networks. Such agreements sometimes are partly political in nature, and one may be faced with the task of computing unpleasant quantities. The checksum chosen for the first round of these international standards ([IS86], [IS87], for example) was first proposed by John Fletcher [Fl82] as a way of providing the same order of protection as the more computationally expensive CRC algorithm. It has some interesting error detection properties, but is still somewhat expensive to compute, and has a decided impact on the total throughput of these protocols [Mc87]. Thus, even a modest improvement of five to ten percent in the speed of checksum computation conceivably could translate to a four to eight percent increase in transport throughput.

More specifically, for a sequence of bytes b_1 through b_n, Fletcher would have us perform the following iterative calculation:

$$C_{0,k+1} = C_{0,k} + b_k$$

$$C_{1,k+1} = C_{1,k} + C_{0,k}$$

with initial conditions $C_{0,0} = C_{1,0} = 0$. One can easily show that these computations have the closed form:

This work was sponsored in part by Xerox Corporation, and in part by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871 monitored by the Space and Naval Warfare Systems Command under contract No. N00039-84-C-0089.

†XNS is a trademark of Xerox Corporation.

$$C_0 = \sum_{i=1}^{n} b_i$$

$$C_1 = \sum_{i=1}^{n} (n + 1 - i) b_i \ .$$

Fletcher analyzes the protection provided by these quantities (which he calls ‘‘check-bytes’’) for detecting errors under quite general circumstances, for bytes consisting of K bits (not just eight), and in fact considers higher order quantities, which we will not make use of. The ISO committees essentially have adopted the case $K = 8$ and the two check-bytes C_0 and C_1 as providing the basis for their checksum. More specifically, they allow space in the packets for two contiguous bytes to be chosen so that when the two quantities C_0 and C_1 are computed in 8–bit, one’s complement arithmetic for the packet as a whole, both sums are 0.

Our paper has two purposes: first, to discuss additional techniques for mustering as much computational efficiency as we can in computing these quantities; second, to propose a modification to this algorithm, which can double computational efficiency and greatly improve error detection properties.

2. The 8–Bit Algorithm.

The 4.3 Berkeley Software Distribution of UNIX† provides implementations of two communications protocols in common use, IP and XNS. The IP and XNS checksum routines supplied employ four techniques for reducing the time required for calculation. In papers analyzing Fletcher’s checksum algorithm (those cited above, [Na88] and [Co87]), we have found references to three of these. First: one’s complement arithmetic can be done by using native two’s complement arithmetic for some number of iterations known not to generate any carries, followed by a reduction step; second, reduction from 32– to 16– or 8– bit arithmetic can be done by merely adding up the constituent halfwords or bytes; and third, unrolling loops can contribute a substantial reduction in processing time.

Additionally, we propose that less obvious or slightly more complex iteration algorithms that access two bytes of memory at a time instead of one may provide additional efficiency for some CPU’s and cache architectures. (Our inspiration for this is the Internet checksum implementation in 4.3BSD, which references memory in 4–byte accesses instead of two.)

These four techniques (and others) are discussed at length in recent papers on computing the IP checksum ([Br89], [Pl89]). The most commonly available reference on the XNS protocols ([Xe81]), describes the checksum algorithm without formulas, so we will present a brief analysis of the XNS checksum in Appendix A for the insatiably inquisitive.

Computer architectures can be distinguished by the manner in which pairs or quadruples of bytes in memory serve as arithmetic operands in ALU’s. In a ‘‘Big-Endian’’ byte-addressible machine, when a pair of bytes b_n and b_{n+1} is fetched from memory, the 16–bit arithmetic quantity $256b_n + b_{n+1}$ is used as the value. By contrast, a ‘‘Little-Endian’’ machine will let the byte with the higher address have higher significance: the quantity $b_n + 256b_{n+1}$ is used. We’ll discuss computations on with Big-Endian machines; modifications for Little-Endians are simple and do not affect the analysis.‡ Initially, our discussion is directed at computing Fletcher’s quantities We’ll also limit ourselves to packets with even numbers of bytes. Suppose we have a sequence of bytes in memory:

$$a_1 \quad b_1 \quad a_2 \quad b_2 \quad \cdots \quad a_n \quad b_n$$

As a notational aid, let capital letters (A_i) denote 256 times the contents of the memory byte denoted by the small letter (a_i). Instead of computing C_0 and C_1 directly, we will compute two

†UNIX is a registered trademark of AT&T Bell Laboratories in the USA and other countries.

