SUN/61.4

Starlink Project
Starlink User Note 61.4

R.F. Warren-Smith

12th January 2006

TRANSFORM — Coordinate Transformation Facility
Version 0.9-6
Programmer's Guide and Reference Manual

Abstract

TRANSFORM provides a standard, flexible method for manipulating coordinate transformations and for transferring information about them between applications. It can handle coordinate systems with any number of dimensions and can efficiently process large (*i.e.* image-sized) arrays of coordinate data using a variety of numerical precisions. No specific support for astronomical coordinate transformations or map projections is included – these features are provided by the AST library.

Contents

1 Introduction

This document describes version 0.9 of the TRANSFORM coordinate transformation facility and shows how it may be used in application programs operating within the Starlink ADAM environment. It is assumed that the reader is familiar with this environment and with the Starlink Hierarchical Data System HDS (SUN/92).

The main text of this document provides a guide for programmers who have not used the TRANSFORM facility before. It contains an introduction to the overall capabilities of the software and the basic concepts involved, followed by a tutorial-style description of all the main features with examples of their use.

The Appendices at the end of the document contain further reference material which will mainly be of value to more experienced users. In particular, Appendix D gives a description of all the user-callable routines provided.

1.1 What is the TRANSFORM Facility?

The TRANSFORM facility is a library of subroutines which may be used by application programs to process information describing the relationships between different coordinate systems.

It provides a standard, flexible method for manipulating coordinate transformations and for transferring information about them between applications. It can handle coordinate systems with any number of dimensions and can efficiently process large (*i.e.* image-sized) arrays of coordinate data using a variety of numerical precisions. No specific support for astronomical coordinate transformations or map projections is included at present, but routines for handling these will appear in future. The current system provides tools for creating a wide variety of coordinate transformations, so it should be possible to construct some of the simpler astronomical transformations explicitly, if required, on an interim basis.

Some possible applications of TRANSFORM routines include:

- Defining linear and non-linear graphics coordinate systems;
- Attaching coordinate systems to datasets (*e.g.* relating image pixels to sky positions);
- Describing distortion in images and spectra;
- Storing and applying instrumental calibration functions.

Note that the TRANSFORM facility uses the Hierarchical Data System (HDS) to store its information in standard data structures for interchange between applications. These data structures may therefore be used as building blocks when constructing larger HDS datasets and also when designing "extensions" to the standard Starlink NDF data structure (see SGP/38).

2 Basic Concepts and Terminology

2.1 Transformations

The TRANSFORM facility stores the information which it requires for conversion between different coordinate systems in HDS structures called *transformations*. They have an HDS type of TRN_TRANSFORM[1] and the purpose of each of these structures is to describe the relationship which exists between two

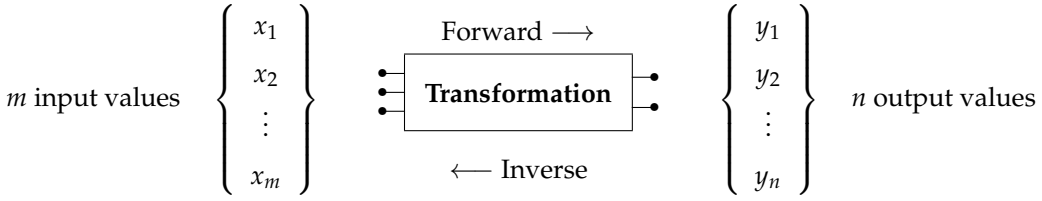
[1] Full internal details of all the HDS structures used by the TRANSFORM facility are given in Appendix C.

different coordinate systems. The two systems in question are termed the *input* and *output* coordinate systems and the HDS transformation structure serves to "link" them together as a pair. In this role, the transformation may be regarded simply as a "black box", thus:

Positions (or *points*) in each coordinate system will be specified by appropriate sets of *coordinates*, and the purpose of the transformation is to define an "association" between corresponding points in each system. Thus, given the coordinates of a point in one system, the coordinates of the associated point in the other system may be derived by using the information stored in the transformation structure.

2.1.1 Transformation variables.

The inter-conversion of coordinates in this way between two different 2-dimensional systems (*e.g.* between *Equatorial* and *Galactic* sky coordinates) is a familiar concept. In general, however, there is no need for the two coordinate systems to be 2-dimensional, nor even for both of them to have the same dimensionality. To cater for this general case, each transformation has a set of *input* and *output variables* associated with it. They are represented here by $(x_1, \ldots x_m)$ and $(y_1, \ldots y_n)$ respectively, where the numbers of input and output variables (m and n) may be any positive integer – not necessarily equal. These variables are akin to the "dummy arguments" used in Fortran subroutines and are the means by which coordinate values are passed to and from the transformation; they may be visualised as a set of input and output ports attached to it, as follows:



Any coordinate values supplied to the ports (*i.e.* variables) at either end of the transformation may be converted into a corresponding set of "transformed" coordinates (using the information within the transformation), to be delivered through the ports (*i.e.* variables) at the opposite end.

As with subroutine arguments, the names assigned to the input/output variables have no absolute significance outside the transformation itself, and coordinate values must be supplied in the correct order (typically within a data array, for instance) in order to "match up" with the appropriate transformation variables. In practice, of course, when describing a transformation, it is convenient to retain meaningful names for its variables; these may then be referred to using the usual notation as (α, δ) or (x, y, z), etc., so that the expected order is clear.

2.2 Mappings

The most important part of any transformation is a pair of "numerical recipes" which provide a description of the precise numerical steps which must be carried out in order to perform coordinate conversion

in each direction. These "recipes" are termed the *forward* and *inverse mappings*.[2] It will frequently be possible to define both of these, so that coordinate conversion may be performed in either direction. However, this may not always be possible (or desirable), so a transformation may also exist even if only one of its two possible mappings is defined. In this case, coordinate conversion will only be possible in one direction.

As will be seen later, it is often convenient to extract the mappings from a transformation and to process them as separate entities.

2.2.1 Notation

A general transformation with m input variables and n output variables, in which both the forward and inverse mappings are defined, is denoted here by $[m \leftrightarrow n]$. The double-ended arrow '$\leftrightarrow$' indicates that coordinate conversion is possible in either direction. Similarly, the notation $[m \rightarrow n]$ and $[m \leftarrow n]$ is used to represent transformations where only the forward or inverse mapping (respectively) is defined, so that coordinate conversion can be performed only in the direction indicated. This notation may also be used to describe individual transformations by including a list of their input and output variables. Thus, a transformation which relates a *Cartesian* coordinate system to a *Polar* system might be denoted by $[(x, y) \leftrightarrow (r, \theta)]$.

To distinguish *transformations* from *mappings*, the notation for the latter uses braces '$\{\ldots\}$' rather than square brackets, so that a mapping which describes how to convert a set of m input values into a set of n output values would be denoted by $\{m \rightarrow n\}$. In this case, the arrow may only point to the right because a single mapping, once separated from its parent transformation, can only perform coordinate conversion in one direction. This is always regarded as its "forward" direction.

2.3 Transformation Functions

In principle, the value of each of a transformation's output variables may depend on the values supplied to **all** of its input variables. Consequently, a general transformation's forward mapping may only be specified in full by giving a complete set of *transformation functions* which define the precise form of this dependence for each of the output variables. The same consideration also applies to the inverse mapping. Thus, in general, a transformation's two mappings may be decomposed into a set of *n forward transformation functions* (denoted $F_1, \ldots F_n$) and a set of *m inverse transformation functions* (denoted $I_1, \ldots I_m$) which act upon the input and output variables, as follows:

$$\text{Forward} \left\{ \begin{array}{lcl} y_1 & = & F_1(x_1, x_2, \ldots x_m) \\ y_2 & = & F_2(x_1, x_2, \ldots x_m) \\ \vdots & & \vdots \\ y_n & = & F_n(x_1, x_2, \ldots x_m) \end{array} \right. \qquad \text{Inverse} \left\{ \begin{array}{lcl} x_1 & = & I_1(y_1, y_2, \ldots y_n) \\ x_2 & = & I_2(y_1, y_2, \ldots y_n) \\ \vdots & & \vdots \\ x_m & = & I_m(y_1, y_2, \ldots y_n) \end{array} \right. \tag{1}$$

Example 1. Drawing graphs.

A simple illustration of mappings, transformation functions and the notation used to describe them may be taken from the $[2 \leftrightarrow 2]$ transformation commonly used to relate "data" coordinates (x_d, y_d) to "graph paper" coordinates (x_p, y_p) when drawing a graph. If the graph is linear, then the transformation's two mappings might typically be defined by the following transformation functions:

[2] Note that the term *mapping* is used here with its usual mathematical meaning, whereas the (mathematically synonymous) term *transformation* is reserved to represent a more complex structure which may contain both a forward and inverse mapping, as well as possible ancillary information. The two terms should not be confused.

$$\text{Forward mapping} \left\{ \begin{array}{lcl} x_p & = & S_x(x_d - x_0) \\ y_p & = & S_y(y_d - y_0) \end{array} \right. \quad \text{Inverse mapping} \left\{ \begin{array}{lcl} x_d & = & (x_p/S_x) + x_0 \\ y_d & = & (y_p/S_y) + y_0 \end{array} \right. \tag{2}$$

where x_0 and y_0 are zero points on the two axes, and S_x and S_y are scale factors. In this example, (x_d, y_d) have been treated as input variables, while (x_p, y_p) are output variables. This particular choice is arbitrary, although a convention would have to be adopted before writing software which used such a transformation.

Using the notation outlined above, this simple transformation would be denoted by:

$$\left[(x_d, y_d) \leftrightarrow (x_p, y_p)\right]$$

i.e. with the input variables appearing on the left. The forward mapping would then be denoted by:

$$\left\{(x_d, y_d) \rightarrow (x_p, y_p)\right\}$$

and the inverse mapping by:

$$\left\{(x_p, y_p) \rightarrow (x_d, y_d)\right\}$$

---

3 Simple use of TRANSFORM Routines

3.1 Formulating a Transformation

The first stage in creating a transformation is to formulate a description of it in a form which may be used in a program. In future, there may be a number of ways of formulating such a description and of creating a corresponding transformation from it. At present, however, only one method is supported, based on the explicit use of transformation functions.

