FITS Checksum Proposal

R.L. Seaman* W.D. Pence†

24 August 1995

Abstract

This paper describes a simple means for embedding a checksum within a *FITS* header, providing a mechanism for later verifying the integrity of the *FITS* file. The method uses ASCII coded 32-bit 1's complement arithmetic to force the checksum of each *FITS* HDU to zero. This technique requires no extension of the *FITS* standard to be used by any given project, in fact the technique may be used to zero the checksum of any ASCII file. The purpose of this proposal is simply to reserve keywords that will allow projects and software packages to seamlessly verify and update each other's data.

1 Introduction

Checksums are used to gain confidence in the continued integrity of all sorts of data. The normal procedure, of course, is to calculate the checksum of the data on the transmitting side of some communication channel (including magnetic media) and later to compare that checksum with the recalculated checksum on the receiving side. The original checksum is transmitted separately over the same communication channel.

This scheme works for *FITS* (the *Flexible Image Transport System*, see ref. 1–5) data as for other data, but separating the checksum from the *FITS* file limits its utility, especially for archival storage. It is also hard to see just how to incorporate a separate checksum into the *FITS* standard.

The internet checksum (ref. 12–14) resolves the similar problem of embedding a checksum within each IP packet by forcing the checksum of each binary packet to the same value (zero). This works by writing the complement of the calculated checksum into the packet instead of the checksum itself.

Arranging to write a binary number into a *FITS* file is unattractive and limiting. However, the properties of commutativity and associativity that make the internet checksum work in the first place, also make it possible to generalize the technique with an ASCII encoding that may be embedded within a *FITS* header keyword (ref. 8).

*IRAF Group, National Optical Astronomy Observatories, *seaman@noao.edu*
†HEASARC, NASA Goddard Space Flight Center, *pence@tetra.gsfc.nasa.gov*

2 Reserved Keywords

Three keywords are reserved by this proposal, `CHECKSUM`, `DATASUM`, and `CHECKVER`. Normally both the `CHECKSUM` and `DATASUM` keywords will be present in a header if either keyword is present. This is not required however, and software should be prepared to handle each keyword separately. Each of the three keywords should be written using the rules of *FITS* fixed format character strings. An exception to this requirement is that the entire 16 characters of the `CHECKSUM` keyword value and also the entire length of the `DATASUM` keyword value are significant since all of the characters are needed to encode the checksum or format the 32-bit unsigned integer value of the datasum. The checksum keywords are constrained to fixed format since the ASCII encoding depends on the byte offset within each header card image (see §3.2).

2.1 `CHECKSUM` Keyword

The value field of the `CHECKSUM` keyword shall contain a 16 character string, left justified starting in column 12, containing the ASCII encoded complement of the checksum of the *FITS* HDU (Header and Data Unit). The algorithm shall be the 32-bit 1's complement checksum and the ASCII encoding that are described in §3. The checksum is accumulated in *FITS* datastream order on the same HDU, identical in all respects, except that the value of the `CHECKSUM` keyword shall be set to the string `'0000000000000000'` (ASCII 0's, hex 30) before the checksum is computed. Other equivalent interpretations of the ASCII encoding algorithm are possible, see §3.6.

2.2 `DATASUM` Keyword

The value field of the `DATASUM` keyword shall be a character string containing the unsigned integer value of the checksum (not the complement of the checksum in this case) of the data records of the HDU. This number is not ASCII encoded, but is formatted as a string since *FITS* support for unsigned integer keyword values is problematic. Any software that needs to read this value should perform normal input conversions to allow handling the value as an unsigned integer as allowed by the host system. For dataless HDU's, this keyword may either be omitted, or the value field shall contain the string value `'0'`, which is preferred. A missing `DATASUM` keyword asserts no knowledge of the checksum of the data records.

2.3 `CHECKVER` Keyword

The value field of the `CHECKVER` keyword shall contain a string, unique in the first 8 characters, which distinguishes between any future alternative checksum algorithms which may be defined. The default value for a missing keyword shall be `'COMPLEMENT'` which will represent the algorithm defined in this paper. It is recommended that this keyword be omitted from headers which implement the default ASCII encoded 32-bit 1's complement algorithm.

2.4 Undefined Keyword Values

For the default case of the ASCII encoded 32-bit 1's complement algorithm, a string of eight ASCII blanks will represent an undefined value for the `CHECKSUM` and `DATASUM` keywords. Keywords with undefined values can remain as placeholders during intermediate processing steps.

2.5 Keyword Usage

The following *strong recommendations* are made regarding keyword usage:

- *FITS* data files should include the checksum keywords in every HDU.
- *FITS* software should update the checksum keywords if present, or add the keywords if not present.
- Software that modifies *FITS* files should at the very least *delete* the keywords if no other checksum handling is implemented.
- All HDUs of a single *FITS* file should use the same `CHECKVER` algorithm.

