emsaplib

January 27, 2025

Abstract

Library of routines developped for EPIC MOS tasks.

1 Instruments/Modes

Instrument	Mode
EPIC MOS	-

2 Description

This library contains 3 modules for F90 :

- `emutils_mod` (Sect 2.1)
- `edusoft_mod` (Sect 2.2)
- `badpixutils_mod` (Sect 2.3)

They are described in the following subsections.

It also contains the test directory `emodf` for the EPIC MOS routines, and the utilities `compare_columns` and `compare_realcols` used in test harnesses.

2.1 emutils_mod module

This module contains F90 routines and functions developped for the MOS tasks, but which are of general interest.

2.1.1 addFilename

Aim: Write history line with the name of an input file, removing the directory.

The routine declaration is:

```
    subroutine addFilename(in_tab, filename, comment)

! in_tab    : handle to the output table
! filename  : name of the file
! comment   : comment string to introduce file name

    type(BlockT), intent(in) :: in_tab
    character(len=*), intent(in) :: filename, comment
```

2.1.2 changeCase

Aim: Switch from lower to upper case or vice-versa.

The routine declaration is:

```
  function changeCase(instring,dir) result(outstring)

! instring : input string
! dir      : direction (1: to lower case; 2: to uppercase; 0: switch case)

    character(len=*), intent(in) :: instring
    integer, intent(in) :: dir
    character(len=len(instring)) :: outstring
```

2.1.3 getCcd

Aim: Get the CCD value from the keywords (returns 0 if error).

The routine declaration is:

```
  integer function getCcd(in_tab)

!  in_tab  : handle to the  input block for the DAL

    type(BlockT), intent(in) :: in_tab
```

2.1.4 getMode

Aim: Return logicals defining the data mode.

The routine declaration is:

```
  subroutine getMode(ev_tab, imaging, timing, redImaging, compTiming)

! ev_tab:     handle to the events extension
! imaging:    set to True if IMAGING (EPIC) or SPECTROSCOPY (RGS) mode
! timing:     set to True if TIMING (EPIC) or HTR (RGS) mode
! redImaging: set to True if REDUCED IMAGING (MOS) mode
! compTiming: set to True if COMPRESSED TIMING (MOS) or BURST (PN) mode

    type(TableT), intent(in) :: ev_tab
    logical, intent(out)     :: imaging, timing, redImaging, compTiming
```

2.1.5 keywordDone

Aim: Write keyword stating that an action governed by a boolean parameter was performed.

The routine declaration is:

```
  subroutine keywordDone(in_tab, taskname, paramname)

! in_tab:    handle to the table where the keyword will be written
! taskname:  name of the calling task
! paramname: name of the boolean parameter

    type(BlockT), intent(in) :: in_tab
    character(len=*), intent(in) :: taskname, paramname
```

2.1.6 keywordRemove

Aim: Remove a keyword written with keywordDone.

The routine declaration is:

```
  subroutine keywordRemove(in_tab, paramname)

! in_tab:    handle to the table where the keyword will be set to 0
! paramname: name of the boolean parameter

    type(BlockT), intent(in) :: in_tab
    character(len=*), intent(in) :: paramname
```

2.1.7 wasDone

Aim: Test using keywords whether an action governed by a boolean parameter was already performed.

The routine declaration is:

```
  function wasDone(in_tab,paramname) result(done)

! in_tab:    handle to the table where to look for the keyword
! paramname: name of the boolean parameter

    type(BlockT), intent(in) :: in_tab
    character(len=*), intent(in) :: paramname
    logical :: done
```

2.1.8 equalKeywords

Aim: Check that two files share a number of attributes.

The routine declaration is:

```
  function equalKeywords(handle1,handle2,keywList,strict,onwarn) result(compat)

! handle1     : handle to the first table or set
! handle2     : handle to the second table or data set
! keywList    : array of keywords to test for compatibility
! strict      : set to true if need to check existence also (default is true)
! onwarn      : set to true if warnings are to be sent (default is true)
!             : if false then messages are sent instead

    type(AttributableT), intent(in) :: handle1, handle2
    character(len=*), dimension(:), intent(in) :: keywList
    logical, intent(in), optional :: strict, onwarn
    logical :: compat
```

2.1.9 putPrimaryKeywords

Aim: Copy general keywords from extension to primary header.