For instance, suppose you wished to create the $[(x, y) \leftrightarrow (r, \theta)]$ transformation relating a two-dimensional *Cartesian* coordinate system (x, y) to a *Polar* system (r, θ). In this case, the transformation's two mappings might be defined in terms of the following transformation functions:

$$\text{Forward} \left\{ \begin{array}{lcl} r & = & \sqrt{x^2 + y^2} \\ \theta & = & \tan^{-1}(y/x) \end{array} \right. \quad \text{Inverse} \left\{ \begin{array}{lcl} x & = & r\cos(\theta) \\ y & = & r\sin(\theta) \end{array} \right. \tag{3}$$

This description of the transformation could now be used in a program by converting it directly into character data, as follows:

Example 2. Formulating a Cartesian-to-Polar transformation.

```
*  Define the number of input and output variables.
      INTEGER NVIN, NVOUT
      PARAMETER ( NVIN = 2, NVOUT = 2 )

*  Declare arrays for the forward and inverse transformation functions.
      CHARACTER * 25  FOR( NVOUT ), INV( NVIN )        [1]

*  Assign the forward transformation functions.
      FOR( 1 ) = 'r = sqrt( x * x + y * y )'           [2][3]
      FOR( 2 ) = 'theta = atan2( y, x )'

*  Assign the inverse transformation functions.
      INV( 1 ) = 'x = r * cos( theta )'
      INV( 2 ) = 'y = r * sin( theta )'
```

Programming notes:

(1) The transformation functions are assigned to the elements of two character arrays *FOR* and *INV*, the number of elements in each array being determined by the number of output and input variables, *NVOUT* and *NVIN* respectively.

(2) The formulae of Equation 3 have been converted into character data for storage in these arrays by using Fortran arithmetic operators and intrinsic functions (see Appendix A for a full description of the syntax of transformation functions, which closely resembles that of Fortran 77 assignment statements).

(3) The names 'x', 'y', 'r' and 'theta' have been used to represent the transformation's input and output variables. Names such as these may be chosen freely (see Appendix A) and are defined implicitly when they appear on the left hand side of a transformation function.

Unspecified mappings. In the above example, both the forward and inverse mappings were defined. If only one of these were required, however, then the other could be left unspecified. This is done simply by omitting the right hand sides (and '=' signs) from the transformation functions which define it. Thus, if the inverse transformation functions had been specified as:

```
INV( 1 ) = 'x'
INV( 2 ) = 'y'
```

then the inverse mapping would not be defined. Note that the left hand sides of the transformation functions must still appear, however, because they define the names of the transformation's input variables.

3.2 Creating a Transformation

The transformation functions formulated above may be used to create an HDS transformation structure by calling the routine TRN_NEW (create new transformation), as follows:

```
CALL TRN_NEW( NVIN, NVOUT, FOR, INV, PREC, COMM, ELOC, NAME, LOCTR, STATUS )
```

where:

- *NVIN* and *NVOUT* are the numbers of input and output variables;
- *FOR* and *INV* are the two character arrays containing the forward and inverse transformation functions;
- *PREC* is a character expression specifying the arithmetic precision with which the transformation functions will be evaluated;
- *COMM* is a comment to be stored with the transformation;
- *ELOC* is a locator to an existing HDS structure;
- *NAME* is the HDS name of the new structure component to be created;
- *LOCTR* returns a locator to the newly created transformation structure;
- *STATUS* is an inherited error status variable.

Example 3. Creating a Cartesian-to-Polar transformation.

The following arrangement might be used to create a transformation called MYTRAN from the *FOR* and *INV* arrays defined earlier:

```
*  Declare variables.
      INCLUDE 'SAE_PAR'
      INCLUDE 'TRN_PAR'                                    [1]
      CHARACTER PREC * ( TRN__SZPRC ), COMM * 80
      CHARACTER * ( DAT__SZLOC ) ELOC, LOCTR

*  Get a locator ELOC to an existing HDS structure.
      <a call to DAT_ASSOC or DAT_FIND, for instance>

*  Specify the arithmetic precision and make a comment.
      PREC = '_REAL:'                                      [2]
      COMM = '2-d Cartesian (x,y) --> 2-d Polar (r,theta)  [3]

*  Create the transformation.
      CALL TRN_NEW( NVIN, NVOUT, FOR, INV, PREC, COMM,     [4][5]
     :              ELOC, 'MYTRAN', LOCTR, STATUS )
```

Programming notes:

(1) The symbolic constant TRN__SZPRC is defined in the include file TRN_PAR. It is used to declare the length of the character variable *PREC* which will contain a precision specification.

(2) *PREC* specifies the type of arithmetic to be used when the transformation functions are evaluated and has been assigned the value '`_REAL:`', which is recommended for general use. This indicates that single precision (*real*) arithmetic should normally be used but that the precision may be increased if *double precision* data are being processed. This is discussed further in Section 5.2.

(3) The '`-->`' (or '`<--`') character sequence may be used in comment strings to indicate the direction of the forward mapping. If the transformation is subsequently *inverted* (Section 4.5), then the '`-->`' and '`<--`' symbols will be interchanged so that the comment remains valid. Comments may be of any length.

(4) TRN_NEW will check the transformation functions for correct syntax and consistent use of variable names before the transformation is created. However, no check is performed to determine whether the forward and inverse transformation functions actually define a pair of complementary mappings. It is the caller's responsibility to ensure that this is so.

(5) On successful exit from TRN_NEW, the *LOCTR* argument returns an HDS locator associated with the newly created transformation.

3.3 Temporary Transformations

The TRN_NEW routine may also be used to create a temporary transformation. This can be very convenient for internal use by an application because it avoids the need for an existing HDS structure to contain it. To create such a transformation, the *ELOC* argument to TRN_NEW should be replaced by a blank string (the *NAME* argument will then be ignored and may also be blank). Thus, if TRN_NEW were invoked by:

```
CALL TRN_NEW( NVIN, NVOUT, FOR, INV, PREC, COMM, ' ', ' ', LOCTR, STATUS )
```

then the locator *LOCTR* would subsequently be associated with a temporary transformation (*i.e.* a temporary HDS structure of type TRN_TRANSFORM).

3.4 Compilation

The transformation structures created by TRN_NEW hold information in a form which can be easily manipulated by HDS. However, this form of storage is not efficient if the transformation is to be used, perhaps repeatedly, to process large arrays of data. Therefore, before a transformation can be used to transform coordinate data, its mapping information must first be *compiled* using TRN_COMP (compile transformation), thus:

```
CALL TRN_COMP( LOCTR, FORWD, ID, STATUS )
```

The TRN_COMP routine accepts a locator *LOCTR* associated with a transformation and checks the information within it. It then compiles one of the transformation's mappings into a different form which is stored internally by the TRANSFORM facility. This internal representation is called a *compiled mapping* and may subsequently be used (for instance) to convert coordinate data from one coordinate system to another. The logical argument *FORWD* is used to specify which mapping is required – the forward mapping is compiled if this argument is .TRUE. and the inverse mapping is compiled if it is .FALSE.. An error will be reported if the requested mapping has not been defined.

Any number of transformations may be compiled by repeatedly calling TRN_COMP. To distinguish the resulting compiled mappings, a unique *integer* identifier is issued for each via the *ID* argument, and these identifiers are subsequently used to pass the mappings to other routines.

It is important to appreciate that a *compiled mapping* is a "uni-directional" object and can only perform coordinate conversion in a single direction, whereas a *transformation* (which may contain up to two mappings) is potentially "bi-directional". Thus, if an $[m \leftrightarrow n]$ transformation is compiled in the forward direction (with the *FORWD* argument of TRN_COMP set to .TRUE.), then a $\{m \rightarrow n\}$ mapping will result. Conversely, compilation in the inverse direction would yield a $\{n \rightarrow m\}$ mapping. Note that in this latter case the numbers of input and output variables will have been interchanged by the compilation process.

3.5 Inquiry Routines

An application may determine the number of input and output variables used by a transformation by calling TRN_GTNV (get numbers of variables), thus:

```
CALL TRN_GTNV( LOCTR, NVIN, NVOUT, STATUS )
```

The information is returned via the *NVIN* and *NVOUT* arguments. Since TRN_GTNV first checks that the locator *LOCTR* is associated with a valid transformation, this also provides a convenient way of validating a transformation.

A similar routine TRN_GTNVC (get numbers of compiled variables) is provided for use with compiled mappings:

```
CALL TRN_GTNVC( ID, NVIN, NVOUT, STATUS )
```

In this case, the mapping is specified by its *integer* identifier *ID*.

3.6 Transforming 1-Dimensional Coordinate Data

A number of routines are provided for applying compiled mappings to arrays of coordinate data (*i.e.* lists of coordinates). They are distinguished according to the number of variables used by the mapping, by the way the data are stored in the arrays and by the type (*i.e.* numerical precision) of the data being transformed. The common and relatively simple cases of 1- and 2-dimensional data are discussed here and in the following Section. Section 4.1 describes the use of more general mappings.

The simplest situation arises when the input and output data points are both 1-dimensional, so that a $\{1 \rightarrow 1\}$ mapping is to be applied. Routines with names of the form TRN_TR1x are provided for this purpose, where x is either **I**, **R** or **D** according to whether the data are specified by *integer*, *real* or *double precision* values respectively. Thus, to apply a compiled $\{1 \rightarrow 1\}$ mapping to a set of 1-dimensional data points specified by a list of *real* coordinates, the following statement might be used:

```
CALL TRN_TR1R( BAD, NX, XIN, ID, XOUT, STATUS )
```

where:

- *BAD* is a logical value specifying whether the input coordinates may contain *bad* values (see Section 4.2);
- *NX* specifies the number of data points to be transformed;
- *XIN* is a 1-dimensional array containing the coordinates of the input data points;
- *ID* is an identifier associated with the compiled $\{1 \rightarrow 1\}$ mapping to be applied;
- *XOUT* is a 1-dimensional array to receive the (transformed) coordinates of the output data points;
- *STATUS* is an inherited error status variable.