The intent of these recommendations is to avoid a situation in which a *FITS* file has been modified, and the checksum allowed to become out of date.

2.6 Future Checksum Algorithms

The development of future *FITS* checksum algorithms should be coordinated with the *FITS* community. The allowed `CHECKVER` algorithm names should be governed by a registry similar to the registry of *FITS* extension names.

3 Discussion

The basic idea is the same as used by the Internet checksum (ref. 12–14). An unsigned integer is embedded within each data packet (*FITS* header) which forces the 1's complement 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.

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 quick and easy to calculate as a 16 bit sum due to this symmetry, providing greater sensitivity to errors (see §4).

In this proposal, the equivalent of zeroing the checksum field is to set the 16 character string value of the `CHECKSUM` keyword to all ASCII 0s (hex 30). The checksum is then accumulated in *FITS* storage order over the header and data records of each separate *FITS* HDU. These HDUs are either the individual extensions of a larger *FITS* file, or are just the primary HDU (whole file) for a simple *FITS* file. The (1's) complement is then formed for the resulting checksum. Finally, the complemented checksum is ASCII encoded and then written into the header replacing the ASCII 0s, which are in effect subtracted back out of the encoding to restore the original value. The HDU's previous checksum and its embedded complement now sum to zero. Alternate interpretations of this encoding algorithm are possible, see §3.6.

3.1 Scope of the Checksum

The checksum described in this proposal applies to each separate *FITS* Header and Data Unit, that is, to each separate *FITS* extension in any given larger multi-extension *FITS* data file. It would be equally possible to define a single checksum that applies to the entire *FITS* file, including all extensions, but since in either case the 1's complement checksum is zeroed for the entire file, there is no advantage to the latter interpretation.

A single, whole file checksum also implicitly assumes that the individual *FITS* extensions are related to one another in some intimate fashion. This is not necessarily the case - the separate extensions may simply be concatenated into larger *FITS* files for I/O or indexing efficiency (as with the NOAO *Save the Bits* archive, see ref. 8).

Even if the individual extensions in a given *FITS* file are related to a single observation, or form some other coherent data structure, a particular user may need only some of the extensions. Stripping unneeded extensions from the *FITS* file is trivial if a per HDU checksum is defined, since an individual extension with a zeroed checksum may be removed without affecting the (zeroed) checksum of the entire file.

A per HDU checksum offers maximum processing flexibility as well as flexibility of interpretation. Even if a programmer decides to update a checksum by re-accumulating the checksum over all the *FITS* records (rather than by using the end-to-end, incremental calculation described in §3.3) this will be far more efficient on a per HDU basis.

3.2 1's Complement Arithmetic

1's complement addition is distinguished from the 2's complement addition typically encountered in (unsigned) computer arithmetic by how overflow bits are handled. 1's complement overflow bits are carried around back into the sum while 2's complement overflow bits are discarded.

In general, the inverse of a number under a given mathematical operation is the value which when operated on with that number returns the identity element. The 1's complement additive inverse of a number is its bitwise complement (replace 0s with 1s and 1s with 0s). This proposal relies on a number and its complement summing to zero (the additive identity element). Actually they sum to *negative* zero—1's complement addition has two identity elements. Recall that an identity element under a given operation is a value which leaves any other number unchanged when the operation is applied. Under 1's complement arithmetic the addition of either zero (all 0's) or negative zero (all 1's) to a number will generate a sum equal to the original number.

1's complement addition is both associative and commutative (it forms an *Abelian group*[1] over the unsigned integers, ref. 11), so it is immaterial whether an identity element is added to a number or the number is added to an identity element, or whether the number operates on its inverse or the inverse operates on the number—both arrangements have the same result. Also note that the operation of subtraction is equivalent to adding the inverse (complement) of the number.

3.3 Incremental Updating of the Checksum

The symmetry of 1's complement arithmetic also means that after modifying a *FITS* file, the checksum may be incrementally updated using simple arithmetic without accumulating the checksum for portions of the file that have not changed.

An incremental update provides the mechanism for end-to-end checksum verification through any number of intermediate processing steps. By *calculating* rather than *accumulating* the intermediate checksums, the original checksum test is propagated through to the final data file. On the other hand, if a new checksum is accumulated with each change to the file, no information is preserved about the file's original state.

The recipe for updating the `CHECKSUM` keyword following some change to the file is: $C' = C - m + m'$, where C and C' represent the file's checksum (that is, the complement of the `CHECKSUM` keyword) before and after the modification and m and m' are the corresponding checksums for the modified *FITS* records or keywords only. Since the `CHECKSUM` keyword contains the complement of the checksum, the correspondingly complemented form of the recipe is more directly useful: $\tilde{}C' = \tilde{}(C + \tilde{}m + m')$, where ˜ (tilde) denotes the (1's) complement operation. (See ref. 12–14.) Note that the tilde on the right hand side of the equation cannot be distributed over the contents of the parentheses due to the dual nature of zero in 1's complement arithmetic (ref. 14).