The routine declaration is:

```
  subroutine putPrimaryKeywords(fr_tab)

! fr_tab    : Handle to the input block
```

```
    type(BlockT), intent(in)   :: fr_tab

! Names of the keywords to be copied into the primary header from the extension

    integer, parameter :: num_keyw_prim = 6
    character(len=8), dimension(num_keyw_prim), parameter :: &
       name_keyw_prim = (/"TELESCOP","INSTRUME", "OBS_ID  ", "EXP_ID  ",  &
                          "DATE-OBS","DATE-END"/)
```

2.1.10 sizeListParam

Aim: Read the size of a parameter list.

The routine declaration is:

```
   function sizeListParam( paramlist ) result( numlist )

! paramlist  : name of the string list parameter
! numlist    : number of strings in list

    character(len=*), intent(in)               :: paramlist
    integer                                    :: numlist
```

2.1.11 readListParam

Aim: Read a parameter list (strings only).

The routine declaration is:

```
  subroutine readListParam(paramlist,list,numlist)

! paramlist  : name of the string list parameter
! list       : list of strings
! numlist    : number of strings in list

    character(len=*), intent(in)                     :: paramlist
    character(len=*), dimension(:), intent(inout)    :: list
    integer, intent(in)                              :: numlist
```

2.2 edusoft_mod module

This module contains F90 declarations and routines used to interface the simulation of the Event Detection Unit (EDU) of the EPIC MOS camera. This simulation (EDUSOFT) is written in C and is here interfaced with F90.

2.2.1 declarations

Here are declarations of parameters and data structures used together with the EDUSOFT routines

- `es_nmax`: Maximum number of events that can be found in a frame by EDUSOFT.
  Type and value :

  ```
  integer, parameter :: es_nmax  = 50000
  ```

- `edu_npat`: Number of EDU patterns.
  Type and value :

  ```
  integer, parameter :: edu_npat = 32
  ```

- `edu_nsid`: Side dimension of the square EDU patterns.
  Type and value :

  ```
  integer, parameter :: edu_nsid = 5
  ```

- `edu_pattern`: Data structure describing each EDU pattern.
  Type :

  ```
  type edu_pattern
     integer               id_patt
     integer               id_mask
     integer(kind=int8)    mat(edu_nsid,edu_nsid)
     integer(kind=int8)    number
     integer(kind=int8)    id_kind
  end type edu_pattern
  ```

- `edu_evt`: Data structure describing an EDU event in output of EDUSOFT.
  Type :

  ```
  type edu_evt
     integer(kind=int16)   x
     integer(kind=int16)   y
     integer(kind=int16)   pattern
     integer(kind=int16)   e1
     integer(kind=int16)   e2
     integer(kind=int16)   e3
     integer(kind=int16)   e4
     integer(kind=int16)   peripix
  end type edu_evt
  ```

- edu_out: Data structure containing the EDUSOFT output event list.
  Type :

```
type edu_out
   integer         nevent
   type(edu_evt)   evt(es_nmax)
   integer         npix
   integer         fifoovf
end type edu_out
```

- sas_evt: Data structure describing an EDU event as used by the SAS. Type :

```
type sas_evt
   integer(kind=int8)  pattern
   integer(kind=int8)  peripix
   integer(kind=int16) rawx
   integer(kind=int16) rawy
   integer(kind=int32) frame
   integer(kind=int32) flag
   integer(kind=int16) e1
   integer(kind=int16) e2
   integer(kind=int16) e3
   integer(kind=int16) e4
end type sas_evt
```

2.2.2 getpixelInE2

Aim: Get the number of pixels making E2 for all patterns, return mask itself if required. CAL must be initialised beforehand.

