Astronomical Data Analysis Software and Systems IV
ASP Conference Series, Vol. 77, 1995
R. A. Shaw, H. E. Payne, and J. J. E. Hayes, eds.

FITS Checksum Verification in the NOAO Archive

R. Seaman

National Optical Astronomy Observatories[1], *P. O. Box 26732, Tucson, Arizona 85726*

Abstract. There is no standard procedure for verifying the integrity of FITS data files. While a FITS file may be subjected to the same checksum or digital signature calculation as any other data file, the resulting sum or signature must normally be carried separately from the FITS file since writing the value into the header will change the checksum.

A simple method for embedding an ASCII coded 32 bit 1's complement checksum within a FITS header (or any ASCII text) is described that is quick to compute and has desirable features such as: the checksum of each FITS file or extension is set to zero; the checksum may be accumulated in any order; and the checksum is easily updated with simple arithmetic. On-line verification of tapes for the NOAO/IRAF *Save the Bits* archive is discussed as an example.

1. Introduction

There is no standard way to verify FITS files. Various checksums may be calculated for FITS as for other data, but the results must be kept separate from the FITS file since writing the value into the header will change the checksum.

There is a tradeoff between the error detection capability of an algorithm and its speed. The overhead of a digital signature or a cyclic redundancy check (CRC) may be prohibitive for multimegabyte files, and a CRC, tuned to be sensitive to the bursty nature of communication line noise, may not represent the best model for FITS bit errors.

A simple method of embedding an ASCII coded 32 bit 1's complement checksum within a FITS header is described. A 1's complement checksum (as used by TCP/IP) is preferable to a 2's complement checksum (as used by the UNIX `sum` command, for example), since overflow bits are permuted back into the sum and therefore all bit positions are sampled evenly. A 32 bit sum is as easy to calculate as a 16 bit sum because of this symmetry, providing greater sensitivity to errors. A binary to ASCII conversion (analogous to `uuencode`) allows writing the checksum, an unsigned integer, into a string valued FITS header keyword, such that the ASCII bytes sum four at a time. This method has several desirable features:

- The checksum of each FITS file is forced to zero by writing the complement of the calculated checksum into the header. Verifying a particular file requires only that the checksum computes to zero.
- Since 1's complement addition is commutative and associative, the checksum may be accumulated in any order.
- If a FITS header is changed, the checksum is updated with simple arithmetic. Only the checksum of keywords that change need be recalculated. A simple rearrangement of keywords leaves the checksum unchanged.

[1]NOAO is operated by AURA, Inc. under contract to the National Science Foundation.

- The checksum of the data records is written into a separate header keyword and is not recomputed unless the data are modified.
- The checksum for individual FITS extensions is separately preserved. Extensions may be added and removed at will from a larger FITS file without disturbing the checksum.

2. Algorithm

The 1's complement checksum is fast and simple to compute. A third of the following C code implementation handles odd length input records—a case that does not apply to FITS. Just zero **sum32** and step through the FITS records:

```
checksum (buf, length, sum32)
char *buf;
int length;                     /* < 2^18, or carry can overflow */
unsigned int *sum32;
{
        unsigned short *sbuf;
        unsigned int hi, lo, hicarry, locarry;
        int len, remain, i;

        sbuf = (unsigned short *) buf;
        len = 2*(length / 4);   /* make sure it's even */
        remain = length % 4;    /* add odd bytes below */

        hi = (*sum32 >> 16);
        lo = (*sum32 << 16) >> 16;
        for (i=0; i < len; i+=2) {
            hi += sbuf[i];
            lo += sbuf[i+1];
        }
        (remain >= 1) ? hi += buf[2*len] * 0x100;
        (remain >= 2) ? hi += buf[2*len+1];
        (remain == 3) ? lo += buf[2*len+2] * 0x100;

        hicarry = hi >> 16;     /* fold carry bits in */
        locarry = lo >> 16;
        while (hicarry || locarry) {
            hi = (hi & 0xFFFF) + locarry;
            lo = (lo & 0xFFFF) + hicarry;
            hicarry = hi >> 16;
            locarry = lo >> 16;
        }
        *sum32 = (hi << 16) + lo;
}
```