[1] Making allowance for the dual zeroes, 1's complement addition and multiplication also form an *integral domain* over the N-bit unsigned integers, see ref. 21, for example.

3.4 ASCII Encoding

Encoding the unsigned integer checksum into an ASCII string that may legally be embedded in a *FITS* header is 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 alphanumerics, including a hex 30 offset to start at ASCII 0 (zero). The ASCII special characters are avoided by the encoding algorithm implemented in appendices A and B of this proposal, as shown in Figure 1. Restricting the encoded value to the ASCII alphanumerics is not required by the *FITS* standard (see §5.3 in ref. 5), but rather promotes human readability and transcription.

0 *30*	1 *31*	2 *32*	3 *33*	4 *34*	5 *35*	6 *36*	7 *37*	8 *38*	9 *39*
: *3a*	; *3b*	< *3c*	= *3d*	> *3e*	? *3f*	@ *40*	A *41*	B *42*	C *43*
D *44*	E *45*	F *46*	G *47*	H *48*	I *49*	J *4a*	K *4b*	L *4c*	M *4d*
N *4e*	O *4f*	P *50*	Q *51*	R *52*	S *53*	T *54*	U *55*	V *56*	W *57*
X *58*	Y *59*	Z *5a*	[*5b*	\ *5c*	] *5d*	^ *5e*	_ *5f*	‘ *60*	a *61*
b *62*	c *63*	d *64*	e *65*	f *66*	g *67*	h *68*	i *69*	j *6a*	k *6b*
l *6c*	m *6d*	n *6e*	o *6f*	p *70*	q *71*	r *72*			

Figure 1. Only ASCII alpha-numerics are used to encode the checksum — punctuation is excluded.

The reason that this encoding works is that unlike the more familiar `uuencode` or `binhex` schemes for representing binary data as printable ASCII characters, this is not just encoding the meaning of the binary data, but the checksum's actual 32-bit unsigned integer value itself. This integer is divided into 4 equal pieces that are placed in four successive integer aligned fields in the (otherwise ASCII) header.

Dividing each of the 4 bytes of the original integer value into 4 separate bytes results in sixteen total bytes. (Any remainder from each bytewise division is added to the first byte in each of the four quartets.) The division by 4 allows each of the 16 bytes to fit within the range of the ASCII alphanumeric characters - with room to spare since there is a range of 75 from ASCII 0 (zero) at hex 30 to ASCII z at hex 7A. Only a range of 64 is needed to quarter each original byte. The spare room in the ASCII chart allows excluding non-alphanumeric characters. The trick is to offset the zero of the character encoding to ASCII zero (hex 30) before accumulating the checksum. This offset is subtracted back out again when the ASCII zeroes are replaced by the final complemented checksum.

The key concept is that dividing the byte into four bytes means dividing the *integer value* by four, with three bytes having a value one-fourth of the original byte (the quotient of dividing by four) and the final byte, quotient plus remainder. Before ASCII zero is added, the four bytes sum to the original. Adding ASCII zero (hex 30) restricts the *character value* of the bytes to be printable ASCII. The adjustment to further restrict each character in the string to only the ASCII alphanumerics consists of decrementing one byte and incrementing another byte by the same amount. Then, when the ASCII zero is subtracted (at the receiving end), the four new bytes will still add to the original number. This whole thing is repeated for all four bytes of the original checksum.

Alternate interpretations of the ASCII encoding algorithm are possible, see §3.6.

3.5 Encoding Example

An example may help make this clearer. Consider a *FITS* HDU whose 1's complement checksum is 868229149. This number was obtained by simply accumulating the 32-bit checksum over the header and data records using 1's complement arithmetic. The `CHECKSUM` keyword was first initialized to the value `'0000000000000000'` as described in §3.8. To zero the checksum, replace the zero initialized `CHECKSUM` field with the complement of the accumulated checksum, 3426738146, which is equivalent to hex `CC3FDFE2`. The steps needed to encode this hex value into ASCII are:

- divide each of the four original bytes into four equal bytes, for 16 total bytes
- by maintaining the byte alignment of each of the four bytes of each of the four quotients, these will sum (including remainder) to the original value
- add the remainder from each of the divisions to the first byte in each case
- offset the result by hex 30 so that a zero quotient now corresponds to ASCII zero
- interpret the 16 resulting bytes as ASCII