The routine declaration is:

```
  subroutine getpixelInE2(pixelInE2, patabove, npatterns)

! pixelInE2: number of pixels in E2 for each pattern
! patabove : mask of E2 for each pattern
! npatterns: number of patterns

    integer(kind=int16), dimension(0:edu_npat-1), intent(out) :: pixelInE2
    integer(kind=int16), dimension(-1:1,-1:1,0:edu_npat-1),  &
                         intent(out), optional                :: patabove
    integer,             intent(out), optional                :: npatterns
```

2.2.3 inMask

Aim: Returns sum of offsets through mask.

The routine declaration is:

```
  integer function inMask(offX, offY, patabove)

! offX, offY: local column and row offsets
! patabove  : 1 means count, 0 means ignore (like output of getpixelInE2)

    integer,             dimension(-1:1),      intent(in) :: offX, offY
    integer(kind=int16), dimension(-1:1,-1:1), intent(in) :: patabove
```

2.2.4 projectEventsCounts

Aim: Project the pixels above threshold of an array of events onto an image.

The routine declaration is:

```
  subroutine projectEventsCounts(evt, patabove, image)

! evt         : Array of event structure
! patabove    : Geometry of event outside central pixel (from getpixelInE2)
! image       : Image upon which to project

    type(sas_evt), dimension(:), intent(in) :: evt
    integer(kind=int16), dimension(-1:1,-1:1,0:edu_npat-1),  &
                   intent(in)    :: patabove
    integer(kind=int32), dimension(-2:,-2:), intent(inout) :: image
```

2.2.5 projectEventsEnergy

Aim: Project the energy of an array of events onto an image.

The routine declaration is:

```
  subroutine projectEventsEnergy(evt, patabove, image)

! evt         : Array of event structure
! patabove    : Geometry of event outside central pixel (from getpixelInE2)
! image       : Image upon which to project

    type(sas_evt), dimension(:), intent(in) :: evt
    integer(kind=int16), dimension(-1:1,-1:1,0:edu_npat-1),  &
                   intent(in)    :: patabove
    integer(kind=int32), dimension(-2:,-2:), intent(inout) :: image
```

2.2.6 pat_init

Aim: Initialization of the pattern library for EDUSOFT.

The F90 calling sequence is:

```
! patterns  : input argument. Pattern library read from the CAL by a call like
!             call CAL_getEventPatterns(patterns, eduThreshold)
  integer(kind=int8),  dimension(:,:,:), pointer :: patterns


! edupat: output argument. Pattern library as used by edusoft routine.
  type(edu_pattern) :: edupat(edu_npat)

   call pat_init(patterns,edupat)
```

2.2.7 edusoft

This is the EDUSOFT call.

The F90 calling sequence is:

```
! Input arguments :

 integer :: edumode                           ! 0  diagnostic mode
                                              ! 1  timing mode
                                              ! 2  reduced imaging (threshold) mode
                                              ! 3  imaging mode

 type(edu_pattern)   :: edupat(edu_npat)  ! initialized by pat_init().
 integer(kind=int32) :: dx                !  x size of the input image.
 integer(kind=int32) :: dy                !  y size of the input image.
 integer(kind=int16), dimension(dx,dy) :: im  ! input image.
 integer(kind=int16) :: threshold         ! EDU threshold.

! x EDU offset (must contain at least x0+dx data).
 integer(kind=int16), dimension(0:x0+dx-1) :: offX

! y EDU offset (must contain at least y0+dy data).
 integer(kind=int16), dimension(0:y0+dy-1) :: offY

! x coordinate of the closest pixel to the output CCD node.
 integer(kind=int32) :: x0

! y coordinate of the closest pixel to the output CCD node.
 integer(kind=int32) :: y0

! es_nmax : Maximum number of events that can be found in a frame by EDUSOFT.
```

```
!           (See declaration subsection).

! Output argument :

! Data structure containing the EDUSOFT output event list.
 type(edu_out)  :: eduout


 call edusoft(edumode,edupat,dx,dy,im,threshold, &
             offX(0:x0+dx-1),offY(0:y0+dy-1),x0,y0,es_nmax,eduout)
```

2.3 badpixutils_mod module

This module contains F90 routines and functions developped for dealing with bad pixels, including generalist Poisson and correlation routines.