‡ The term Low-End and High-End are also used; Big-Endian appears to have originated with Danny Cohen, in Internet Engineering Note 137.

quantities S_i and T_i, which have the same values when reduced modulo 255. They are computed iteratively by:

$$S_i = S_{i-1} + (A_i + b_i)$$
$$T_i = T_{i-1} + S_{i-1} + S_i + A_i ,$$

or, equivalently,

$$T_i = T_{i-1} + 2S_i - b_i ,$$

with initial conditions $S_0 = T_0 = 0$, and where $(A_i + b_i)$ is the result of fetching two bytes of memory. Having a closed form expression for S and T will both help us to understand why they are equivalent to Fletcher's check-bytes, and to determine how many iteration steps may be taken before a carry can occur. The expression is not elegant, but can be easily verified by induction:

$$S_n = \sum_{i=1}^{n} A_i + \sum_{i=1}^{n} b_i$$

$$T_n = \sum_{i=1}^{n} 2(n + 1 - i)A_i + \sum_{i=1}^{n} (2n + 1 - 2i)\ b_i .$$

Since multiplication distributes over addition and 256 is congruent to 1 mod 255, it is not hard to see that S reduces to C_0. Persistence and diligence in re-arranging terms will convince the reader that T reduces to C_1.

Since each term a_i and b_i is no bigger than 255, and by using closed form expressions for $1 + 2 + 3 + \cdots + n$ and $1 + 3 + 5 + \cdots + n$, we have

$$T_n \le 255 * \left(256n(n + 1) + n^2 \right) ,$$

so that $T_{255} \le 4278059775 < 2^{32}$. Consequently, no overflow can occur in 255 iterations, starting with T_0 being 0. This bound is sharp, since in the case where all coefficients are 255, $T_{256} = 4311613440 > 2^{32}$.

In an actual implementation, we will have to compute the values S and T for packets longer than 255 bytes. Every so often, we *''fold''* the quantities S and T by replacing each by the value obtained from adding the upper and lower 16 bits in its two's complement representation. (Folding a quantity has no effect on the value it represents modulo 65535). The results of a folding operation are bounded by the maximum values for S and T after two iterations, so no carries can occur if an additional 253 iteration steps are made after a folding operation.

A C language implementation of our induction step might look like this:

```
unsigned short *wp;
long S, T, upper;

upper = *wp++;
T += S;
S += upper;
T += S;
upper &= 0xff00;
T += upper;
```

On CPU's such as the DEC VAX, the CCI power 6, and the Motorola 68000†, this sequence requires six machine instructions per two bytes of data, which is admittedly the same instruction count as computing C_0 and C_1 in the standard way:

```
unsigned char *cp;
long C0, C1, Reg;

Reg = *cp++;
C0 += Reg;
C1 += C0;
Reg = *cp++;
C0 += Reg;
C1 += C0;
```

Although the instruction count is the same, the very fact of accessing memory in words may be faster for some machine architectures, as our experiments described below will show.

On the DEC VAX [DE81] and on the CCI Power 6 [Ha87], which both have a special ‘‘double indexed’’ addressing mode, it is possible to perform the addition of twice S to T in a single instruction, "move address of word". This reduces the instruction count from 6 per pair of bytes to 5 per pair of bytes. Here the induction step would look like:

```
movzwl  n(r11),r8;
addl2   r8,r10;
movaw   (r9)[r10],r9;
andl2   $255,r8;
subl2   r8,r9
```

There remain two minor considerations: auto-increment, and loop control for reduction back to 16 bits. The CCI machine does not support auto-increment addressing, so, when the induction step is unrolled, one can use register displacement addressing with different displacements for each step unrolled. Although the VAX family of computers supports auto-increment, it appears that certain processors in the series compute this checksum more rapidly by using the method for the CCI machine, ignoring auto-increment.