Schematically:

```
    Byte                          Preserve byte alignment
 1  2  3  4       1  2  3  4   1  2  3  4   1  2  3  4   1  2  3  4

CC 3F DF E2  ->  33 0F 37 38  33 0F 37 38  33 0F 37 38  33 0F 37 38
+ remainder       0  3  3  2

      = hex      33 12 3A 3A  33 0F 37 38  33 0F 37 38  33 0F 37 38
 + 0 offset      30 30 30 30  30 30 30 30  30 30 30 30  30 30 30 30

      = hex      63 42 6A 6A  63 3F 67 68  63 3F 67 68  63 3F 67 68
      ASCII       c  B  j  j   c  ?  g  h   c  ?  g  h   c  ?  g  h
```

We further require that only ASCII alphanumerics be used by the encoding. This is purely for cosmetic reasons to improve readability. Non-alphanumeric bytes are shifted higher and lower by pairs as in the example on the top of the next page. Byte A2 (originally ASCII `B`) is shifted higher (to ASCII `H`) to balance byte B2 (originally ASCII `?` being shifted lower (to ASCII `9`). The same holds true in this example for the paired bytes C2 and D2. This is possible since the two sequences of ASCII punctuation characters that can occur in encoded checksums are both preceded and followed by longer sequences of ASCII alphanumeric characters as shown in Figure 1 (above).

```
            A      B      C      D
step      1234   1234   1234   1234

  1       cBjj   c?gh   c?gh   c?gh    initial encoded value
  2       cCjj   c>gh   c@gh   c>gh
  3       cDjj   c=gh   cAgh   c=gh
  4       cEjj   c<gh   cBgh   c<gh
  5       cFjj   c;gh   cCgh   c;gh
  6       cGjj   c:gh   cDgh   c:gh
  7       cHjj   c9gh   cEgh   c9gh      final encoded value
```

3.6 Alternate Encoding Interpretation

The 16-byte (128-bit) ASCII encoding for the 32-bit checksum provides a large degree of redundancy in the precise string that is used to express any given value. A straightforward estimate suggests that there are about $10^{29} = 2^{128}/2^{32}$ different ASCII patterns that represent each unsigned 32-bit integer. If only the 62 alpha-numeric characters are allowed in the encoded strings, this number is reduced by a factor of about $10^{10} = (62/256)^{16}$ to 10^{19} total ASCII expressions for each possible FITS checksum.[2]

This proposal does not mandate a single ASCII encoding algorithm selected from this very large number of possibilities, although the encoding described in §3.4 and §3.5 and in the code in the appendices is recommended. The following constraints on the ASCII encoded values are required, however:

- The 32-bit unsigned integer checksum will be encoded into a 16 character ASCII string.
- Only the ASCII alpha-numeric characters, 0-9, A-Z and a-z, are permitted.[3]
- The quotes enclosing the ASCII string in the `CHECKSUM` keyword value will lie in columns 11 and 28. The implications of this are discussed in §3.7 below.
- The encoded ASCII value will cause the 32-bit 1's complement checksum of the particular *FITS* HDU to be zeroed.

3.7 Checksum Keyword Alignment

Note that the checksum field must be four-byte integer aligned, whether the checksum is being stored as an integer or as an encoded string (equivalent to four sequential integers). In either case, this requirement only applies byte-by-byte due to the commutativity and associativity of 1's complement arithmetic—any file can be reshuffled

[2]This estimate may not be exactly right due to binary and higher order interactions between the characters from excluding the intervening punctuation characters from the strict alpha-numeric encoding.

[3]The encoding algorithm described in §3.4 and §3.5 only uses the characters, 0-9, A-Z and a-r, as expressed in Figure 1.

without affecting the checksum as long as the location of each byte modulo 4 is preserved. This may be considered a weakness of the 1's complement checksum, but only if such a wholesale reshuffling is a likely error that must be guarded against. It is also a strength of the algorithm since the checksum field may be positioned at an arbitrary odd byte offset just by permuting the bytes while preserving their relative offsets. This would not be possible for algorithms such as the 2's complement checksum and the CRC (§4) whose values depend on the order in which the bytes are summed.

This proposal requires that the initial quote for the `CHECKSUM` keyword must be located in column 11 to agree with the *FITS* fixed format character string conventions. Column 12 contains the final byte of the same four byte word (that contains column 11). To ensure proper integer alignment the final byte (character) of the encoded string must either be permuted to the beginning of the string, or column 12 (the first character of the string) must be left blank. (Recall that leading blanks are significant in *FITS* keyword values.) This proposal mandates that the bytes be permuted. (ASCII blanks are represented by "␣".)

```
           1         2
1234 5678 9012 3456 7890 1234 5678
CHEC KSUM =␣'h cHjj c9gh cEgh c9g'   ⇐ correct
CHEC KSUM =␣'␣ cHjj c9gh cEgh c9gh'
```

The two arrangements (other more complicated permutations are possible) are shown with the alignment on four byte word boundaries indicated by spacing. The 16 character values of each encoded unsigned integer have identical effects on the file checksum, ignoring punctuation, but only the first is permitted by this proposal.

