MEMORANDUM

To: IMP Guys

From: J. McQuillan

Subject: Software Checksum on All IMP/IMP Transmissions

Date: 12 March 1973

IG #15

During the last few weeks of intensified work on the problem of network reliability, it has become increasingly apparent that the present network programs are lacking in two fundamental areas: fault tolerance and diagnosis. The IMP can be made more reliable and also more useful in fault isolation, but a large effort is required. As an important first step, I investigated the technique of generating and verifying a checksum on all IMP transmissions, in software. I have devised an implementation which is attractive enough to merit inclusion in the IMP on a long-term basis.

1. Checksum on all IMP-to-IMP Transmissions

The recently discovered data errors in the network (see IMP Guys Note #14) make it imperative to check the modem interfaces, the DMCs, and the memories in the IMPs. The key problem with including a checksum calculation and verification in the store-and-forward loop of the IMP is speed. That loop is designed to be as fast as possible, and checksums take a long time to generate. The processing bandwidth of the IMP for store-and-forward traffic can be computed as follows:

- the program time for Modem Input, TASK, and Modem Output = 550 cycles

- the I/O time for a packet = 8 cycles/word for 516, 4 cycles/word for 316

These figures are for sending a packet from one IMP, receiving it at the IMP, and processing it (including acknowledge overhead which is piggybacked). A full length packet is about 1000 data bits and takes 67 words for the I/O transfers. Thus, it takes the 516 about 1 ms to process 1000 bits, for a bandwidth of 1 megabit per second, and the 316 has a bandwidth of about 750 Kbs. In other words, the 516 IMP can handle twenty (20) 50 Kbs lines full duplex, in terms of processing time.

The fastest way to add a table of numbers in a loop on the 516 or 316 is to use the index register as a counter and add indirect through a post-indexed pointer:

```
          ADD  PNTR I
          IRS  Ø
          JMP  .-2
```

where PNTR is TABLE + TABLELENGTH X and the X starts at -(TABLE-LENGTH) and is incremented to zero. This loop takes 7 cycles per word. Including it on the IMP store-and-forward loop adds 14 cycles per word, cutting processor bandwidth by half on the 516 and by even more on the 316.

An alternative is to code the add loop on a straight line. The number of words in a packet, including header is 67, and an add chain determines the length of the packet, sets up the index register to point to the packet, and jumps into the sequence at the appropriate place, adding the last word of the packet first, and ending with the first word:

```
ADD  67 X
ADD  66 X
      .
      .
      .
ADD  1 X
ADD  Ø
JMP  RETURN
```

This technique costs only 2 cycles per word, since pre-indexing is free on the X16. Note that this is nearly as fast as special-purpose hardware can run!

One drawback to this approach is that the code takes a lot of memory, about a buffer's worth. Both Modem Input and Modem Output need to perform checksums. Two ways to permit this are to provide separate code for each program, which is expensive in storage, or to have each program lock the others out, which is slow. It turns out to be very easy to escape this dilemma by making the add chain reentrant, so that any number of interrupt programs can use it without locking each other out. The state of any process using the add chain is given by the three registers (program counter, accumulator, index register) which are saved and restored by each process, and by the identity of the owner of the add chain. Thus, the end of the chain can be JMP RETURN 1, and each process can set up RETURN for itself, saving the current lower priority owner it interrupted, and then restore RETURN when it is finished. Then arbitrarily many levels of interrupts can share the same add chain without locking each other out.

It is possible, then, to provide a software checksum on the IMP store-and-forward path for a reasonable cost in speed and storage. The output program adds on a software-generated checksum at the end of each packet, which includes every header and data word, just before issuing the output instruction. Likewise, the input routine verifies the checksum on each packet before it begins to examine the header. This checks the two modem interfaces and DMC transfers. It also checks the memory of the sending IMP. (A way to check the memory of the receiving IMP is discussed below.) This approach costs 4 cycles per word, which reduces the store-and-forward processing bandwidth of the 516 IMP to 750 Kbs and that of the 316 IMP to 600 Kbs, or 15 and 12 fifty (50) Kbs lines respectively.

I am releasing a new version of the IMP program tomorrow, March 13, which includes a software checksum on all IMP/IMP transmissions. The change to the program cost only two buffers of storage.

2. Implementation Considerations

There are several advantages arising from the simple, straight-line code approach to checksum calculation. First, the code is read-only or pure and can be located on page 1, the protected page of 516 IMP memory. This saves a buffer of storage, since it does not take away from the buffer pool (page 1 cannot be used for buffers). Second, it provides an extra measure of reliability for this extremely important code. If a machine has the memory protect option, the code should never get changed. But even machines without memory protect can check to see if this code is correct. It is simple enough so that it could even be regenerated by the program! This may sound extreme, but if the add chain code fails, no messages can get in or out of the system, so precautions are worthwhile.

One issue to be raised is the question of how good this software checksum is. I have implemented a simple "sum of all words minus the length" checksum. The checksum equals the twos complement of [the sum of all words in the packet - number of words in the packet], and it is the last word in the packet. Thus, adding all the words in the packet, including the checksum, then subtracting the packet length (not including the checksum) should give zero. If a packet has errors which are all picked bits or all dropped bits, it will be detected as in error. Picked bits and dropped bits in the same packet will be detected if they do not occur in the same bit position, or otherwise cancel out. Missing words or words of zero are caught by including the length in the checksum. On balance, this checksum is probably good enough for now. We cannot get too much fancier in software. We could never replace the cyclic checksum in the modem interface, for instance. However, we might want to change the code to do an add and shift sequence which is a lot better but twice as slow.