For periodic reductions to 16 bits (to prevent overflow), it is convenient to test whether to reduce at the end of the largest unrolled loop, which in our implementation passes through the data 32 bytes at a time. Although one might be tempted to decrement a counter each time through the end of the unrolled loop (thus saving reductions for every sixth pass), it turns out to be 2 percent faster to check two bits for simultaneously being zero in the register counting the number of bytes remaining; even though this causes a reduction operation every fourth pass, the test is quite cheap, there is no overhead in resetting the counter, and the reduction itself is not expensive, merely being a store, two loads and an add.

3. The 16–Bit Algorithm.

Fletcher’s original paper carries out an error analysis of his algorithm for arbitrary K bit bytes, not necessarily just for $K = 8$. We would like to suggest that it may be profitable for future versions of OSI protocols to use the algorithm with $K = 16$. We have determined by measurement that the computation is significantly less costly, and show here that it has dramatically better error detection properties.

An interesting feature of either checksum is that for sufficiently small packets, it will detect all double bit errors. Fletcher gives the bounding size as $2^K - 1$ ‘‘bytes’’. In the 8–bit case, this gives 255 bytes for the packet size, somewhat less than conventional Ethernet packet sizes, for

†DEC and VAX are trademarks of Digital Equipment Corporation; CCI and Power 6 are trademarks of Computer Consoles, Incorporated; SUN is a trademark of Sun Microsystems.

large data transfers. In the 16–bit case, this gives us 64K 16–bit bytes or 128K 8–bit bytes. Doubling the size of the bytes generally squares the probability of other sorts of undetected errors: the fraction of all undetected errors is on the order of $2.37 * 10^{-10}$ (as opposed to $1.58 * 10^{-5}$ in the 8–bit case). The probability of undetected 32 bit burst errors is on the order of 2^{-40} (compared with a probability of undetected 16–bit burst errors of 2^{-20} in the 8–bit case). These numbers are computed from the formulas given in Fletcher's paper.

A precise statement of the algorithm would be as follows. If the packet is of an odd number of 8–bit bytes, logically extend it with a zero byte. One then computes the two sums C_0 and C_1 using 16–bit one's complement arithmetic, thinking of each pair of bytes as a 16–bit number. The packet has been properly prepared and transmitted if the two sums are zero.

We wish to show that the verification procedure is independent of the ‘‘Endian-ness’’ of the machine. The effect of the computation being performed by a machine of the opposite Endian-ness is the same as reversing the order of every pair of bytes in the packet. We state that the 16–bit checksum of a byte swapped packet is the same as swapping the bytes of the natively computed checksum for the original packet. The proof hinges on two observations: first, if we think of two bytes as representing a number modulo 65535, and we multiply the number by 256, the resulting number modulo 65535 is represented by the two bytes in the opposite order; second, multiplication distributes over sums and commutes with the multiplicative ‘‘weights’’ in sum C_1. Clearly, byte-swapping zero gives zero.

As noted earlier, the way the checksum algorithm would be likely to be employed in OSI protocols would be to have a header option, namely a type byte, length byte, and four contiguous bytes whose values would be chosen so that both check–quantities (now 16 bits each, giving us 32 total) would be zero. It is not hard to see that if the four bytes were word-aligned, the same formal computation that generates the present checksum option (for 8–bit Fletcher) would work for our replacement checksum.

Nonetheless, it is still possible to choose values for the four bytes, even in the case of odd positioning, so that the two sums C_0 and C_1 will be zero. Using the naming scheme in the previous section, let us first assume we are to adjust a sequence

$$b_{k-1} \quad a_k \quad b_k \quad a_{k+1} \ .$$

We first replace all four bytes by zero, then compute

$$C_0 = \sum_{i=1}^{n} w_i$$

$$C_1 = \sum_{i=1}^{n} (n + 1 - i) w_i \ ,$$

where each w_i is obtained by adding $A_i + b_i$ as a one's complement 16–bit quantity in the Big-Endian case. This gives us two equations to satisfy:

$$C_0 + b_{k-1} + A_k + b_k + A_{k+1} = 0 \quad ,$$

$$C_1 + (k-1)b_{k-1} + kA_k + kb_k + (k+1)A_{k+1} = 0 \quad ;$$

subtracting k times the first equation from the second we get

$$C_1 - kC_0 - b_{k-1} + A_{k+1} = 0 \quad .$$

This is satisfied if we let b_{k-1} be the lower byte of $C_1 - kC_0$, and A_{k+1} be the one's complement of the upper byte. One can then get the other two quantities by substitution in the equation for C_0 .