3.8 Checksum Keyword Example

The following is an example of how the `CHECKSUM`, `DATASUM`, and `CHECKVER` keywords are used. For the default case of the ASCII coded 32-bit 1's complement checksum, omitting the `CHECKVER` keyword is preferred, but it is included here for the sake of completeness. The example uses a primary *FITS* header, but the identical usage applies for extension headers.

```
         1         2         3         4         5         6
12345678901234567890123456789012345678901234567890123456789012345...

SIMPLE  =                    T  /  standard FITS format
...
CHECKSUM= 'hcHjjc9ghcEghc9g'    /  ASCII 1's complement checksum
DATASUM = '2503531142'          /  checksum of data records
CHECKVER= 'COMPLEMENT'          /  checksum version ID
END
```

To calculate the two checksums, first accumulate the checksum of the data records, format this unsigned integer into a character string (for portability) and update the `DATASUM` keyword. Next, set the `CHECKSUM` keyword value to `'0000000000000000'` and calculate the checksum of the header records. Add the `DATASUM` to the header checksum and bit complement the total. This is the 1's complement additive inverse of the total. ASCII encode this value and write it into the `CHECKSUM` keyword, replacing the 0's. The checksum of the *FITS* HDU is now zeroed. A few points:

- If this same procedure is followed for all extensions in any given *FITS* file as well as for the primary HDU, the checksum of the entire file will be zeroed since each extension's checksum is the additive identity element.
- To later verify a *FITS* file or an individual extension, simply accumulate the checksum and compare it to negative zero (all one's).
- To update the checksum after modifying some part of the *FITS* file, subtract from the total checksum the previous checksum of the changed *FITS* records and add the new checksum of those records.
- To invert the ASCII encoding, permute the encoded string back to an alignment of zero bytes, subtract the hex 30 offset and calculate the checksum of the resulting string. This can be used, for instance, to read the value of `CHECKSUM` into the software when verifying or updating a file.
- Any ASCII encoding algorithm could potentially be used that represents the pertinent unsigned integer as a 16 character string with a per byte offset of hex 30 (for instance, not excluding the punctuation). The encoding defined by example in appendices A and B is recommended by this proposal, however. Restraints on allowed encodings are discussed in §3.6.

4 Alternate Checksum Algorithms

There are a variety of checksum schemes (for examples, see ref. 11,15–16) other than the 1's complement algorithm described in this proposal, although other checksums are significantly less easy (often computationally impractical or impossible) to embed in *FITS* headers in the same fashion.

Checksums, *cyclic redundancy checks* (or *CRCs*, see ref. 10 for example), and *message digests* such as MD5 (ref. 19) 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), to one part in however many bits, depends on whether it evenly samples the likely errors.

For instance, a 32-bit checksum or CRC each detects the same fraction of all bit errors (ref. 16), missing only $1/2^{32}$ of all errors (about 1 out of 4.3 billion) in the limit of long transmissions (the extra zero of 1's complement arithmetic changes this by only a small amount).

CRCs and message digests are basically checksums that use higher order polynomials, thus removing the arithmetic symmetry on which this proposal relies. CRCs are tuned to be sensitive to the bursty nature of communication line noise and will detect all bitstream errors shorter than the size of the CRC. Note that the 1's complement sum is not insensitive to these bit error patterns, it is just not *especially* sensitive to them. The extra sensitivity of a CRC to burst errors must come at the expense of lessened sensitivity to other bit pattern errors (since the total fraction of errors detected remains the same) and does not necessarily represent the best model for *FITS* bit errors. CRCs are also designed to be implemented in hardware using XOR gates and shift registers that accumulate the function "on-the-fly" and emit the CRC *after* transmitting the data. This is not well matched to the *FITS* convention of writing the metadata as a header which preceeds the data records.

4.1 Digital Signatures

The particular intent of a message digest, on the other hand, is to protect against human tampering by relying on functions that are computationally infeasible to spoof. A message digest should also be much longer than a simple checksum so that any given message may be assumed to result in a unique value.

A *digital signature* may be formed by reverse encrypting a message digest using the private key of a public key encryption pair (ref. 20). A later decryption using the corresponding publically available key guarantees that the signature could only have been generated by the holder of the private key, while the message digest uniquely identifies the document (or image) that was signed. Support for digital signatures could be added to the *FITS* standard by defining a *FITS* extension format to contain the digital signature certificates, or perhaps by simply embedding them in an appended *FITS* table extension.

There is a tradeoff between the error detection capability of these algorithms and their speed. The overhead of a digital signature (or a software emulated CRC) is prohibitive for multi-megabyte files today, but may be essential for certain purposes (for instance, archival storage) in the future. The checksum defined by this proposal provides a way to verify *FITS* data against likely random errors, while on the other hand a full digital signature may be required to protect the same data against systematic errors, especially human tampering.