Example 4. Plotting a graph of a user-specified function.

In the following, TRN_TR1R is used to generate data for a graph. The function to be plotted is specified by the user via an expression obtained through the ADAM parameter system. For instance, if the user entered:

```
X*EXP(-4.4*X)
```

then a graph of the function $y = xe^{-4.4x}$ would be plotted over the range $0 \leq x \leq 1$.

```
*  Declare variables.
      INCLUDE 'SAE_PAR'
      INTEGER NPTS
      PARAMETER ( NPTS = 1000 )
      INTEGER STATUS, ID, IPOINT
      REAL X( NPTS ), Y( NPTS )
      CHARACTER EXPRS * 80, FOR( 1 ) * 82, INV( 1 ) * 1
      CHARACTER * ( DAT__SZLOC ) LOCTR

*  Obtain an expression and formulate the transformation functions.
      CALL PAR_GETOC( 'EXPRESSION', EXPRS, STATUS )          [1]
      FOR( 1 ) = 'Y=' // EXPRS
      INV( 1 ) = 'X'

*  Create a temporary transformation and compile it.
      CALL TRN_NEW( 1, 1, FOR, INV, '_REAL:',                [2]
                    'X --> f( X )', ' ', ' ', LOCTR, STATUS )
      CALL TRN_COMP( LOCTR, .TRUE., ID, STATUS )

*  Set up the X values, then transform to yield the Y values.
      DO 1 IPOINT = 1, NPTS
         X( IPOINT ) = REAL( IPOINT - 1 ) / REAL( NPTS - 1 )
    1 CONTINUE
      CALL TRN_TR1R( .FALSE., NPTS, X, ID, Y, STATUS )       [3]

*  Plot the graph and clean up.
      CALL GPL( NPTS, X, Y )                                 [4]
      CALL DAT_ANNUL( LOCTR, STATUS )                        [5]
      CALL TRN_ANNUL( ID, STATUS )
```

Programming notes:

(1) An expression is obtained via the parameter system and used to formulate suitable transformation functions. Only the forward mapping is specified.

(2) A temporary transformation is created (TRN_NEW) and is then compiled (TRN_COMP).

(3) After generating the X values, TRN_TR1R is called to apply the compiled mapping, thereby generating the corresponding Y values. The *BAD* argument is set to .FALSE. since there are no *bad* input coordinates.

(4) A graph is plotted using the GKS polyline routine GPL. It is assumed that a suitable coordinate system has been established.

(5) Lastly, a clean up is performed by annulling the temporary transformation (DAT_ANNUL) and the compiled mapping (TRN_ANNUL – described later in Section 3.8).

3.7 Transforming 2-Dimensional Coordinate Data

Transformations which inter-relate 2-dimensional coordinate systems are common in graphical and image-processing applications, so a set of routines is provided for applying the associated $\{2 \rightarrow 2\}$ mappings to coordinate data. These routines have names of the form TRN_TR2x, where x is either **I**, **R** or **D** according to whether the data are specified by *integer*, *real* or *double precision* values respectively. Thus, to apply a compiled $\{2 \rightarrow 2\}$ mapping to a set of 2-dimensional data points specified by two lists of *real* coordinates X_{IN} and Y_{IN}, the following statement might be used:

```
CALL TRN_TR2R( BAD, NXY, XIN, YIN, ID, XOUT, YOUT, STATUS )
```

where:

- *BAD* is a logical value specifying whether the input coordinates may contain *bad* values (see Section 4.2);
- *NXY* specifies the number of data points to be transformed;
- *XIN* and *YIN* are 1-dimensional arrays containing the X_{IN} and Y_{IN} coordinates of the input data points;
- *ID* is an identifier associated with the compiled $\{2 \rightarrow 2\}$ mapping to be applied;
- *XOUT* and *YOUT* are 1-dimensional arrays to receive the (transformed) X_{OUT} and Y_{OUT} coordinates of the output data points;
- *STATUS* is an inherited error status variable.

Example 5. Transforming a cursor position.

The following shows TRN_TR2R being used to transform 2-dimensional coordinate data obtained by reading a cursor. It is assumed that the identifier *ID* is associated with a compiled $\{2 \rightarrow 2\}$ mapping which has been established to convert cursor coordinates into "user" coordinates:

```
*  Declare variables.
      INTEGER STATUS, N, ID
      REAL XCUR( 1 ), YCUR( 1 ), XUSER( 1 ), YUSER( 1 )

*  Set up cursor choice device.
      CALL SGS_SELCH( 0 )
      CALL SGS_DEFCH( '.' )

*  Loop to read cursor positions.
      DO WHILE ( STATUS .EQ. SAI__OK )
         CALL SGS_REQCU( XCUR( 1 ), YCUR( 1 ), N )      [1]
         IF( N .NE. 0 ) GO TO 1

*  Transform the cursor coordinates.
         CALL TRN_TR2R( .FALSE., 1, XCUR, YCUR, ID,     [2]
     :                  XUSER, YUSER, STATUS )

*  Display the position in user coordinates.
         CALL MSG_SETR( 'X', XUSER( 1 ) )
         CALL MSG_SETR( 'Y', YUSER( 1 ) )
         CALL MSG_OUT( ' ', 'X = ^X ; Y = ^Y', STATUS ) [3]
      ENDDO
    1 CONTINUE
```

Programming notes:

(1) The SGS cursor position is read repeatedly until the user terminates interaction (by pressing the keyboard '.' key in this case).

(2) TRN_TR2R is used to transform each cursor position in turn into "user" coordinates. Only a single data point is transformed here, but in a non-interactive application it would be more efficient to transform a whole set of data points in a single call to this routine.

(3) The transformed positions are displayed in "user" coordinates.

3.8 Clearing Up and Closing Down

When a compiled mapping is no longer required, the resources associated with it should be released. This is done by calling TRN_ANNUL (annul compiled mapping), thus:

```
CALL TRN_ANNUL( ID, STATUS )
```

This causes the compiled mapping to be deleted and the *ID* value to be reset to TRN__NOID.[3] Finally, before an application finishes, the statement:

```
CALL TRN_CLOSE( STATUS )
```

should be executed. Calling TRN_CLOSE will first annul any compiled mappings which are still active and will then close the TRANSFORM facility down, recovering any resources associated with it.

Both of these cleaning up routines will attempt to execute regardless of the value of their *STATUS* argument.

4 Additional Features

4.1 Transforming General Coordinate Data

As well as providing routines for applying compiled mappings to 1- and 2-dimensional data (Sections 3.6 & 3.7), the TRANSFORM facility also has a set of routines for applying more general mappings. These are appropriate, for instance, when the number of input or output data coordinates is greater than 2 (or when these numbers are unequal) or when the number of coordinates is not known in advance. In such cases, all the input and output coordinates must each reside in single data arrays.

These general routines have names of the form TRN_TRNx, where x is either **I**, **R** or **D** according to whether the data are specified by *integer*, *real* or *double precision* values respectively. Thus, to transform a general set of data points specified by an array of *real* coordinates, the routine TRN_TRNR would be used, thus:

```
CALL TRN_TRNR( BAD, ND1, NCIN, NDAT, DATA, ID, NR1, NCOUT, RESULT, STATUS )
```

[3] The symbolic constant TRN__NOID is defined in the include file TRN_PAR. It is provided as a "null" value, which is guaranteed never to be used as a compiled mapping identifier. It may be used, when necessary, to indicate that an identifier is not currently associated with a compiled mapping.

where:

- *BAD* is a logical value specifying whether the input coordinates may contain *bad* values (see Section 4.2);
- *ND1* specifies the first dimension of the *DATA* array;
- *NCIN* specifies the number of coordinates per input data point;
- *NDAT* specifies the number of data points to be transformed;
- *DATA* is a 2-dimensional array containing a list of coordinates for the input data points;
- *ID* is an identifier associated with the compiled $\{NCIN \rightarrow NCOUT\}$ mapping to be applied;
- *NR1* specifies the first dimension of the *RESULT* array;
- *NCOUT* specifies the number of coordinates per output data point;
- *RESULT* is a 2-dimensional array to receive a list of (transformed) coordinates for the output data points;
- *STATUS* is an inherited error status variable.

Example 6. Mapping 3-dimensional coordinates into 2 dimensions.

In the following, a set of data points representing a 3-dimensional object is converted into a related set of 2-dimensional points using TRN_TRNR. The resulting points are then plotted using GKS. With a suitable choice of mapping, such an arrangement might be used to generate perspective pictures of 3-dimensional objects:

```
*  Define dimensions of the DATA and RESULT arrays.
      PARAMETER ( MAXPTS = 5000, MAXDIM = 3 )

*  Declare variables and arrays.
      INTEGER IPOINT, STATUS, ID
      REAL X, Y, Z, DATA( MAXPTS, MAXDIM ), RESULT( MAXPTS, MAXDIM )

*  Generate 3-d coordinates of helix.
      DO 1 IPOINT = 1, 3601
         THETA = REAL( IPOINT - 1 ) * 2.0 * 3.14159 / 360.0
         X = COS( THETA )                              [1]
         Y = SIN( THETA )
         Z = REAL( IPOINT - 1 ) / 360.0

*  Enter coordinates into the DATA array.
         DATA( IPOINT, 1 ) = X                         [2]
         DATA( IPOINT, 2 ) = Y
         DATA( IPOINT, 3 ) = Z
    1 CONTINUE

*  Transform the data points.
      CALL TRN_TRNR( .FALSE., MAXPTS, 3, 3601, DATA, ID, MAXPTS, 2,
     :               RESULT, STATUS )                [3]

*  Plot the resulting 2-dimensional data points.
      CALL GPL( 3601, RESULT( 1, 1 ), RESULT( 1, 2 ) ) [4]
```

Programming notes:

(1) The coordinates of a set of 3-dimensional data points are generated.

(2) The coordinates are stored so that *DATA(I, J)* contains the *J'th* coordinate of data point *I*. These coordinates need not fill the entire array.

(3) TRN_TRNR is called to apply the compiled mapping. The size of the input and output coordinate arrays are specified, together with the numbers of coordinates associated with each input and output data point. An error will result if the compiled mapping does not have the appropriate numbers of input and output variables.

