4.4.2.7 Data Integrity Keywords

The two keywords described here provide an integrity check on the information contained in *FITS* HDUs.

DATASUM Keyword. The value field of the DATASUM keyword *shall* consist of a character string that *should* contain the unsigned integer value of the 32-bit 1's complement checksum of the data records in the HDU (i.e., excluding the header records). For this purpose, each 2880-byte *FITS* logical record should be interpreted as consisting of 720 32-bit unsigned integers. The 4 bytes in each integer *must* be interpreted in order of decreasing significance where the most significant byte is first, and the least significant byte is last. Accumulate the sum of these integers using 1's complement arithmetic in which any overflow of the most significant bit is propagated back into the least significant bit of the sum.

The DATASUM value is expressed as a character string (i.e., enclosed in single quote characters) because support for the full range of 32-bit unsigned integer keyword values is problematic in some software systems. This string may be padded with non-significant leading or trailing blank characters or leading zeros. A string containing only 1 or more consecutive ASCII blanks may be used to represent an undefined or unknown value for the DATASUM keyword. The DATASUM keyword may be omitted in HDUs that have no data records, but it is preferable to include the keyword with a value of 0. Otherwise, a missing DATASUM keyword asserts no knowledge of the checksum of the data records. Recording in the comment field the ISO-8601-formatted Datetime when the value of this keyword record is created or updated is *recommended*.

CHECKSUM Keyword. The value field of the CHECKSUM keyword *shall* consist of an ASCII character string whose value forces the 32-bit 1's complement checksum accumulated over the entire *FITS* HDU to equal negative 0. (Note that 1's complement arithmetic has both positive and negative zero elements). It is *recommended* that the particular 16-character string generated by the algorithm described in Appendix J be used. A string containing only 1 or more consecutive ASCII blanks may be used to represent an undefined or unknown value for the CHECKSUM keyword.

The CHECKSUM keyword value *must* be expressed in fixed format, when the algorithm in Appendix J is used, otherwise the usage of fixed format is *recommended*. Recording in the comment field the ISO-8601-formatted Datetime when the value of this keyword record is created or updated is *recommended*.

If the CHECKSUM keyword exists in the header of the HDU and the accumulated checksum is not equal to -0, or if the DATASUM keyword exists in the header of the HDU and its value does not match the data checksum then this provides a strong indication that the content of the HDU has changed subsequent to the time that the respective keyword value was computed. Such an invalid checksum may indicate corruption during a prior file copy or transfer operation, or a corruption of the physical media on which the file was stored. It may alternatively reflect an intentional change to the data file by subsequent data processing if the CHECKSUM value was not also updated.

Normally both keywords will be present in the header if either is present, but this is not required. These keywords apply *only* to the HDU in which they are contained. If these keywords are written in one HDU of a multi-HDU *FITS* file then it is *strongly recommended* that they also be written to every other HDU in the file with values appropriate to each HDU in turn; in that case the checksum accumulated over the entire file will equal -0 as well. The DATASUM keyword *must* be updated before the CHECKSUM keyword. In general updating the two checksum keywords should be the final step of any update to either data or header records in a FITS HDU. It is highly *recommended* that if a *FITS* file is intended for public distribution, then the checksum keywords, if present, should contain valid values.

4.4.3. Additional keywords

New keywords *may* be devised in addition to those described in this standard, so long as they are consistent with the generalized rules for keywords and do not conflict with mandatory or reserved keywords. Any keyword that refers to or depends upon the existence of other specific HDUs in the same or other files should be used with caution because the persistence of those HDUs cannot always be guaranteed.

Appendix J: CHECKSUM Implementation Guidelines

This Appendix is not part of the FITS *standard, but is included for informational purposes.*

J.1. Recommended CHECKSUM *Keyword Implementation*

The *recommended* CHECKSUM keyword algorithm described here generates a 16-character ASCII string that forces the 32-bit 1's complement checksum accumulated over the entire *FITS* HDU to equal negative 0 (all 32 bits equal to 1). In addition, this string will only contain alphanumeric characters within the ranges 0–9, A–Z, and a–z to promote human readability and transcription. If the present algorithm is used, the CHECKSUM keyword value *must* be expressed in fixed format, with the starting single quote character in column 11 and the ending single quote character in column 28 of the *FITS* keyword record, because the relative placement of the value string within the keyword record affects the computed HDU checksum. The steps in the algorithm are as follows:

1. Write the CHECKSUM keyword into the HDU header with an initial value consisting of 16 ASCII zeros (`'0000000000000000'`) where the first single quote character is in column 11 of the *FITS* keyword record. This specific initialization string is required by the encoding algorithm described in Sect. J.2 The final comment field of the keyword, if any, must also be written at this time. It is recommended that the current date and time be recorded in the comment field to document when the checksum was computed.
2. Accumulate the 32-bit 1's complement checksum over the *FITS* logical records that make up the HDU header in the same manner as was done for the data records by interpreting each 2880-byte logical record as 720 32-bit unsigned integers.
3. Calculate the checksum for the entire HDU by adding (using 1's complement arithmetic) the checksum accumulated over the header records to the checksum accumulated over the data records (i.e., the previously computed DATASUM keyword value).
4. Compute the bit-wise complement of the 32-bit total HDU checksum value by replacing all 0 bits with 1 and all 1 bits with 0.
5. Encode the complement of the HDU checksum into a 16-character ASCII string using the algorithm described in Sect. J.2
6. Replace the initial CHECKSUM keyword value with this 16-character encoded string. The checksum for the entire HDU will now be equal to negative 0.