4. Measurements and Comparisons

We prepared two timing tests to give some feeling for the way checksum routines would compare under different limiting sorts of conditions. The tests were run typically on lightly loaded machines with large memories, but in off hours. Each measurement was taken a few times to establish reasonable credibility. By contrast, the measurements in [Na88] were made in a much more careful manner. Nonetheless, we believe that the data discussed here are sufficient to make qualitative comparisons.

The first test is the equivalent of checksumming a single 16 megabyte packet. This is designed to exercise mostly the inner unrolled loop of each algorithm. The second is checksumming 90 minutes worth of packets as found on one large Ethernet connecting SUN workstations to file servers from trace data giving the length and type of every packet [Gu87].

Most of the traffic on the workstation network consisted of paging and non-checksummed file system references, which we exclude entirely in our test. The other packets were largely single character virtual terminal packets, or acknowledgements, with about a quarter of the remaining packets being file transfers, and mail activity. We discarded the paging and file activity (comprising 60 per cent of the network traffic) as being as yet relatively uncommon in communications over long-haul networks.

Under realistic conditions, if we had a dedicated mail gateway, although most of the data packets would be large, there would likely to be about as many acknowledgement packets as data packets, and acknowledgement packets are typically quite small. Thus, one would not expect to see a mix of more than 50 percent of large packets.

We compared the following routines:

IP	The standard IP checksum routine for the machine.
PIP	A ‘‘portable’’ IP checksum routine accessing memory a word at a time.
XNS	The standard XNS checksum routine for the machine.
simple	A naive OSI implementation (one byte at a time), but with loop unrolling.
NoMagic	Our OSI implementation (two bytes at a time), but without the ‘‘movaw’’ instruction
Magic	Our OSI implementation (two bytes at a time), with the ‘‘movaw’’ instruction.
Ediv	The OSI implementation given in [Mc87] (no unrolling, using ediv, aob, etc.)
F16	A 16–Bit Fletcher Checksum Routine.
F16-NOAI	A 16–Bit Fletcher Checksum Routine without auto-increment.
null	A routine merely returning 0, to measure overhead in the second test.

Each of these routines includes the mechanism (and consequent overhead) to process packets buffered in a segmented fashion. An anonymous reviewer of an earlier draft of this paper calls this ‘‘the curse of 4BSD: mbufs’’; an amusingly described but regrettably pertinent detail that must be considered when trying to optimize the last iota out of checksum algorithms.

Our test bed included the following machines: CCI Power 6, Vax 8800, Vax 8600, Vax 785, Vax 750, and Sun 3/280. We will present all the raw data collected, in Tables 1 showing the amount of cpu time in seconds to perform the the single huge checksum, and Table 2 showing the amount of cpu time in seconds to checksums of approximately 250,000 packets of varying and more reasonable sizes, respectively. We will then draw some comparisons using more specific tables, so that it should not be necessary for the reader to memorize the first two completely.

TABLE 1. Times to Checksum a Single Huge Packet.

Algorithm	Machine Type					
	CCI	8800	8600	785	750	Sun
IP	2.6	1.5	1.9	4.9	11.5	2.3
PIP	5.0	3.3	4.1	11.2	35.7	4.2
XNS	7.2	5.7	7.7	15.0	41.6	5.2
Simple	10.2	8.5	10.7	24.2	81.8	10.6
NoMagic	8.9	8.0	9.7	25.0	79.8	9.9
Magic	11.5	7.0	9.6	29.4	84.1	†
Ediv	†	13.6	18.6	34.4	95.5	†
F16	†	6.8	7.8	19.4	62.5	5.0
F16-NOAI	6.2	4.8	6.4	15.2	48.8	5.5

TABLE 2. Times for checksuming a variety of packets.