2.3.1 readBadpix

Aim: Read bad pixels table into an array.

The routine declaration is:

```
    subroutine readBadpix(bad_tab, incremental,  &
                          xbad, ybad, yext, tbad, fbad, nbad)

! bad_tab : Handle to the bad pixels table
! incremental : Should normally be set to True. If False, no bad pixel
!           is read (nbad is set to 0) and the bad pixels columns
!           (RAWX,RAWY,TYPE,YEXTENT,BADFLAG) are added to the bad_tab table
! xbad    : array of RAWX coordinates
! ybad    : array of RAWY coordinates
! yext    : array of RAWY extensions (YEXTENT)
! tbad    : array of bad pixel types (TYPE)
! fbad    : array of bad pixel status (BADFLAG: uplinked, CCF or new)
! nbad    : number of bad pixels
! Those arrays must be dimensioned (large enough) in the calling program.

    type(TableT),                      intent(in)  :: bad_tab
    logical,                           intent(in)  :: incremental
    integer(kind=int16), dimension(:), intent(out) :: xbad, ybad, yext,  &
                                                      tbad, fbad
    integer,                           intent(out) :: nbad
```

2.3.2 writeBadpix

Aim: Write bad pixels array into a table.

The routine declaration is:

```
    subroutine writeBadpix(bad_tab, xbad, ybad, yext, tbad, fbad, nbad)

! bad_tab : Handle to the output bad pixels table
!           The columns should exist already
! xbad    : array of RAWX coordinates
! ybad    : array of RAWY coordinates
! yext    : array of RAWY extensions (YEXTENT)
! tbad    : array of bad pixel types (TYPE)
! fbad    : array of bad pixel status (BADFLAG: uplinked, CCF or new)
! nbad    : number of bad pixels

    type(TableT),                       intent(in) :: bad_tab
    integer(kind=int16), dimension(:), intent(in) :: xbad, ybad, yext,  &
                                                     tbad, fbad
    integer,                            intent(in) :: nbad
```

2.3.3 mergeBad

Aim: Compute Y extent of bad pixels, remove redundancies. Column segments are built only for identical type and status. In case of redundancy, the lower status is kept (uplinked ¿ CCF ¿ new) and for the types the precedence is set as follows: HOT(1) ¿ FLICKERING(2) ¿ PIN_HOLE(4) ¿ DEAD(3) ¿ UNSPECIFIED(5) ¿ INTACT(0)

The routine declaration is:

```
    subroutine mergeBad(xbad, ybad, yext, tbad, fbad, nbad)

! xbad     : array of RAWX coordinates
! ybad     : array of RAWY coordinates
! yext     : array of RAWY extensions
! tbad     : array of bad pixel types
! fbad     : array of bad pixel status (uplinked, CCF or new)
! nbad     : number of bad pixels

    integer(kind=int16), dimension(:), intent(inout) :: xbad, ybad, yext,  &
                                                        tbad, fbad
    integer,                            intent(inout) :: nbad
```

2.3.4 readBadOffsets

Aim: Read bad offset values in SAS coordinates 1-600. Beware: contains under/overscans ¡ 1 and ¿ 600.