(4) The 2-dimensional output data points are plotted. Their coordinates are stored in the *RESULT* array in a similar manner to those in the *DATA* array. As before, these coordinates need not entirely fill the *RESULT* array.

4.2 Handling of *Bad* Coordinate Values

During the process of transforming coordinate data, numerical errors (such as division by zero or overflow) will inevitably occur from time to time. There is no need to include specific protection against this when a transformation is formulated, however, because such errors will automatically be trapped and converted into one of the standard Starlink *bad* values.[4]

This form of error trapping is performed by all the routines which apply compiled mappings to coordinate data. The resulting "*bad* coordinates" will, where necessary, be propagated through each stage in the evaluation of a compiled mapping, so that all the results which have been affected by numerical errors can later be identified. No error report is currently made (or *STATUS* value set) if a numerical error of this nature occurs.

All the routines which apply compiled mappings to coordinate data can also recognise *bad* coordinate values supplied as input, and will propagate these if required. Frequently, however, there will not be any *bad* coordinates in the input data stream and it may be possible to save appreciable amounts of processing time by disabling recognition of these values (thereby eliminating unnecessary checking), especially when large arrays are being processed. Each relevant routine therefore carries a *logical* argument called *BAD*, which specifies whether recognition of *bad* input data is required (execution will generally be faster if *BAD* is .FALSE.). Note that this argument only affects recognition of *bad* data in the **input** stream, while numerical errors which occur during the computation are always converted into *bad* values and correctly propagated to the output.

4.3 Concatenating Transformations

It is common to find that the relationship between two coordinate systems is most easily expressed in terms of several transformations applied in succession. For instance, the transformation between positions on the sky and the pixel coordinates of a CCD image might be divided into two stages; the first representing the effect of imaging the sky into the focal plane of the telescope, and the second taking account of the position, size and orientation of the detector in the focal plane. The TRANSFORM facility makes explicit provision for cases such as this by allowing transformations to be *concatenated*.

A rather general case of concatenation is illustrated in Figure 1. In this example, *Transformation 1* relates two input variables (x_1, x_2) to three "intermediate" variables (y_1, y_2, y_3), which are also the input

[4] The Starlink *bad* values are symbolic constants with names of the form VAL__BADx, where x is either **D**, **R**, **I**, **W**, **UW**, **B** or **UB** according to the data type in question. These constants are defined in the include file PRM_PAR (see SGP/38 & SUN/39 for further information).

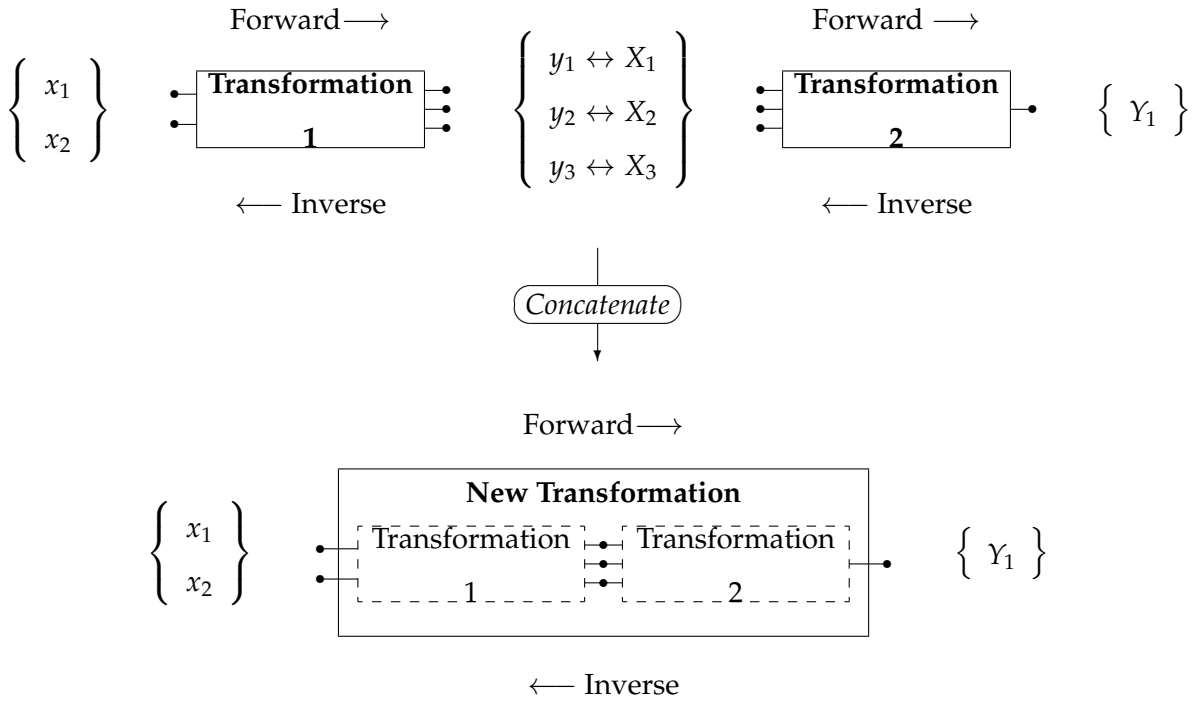


Figure 1: Concatenating two transformations.

variables (X_1, X_2, X_3) of *Transformation 2*. This transformation, in turn, has a single final output variable (Y_1). The concatenation process involves eliminating the three intermediate variables and storing the two transformation definitions together in a single new transformation. This new transformation then has two input variables (x_1, x_2) and a single output variable (Y_1). Using the '.' symbol to represent concatenation, this entire process may be summarised as:

$$[2 \leftrightarrow 3].[3 \leftrightarrow 1] = [2 \leftrightarrow 1]$$

or, in general:

$$[i \leftrightarrow j].[j \leftrightarrow k] = [i \leftrightarrow k]$$

Note that the number of output variables from the first transformation must be equal to the number of input variables to the second transformation. For obvious reasons, also, it is not permitted to concatenate a transformation in which only the forward mapping is defined with another in which only the inverse mapping is defined (*e.g.* $[i \rightarrow j]$ could not be concatenated with $[j \leftarrow k]$).

The result of concatenating two transformations is itself a transformation, so the process may be repeated indefinitely, making it possible for a whole sequence of transformations to be joined together and processed as a single unit. Once transformations have been combined in this way, however, they cannot later be separated.

4.3.1 Performing concatenation

Concatenation of a pair of transformations is performed by the routine TRN_JOIN (concatenate transformations), as follows:

```
CALL TRN_JOIN( LOCTR1, LOCTR2, ELOC, NAME, LOCNEW, STATUS )
```

where:

- *LOCTR1* and *LOCTR2* are locators to two existing transformations;
- *ELOC* is a locator to an existing HDS structure;
- *NAME* is the HDS name of the new structure component to be created;
- *LOCNEW* returns a locator to the newly created transformation structure;
- *STATUS* is an inherited error status variable.

As with the TRN_NEW routine (Section 3.3), a temporary transformation may also be produced by leaving the *ELOC* argument (and optionally the *NAME* argument) blank when TRN_JOIN is invoked.

4.4 Prefixing and Appending Transformations

In addition to producing new transformations by concatenating existing ones, it is also possible to modify an existing transformation by *prefixing* or *appending* another one to it. For example, to prefix one transformation to another, TRN_PRFX (prefix transformation) would be used, thus:

```
CALL TRN_PRFX( LOCTR1, LOCTR2, STATUS )
```

In this case, the two transformations with locators *LOCTR1* and *LOCTR2* are concatenated (as above), but the resultant transformation replaces the second one, so that the first transformation is, in effect, prefixed to it. The first transformation itself is not altered.

The routine TRN_APND (append transformation) behaves in a similar way, except that it appends the second transformation to the first. In this case the second transformation remains unchanged.

Example 7. Adjusting an astrometric calibration.

Suppose *LOCTRS* is a locator to a transformation which relates the pixel coordinates (x, y) of a CCD image to positions on the sky (α, δ). If the size of this image is altered during data reduction, then its pixel coordinates will change and the astrometric calibration will need adjustment to take account of this.

If the x and y pixel coordinates are reduced by (say) 19 and 25 units, then a transformation inter-relating the old and new pixel coordinates could be formulated as follows:

$$\text{Forward} \begin{cases} x_{old} = x_{new} + 19 \\ y_{old} = y_{new} + 25 \end{cases} \qquad \text{Inverse} \begin{cases} x_{new} = x_{old} - 19 \\ y_{new} = y_{old} - 25 \end{cases} \tag{4}$$

and then created. This transformation (with locator *LOCTRA*, say) represents the necessary adjustment which should be applied as a prefix to the original transformation, thus:

```
CALL TRN_PRFX( LOCTRA, LOCTRS, STATUS )
```

The modified transformation (*LOCTRS*) will then correctly relate the image's **new** pixel coordinates to positions on the sky, the necessary adjustment having been achieved through the following concatenation process:

$$[(x_{new}, y_{new}) \leftrightarrow (x_{old}, y_{old})].[(x_{old}, y_{old}) \leftrightarrow (\alpha, \delta)] = [(x_{new}, y_{new}) \leftrightarrow (\alpha, \delta)]$$

Note that it is not necessary to know anything about the nature of the original transformation $[(x_{old}, y_{old}) \leftrightarrow (\alpha, \delta)]$ in order to apply such a correction.

4.5 Inverting Transformations

Inversion of a transformation is a straightforward process involving the inter-change of the forward and inverse mappings (the numbers of input and output variables will also be inter-changed). It is performed by TRN_INV (invert transformation), thus:

```
CALL TRN_INV( LOCTR, STATUS )
```

where *LOCTR* is a locator to the transformation to be inverted. After such a call, compiling the transformation in the **forward** direction will yield the same compiled mapping as would have been obtained by compiling in the **inverse** direction prior to calling TRN_INV.

The main use for TRN_INV is in adapting transformations which have been created or supplied the "wrong way round" for some purpose (*e.g.* concatenation with another transformation).