Algorithm	Machine Type					
	CCI	8800	8600	785	750	Sun
IP	13.2	10.3	12.8	37.5	90.8	14.3
PIP	16.7	14.5	18.4	52.0	148.0	19.6
XNS	22.8	18.7	24.7	58.7	154.2	19.0
Simple	31.3	26.1	35.6	86.0	273.3	36.0
NoMagic	28.7	26.2	32.0	88.5	265.1	35.1
Magic	33.3	26.2	32.3	96.6	263.8	†
Ediv	†	35.6	49.9	101.8	263.6	†
F16	†	22.1	26.6	74.9	217.5	23.5
F16-NOAI	21.7	18.3	24.2	65.6	190.0	23.5
Null	1.8	1.8	3.4	11.3	23.4	2.5

As one might guess, one sees less dramatic differences between protocol checksum times in the mixed packet test than in the large packet test, due to increased overhead in the latter. Here, for a single CPU (the DEC VAX 8800) we have the following ranges of times:

TABLE 3. Big versus Mixed CPU Time for 8800.

Test	Algorithm								
	IP	PIP	XNS	Simple	Nomagic	Magic	Ediv	F16	F16-NOAAI
Big	1.5	3.3	5.7	8.5	8.0	7.0	13.6	6.8	4.8
Mixed	10.3	14.5	18.7	26.1	26.2	26.2	35.6	22.1	18.3

We notice that the ratio of slowest time to fast time in the big packet test is about 9.1 whereas in the mixed packet test it is only about 3.4. A similar amelioriation occurs across all machines:

† Machine lacks addressing mode or special instruction.

TABLE 4. Ratios of Slowest to Fastest Algorithm by machine.

Test	Machine Type					
	CCI	8800	8600	785	750	Sun
Big	4.4	9.1	9.7	6.7	6.9	4.6
Mixed	2.5	3.4	3.8	2.7	3.0	2.5

The regular OSI checksum can be made to run only two and a half times more slowly than the IP checksum in the case of a realistic mix of packets, even though it takes longer by a factor of at least 4.4 for very large packets. It is amusing to note that using the special machine-specific instruction pays off only on extremely sophisticated CPU's, and then only for large packets. In the following table we note which was the best OSI algorithm for the given machine type and test; the ratio of OSI to IP performance; and give the per cent improvment of the best OSI algorithm for that machine and test over the simple OSI implementation.

TABLE 5. Ratios of Best OSI to IP by machine.

Test	Machine Type					
	CCI	8800	8600	785	750	Sun
Big/Best	NoMagic	Magic	Magic	Simple	NoMagic	NoMagic
OSI/IP	3.4	4.7	5.0	4.9	8.0	4.3
Marginal%	14.	21.	11.	0.	3.	7.
Mixed/Best	NoMagic	Simple	NoMagic	Simple	Ediv	NoMagic
OSI/IP	2.1	2.5	2.5	2.3	2.9	2.4
Marginal%	9.	0.	11.	0.	.5	2.

Almost in every case, the extended Fletcher algorithm is comparable to the XNS checksum, though, at best, it runs twice as slowly as an optimized IP checksum for large packets. Furthermore, the simple act of writing the C code for the extended Fletcher algorithm in such a way to avoid having auto-increment addressing generated, seems to made an improvement in the efficiency on the same order that our proposed technique for improving the existing algorithm has. In our final table, we provide the ratios of times for extended Fletcher over IP, XNS over ‘‘new’’ Fletcher, and percentage of marginal improvement by choosing the best auto-increment strategy for both big and small packet tests.

TABLE 6. Ratios of Extended Fletcher to IP and XNS by machine.

Test	Machine Type					
	CCI	8800	8600	785	750	Sun
BigF/IP	2.4	3.2	3.3	3.1	4.2	2.1
XNS/F	1.1	1.2	1.2	.99	.85	1.04
Marginal%	†	40.	22.	27.	28.	-10.
MixedF/IP	1.6	1.8	1.9	1.7	2.1	1.6
XNS/F	1.05	1.02	1.02	.89	.81	.80
Marginal%	†	21.	9.	14.	14.	0.

5. Conclusions.

We have proposed some alternative techniques for computing the OSI checksum. As we have seen, not every architecture makes it worthwhile to trade more numerous or complicated instructions instead of fewer references to data.