4.2 Fletcher's Checksum

One other checksum algorithm shows some promise of being embeddable in an ASCII *FITS* header. This is *Fletcher's checksum* (ref. 16–18) which is a variant of the 1's complement checksum that is tuned to trap bit error patterns similar to those trapped by a CRC. It is somewhat slower than the 1's complement checksum and more finicky to implement. The checksum is divided into two (16 bit) pieces—a straight 1's complement sum and a higher order sum of the running sums. The procedure for updating the two checksum fields (zeroing the checksum of the file) involves solving a pair of simultaneous equations. ASCII encoding the checksum would require an iterative solution spread over the four separate ASCII encoded integer words (and including the constraint of the hex 30 offset). Incremental updating of the checksum would incur a similar penalty for each word of the *FITS* file that was modified.

The added complexity and overhead of handling Fletcher's checksum (see ref. 17–18) are unwarranted for *FITS*, at least as the default algorithm, but this checksum is an interesting possibility for binary applications. Other checksums are also options in the binary case, especially if the checksum fields can be located at the end of the file, which simplifies the arithmetic significantly.

4.3 Error Correcting Algorithms

Error *correcting* (see ref. 9), as opposed to error *detecting*, algorithms are beyond the scope of this proposal, as are non-systematic codes for either error detection or correction. *Systematic* codes are those, such as the 1's complement checksum, that require no change to the data when applied to a message. Simply appending a checksum to a file is systematic, as is appending parity or other check bits to each byte or record of the data without otherwise modifying the data bits. Codes that are not systematic involve recoding the individual data bits in some fashion (see the discussion of *product* codes in ref. 11, for example).

5 Summary

The ASCII encoded 1's complement checksum has several features that make it a good choice for safeguarding FITS data:

- The checksum of each *FITS* HDU is forced to zero by writing the complement of the calculated checksum into the header. Verifying a particular HDU requires only that the checksum computes to zero.
- If the checksums of all HDUs in a *FITS* file are zeroed, the checksum for the entire file will be zeroed. All HDU's in such a file can be verified by accumulating a single checksum.
- Since 1's complement addition is commutative and associative, the checksum may be accumulated in any order.
- Specifically, this means that the checksum is transparent to the byte order of any particular machine architecture.
- The checksum is straightforward to update as the header or data records are changed during a sequence of processing steps. Since the checksum is incrementally updated, the end-to-end nature of the verification is maintained.
- A simple rearrangement of keywords leaves the checksum unchanged.
- The checksum of the data records is written into a separate header keyword, `DATASUM`, and is not recomputed unless the data records are modified. The `DATASUM` keyword is not ASCII encoded as with the `CHECKSUM` keyword, but is rather formatted as an unsigned integer valued character string.
- Individual *FITS* extensions have separate checksums. Extensions with zeroed checksums may be added and removed from a larger *FITS* file without disturbing the aggregate checksum.
- The recipe for updating the `CHECKSUM` keyword following some change to the file is: $\tilde{}C' = \tilde{}(C + \tilde{}m + m')$, where C and C' represent the file's checksum before and after the modification, m and m' are the corresponding checksums for the modified *FITS* records or keywords only, and ˜ (tilde) denotes the (1's) complement operation.

6 References

FITS topics:

1. Wells, D.C., Greisen, E.W., Harten, R.H. 1981, "*FITS*: A Flexible Image Transport System", *Astron. Astrophys. Suppl.*, **44**, 363-370.

2. Greisen, E.W., Harten, R.H. 1981, "An Extension of *FITS* for Small Arrays of Data", *Astron. Astrophys. Suppl.*, **44**, 371-374.

3. Grosbøl, P., Harten, R.H., Greisen, E.W., Wells, D.C. 1988, "Generalized Extensions and Blocking Factors for *FITS*", *Astron. Astrophys. Suppl.*, **73**, 359-364.

4. Harten, R.H., Grosbøl, P., Greisen, E.W., Wells, D.C. 1988, "The *FITS* Table Extension", *Astron. Astrophys. Suppl.*, **73**, 365-372.

5. NASA/Science Office of Standards and Technology 1993, "Definition of the Flexible Image Transport System (*FITS*)" *NOST 100-1.0 Standard.*

6. Pence, W.D. 1991, "FITSIO and FITS File Utility Software" in *Astronomical Data Analysis Software and Systems I*, edited by Worrall, D.M., Biemesderfer, C. and Barnes, J., *A.S.P. Conf. Ser.*, **25**, 22-23.

7. Pence, W.D. 1994, "FITSIO Subroutine Library Update", presented at the conference *Astronomical Data Analysis Software and Systems IV*, to appear in the *A.S.P. Conf. Ser.*.