4.6 Formatting Transformation Functions

To simplify the process of formatting transformation functions, a set of routines is provided which allows numerical or textual values to be inserted into "template" character strings. These routines largely eliminate the need to use Fortran WRITE statements to construct transformation functions containing numerical values.

The routines have names of the form TRN_STOK[x], where x is either **I**, **R** or **D** according to whether the type of value to be inserted is *integer*, *real* or *double precision* respectively. The routine TRN_STOK (*i.e.* with x omitted altogether) is used for making textual (rather than numerical) insertions but otherwise functions in the same way. These routines rely on the concept of a *token*,[5] which is placed in a character string at the point where an insertion is to be made and which is subsequently replaced by the value to be inserted. For example, if the character variable *TEXT* had the value:

```
'magnitude = - 2.5 * log10( count ) + zero_point'
```

then TRN_STOKR (substitute a *real* token value) could be used to substitute a numerical value for the '`zero_point`' token, as follows:

```
CALL TRN_STOKR( 'zero_point', 17.7, TEXT, NSUBS, STATUS )
```

This causes all occurrences of the '`zero_point`' token to be replaced with the formatted *real* number '`17.7`', while *NSUBS* returns the number of substitutions made (in this case there will only be one). The value of *TEXT* then becomes:

```
'magnitude = - 2.5 * log10( count ) + 17.7'
```

Note that the TRN_STOKx routines will respect the syntax of transformation functions, so that negative numerical values will be enclosed in parentheses before a substitution is made. For instance, if the call to TRN_STOKR had been:

```
CALL TRN_STOKR( 'zero_point', -17.7, TEXT, NSUBS, STATUS )
```

then the '`zero_point`' token would be replaced by '`(-17.7)`' to prevent the illegal expression '`...+ -17.7`' from being produced. The need for two extra characters to accommodate these parentheses must be remembered when declaring the size of character strings.

[5] Tokens take the same form as the variable names used in transformation functions – *i.e.* they may contain only alpha-numeric characters (including underscore) and must begin with an alphabetic character. Tokens may be of any length and may use mixed case (token substitution is not case-sensitive) but embedded blanks are not allowed.

Example 8. A "packaged" transformation.

This illustrates the use of TRN_STOKR in a subroutine to "package up" the creation of a simple [1 ↔ 1] transformation which contains several adjustable numerical parameters. The same principles can be followed to write routines for creating a variety of specialised transformations.

```
      SUBROUTINE LINEAR( SCALE, ZERO, ELOC, NAME, LOCTR, STATUS )

*  Declare variables.
      INCLUDE 'SAE_PAR'
      INTEGER STATUS, NSUBS
      REAL SCALE, ZERO, INVSCL
      CHARACTER * ( * ) ELOC, NAME, LOCTR
      CHARACTER FOR( 1 ) * 80, INV( 1 ) * 80

*  Check STATUS.
      IF ( STATUS .NE. SAI__OK ) RETURN

*  Formulate the forward transformation function.
      FOR( 1 ) = 'out = ( in - zero ) * scale'                  [1]
      CALL TRN_STOKR( 'zero', ZERO, FOR( 1 ), NSUBS, STATUS )
      CALL TRN_STOKR( 'scale', SCALE, FOR( 1 ), NSUBS, STATUS )

*  If possible, formulate the inverse transformation function.
      IF( SCALE .NE. 0.0 ) THEN
         INVSCL = 1.0 / SCALE
         INV( 1 ) = 'in = ( out * inv_scale ) + zero'           [2]
         CALL TRN_STOKR( 'zero', ZERO, INV( 1 ), NSUBS, STATUS )
         CALL TRN_STOKR( 'inv_scale', INVSCL, INV( 1 ), NSUBS, STATUS )

*  Inverse is undefined...
      ELSE
         INV( 1 ) = 'in'                                        [3]
      ENDIF

*  Create the transformation.
      CALL TRN_NEW( 1, 1, FOR, INV, '_REAL:',                   [4]
     :              'Shift and linear scaling: in --> out',
     :              ELOC, NAME, LOCTR, STATUS )

      END
```

Programming notes:

(1) The forward transformation function is assigned and the numerical parameters *SCALE* and *ZERO* are substituted into it.

(2) If possible, the inverse transformation function is defined similarly. Note that the reciprocal of *SCALE* is taken so that multiplication may be used in preference to the less efficient division operation.

(3) If *SCALE* is zero, then the inverse mapping cannot be defined so the inverse transformation function is assigned a value to indicate this.

(4) The transformation is created.

Token delimiters. Tokens should normally be delimited from surrounding text (such as variable names) by non-alphanumeric characters, otherwise they will not be recognised. The syntax of transformation functions ensures that this will normally be so, but, when this is not possible, enclosing angle brackets '<...>' may also be used as token delimiters. In this case the brackets will be replaced as if they were part of the token itself. By this means, values may be substituted to form part of a variable name if required.

Example 9. Substituting values into variable names.

This simple example formulates a set of transformation functions to multiply each ordinate of a 3-dimensional coordinate system (X_1, X_2, X_3) by 2.0 to yield the coordinates (Y_1, Y_2, Y_3):

```
      INTEGER I, NSUBS, STATUS
      CHARACTER FOR( 3 ) * 30

      DO 1 I = 1, 3
         FOR( I ) = 'Y<I> = X<I> * 2.0'
         CALL TRN_STOKI( 'I', I, FOR( I ), NSUBS, STATUS )
    1 CONTINUE
```

Recursive substitution. The routine TRN_STOK, which allows text to be substituted in place of a token, admits some more sophisticated possibilities, including the insertion of text which itself contains tokens (perhaps including further instances of the original token). Although the substitution performed by a single call to TRN_STOK will not be affected by the presence of tokens within the substituted text (*i.e.* the substitution is not recursive), the routine may nevertheless be invoked repeatedly to perform recursive substitution if required. This capability can be used to construct more complex expressions which have a suitable recursively defined form.

Example 10. Constructing a polynomial expression.

The following constructs an expression representing a polynomial in X with an arbitrary number of numerical coefficients:

```
*  Declare variables.
      INTEGER NCOEFF, STATUS, I, NSUBS
      REAL COEFF( NCOEFF )
      CHARACTER * ( * ) EXPRS

*  Initialise the expression.
      EXPRS = '<next_term><coeff>'                    [1]

*  Substitute each coefficient value.
      DO 1 I = 1, NCOEFF
         CALL TRN_STOKR( 'coeff', COEFF( I ), EXPRS,  [2]
     :                   NSUBS, STATUS )

*  Expand the token representing the next term.
         IF( I .NE. NCOEFF ) THEN
            CALL TRN_STOK( 'next_term',               [3]
     :                     '(<next_term><coeff>)*X+',
```

```
      :                    EXPRS, NSUBS, STATUS )

*  Eliminate the final '<next_term>' token.
         ELSE
            CALL TRN_STOK( 'next_term', ' ',          [4]
      :                    EXPRS, NSUBS, STATUS )
         ENDIF
    1  CONTINUE
```

Programming notes:

(1) The character variable *EXPRS*, which is to contain the expression, is initialised to the value '`<next_term><coeff>`'.

(2) The '`<coeff>`' token is substituted with a coefficient value (say 0.1) to give '`<next_term>0.1`'.

(3) The '`<next_term>`' token is expanded to give '`(<next_term><coeff>)*X+0.1`'. The process is then repeated to replace the '`<next_term><coeff>`' part of this expression using the value of the next coefficient.

(4) After the last coefficient value has been substituted the remaining '`<next_term>`' token is eliminated by replacing it with a blank string.

The effect of this algorithm with (say) the 5 polynomial coefficients 0.1, 0.12, 1.06, −4.4 & 1E−7, would be to produce the following expression:

```
'((((1E-7)*X+(-4.4))*X+1.06)*X+0.12)*X+0.1'
```

which casts the polynomial into a form for efficient evaluation using Horner's method.

---

5 More Advanced Topics

5.1 Classifying Transformations

As well as holding mapping information describing how to convert from one coordinate system to another, a transformation may also carry information about the character of its mappings. When plotting a graph, for instance, it may be important to know whether the mapping being used is linear, because a more complex (and costly) algorithm may be required if it is not. Indeed, in many circumstances, the absence of a vital property such as linearity could actually make it impossible for an application to proceed, in which case it must issue an appropriate error message and abort.

There are a number of mapping characteristics, in addition to linearity, which can influence or simplify the coding of applications in this way. However, it is often difficult to determine by indirect means (such as transforming test points) whether the necessary special properties are present. Provision has therefore been made for handling this type of information explicitly.

Property	*Brief Description*	*Symbolic Constant*
LINEAR	Preserves straight lines.	TRN__LIN
INDEPENDENT	Preserves the independence of the axes.	TRN__INDEP
DIAGONAL	Preserves the axes themselves.	TRN__DIAG
ISOTROPIC	Preserves angles and shapes (*e.g.* circles).	TRN__ISOT
POSITIVE_DET	A component of reflection is absent.	TRN__POSDT
NEGATIVE_DET	A component of reflection is present.	TRN__NEGDT
CONSTANT_DET	The area (or volume) scale factor is constant.	TRN__CONDT
UNIT_DET	Areas (or volumes) are preserved.	TRN__UNIDT

Table 1: The basic classification properties which may be declared for a transformation and the symbolic constants associated with each. The constants are defined in the include file TRN_PAR.

Classification properties. Usually, information about any special characteristics which are present is directly available only to the application which creates a transformation. Consequently, this application should be responsible for declaring that such properties are present, so that other applications may subsequently enquire about them. This process is termed *classification* and the *basic classification properties* which may be declared are indicated in Table 1, where each is briefly described. These properties are defined more precisely (and mathematically) in Appendix B.

In most situations, a transformations's mappings are not adequately described by any one of the basic properties alone, but require a *composite* classification comprising a set of several of these properties. For instance, the combination:

LINEAR *and* ISOTROPIC *and* POSITIVE_DET *and* UNIT_DET

would indicate that a mapping represents a rigid rotation about an axis (or a point in two dimensions). Facts such as this may not be obvious without some thought, however, so those classifications which apply to a number of the most common types of mapping are set out in Table 4 in Appendix B.