J.2. Recommended ASCII Encoding Algorithm

The algorithm described here is used to generate an ASCII string which, when substituted for the value of the CHECKSUM keyword, will force the checksum for the entire HDU to equal negative 0. It is based on a fundamental property of 1's complement arithmetic that the sum of an integer and the negation of that integer (i.e, the bitwise complement formed by replacing all 0 bits with 1s and all 1 bits with 0s) will equal negative 0 (all bits set to 1). This principle is applied here by constructing a 16-character string which, when interpreted as a byte stream of 4 32-bit integers, has a sum that is equal to the complement of the sum accumulated over the rest of the HDU. This algorithm also ensures that the 16 bytes that make up the 4 integers all have values that correspond to ASCII alpha-numeric characters in the range 0–9, A–Z, and a–z.

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.

1. Begin with the 1's complement (replace 0s with 1s and 1s with 0s) of the 32-bit checksum accumulated over all the FITS records in the HDU after first initializing the CHECKSUM keyword with a fixed-format string consisting of 16 ASCII zeros (`'0000000000000000'`).
2. Interpret this complemented 32-bit value as a sequence of 4 unsigned 8-bit integers, A, B, C and D, where A is the most significant byte and D is the least significant. Generate a sequence of 4 integers, A1, A2, A3, A4, that are all equal to A divided by 4 (truncated to an integer if necessary). If A is not evenly divisible by 4, add the remainder to A1. The key property to note here is that the sum of the 4 new integers is equal to the original byte value (e.g., A = A1 + A2 + A3 + A4). Perform a similar operation on B, C, and D, resulting in a total of 16 integer values, 4 from each of the original bytes, which should be rearranged in the following order:

   `A1 B1 C1 D1 A2 B2 C2 D2 A3 B3 C3 D3 A4 B4 C4 D4`

   Each of these integers represents one of the 16 characters in the final CHECKSUM keyword value. Note that if this byte stream is interpreted as 4 32-bit integers, the sum of the integers is equal to the original complemented checksum value.
3. Add 48 (hex 30), which is the value of an ASCII zero character, to each of the 16 integers generated in the previous step. This places the values in the range of ASCII alphanumeric characters '0' (ASCII zero) to 'r'. This offset is effectively subtracted back out of the checksum when the initial CHECKSUM keyword value string of 16 ASCII 0s is replaced with the final encoded checksum value.
4. To improve human readability and transcription of the string, eliminate any non-alphanumeric characters by considering the bytes a pair at a time (e.g., A1 + A2, A3 + A4, B1 + B2, etc.) and repeatedly increment the first byte in the pair by 1 and decrement the 2nd byte by 1 as necessary until they both correspond to the ASCII value of the allowed alphanumeric characters 0–9, A–Z, and a–z shown in Figure 1. Note that this operation conserves the value of the sum of the 4 equivalent 32-bit integers, which is required for use in this checksum application.
5. Cyclically shift all 16 characters in the string one place to the right, rotating the last character (D4) to the beginning of the string. This rotation compensates for the fact that the fixed format *FITS* character string values are not aligned on 4-byte word boundaries in the *FITS* file. (The first character of the string starts in column 12 of the header card image, rather than column 13).

6. Write this string of 16 characters to the value of the CHECKSUM keyword, replacing the initial string of 16 ASCII zeros.

To invert the ASCII encoding, cyclically shift the 16 characters in the encoded string one place to the left, subtract the hex 30 offset from each character, and calculate the checksum by interpreting the string as 4 32-bit unsigned integers. This can be used, for instance, to read the value of CHECKSUM into the software when verifying or updating a HDU.