The question now arises as to what we should do with a packet with a bad checksum. I think that we want to begin to expand the role of the NCC in fault isolation and diagnosis. The NCC machine could, with more c re and an additional output devise, log all occurrences of trouble on the network. This is conceptually an expansion of the trap messages from three numbers to a message of arbitrary length. Accordingly, I have built a new statistics program into the IMP, called DIAG, whose function is to send reports of malfunctions to a specified destination. At first, these will simply be a copy of a packet with a bad checksum, a spurious ack, or the like, along with the program counter, accumulator and index register which identify the error condition. The packet location in memory will also be included. This statistics program will be independent of NCC trouble reports. It uses the slot of the now-defunct arrival statistics. That is,

Parameter 4	= DIAG on/off
Parameter 13	= DIAG link
Parameter 21	= DIAG destination
Parameter 27	= DIAG frequency

Currently, the DIAG destination is set to be the TTY on the Tinker IMP which is located in the IMP room. The DIAG on/off flag is used by the program as the queue of packets to send, so DIAG is turned on when an error is detected, and turned off when all errors have been reported. The frequency is kept at zero, meaning DIAG is runnable whenever it is turned on, with no timed latency.

Another set of issues concers the release of the IMP program which generates and expects checksums into a network of IMPs running a program which does not. There are three differences in the two systems. The new system has a packet length one larger to include the checksum. It generates a checksum for each outgoing packet. And it verifies the checksum on each incoming packet. Therefore, the release should have the following three phases:

(1) The new system is propagated everywhere with:

- bigger packet buffers
- checksum generation disabled
- checksum checking disabled *except* for packets marked as "checksummed" with a special bit in the header (currently unused). These packets will be checked, and, if validated, the data end pointer, BUFE, will be decremented.
- all output packets marked as unchecksummed

(2) After the new system is running everywhere, change each IMP in turn to have:

- checksum generation enabled, with output packets marked as checksummed

(3) Finally, when all systems are generating checksums:

- checksum checking enabled unconditionally

The last step prevents the "checksummed" bit from serving as a one-bit checksum and admitting bad packets. Thus, the "check-summed" bit is only a temporary measure to facilitate the release. There is one complication: core reload requests. These messages can easily have the correct checksum assembled in, but the IMPLOD paper tapes in the field do not have a checksum for this message.

Therefore, steps (1) and (2) will happen on the Tuesday, March 13 release, and step (3) can happen (and the "checksummed" bit can go away) after new IMPLOD paper tapes are distributed to all sites. In the meantime, the program will check to see that an unchecksummed message is a reload request. If not, it will treat it as a checksum error and give the packet to DIAG. Therefore, it is actually this fast check which is enabled as a kind of temporary step 2.5.

3. Extensions to the IMP-to-IMP Checksum

After we have a software checksum on all IMP-to-IMP transmissions, it is natural to ask if we can get more. Consider the case of 3 IMPs:

$$\begin{tikzcd} A \arrow[r, dash] & B \arrow[r, dash] & C \end{tikzcd}$$

When A generates the checksum and sends the packet to B, B is able to verify that the packet did not change while being re-written into core after being read to calculate the checksum. It also verifies that the DMC read the same thing out, and that the interfaces did not change it at either end. It arrived in the memory of B exactly the same as it left the memory of A. Likewise, we can be certain that it will arrive in the memory of C just as it left the memory of B. But what about from the time it arrives in B to the time when it leaves B? It may change during the core re-write as the checksum is checked on input, it may be fetched incorrectly for checksum generation on out, or may be changed in between by almost anything. Clearly, one should check the output checksum on leaving B against the checksum that the packet carried on input. The only change made to the packet in the store-and-forward loop is to the acknowledge header, and therefore the change to this word can be compared to the change in the checksum. If they do not agree, a data error occurred while the packet was resident in the memory of B. Note that this extra calculation costs very little in speed or space, since the checksum must be generated afresh anyway.

At this point, it is only a single step to an end-to-end packet checksum. If one stipulates that the Host Input routine in the IMP generate a checksum for each packet it gives to TASK, and that the Host Output routine verify the checksum on each packet it sends to the Host, then we have an end-to-end checksum on packets. It has no time "gaps", or times when the data is not checked. It uses only 16 bits per 1000 bit packet. And it only costs 4 cycles per word per hop on the network path, 2 cycles on input, and 2 on output. Note also that it does not cost any extra code, since the add chain which performs the checksum is reentrant, and different interrupt levels may share it.

Clearly, we should look hard at this scheme with regard to the HSMIMP hardware design effort. It is not certain that the 516s and 316s in the network could ever be fitted with hardware to do a checksum job a great deal more efficiently than this software approach. It is certain that the 516s and 316s could never perform a fancy cyclic checksum in software to imitate some HSMIMP hardware. Perhaps a compromise can be reached.

In summary, the checksum scheme presented here answers both of the needs described at the beginning of this note. *The IMPs are much more reliable and fault-tolerant, since all IMP-to-IMP and source-to-destination transmission are checked.* Moreover, the IMP-to-IMP checking dovetails with the end-to-end checking so there are no time gaps in the path. *Therefore, since every packet is checked both when it enters an IMP and when it leaves, the IMPs provide a maximum of diagnostic information. We have both an end-to-end checksum on packets and a means for determining exactly where failures occur.*

JMCQ/nlg