The routine declaration is:

```
  subroutine readBadOffsets(ev_set, offX, offY, ccdnr)

! ev_set    : Handle to the data set where the OFFSETS extension is
! offX, offY: additional offset values
! ccdnr     : CCD number as CCDNR column (if merged table)

    type(DataSetT),                       intent(in)  :: ev_set
    integer, dimension(-10:EMOS_MAX_X+20), intent(out) :: offX
    integer, dimension(-10:EMOS_MAX_Y+20), intent(out) :: offY
    integer, optional,                    intent(in)  :: ccdnr
```

2.3.5 cumulativeBinomial

Aim: Compute cumulative binomial distribution. cumulativeBinomial(Non,Noff,p) = Sum(Non to Non+Noff) PB(Non,Noff,p) PB(Non,Noff,p) is the probability to get Non source counts and Noff background counts, if p is the a priori probability that a count is attributed to the source (on assumption of no source)

The routine declaration is:

```
  real(double) function cumulativeBinomial(Non, Noff, p)

! Input:
! Non  : number of observed source counts
! Noff : number of observed background counts
! p    : a priori probability that a count is attributed to the source

    real(double), intent(in) :: p
    integer,      intent(in) :: Non, Noff
```

2.3.6 cumulativePoisson

Aim: Compute cumulative Poisson distribution over some range. cumulativePoisson(k) = Sum(0 to k) P(k)

The routine declaration is:

```
  subroutine cumulativePoisson(mu, kmin, kmax, cvf)

! mu      : average value
! kmin    : minimum number of counts
! kmax    : maximum number of counts
! cvf(1:kmax-kmin+1) : cumulative Poisson distribution from kmin to kmax

    real(double), intent(in) :: mu
    integer,      intent(in) :: kmin, kmax
```

```
    real(double), dimension(:), intent(out) :: cvf
```

2.3.7 compCumulPoisson

Aim: Compute complementary cumulative Poisson distribution over some range. compCumulPoisson(k) = Sum(k to infinity) P(k)

The routine declaration is:

```
  subroutine compCumulPoisson(mu, kmin, kmax, cvf)

! Input:
! mu      : average value
! kmin    : minimum number of counts
! kmax    : maximum number of counts
! Output:
! cvf(1:kmax-kmin+1) : complementary cumulative Poisson distribution
!                      from kmin to kmax

    real(double), intent(in) :: mu
    integer,      intent(in) :: kmin, kmax
    real(double), dimension(:), intent(out) :: cvf
```

2.3.8 quantilePoisson

Aim: Return quantiles for the Poisson distribution. Probability to get quantile or less is always ¿= 1 - epsilon. Probability to get 1+quantile or more is always ¡ epsilon. Return lower quantile if epsilon ¡ 0 such that probability to get quantile or more is always ¿= 1 - epsilon. Probability to get quantile-1 or less is always ¡ epsilon.

The routine declaration is:

```
  integer function quantilePoisson(mu, epsilon)

! mu      : average value
! epsilon : probability level

    real, intent(in) :: mu, epsilon
```

2.3.9 corrCoeff

Aim: Compute correlation coefficient and main axes from a list of (X,Y) coordinates assuming identical and independent errors on X and Y.

The routine declaration is:

```
  real function corrCoeff(x, y, theta, sigma1, sigma2)

! x, y   : list of X and Y values
! Returns if present:
! theta  : rotation angle (radians, between -pi/4 and pi/4) to main axes
! sigma1 : dispersion along theta (not necessarily major axis)
! sigma2 : dispersion perpendicular to theta

    real, intent(in) , dimension(:) :: x, y
    real, intent(out), optional     :: theta, sigma1, sigma2
```

2.3.10 localMedian

Aim: Compute median of an array (either integer or real).

The routine declaration is:

```
  function localMedian(toto,nval)

! toto: 1-D input array of integer or real values
! nval: number of values in toto to consider (optional)

    integer or real, dimension(:), intent(in)  :: toto
    integer, optional,     intent(in)  :: nval
```

2.4 energy combination

CAL_mosPhaBuild call:

Computes a single energy `PHA` (in ADU) for each event from a weighted sum of the E_i, and the residual background $Bkg(x, y)$ computed in CCDBKG, assumed not to vary with time (*i.e.* the time series output from CCDBKG is not used).
The coefficients α_1,α_2,α_3,α_4 are defined by the CAL from a CCF file. They have 1 value for each of the 32 patterns.
E_1 is entered as real in order to allow randomisation before calling CAL_mosPhaBuild.

Two different formulaes are used depending on whether α_4 is positive or negative.