We have proposed a related checksum algorithm which has significantly better error-detection properties. We showed that this checksum presents no special burden on machines of differing ‘‘Endian-ness’’. Our data shows that for every machine, the proposed checksum was faster to compute than the existing one, up to a factor of two for large packets.

For those architectures where it is faster to use more complicated instructions and reference memory less often, one might ask, if two bytes at a time is a good thing for the standard OSI checksum, might not four be even better? Unfortunately, we have not been able to devise an induction step of sufficiently few instructions to make this pay off, as two instructions are required to perform a 64–bit add on most of the machines at our disposal. However, we are tantalized by the prospect of other machines having single instruction 64–bit arithmetic, or possibly even using floating point arithmetic to get us a one-instruction 53–bit (i.e. mantissa-sized) register.

Lastly, we have proposed an alternative checksum algorithm which has significantly better error detection properties than any of the existing three, and is far from the worst among them in computational requirements.

6. References

[DE81] Digital Equipment Corporation, *VAX Architecture Handbook*, Digital Press, 1981.

[Xe81] Xerox Corporation, *Internet Transport Protocols*, Xerox System Integration Standard 028112, 1981.

[Fl82] Fletcher, J., ‘‘An Arithmetic Checksum for Serial Transmissions’’ *IEEE Trans Commun.*, Vol. COM-30, No. 1, January, 1982, pp. 247-252.

[IS86] International Organization for Standardization, ‘‘Connection oriented transport protocol specification’’, *International Standard ISO 8073–1986 (E)*.

[Gu87] Gusella, R., ‘‘The Analysis of Diskless Workstation Traffic on an Ethernet’’, Technical Report No. UCB/CSD 87/379, Computer Science Division (EECS), University of California, Berkeley.

[Ha87] Harris Corporation, *HCX-7 and HCX-9 Architecture*, Reference Manual, Pub. No. 0830022-100, Change 2, reissue 1.

[IS87] International Organization for Standardization, ‘‘Protocol for providing the connectionless-mode network service’’, *Draft International Standard ISO 8473*, ISO/TC 97/SC 6 N4542.

[Co87] Cockburn, A., ‘‘Efficient Implementation of the ISO Transport Protocol Checksum’’ *ACM Comp. Commun. Rev.*, Vol. 17, No. 3, July/August, 1987, pp. 13-20

[Mc87] McCoy, W., RFC 1008. ‘‘Implementation Guide for the ISO transport protocol.’’

[Na88] Nakassis, T., ‘‘Fletcher’s Error Detection Algorithm: How to implement it efficiently and how to avoid the most common pitfalls’’, *ACM Comp. Commun. Rev.*, Vol. 18, No. 5, October, 1988, pp 63-88.

[Br89] Braden, R., Borman, D., Partridge C., ‘‘Computing the Internet Checksum’’, *ACM Comp. Commun. Rev.*, Vol. 19, No. 2, April, 1989, pp 86-94.

[Pl89] Plummer, W. W., ‘‘IEN-45: TCP Checksum Function Design’’, *ACM Comp. Commun. Rev.*, Vol. 19, No. 2, April, 1989, pp 95-101.

7. Acknowledgements

The author wishes to acknowledge careful reading of drafts, numerous suggestions for improvements and general encouragement of the following people (among others): Lawrence Landweber, Two unnamed reviewers for Sigcomm 88, Riccardo Gusella, Domenico Ferrari, Tassos Nakassis, Van Jacobsen, and the Editor, Craig Partridge.

8. Appendix A -- The XNS Checksum Algorithm

The definition given for the checksum in [Xe81] (pp 19-20.) is

> ''an optional ones complement add-and-left-cycle (rotate) of all the 16–bit words of the internet packet excluding the checksum word itself.''

We note that multiplication by 2 modulo $2^{16}-1$ has the same effect as left cycling a 16–bit quantity. Horner's rule for evaluating a polynomial in X is precisely an iterative process of multiplying by the fixed quantity X, and adding coefficients; this action is very similar (with $X = 2$).