Classification information is conveniently processed in the form of a 1-dimensional *logical classification array*, each of whose elements indicates the presence or absence of one particular property. Each basic property therefore has an *integer* symbolic constant associated with it (see Table 1) which identifies the array element to be used (the precise mechanism is illustrated below). Not all possible combinations of the basic properties are permitted (see Appendix B) but the number of different classifications possible is nevertheless still quite large.

Inserting classification information. It is important to appreciate that classifying a transformation is not mandatory, but merely assists other applications in making effective use of it. The desirability of classification therefore depends largely on the type of applications which are likely to process the transformation. For instance, a comprehensive general-purpose application, which might be used in conjunction with a wide range of other software, would probably take care to classify a transformation fully, whereas a simpler application, perhaps designed for personal use, might not. However, a declaration of linearity, if it applies, is generally recommended.

When a transformation is first created using TRN_NEW it has no classification information associated with it, so this information has to be inserted explicitly. This is done by calling TRN_PTCL (put classification), as follows:

Example 11. Inserting classification information.

```
*  Declare variables.
      INCLUDE 'TRN_PAR'
      LOGICAL CLASS( TRN__MXCLS )                        [1]

*  Create a (temporary) transformation.
      CALL TRN_NEW( NVIN, NVOUT, FOR, INV, PREC,         [2]
     :              ' ', ' ', LOCTR, STATUS )

*  Set up the logical classification array.
      DO 1 I = 1, TRN__MXCLS                             [3]
         CLASS( I ) = .FALSE.
    1 CONTINUE
      CLASS( TRN__LIN ) = .TRUE.                         [4]
      CLASS( TRN__DIAG ) = .TRUE.
      CLASS( TRN__ISOT ) = .TRUE.
      CLASS( TRN__POSDT ) = .TRUE.

*  Enter the classification information.
      CALL TRN_PTCL( CLASS, LOCTR, STATUS )              [5]
```

Programming notes:

(1) A 1-dimensional *logical* array *CLASS* is declared with *TRN__MXCLS* elements (this symbolic constant specifies the number of basic classification properties currently recognised and is defined in the include file TRN_PAR).

(2) A transformation is created. At this point it does not contain any classification information.

(3) All elements of the *CLASS* array are explicitly initialised to .FALSE. – this precaution is recommended so that the number of basic classification properties may be increased in future without adversely affecting existing software.

(4) The required array elements are set to .TRUE. – in this example the classification describes a simple linear magnification about a point with a positive magnification factor (see Table 4 in Appendix B). Symbolic constants (also defined in the include file TRN_PAR) are used to identify the array elements concerned.

(5) The classification information is entered into the transformation by calling TRN_PTCL.

Retrieving classification information. The routine TRN_GTCL (get classification) is provided for retrieving classification information from a transformation so that applications may enquire about the properties which have been declared. Unless the context dictates otherwise, all applications which import transformations should make such an enquiry and should not assume that a transformation has any special property unless its classification information indicates that this is so.

Example 12. Retrieving classification information.

```
*  Declare variables.
      INCLUDE 'TRN_PAR'
```

```
      LOGICAL CLASS( TRN__MXCLS ), OK                  [1]

*  Get the classification information.
      CALL TRN_GTCL( LOCTR, .TRUE., CLASS, STATUS )    [2]

*  Test for the required properties.
      OK = CLASS( TRN__LIN ) .AND.                     [3]
     :     CLASS( TRN__ISOT )
```

Programming notes:

(1) A *logical* array *CLASS* is declared with *TRN__MXCLS* elements.

(2) TRN_GTCL is called to retrieve classification information from a transformation, returning it in the *CLASS* array. The second (*logical*) argument to TRN_GTCL is set .TRUE. in this example, which indicates that information is required about the forward mapping (as opposed to the inverse mapping). An error would result if this mapping were not defined.

(3) The appropriate elements of the CLASS array are tested to determine whether the mapping has the required property. In this case, OK is set .TRUE. if the forward mapping is LINEAR and ISOTROPIC. In two dimensions this would ensure that it preserves straight lines, angles and shapes (*e.g.* circles).

The routine TRN_GTCLC (get compiled classification) is also provided to retrieve classification information from compiled mappings (Appendix D).

Automatic classification processing. The TRANSFORM software provides for a certain amount of automatic processing to take place whenever an exchange of classification information occurs. The simplest form which this takes is *validation*, which will detect errors such as an attempt to declare an inconsistent set of classification properties. In addition, the software is capable of "filling in" any properties which are missing but which can be deduced from the others supplied. Some instances where properties can be deduced in this way are indicated by the open circles in Table 4 – it would not be necessary to specify these items explicitly, as TRANSFORM routines would supply them automatically.

Automatic processing also takes place whenever transformations are *concatenated*, *prefixed* or *appended* (Sections 4.3 & 4.4). In this case, the classification information available from each of the contributing transformations is combined to deduce the set of properties which apply to the result. Note, however, that only those properties which **necessarily** follow can be deduced in this way. For instance, it would be possible for the non-linearities in two mappings to cancel when their transformations are concatenated, giving an overall linear result. However, since this is not the case in general, it could not be deduced automatically. If the automatic classification processing provided should prove inadequate (for instance, the application may have some additional information available to it), then a new classification may be derived explicitly and "re-declared" by calling TRN_PTCL, which will replace any pre-existing classification information.

5.2 Arithmetic Precision

When a transformation is created, a *precision specification* is associated with it (*i.e. via* the *PREC* argument to TRN_NEW – Section 3.2) and this subsequently determines the type of arithmetic (*integer*, *real* or *double precision*) which will be used to evaluate the transformation functions. The correct choice of this specification is important if the desired behaviour is to be achieved. Two considerations normally apply:

(1) **The type of calculation being performed.** For instance, *integer* arithmetic might be required if effects due to rounding were being exploited, whereas transformation functions representing a 2-dimensional rotation would probably need to use *real* arithmetic, even if the coordinate data were to be stored as integers. Some functions may also require *double precision* arithmetic to reduce internal rounding errors.

(2) **The type of data being transformed.** It would clearly be inappropriate to use *real* arithmetic on *double precision* data but, conversely, it would be inefficient to use a higher precision than was necessary to preserve the accuracy of *real* data.

It can be seen that an appropriate arithmetic precision cannot necessarily be selected solely on the basis of the type of calculation to be performed, because the type of coordinate data to be processed may also be relevant. Unfortunately, this latter information may not be available to the application which creates the transformation. Furthermore, whenever transformations are concatenated (Section 4.3), each must be able to accommodate data passed to it by neighbouring transformations although it may not have any advance knowledge of the type of arithmetic its neighbours will be using.

To accommodate these possibilities without unnecessary loss of data precision, transformations are allowed some flexibility in adapting their internal arithmetic to the external data being processed. This is controlled by the precision specification, which is a character string taking one of the following six values:

Fixed precisions { '_INTEGER' '_REAL' '_DOUBLE' } Elastic precisions { '_INTEGER:' '_REAL:' '_DOUBLE:' }

Fixed precisions specify the type of arithmetic to be used explicitly. In this case, the data used in the calculation (both incoming coordinates and explicit numerical constants in the transformation functions) are first converted to the specified numerical type (*integer*, *real* or *double precision*) and the transformation functions are then evaluated using the appropriate type of arithmetic.

N.B. It is currently assumed that the data types of all the input and output coordinates are the same and all the transformation functions are therefore evaluated using the same type of arithmetic. This may change in future to allow coordinates to have mixed data types. The present arrangement is designed so that changes to existing applications and datasets will not generally be necessary.

Elastic precisions are similar, except that some subsequent adjustment is also allowed. In this case, the precision specification indicates the **minimum** precision of the arithmetic to be used, but this may be increased if the data type warrants it. Thus, a precision specification of '_REAL:' would request that *real* arithmetic be used unless *double precision* data were being processed (in which case *double precision* arithmetic would be used instead). In general, the type of arithmetic used is related to the precision specification and the type of data being processed as follows:

Elastic Precision Specification

Data Type	'_INTEGER:'	'_REAL:'	'_DOUBLE:'
_INTEGER	*integer*	*real*	*double precision*
_REAL	*real*	*real*	*double precision*
_DOUBLE	*double precision*	*double precision*	*double precision*

Once the arithmetic precision has been decided, the transformation functions are evaluated in the same way as for a fixed precision (*i.e.* by converting all incoming data and constants to the appropriate data type and then performing the arithmetic).

When several transformations have been concatenated, the type of arithmetic to be used is determined individually for each stage in the overall transformation (data type conversion being performed automatically between each stage as necessary). If an elastic precision (which is sensitive to the input data type) has been specified, then the input data type is taken to correspond with the type of arithmetic used in the previous mapping to be evaluated (or the input data type itself if appropriate). This arrangement avoids any unnecessary loss of data precision regardless of the order in which transformations are concatenated.

6 Compiling and Linking

The files required for compiling and linking applications which use the TRANSFORM facility reside in the standard locations – usually `/star/include` for the Fortran INCLUDE files, `/star/lib` for the object library and `/star/bin` for the development and link scripts.

Files and routines from the PRIMDAT facility (SUN/39) are also required. The INCLUDE file names used by both of these facilities must first be defined by executing the commands:

```
% prm_dev
% trn_dev
```

Applications which contain Fortran INCLUDE statements associated with the TRANSFORM facility may then be compiled.

ADAM applications should be linked using the link script `trn_link_adam`. This script automatically links in the required PRIMDAT library as well as all the other required Starlink libraries. For instance, to link an application called PROG, the ADAM command:

```
% alink prog.f -o prog `trn_link_adam`
```

should be used.

A Transformation Functions

This Appendix describes the form and syntax of transformation functions and the way in which they are used to define transformations.

A.1 General Form

A transformation is defined by specifying two ordered sets of transformation functions which define its forward and inverse mappings. These functions are stored as character data (typically in the elements of two CHARACTER arrays) and have the general form:

<variable> [= <expression>]

where <variable> is a valid *variable* name and <expression> is an arithmetic *expression*. Transformation functions therefore resemble Fortran 77 assignment statements, except that the contents of the square brackets [...] may be omitted in appropriate circumstances (see below).