If $\alpha_4 \leq 0$ (and normally ≥ -1), then the idea is to use a weighted average of E_4 and Bkg to estimate the local background. This is adapted to compact events.

$$\begin{aligned} Wght &= \alpha_1 + \alpha_2 N_{above} + \alpha_3(8 - N_{above}) \\ Pha &= \alpha_1 E_1 + \alpha_2 E_2 + \alpha_3 E_3 - Wght\left((1+\alpha_4)Bkg - \frac{\alpha_4 E_4}{16 - Peripix}\right) \end{aligned} \tag{1}$$

If $\alpha_4 \geq 0$, then the idea is to use E_4 as part of the signal, and estimate the local background entirely from Bkg. This is adapted to events spread out over many pixels.

$$\begin{aligned} Wght &= \alpha_1 + \alpha_2 N_{above} + \alpha_3(8 - N_{above}) + \alpha_4(16 - Peripix) \\ Pha &= \alpha_1 E_1 + \alpha_2 E_2 + \alpha_3 E_3 + \alpha_4 E_4 - Wght\, Bkg \end{aligned} \quad (2)$$

In both cases N_{above} is the number of secondary pixels above threshold (for example 1 for bipixels). E_4 is used only where PERIPIX < 7. E_3 and E_4 are not used if next to a bad line or column.

Depending on the calibration results (not yet known) the α_i may depend on the pattern and possibly also on energy. The idea is then to loop on **emenergy** for different selections on the events.

The C++ possible call are :

```
CalReal32Vector &EnergyCombinator::combine(const CalReal32Vector &energye1,
                                           const CalInt16Vector &energye2,
                                           const CalInt8Vector   &pattern,
                                           CalReal32Vector &pha,    // out

                                           const CalReal32Vector &locbkg,
                                           const CalInt16Vector &energye3,
                                           const CalInt16Vector &energye4,
                                           const CalInt8Vector &peripix,
                                           const CalInt32Vector &flag
                                          )
```

Input: energye1 : array of real32 with event energy E1 energye2 : array of int16 with event energy E2 energye3 : array of int16 with event energy E3 (optional) energye4 : array of int16 with event energy E4 (optional) pattern : array of int8 with event pattern number peripix : array of int8 with event peripix number (optional) flag : array of int32 with event flag (optional) locbkg : array of real32 with event local background (optional) Out: pha : array of computed event PHA

energye3, energye4, peripix and flag : are present or not in the same time, while locbkg is optional independently. Which leads to 4 possible calls.

```
CalReal32Vector &EnergyCombinator::combine(const CalReal32Vector &energye1,
                                           const CalInt16Vector &energye2,
                                           const CalInt8Vector   &pattern,
                                           CalReal32Vector &pha     // out
                                          )

CalReal32Vector &EnergyCombinator::combine(const CalReal32Vector &energye1,
                                           const CalInt16Vector &energye2,
                                           const CalInt8Vector   &pattern,
                                           CalReal32Vector &pha,    // out

                                           const CalReal32Vector &locbkg
                                          )
CalReal32Vector &EnergyCombinator::combine(const CalReal32Vector &energye1,
                                           const CalInt16Vector &energye2,
                                           const CalInt8Vector   &pattern,
                                           CalReal32Vector &pha,    // out
```

```
                                         const CalInt16Vector &energye3,
                                         const CalInt16Vector &energye4,
                                         const CalInt8Vector &peripix,
                                         const CalInt32Vector &flag
                                        )
CalReal32Vector &EnergyCombinator::combine(const CalReal32Vector &energye1,
                                         const CalInt16Vector &energye2,
                                         const CalInt8Vector   &pattern,
                                         CalReal32Vector &pha,    // out

                                         const CalReal32Vector &locbkg,
                                         const CalInt16Vector &energye3,
                                         const CalInt16Vector &energye4,
                                         const CalInt8Vector &peripix,
                                         const CalInt32Vector &flag
                                        )
```