Encoding the unsigned integer checksum into an ASCII string is simply a matter of dividing each initial byte into four bytes—this permits each quarter of the original 8-bit byte to fit within the range of the ASCII alpha-numerics, including an offset from ASCII zero (hex 0x30).

```
unsigned exclude[13] = { 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f, 0x40,
                         0x5b, 0x5c, 0x5d, 0x5e, 0x5f, 0x60 };

int offset = 0x30;                      /* ASCII 0 (zero) */

char_encode (value, ascii)
unsigned int value;
char *ascii;
{
        int byte, quotient, remainder, ch[4], check, i, j, k;

        for (i=0; i < 4; i++) {
            byte = (value << 8*i) >> 24;    /* each byte becomes four */
            quotient = byte / 4 + offset;
            remainder = byte % 4;
            for (j=0; j < 4; j++)
                ch[j] = quotient;
            ch[0] += remainder;

            for (check=1; check;)           /* avoid ASCII punctuation */
                for (check=0, k=0; k < 13; k++)
```

```
                for (j=0; j < 4; j+=2)
                    if (ch[j]==exclude[k] || ch[j+1]==exclude[k]) {
                        ch[j]++;
                        ch[j+1]--;
                        check++;
                    }

            for (j=0; j < 4; j++)           /* assign the bytes */
                ascii[4*j+i] = ch[j];
        }
        ascii[16] = 0;
}
```

The basic idea is the same as used by the Internet checksum (Braden et al. 1988; Mallory & Kullberg 1990). See Stevens (1994) for an overview, and Zweig & Partridge (1990) for alternatives. An integer is embedded within each data packet (FITS header) which forces the checksum of the entire packet (FITS HDU) to zero. To find this integer, zero the checksum field in the packet and accumulate the checksum—the necessary value is just the complement (additive inverse) of the checksum.

In this case, the equivalent of zeroing the checksum field is to set the 16 character string value of the `CHECKSUM` keyword to all ASCII 0s (hex 0x30). The checksum is accumulated and complemented in the same fashion. The ASCII encoded complement of the checksum is written into the header replacing the ASCII 0s, which are in effect subtracted back out of the encoding to restore the original value. The checksum and its complement sum to zero. (Actually they sum to *negative* zero, all 1's—1's complement addition has two identity elements.)

Note that the checksum field must be integer aligned, whether the checksum is being stored as an integer or an encoded string. In either case, this requirement only applies byte-by-byte. To begin the string at an arbitrary odd byte offset, just permute the bytes. Note also that the same zeroing effect could be gained by embedding the complemented value in a comment as well as in a keyword.

3. Verification in the NOAO Archive

The NOAO/IRAF *Save the Bits* archive is described in Seaman (1994). Images from several telescopes on Kitt Peak are multiplexed onto tape as large FITS image extension files. As each image is processed, the checksum of the resulting FITS extension is forced to zero by writing its complement into the header:

```
XTENSION= 'IMAGE   '            /  FITS image extension
   ...                    ...                     ...

RECID   = 'kp09m.940909.082728' /  archive ID for observation
RECNO   =               318747  /  NOAO archive sequence number
CHECKSUM= ' cHjjc9ghcEghc9gh '  /  ASCII 1's complement checksum
DATASUM = ' 5ZNF4XME4XME4XME '  /  checksum of data records
END
```

As the tape files are assembled from the individual extension files, the checksum for the primary FITS header is zeroed. This zeroes the checksum for the entire multiple image file since each extension's checksum is the additive identity. After each tape (actually a duplicate pair) fills up, the archive takes the drive off-line and verifies the checksums.

The checksum of the data records is saved separately in the `DATASUM` keyword. This simplifies updating the checksum during subsequent header operations, as when an image is later extracted from the archive. Simple arithmetic suffices to recalculate the checksum no matter where in a file changes occur.

Other checksum schemes are possible (Peterson & Weldon 1972). Checksums, CRCs, and digital signatures such as MD5 (Rivest 1992) are all examples of hash functions. Many possible images will hash to the same checksum—how many depends on the number of bits in the image versus the number of bits in the sum. The utility of a checksum to detect errors (but not forgeries) depends on whether it evenly samples the likely errors. The 1's complement checksum is a good, quick way to do this.

References

Braden, R. T., Borman, D. A., & Partridge, C. 1988 (September), "Computing the Internet Checksum", Internet RFC 1071

Mallory, T. & Kullberg, A. 1990 (January), "Incremental Updating of the Internet Checksum", Internet RFC 1141

Peterson, W. W., & Weldon Jr., E. J. 1972, Error-Correcting Codes, Second Edition (Cambridge, Mass., MIT Press)

Rivest, R. 1992 (April), "The MD5 Message Digest Algorithm", Internet RFC 1321 (see also RFC 1319 and RFC 1320)

Seaman, R. 1994, in Astronomical Data Analysis Software and Systems III, ASP Conf. Ser., Vol. 61, eds. D. R. Crabtree, R. J. Hanisch, & J. Barnes (San Francisco, ASP), p. 119seamanr

Stevens, W. R. 1994, TCP/IP Illustrated Vol. 1 (Reading, Mass., Addison-Wesley)

Zweig, J., & Partridge, C. 1990 (March), TCP Alternate Checksum Options, Internet RFC 1146