8. Seaman, R.L. 1994, "*FITS* Checksum Verification in the NOAO Archive", presented at the conference *Astronomical Data Analysis Software and Systems IV*, to appear in the *A.S.P. Conf. Ser.*.

Checksum and error detection topics:

9. Peterson, W.W. and Weldon Jr., E.J. 1972, *Error-Correcting Codes*, Second Edition (MIT Press).

10. McNamara, J.E. 1982, *Technical Aspects of Data Communication*, Second Edition (Digital Press).

11. Plummer, W.W. 1978, "TCP Checksum Function Design", *ACM Computer Communication Review*, **19**, no. 2, 95-101, this is an appendix to *Internet RFC 1071.*

12. Braden, R. T., Borman, D.A., and Partridge, C. 1988 (September), "Computing the Internet Checksum", *ACM Computer Communication Review*, **19**, no. 2, 86-94, this is *Internet RFC 1071.*

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

14. Rijsinghani, A. (ed.) 1994 (May), "Computation of the Internet Checksum via Incremental Update", *Internet RFC 1624.*

15. Zweig, J. and Partridge, C. 1990 (March), "TCP Alternate Checksum Options", *Internet RFC 1146.*

16. Fletcher, J.G. 1982, "An Arithmetic Checksum for Serial Transmission", *IEEE Transactions on Communications*, **COM-30**, no. 1, 247-252.

17. Nakassis, A. 1988, "Fletcher's Error Detection Algorithm: How to implement it efficiently and how to avoid the most common pitfalls", *ACM Computer Communication Review*, **18**, no. 5, 63-88.

18. Sklower, K. 1989, "Improving the Efficiency of the OSI Checksum Calculation", *ACM Computer Communication Review*, **19**, no. 5, 32-43.

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

20. Zimmermann, P. 1995, *The Official PGP User's Guide* (MIT Press), PGP is available from *http://net-dist.mit.edu/pgp.html* or *ftp://ftp.csn.net/mpj/README*, which also provide United States export and licensing requirements.

21. Hungerford, T.W. 1987, *Algebra*, Fourth Printing (Springer-Verlag).

Internet *Requests for Comments*, or *RFCs*, are the written design documents for internet protocols. They are available at many locations on the internet, including *http://www.cis.ohio-state.edu/htbin/rfc/rfc-index.html.*

7 Acknowledgements

The authors gratefully acknowledge the many helpful comments of Barry Schlesinger and Arnold Rots.

A Example *C* Code

The example *C* routines are adapted from the online software for the NOAO[4]/IRAF *Save the Bits* archive (ref. 8).

A.1 Accumulating the Checksum

The 1's complement checksum is simple to compute and very fast. 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 (after initializing the `CHECKSUM` keyword to `'0000000000000000'`):

```
checksum (buf, length, sum32)
char *buf;
int length;                     /* < 2^18, or carry can overflow */
unsigned int *sum32;
{
        /* ints are assumed to be 32 bits */
        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];
        }
        if (remain >= 1) hi += buf[2*len] * 0x100;
        if (remain >= 2) hi += buf[2*len+1];
        if (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;
}
```

If this routine is adapted to operate on non-*FITS* data, the restriction on the length of the individual buffers can be relaxed by explicitly folding the carry bits back into the sum frequently enough to avoid overflowing the unsigned integer arithmetic. This is not a problem with *FITS* data since the logical record size is at most 28,800 bytes.

[4]The National Optical Astronomy Observatories are operated by the Association of Universities for Research in Astronomy, Inc. under cooperative agreement with the National Science Foundation.

A.2 ASCII Encoding

The byte alignment is permuted one byte forward for *FITS* to left justify the string value starting in column 12. A similar production routine might receive the desired number of bytes to permute as an argument.

```
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;
        char asc[32];

        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 */
                asc[4*j+i] = ch[j];
        }

        for (i=0; i < 16; i++)              /* permute the bytes for FITS */
            ascii[i] = asc[(i+15)%16];

        ascii[16] = 0;
}
```

A.3 ASCII Decoding

To invert the encoding, permute the string back to an alignment of zero bytes, subtract the hex 30 offset from each byte and pass the result to the checksum routine. A specific decoding routine is not required, but is provided as an example.

```
unsigned int char_decode (ascii)
char *ascii;                    /* assumed to be 16 bytes long */
{
        char cbuf[16];
        unsigned short *sbuf;
        unsigned int hi=0, lo=0, hicarry, locarry;
        int i;

        /* remove the permuted FITS byte
         * alignment and the ASCII 0 offset
         */
        for (i=0; i < 16; i++) {
            cbuf[i] = ascii[(i+1)%16];
            cbuf[i] -= 0x30;
        }

        sbuf = (unsigned short *) cbuf;

        for (i=0; i < 8; i+=2) {
            hi += sbuf[i];
            lo += sbuf[i+1];
        }

        hicarry = hi >> 16;
        locarry = lo >> 16;
        while (hicarry || locarry) {
            hi = (hi & 0xFFFF) + locarry;
            lo = (lo & 0xFFFF) + hicarry;
            hicarry = hi >> 16;
            locarry = lo >> 16;
        }

        return ((hi << 16) + lo);
}
```

If this routine is used to decode the value of the `CHECKSUM` keyword, the decoded result must be complemented to restore the original HDU checksum (including the contribution of the zero initialized `CHECKSUM` keyword value).