J.3. Encoding Example

This example illustrates the encoding algorithm given in Sect. J.2 Consider a *FITS* HDU whose 1's complement checksum is 868229149, which is equivalent to hex 33C0201D. This number was obtained by accumulating the 32-bit checksum over the header and data records using 1's complement arithmetic after first initializing the CHECKSUM keyword value to '0000000000000000'. The complement of the accumulated checksum is 3426738146, which is equivalent to hex CC3FDFE2. The steps needed to encode this hex value into ASCII are shown schematically below:

```
    Byte                        Preserve byte alignment
 A  B  C  D    A1 B1 C1 D1   A2 B2 C2 D2   A3 B3 C3 D3   A4 B4 C4 D4

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

                          Eliminate punctuation characters
initial values  c  B  j  j   c  ?  g  h   c  ?  g  h   c  ?  g  h
    .           c  C  j  j   c  >  g  h   c  @  g  h   c  >  g  h
    .           c  D  j  j   c  =  g  h   c  A  g  h   c  =  g  h
    .           c  E  j  j   c  <  g  h   c  B  g  h   c  <  g  h
    .           c  F  j  j   c  ;  g  h   c  C  g  h   c  ;  g  h
    .           c  G  j  j   c  :  g  h   c  D  g  h   c  :  g  h
final values    c  H  j  j   c  9  g  h   c  E  g  h   c  9  g  h

final string   "hcHjjc9ghcEghc9g"   (rotate 1 place to the right)
```

In this example byte B1 (originally ASCII B) is shifted higher (to ASCII H) to balance byte B2 (originally ASCII ?) being shifted lower (to ASCII 9). Similarly, bytes B3 and B4 are shifted by opposing amounts. This is possible because 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. This operation is purely for cosmetic reasons to improve readability of the final string.

This is how these CHECKSUM and DATASUM keywords would appear in a *FITS* header (with the recommended time stamp in the comment field).:

```
DATASUM = '2503531142'          /  2015-06-28T18:30:45
CHECKSUM= 'hcHjjc9ghcEghc9g'    /  2015-06-28T18:30:45
```

J.4. Incremental Updating of the Checksum

The symmetry of 1's complement arithmetic also means that after modifying a *FITS* HDU, the checksum may be incrementally updated using simple arithmetic without accumulating the checksum for portions of the HDU that have not changed. The new checksum is equal to the old total checksum plus the checksum accumulated over the modified records, minus the original checksum for the modified records.

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 HDU, no information is preserved about the HDU's original state.

The recipe for updating the CHECKSUM keyword following some change to the HDU is: $C' = C - m + m'$, where C and C' represent the HDU'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 $\tilde{}$ (tilde) denotes the (1's) complement operation. See Braden et al. (1988); Mallory & Kullberg (1990); Rijsinghani (1994). 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 (Rijsinghani 1994).

J.5. Example C Code for Accumulating the Checksum

The 1's complement checksum is simple and fast to compute. This routine assumes that the input records are a multiple of 4 bytes long (as is the case for *FITS logical records*), but it is not difficult to allow for odd length records if necessary. To use this routine, first initialize the CHECKSUM keyword to '0000000000000000' and initialize sum32 = 0, then step through all the *FITS* logical records in the FITS HDU.

```
void checksum (
   unsigned char *buf,  /* Input array of bytes to be checksummed */
                        /*  (interpret as 4-byte unsigned ints)   */
   int length,          /* Length of buf array, in bytes          */
                        /*  (must be multiple of 4)               */
   unsigned int *sum32) /* 32-bit checksum                        */
{
/*
   Increment the input value of sum32 with the 1's complement sum
   accumulated over the input buf array.
*/
   unsigned int hi, lo, hicarry, locarry, i;

   /*  Accumulate the sum of the high-order 16 bits and the */
   /*  low-order 16 bits of each 32-bit word, separately.  */
   /*  The first byte in each pair is the most significant. */
   /*  This algorithm works on both big and little endian machines.*/
   hi = (*sum32 >> 16);
   lo = *sum32 & 0xFFFF;
   for (i=0; i < length; i+=4) {
       hi += ((buf[i]   << 8) + buf[i+1]);
       lo += ((buf[i+2] << 8) + buf[i+3]);
   }

   /* fold carry bits from each 16 bit sum into the other sum */
   hicarry = hi >> 16;
   locarry = lo >> 16;
   while (hicarry || locarry) {
       hi = (hi & 0xFFFF) + locarry;
       lo = (lo & 0xFFFF) + hicarry;
       hicarry = hi >> 16;
       locarry = lo >> 16;
   }

   /* concatenate the full 32-bit value from the 2 halves */
   *sum32 = (hi << 16) + lo;
}
```

J.6. Example C Code for ASCII Encoding

This routine encodes the complement of the 32-bit HDU checksum value into a 16-character string. The byte alignment of the string is permuted one place to the right for *FITS* to left justify the string value starting in column 12.

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

int offset = 0x30;                          /* ASCII 0 (zero) */
unsigned long mask[4] = { 0xff000000, 0xff0000, 0xff00, 0xff };

void char_encode (
  unsigned int value, /* 1's complement of the checksum     */
                      /*     value to be encoded            */
  char *ascii)        /* Output 16-character encoded string */
{
   int byte, quotient, remainder, ch[4], check, i, j, k;
   char asc[32];

   for (i=0; i < 4; i++) {
       /* each byte becomes four */
       byte = (value & mask[i]) >> ((3 - i) * 8);
       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;               /* terminate the string */
}
```