Variables and data coordinates. Variables are used in transformation functions to represent the coordinates of data points which will be transformed. Their names may be chosen freely to make the definition as comprehensible as possible; they may be of any length and may contain any alphanumeric character (including underscore) although they must start with an alphabetic character.

A variable becomes defined when it appears on the left hand side of a transformation function and the name of each variable must appear on the left hand side of one (and only one) such function. Variables which appear on the left hand side of **forward** transformation functions are termed **output** variables, while those appearing on the left hand side of **inverse** transformation functions are termed **input** variables.

Variables act only as "dummy arguments" and their names have no significance outside the transformation being defined. The correspondence between variables and the coordinates of external data points is established by the order in which the variables are defined. In the following, for instance, the first coordinate of an input data point would correspond with the variable 'ALPHA' and the second coordinate with 'OMEGA', while the two coordinates of an output data point would correspond with the variables 'p' and 'q' respectively:

$$\text{Forward} \begin{cases} \text{'p} = \texttt{ALPHA} + 2' \\ \text{'q} = 44 - 2 * \texttt{OMEGA}' \end{cases} \qquad \text{Inverse} \begin{cases} \text{'ALPHA} = \texttt{p} - 2' \\ \text{'OMEGA} = (44 - \texttt{q})/2' \end{cases}$$

Note that expressions appearing on the right hand side of **forward** transformation functions may only refer to **input** variables, while those appearing on the right hand side of **inverse** transformation functions may only refer to **output** variables.

Unspecified mappings. Either (although not both) of a transformation's mappings may be left unspecified by omitting the right hand sides (and '=' signs) from all the relevant transformation functions. Thus, the transformation given by:

$$\text{Forward} \begin{cases} \text{'p} = \texttt{ALPHA} + 2' \\ \text{'q} = 44 - 2 * \texttt{OMEGA}' \end{cases} \qquad \text{Inverse} \begin{cases} \text{'ALPHA}' \\ \text{'OMEGA}' \end{cases}$$

has only the forward mapping defined (the remainder of the inverse functions serves simply to define the names of the input variables). To define only the **inverse** mapping, the right hand sides of all the **forward** transformation functions would be omitted instead.

A.2 Expression Syntax

The right hand sides of transformation functions closely follow the syntax of Fortran 77 arithmetic expressions. The main differences (in addition to the less restrictive naming rules for variables) are:

- Transformation functions may be written in upper, lower or mixed case and there there is no limit on their length;
- Only the generic form of intrinsic functions is available, but some additional built-in functions are provided;
- The data types of variables and constants are interpreted differently (see below and Section 5.2).

The following describes particular features of the expression syntax in more detail.

Constants. Numerical constants may be written using any of the standard Fortran 77 forms (*integer*, *real* or *double precision*). For positive constants, a preceding + sign is optional. Thus, all the following are valid constants:

0	−1	57	+666	1.0	+3.	0.5438
.303	−.5	1.234d6	−4.6e−3	9E4	+.44D+19	3e0

The special constant '<BAD>' may also be used and represents a *bad* (*i.e.* undefined) value; any expression containing it evaluates to the standard Starlink *bad* value for the data type being transformed.

At present there is no distinction between the data types of constants, so the form in which they are written does not matter and their interpretation depends only on the type of arithmetic in use when the expression is evaluated. This, in turn, may depend on the type of data being transformed (Section 5.2) so constant values are converted automatically to the data type required. For instance, a constant written as '2.1' might be interpreted as

integer (2), *real* (2.1E0), or *double precision* (2.1D0) according to the type of arithmetic being used. *N.B. Handling of data types may change in future to allow implicit type conversion and "mixed mode" arithmetic. To avoid possible problems, floating-point to integer conversion of constants should currently be avoided if the constant has a fractional part. In practice such cases are rare.*

Arithmetic operations. The standard arithmetic operators +, −, ∗, / and ∗∗ are available and their use is identical to Fortran 77, thus:

'Q + 3.0'	...	add 3.0 to Q
'-6.7 - DATA'	...	subtract DATA from −6.7
'INPUT * 14'	...	multiply INPUT by 14
'4 / V2'	...	divide 4 by V2
'NUMBER ** INDEX'	...	raise NUMBER to the power INDEX

The normal rules of operator precedence apply. Matching pairs of parentheses may also be used (and are recommended) to explicitly define the order of expression evaluation; they may be nested arbitrarily deeply. Note that the precision with which arithmetic is performed is not determined until an expression is actually evaluated (Section 5.2).[6]

[6] Expressions such as 'X∗∗2.0' and 'X∗∗2' cannot currently be distinguished, whereas in Fortran they are different and the latter will evaluate more efficiently. Consequently (unless *integer* arithmetic is being used) the exponentiation operator '∗∗' is an inefficient way of squaring a number and the equivalent expression 'X∗X' is preferred. This deficiency will be removed in future.

Boolean operations. The standard logical/boolean operators .EQ., .NE., .GT., .LT., .GE., .LE., .OR., .AND. and .NOT. are available and their use is identical to Fortran 77, except that there is no distinct logical data type. These boolean operators have numerical operands and return numerical values. A numerical value is considered "true" if it is non-zero and "false" if it is zero.[7] The above operators return a value of 1 for "true" and zero for "false".

These operators may also be specified within an expression using the equivalent C forms: `==`, `!=`, `>`, `<`, `>=`, `<=`, `||`, `&&` and `!`.

These operators can be used with the built-in function "QIF". This function has 3 arguments. It returns the value of its second argument if its first argument is non-zero (i.e. "true"), and returns the value of its third argument otherwise. It can be used like the "?" operator in the C programming language.

A.3 Built-in Functions

Standard functions. A set of standard built-in functions is available and corresponds closely with the Fortran 77 set of intrinsic functions (Table 2). Function invocations take the same form as in Fortran and may be nested arbitrarily deeply.

These built-in functions are all *generic* and will adapt to the type of arithmetic being used. Some, however, do not support *integer* arithmetic and a *bad* value will be returned if this is in use (see Table 2). There are no data type conversion functions at present.

Additional functions. Some additional built-in functions, which do not match the standard Fortran 77 set, are also defined. At present these simply comprise the VAX Fortran extensions to this set (Table 3). They are also generic but do not support *integer* arithmetic.

In addition, the following non-standard functions are provided:

IDV - Performs integer division. It has two arguments. The function value is formed by taking the integer part of the two arguments, dividing them, and then returning the integer part of the result.

QIF - Has three arguments. It returns the value of its second argument if its first argument is non-zero (i.e. "true"), and returns the value of its third argument otherwise. It can be used like the "?" operator in the C programming language.

[7] This is the same convention used in the C programming language.

Function	*Number of arguments*	*Description*
SQRT	1	square root: $\sqrt{\text{arg}}$
LOG	1	natural logarithm: $\ln(\text{arg})$
LOG10	1	common logarithm: $\log_{10}(\text{arg})$
EXP	1	exponential: $\exp(\text{arg})$
ABS	1	absolute (positive) value: $\lvert\text{arg}\rvert$
NINT	1	nearest integer value to arg
MAX	2 or more	maximum of arguments
MIN	2 or more	minimum of arguments
DIM	2	Fortran DIM (positive difference) function
MOD	2	Fortran MOD (remainder) function
SIGN	2	Fortran SIGN (transfer of sign) function
SIN*	1	sine function: $\sin(\text{arg})$
COS*	1	cosine function: $\cos(\text{arg})$
TAN*	1	tangent function: $\tan(\text{arg})$
ASIN*	1	inverse sine function: $\sin^{-1}(\text{arg})$
ACOS*	1	inverse cosine function: $\cos^{-1}(\text{arg})$
ATAN*	1	inverse tangent function: $\tan^{-1}(\text{arg})$
ATAN2*	2	Fortran ATAN2 (inverse tangent) function
SINH*	1	hyperbolic sine function: $\sinh(\text{arg})$
COSH*	1	hyperbolic cosine function: $\cosh(\text{arg})$
TANH*	1	hyperbolic tangent function: $\tanh(\text{arg})$
*Function does not support *integer* arithmetic.		

Table 2: The standard built-in functions. The angular arguments/results of all trigonometric functions are in radians.

Function	*Number of arguments*	*Description*
SIND	1	sine function: sin(arg)
COSD	1	cosine function: cos(arg)
TAND	1	tangent function: tan(arg)
ASIND	1	inverse sine function: $\sin^{-1}$(arg)
ACOSD	1	inverse cosine function: $\cos^{-1}$(arg)
ATAND	1	inverse tangent function: $\tan^{-1}$(arg)
ATAN2D	2	VAX Fortran ATAN2D (inverse tangent) function

Table 3: Additional built-in functions; angular arguments/results are in degrees. None of these functions supports *integer* arithmetic.

B Classification Properties

This Appendix describes the classification properties which may be declared and associated with a transformation (Section 5.1 shows how this is done and how the information may subsequently be retrieved). In order to be precise, the definitions given here are necessarily mathematical. Readers who require simpler and more specific information about how to classify a particular transformation may find Table 4 helpful.

B.1 General

Many of the properties described here depend on the nature of a *Jacobian matrix* associated with a transformation; there are potentially two of these matrices, corresponding with the forward and inverse mappings. Using the notation of Equation 1, the Jacobian matrix $\mathbf{J}_F$ associated with the forward mapping is the $n \times m$ matrix of partial derivatives:

$$\mathbf{J}_F = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_m} \\ \vdots & \vdots & & \vdots \\ \frac{\partial y_n}{\partial x_1} & \frac{\partial y_n}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix} \tag{5}$$

while that associated with the inverse mapping $\mathbf{J}_I$ is the equivalent $m \times n$ matrix obtained by interchanging input and output variables (x and y) throughout.