If we have a sequence of 16–bit words

$$w_0 \quad w_1 \quad w_2 \quad \cdots \quad w_n$$

We are instructed to compute

$$S_i \;=\; 2(S_{i-1} + w_i) \quad \text{with} \quad S_0 = 0$$

The closed form solution can be shown to be

$$w_0 \;=\; S_n \;=\; \sum_{i=1}^{n} 2^{n-i+1}\, w_i \qquad (mod 2^{16}-1)$$

Each iteration step can add at most two high bits of significance. The implementation in 4.3BSD performs eight iteration steps before reducing the partial sum back to a 16–bit quanitity, (and thus avoids bit rotations altogether). The closed form solution can be used to show that the XNS checksum is independent of endian-ness: multiplication by 2^8 has the same effect as swapping bytes, distributes over sums and commutes with mutliplication by other powers of 2.

In fact, a little more care in the analysis (including the contribution of a non-zero S_0 as $2^n S_0$, bounding each w_i, and summing the powers of 2), would show that one could perform 14 iteration steps without overflow; however whether this translates to any actual savings is left as a measurement for the reader!

9. Appendix B -- Sample Checksum routine (Little-Endian Version)

```
/*
 * Copyright (c) 1988, 1989 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms are permitted
 * provided that this notice is preserved and that due credit is given
 * to the University of California at Berkeley. The name of the University
 * may not be used to endorse or promote products derived from this
 * software without specific prior written permission. This software
 * is provided ``as is´´ without express or implied warranty.
 *
 *      @(#)os_cksum.c   1.2 (Berkeley) 6/2/89
 */
#include "types.h"
#include "mbuf.h"

/*
 * Checksum routine for OSI Protocol family headers (VAX Version).
 *
 * This routine is very heavily used in the network
 * code and should be modified for each CPU to be as fast as possible.
 */

os_cksum(m, len)
```

```
                register struct mbuf *m;
                int len;
{
                register u_short *w;
                register int sum = 0;
                register int alt = 0;
                register int lower;
                register int mlen;

                union {
                                u_char          c[2];
                                u_short         s;
                } s_util;
                union {
                                u_short s[2];
                                long            l;
                } l_util;

#define ADDCARRY(x)   (x > 65535 ? x -= 65535 : x)
#define FOLD(x) {l_util.l = x; x = l_util.s[0] + l_util.s[1];}
#define REDUCE(x) {FOLD(x); ADDCARRY(x);}
#define BYTECARRY(x)   (x > 255 ? x -= 255 : x)
#define BYTEFOLD(x) {s_util.s = x; x = s_util.c[0] + s_util.c[1]; BYTECARRY(x);}

                for (;m && len; m = m->m_next) {
                                if (m->m_len == 0)
                                                continue;
                                w = mtod(m, u_short *);
                                mlen = m->m_len;
                                if (len < mlen)
                                                mlen = len;
                                len -= mlen;
                                /*
                                 * Force to even boundary.
                                 */
                                if (1 & (int) w) {
                                                sum += *(u_char *)w;
                                                alt += sum;
                                                w = (u_short *)((char *)w + 1);
                                                mlen--;
                                }
                                /*
                                 * Unroll the loop to make overhead from
                                 * branches &c small.
                                 */
#define STEP(n) {lower = w[n]; alt += sum; sum += lower; alt += sum; \
                                                (lower &= 0xff); alt += lower;}

                                while ((mlen -= 32) >= 0) {
                                                STEP(0); STEP(1); STEP(2); STEP(3);
                                                STEP(4); STEP(5); STEP(6); STEP(7);
                                                STEP(8); STEP(9); STEP(10); STEP(11);
                                                STEP(12); STEP(13); STEP(14); STEP(15);
                                                w += 16;
```

```
                        if ((mlen & 96) == 0) {FOLD(alt); FOLD(sum);}
                }
                mlen += 32;
                while ((mlen –= 8) >= 0) {
                        STEP(0); STEP(1); STEP(2); STEP(3);
                        w += 4;
                }
                mlen += 8;
                while ((mlen –= 2) >= 0) {
                        STEP(0); w++;
                }
                if (mlen == –1) {
                        sum += *(u_char *)w;
                        alt += sum;
                }
                FOLD(alt);
                FOLD(sum);
        }
        if (len)
                printf("cksum: out of data\n");
        REDUCE(alt); BYTEFOLD(alt);
        REDUCE(sum); BYTEFOLD(sum);
        return ( (alt << 8) + sum);
}
```