The F90 possible call will be :

```
   subroutine CAL_mosPhaBuild(energye1, energye2, pattern, pha )


   subroutine CAL_mosPhaBuild(energye1, energye2, pattern, pha,  &
                              locbkg )

   subroutine CAL_mosPhaBuild(energye1, energye2, pattern, pha,  &
                               energye3, energye4, peripix, flag)

   subroutine CAL_mosPhaBuild(energye1, energye2, pattern, pha,  &
                              locbkg, energye3, energye4, peripix, flag)
```

2.5 emodf directory

This directory contains a very simple ODF with a single scientific exposure with CCDs 2 and 7 in Imaging mode, CCD 6 in Reduced Imaging mode, and CCD 1 in Timing mode.

2.6 compare_columns utility

This sh script allows to compare columns from two files. It ends in error whenever two values don't match.

The calling sequence is:

```
compare_columns reffile[ext] newfile[ext] "column list" "comment"

# reffile[ext]:  name of the first  file with extension number
# newfile[ext]:  name of the second file with extension number
# "column list": names of the columns to be compared (separated by a blank)
```

```
# "comment":     comment used to make the output message specific
```

2.7 compare_realcols utility

This SAS routine allows to compare real columns from two files, with absolute and relative tolerance. It ends in error whenever two values don't match.

The calling sequence is:

```
compare_realcols table=newfile(:table) reftable=reffile(:table)
                 colnames="column list" (abstol=tol1 reltol=tol2 operation=op)

# newfile(:table): name of the first  file with table (first by default)
# reffile(:table): name of the second file with table (first by default)
# "column list"  : names of the columns to be compared (separated by a blank)
# tol1           : absolute tolerance on difference (1E-4 is default)
# tol2           : relative tolerance on difference (1E-4 is default)
# op             : OR (default) to apply the less stringent of both tests
                   AND          to apply the most stringent of both tests
```

3 Errors

This section documents warnings and errors generated by this task (if any). Note that warnings and errors can also be generated in the SAS infrastructure libraries, in which case they would not be documented here. Refer to the index of all errors and warnings available in the HTML version of the SAS documentation.

getMode01 *(error)*
unexpected DATATYPE in input table

getMode02 *(error)*
No DATATYPE in input table

readBadpix01 *(error)*
Array size smaller than the number of bad pixels

getCcd10 *(warning)*
No CCDID in input table
corrective action: return 0

getCcd11 *(warning)*
No INSTRUME in input table
corrective action: return 0

getCcd12 *(warning)*
INSTRUME=EMOS and no CCDNODE in input table
corrective action: return 0

getCcd13 *(warning)*
INSTRUME=EPN and CCDID not 0,1,2 in input table
corrective action: return 0

getCcd14 *(warning)*
INSTRUME=EMOS and CCDID not in 1-7 in input table
corrective action: return 0

getCcd15 *(warning)*
INSTRUME=EPN and no QUADRANT in input table
corrective action: return 0

getCcd16 *(warning)*
Unrecognized INSTRUME in input table
corrective action: return 0

getCcd17 *(warning)*
INSTRUME=EPN and QUADRANT not in 0-3 in input table
corrective action: return 0

getCcd18 *(warning)*
INSTRUME=EMOS and CCDNODE not 0,1 in input table
corrective action: return 0

equalKeywords11 *(warning)*
One of the keywords is not present in the first input table
corrective action: return False

equalKeywords12 *(warning)*
One of the keywords is not present in the second input table
corrective action: return False

equalKeywords13 *(warning)*
One of the keywords is not identical in both tables
corrective action: return False

equalKeywords14 *(warning)*
One of the keywords has unknown type in the first input table
corrective action: ignore keyword

equalKeywords15 *(warning)*
One of the keywords does not have the same type in both tables
corrective action: return False

putPrimaryKeywords10 *(warning)*
One of the keywords is not present in the input table
corrective action: continue

sizeListParam10 *(warning)*
parameterCount returned a negative value
corrective action: return 0

readListParam10 *(warning)*
string parameter too long
corrective action: truncate

4 Future developments

References