B Example *Fortran 77* Code

The *Fortran 77* routines are adapted from the FITSIO package (ref. 6–7).

B.1 Accumulating the Checksum

```
      subroutine checksum(buffer,length,sum32)

C     Calculate a 32-bit 1's complement checksum of the input buffer, adding
C     it to the value of sum32.  This algorithm assumes that the buffer
C     length is a multiple of 4 bytes.

C     a double precision value (which has at least 48 bits of precision)
C     is used to accumulate the checksum because standard Fortran does not
C     support an unsigned integer datatype.

C     buffer  - integer buffer to be summed
C     length  - number of bytes in the buffer (must be multiple of 4)
C     sum32   - double precision checksum value (The calculated checksum
C               is added to the input value of sum32 to produce the
C               output value of sum32)

      integer buffer(*),length,i,hibits
      double precision sum32,word32
      parameter (word32=4.294967296D+09)
C               (word32 is equal to 2**32)

C     LENGTH must be less than 2**15, otherwise precision may be lost
C     in the sum
      if (length .gt. 32768)then
          print *, 'Error: size of block to sum is too large'
          return
      end if

      do i=1,length/4
          if (buffer(i) .ge. 0)then
              sum32=sum32+buffer(i)
          else
C             sign bit is set, so add the equivalent unsigned value
              sum32=sum32+(word32+buffer(i))
          end if
      end do

C     fold any overflow bits beyond 32 back into the word
10    hibits=sum32/word32
      if (hibits .gt. 0)then
          sum32=sum32-(hibits*word32)+hibits
          go to 10
      end if

      end
```

B.2 ASCII Encoding

```
      subroutine check_encode(sum32,complement,string)

C     sum32     double precision checksum value
C     complement  (logical) whether to encode the complement of the sum (true)
C                           or encode the sum value itself (false).
C     string  c*16  output ASCII encoded checksum

      double precision sum32,tmpsum,all32
      logical complement
      character*16 string,tmpstr
      integer offset,exclud(13),nbyte(4),ch(4),i,j,k
      integer quot,remain,check,nc

C     all32 is equivalent to a 32 bit unsigned integer with all bits set
      parameter (all32=4.294967295D+09)

C     ASCII 0 is the offset value
      parameter (offset=48)

C     this is the list of ASCII punctuation characters to be excluded
      data exclud/58,59,60,61,62,63,64,91,92,93,94,95,96/

      if (complement)then
C         complement the 32-bit unsigned integer equivalent (flip every bit)
          tmpsum=all32-sum32
      else
C         just encode the sum, not its complement
          tmpsum=sum32
      end if

C     separate each byte of the 32-bit integer into separate 8-bit integers
      nbyte(1)=tmpsum/16777216.
      tmpsum=tmpsum-nbyte(1)*16777216.
      nbyte(2)=tmpsum/65536.
      tmpsum=tmpsum-nbyte(2)*65536.
      nbyte(3)=tmpsum/256.
      nbyte(4)=tmpsum-nbyte(3)*256.

C     encode each 8-bit integer as 4-characters
      do i=1,4
          quot=nbyte(i)/4+offset
          remain=nbyte(i) - (nbyte(i)/4*4)
          ch(1)=quot+remain
          ch(2)=quot
          ch(3)=quot
          ch(4)=quot

C         avoid ASCII punctuation characters by incrementing and
C         decrementing adjacent characters thus preserving checksum value
10        check=0
          do  k=1,13
              do  j=1,4,2
                  if (ch(j)   .eq. exclud(k) .or.
     &                ch(j+1) .eq. exclud(k))then
                      ch(j)=ch(j)+1
                      ch(j+1)=ch(j+1)-1
                      check=1
                  end if
              end do
```

```
            end do

C           keep repeating, until all punctuation character are removed
            if (check .ne. 0)go to 10

C           convert the byte values to the equivalent ASCII characters
            do j=0,3
                nc=4*j+i
                tmpstr(nc:nc)=char(ch(j+1))
            end do
        end do

C       shift the characters 1 place to the right, since the FITS character
C       string value starts in column 12, which is not word aligned
        string(2:16)=tmpstr(1:15)
        string(1:1)=tmpstr(16:16)

        end
```