The significance of these matrices can be seen by considering a simple linear mapping in two dimensions. Such a mapping is capable of representing a combination of a shift of origin, magnification, rotation, reflection and shearing deformation:

$$\begin{aligned} y_1 &= ax_1 + bx_2 + c \\ y_2 &= dx_1 + ex_2 + f \end{aligned} \tag{6}$$

It may be re-written as the matrix equation:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \mathbf{J} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} c \\ f \end{bmatrix} \quad \text{where } \mathbf{J} = \begin{bmatrix} a & b \\ d & e \end{bmatrix} \tag{7}$$

The Jacobian matrix **J** therefore contains the coefficients which define this mapping and determine its character, apart from a shift of origin. The determinant of **J** ($\det \mathbf{J} = ae - bd$) is the signed "area scale factor" which the mapping introduces (*i.e.* the area of the parallelogram produced when the mapping acts on a unit square). In more than two dimensions, $\det \mathbf{J}$ would be the equivalent "volume scale factor".

If the mapping is not linear, then the Jacobian matrix will vary from point to point. Nevertheless, it may still be regarded as a local linear approximation to the true mapping (apart from re-location of the origin) and $\det \mathbf{J}$ can still be interpreted as the **local** area (or volume) scale factor, which may now change from point to point.

Changing dimensionality. For transformations with an equal number of input and output variables ($m = n$), the Jacobian matrices $\mathbf{J}_F$ and $\mathbf{J}_I$ associated with the forward and inverse mappings (if specified) will both be square. If the transformation functions are correctly formulated, then these two matrices will be mutually inverse and will satisfy:

$$\mathbf{J}_F \mathbf{J}_I = \mathbf{J}_I \mathbf{J}_F = \mathbf{I} \tag{8}$$

where **I** is an identity matrix. Their determinants will also be related by:

$$\det \mathbf{J}_I = \frac{1}{\det \mathbf{J}_F} \tag{9}$$

As a consequence of this (and the definitions of the basic classification properties given below) any property which applies to one of a transformation's two mappings will necessarily apply to the complementary mapping also.

If the transformation affects a change of dimensionality, however, so that $m \neq n$, then it is possible that certain properties may only apply to one of its two mappings. It is still acceptable to associate such properties with the transformation, however, because the TRANSFORM software will take account of the number of input/output variables, and will omit properties which it knows cannot apply when information about a particular mapping is requested. In general, therefore, a classification property may be declared for a transformation if **either** of its mappings has that property.

B.2 Basic Properties

The basic classification properties are defined as follows:

LINEAR: A mapping has this property if all its output variables are related to its input variables by linear arithmetic expressions. Such a mapping will preserve straight lines. In two dimensions, examples of LINEAR mappings include shifts of origin, rotations, reflections, magnifications and shearing deformations.

- In general, a mapping is LINEAR if all its first derivatives are constant (*i.e.* do not change from point to point).

INDEPENDENT: A mapping has this property if a change in each input variable causes a corresponding change in only a single distinct output variable. Such a mapping will preserve the independence of the coordinate axes. A simple example in two dimensions would be the interchange of the two axes.

- In general, a mapping is INDEPENDENT if there is at most one element in each row and column of its Jacobian matrix which is not identically zero.

DIAGONAL: A mapping has this property if each output variable depends only on the **corresponding** input variable, so that the coordinate axes are preserved. There are many examples of such mappings in two dimensions, including those normally used for scaling linear (and logarithmic) graphs. Note that a DIAGONAL mapping is more strongly constrained than an INDEPENDENT mapping (above) in which the coordinate axes may be interchanged. A DIAGONAL mapping is necessarily always INDEPENDENT.

- In general, a mapping is DIAGONAL if its Jacobian matrix is square and diagonal (*i.e.* all its off-diagonal terms are identically zero). If the Jacobian matrix is not square, then the mapping is DIAGONAL if $\frac{\partial y_i}{\partial x_j}$ is identically zero for all $i \neq j$.

ISOTROPIC: A mapping has this property if it locally preserves shapes and the angles between lines. Such a mapping may apply a local scale factor to the distances between neighbouring points, but this factor will not depend on the orientation of the line between the two points, although it may vary from point to point. In two dimensions, an ISOTROPIC mapping will convert a circle at any point into another circle (but possibly of a different size and in a different place), whereas a non-ISOTROPIC mapping would produce an ellipse. If the mapping is also LINEAR (see above) then circles of any size will behave in this way, whereas with a non-LINEAR mapping this may only be true for circles of infinitely small size. Isotropy is an important property of *conformal* map projections.

- In general, a mapping is ISOTROPIC if it introduces a distance scale factor between neighbouring points which does not depend on the relative orientation of the points. This will be true if its Jacobian matrix **J** satisfies:

$$\tilde{\mathbf{J}}\mathbf{J} = \lambda \mathbf{I} \tag{10}$$

  where $\tilde{\mathbf{J}}$ denotes the transpose of **J**, λ is a constant and **I** is an identity matrix. A mapping cannot have this property if the number of output variables is less than the number of input variables. A mapping with only one input and one output variable is necessarily always ISOTROPIC.

POSITIVE_DET: A mapping has this property if the determinant of its Jacobian matrix is greater than zero at all points. In two dimensions, such a mapping can locally represent rotations, magnifications and shearing deformations and can globally represent "rubber-sheet" distortions, but it will lack any component of reflection. A string of text subjected to such a mapping would remain legible (although possibly highly distorted) and would not be converted into a mirror image of itself.

- This property can only apply to mappings with an equal number of input and output variables.

NEGATIVE_DET: A mapping has this property if the determinant of its Jacobian matrix is less than zero at all points. In two dimensions, such a mapping will locally include a component of reflection (possibly also combined with rotation, magnification and shearing deformation) and can globally represent "rubber-sheet" distortion combined with a reflection. A string of text subjected to such a mapping would be converted into a mirror image of itself (in addition to any other distortion present).

- This property can only apply to mappings with an equal number of input and output variables.

N.B. A mapping may not have both the POSITIVE_DET and NEGATIVE_DET properties simultaneously. It is also possible that neither of these properties may apply if the determinant is positive at some points and negative at others.

CONSTANT_DET: A mapping has this property if its area (or volume) scale factor has the same value at all points. If the mapping has an equal number of input and output variables, then this will be true if the determinant of its Jacobian matrix has the same value at all points. Mappings which are LINEAR (see above) necessarily have the CONSTANT_DET property, but it can also apply to non-LINEAR mappings and is an important property of *equal area* map projections.

- A mapping cannot have this property if the number of output variables is less than the number of input variables.

UNIT_DET: A mapping has this property if the absolute value of its area (or volume) scale factor is unity (and it has the same sign) at all points. If the mapping has an equal number of input and output variables, then this will be true if the determinant of its Jacobian matrix has an absolute value of unity (and the same sign) at all points. This is a stronger constraint than the CONSTANT_DET property (above) and a mapping with the UNIT_DET property necessarily has the CONSTANT_DET property also. In addition, one of the two properties POSITIVE_DET or NEGATIVE_DET will apply.

- A mapping cannot have this property if the number of output variables is less than the number of input variables.

B.3 Composite Properties

Many important mapping properties are *composite*; *i.e.* they depend on the presence of several of the basic properties above in combination. Table 4 lists the more important of these and the following notes augment the information in this Table. The presence of a possible shift of origin is disregarded throughout:

A – Shift of origin. The mapping implements a simple shift of coordinate origin, the nature of which must be determined by transforming a test point.

B – Rotation about an axis. The mapping represents a simple rotation about an axis (a point in two dimensions) without associated magnification or distortion. If the DIAGONAL property also applies, then the amount of rotation will be zero, so the mapping reduces to a shift of origin (see A above).

C – Magnification about a point. The mapping applies a simple positive magnification (a zoom) factor about a point without any associated rotation or other form of distortion. If the magnification factor is negative, then a component of reflection will be introduced if the number of input/output variables is odd. In this case the POSITIVE_DET property should be replaced by NEGATIVE_DET.

D – Graphical scaling (linear). This type of mapping is commonly used to scale the axes of a graph, with different scale factors being applied to each axis. Either POSITIVE_DET or NEGATIVE_DET will also apply depending on the sign of the scale factors in use and whether they result in a mirror image. POSITIVE_DET will apply if the number of negative scale factors is even and NEGATIVE_DET will apply if this number is odd.

E – Graphical scaling (non-linear). This type of mapping is commonly used to non-linearly scale the axes of graphs (to produce a log-log plot for instance). Since the non-linear functions used are normally monotonic, either the POSITIVE_DET or NEGATIVE_DET property will usually apply, depending on the sign of the scaling functions' derivatives along each axis. POSITIVE_DET will

Type of Mapping

Basic Property	A	B	C	D	E	F	G	H	I
LINEAR	●	●	●	●	×	●	●	–	–
INDEPENDENT	○	–	○	○	○	●	○	–	–
DIAGONAL	●	–	●	●	●	×	●	–	–
ISOTROPIC	●	●	●	–	–	●	●	●	–
POSITIVE_DET	●	●	●	?	?	?	?	●	●
NEGATIVE_DET	×	×	×	?	?	?	?	×	×
CONSTANT_DET	○	○	○	○	–	○	○	–	●
UNIT_DET	●	●	–	–	–	●	●	–	–

Mapping types:
A – Shift of origin
B – Rotation about an axis
C – Magnification about a point
D – Graphical scaling (linear)
E – Graphical scaling (non-linear)
F – Interchange of axes
G – Axis reversal
H – Conformal map projection
I – Equal area map projection

Symbols:
● – Required
○ – Implied
× – Prohibited
– – Irrelevant
? – See note in text

Table 4: Common types of mapping with their composite classification properties.

apply if the number of negative derivatives is even and NEGATIVE_DET will apply if this number is odd.

F – Interchange of axes. The mapping simply interchanges coordinate values. The property POSITIVE_DET will apply if the resulting axis permutation is *cyclic* and NEGATIVE_DET will apply if the permutation is *non-cyclic*.

G – Axis reversal. The mapping reverses one or more of the axes (*i.e.* changes the sign of the coordinates with or without the addition of a constant). The POSITIVE_DET property will apply if the number of axes reversed is even, while NEGATIVE_DET will apply if this number is odd.

H – Conformal map projection. This implements a conformal map projection which locally preserves shapes and angles but may introduce a scale factor which varies from point to point.

I – Equal area map projection. The mapping implements an equal area map projection in which the area scale factor does not vary from point to point, although shapes and the angles between lines may be distorted.

C HDS Structures

This Appendix describes the HDS structures used by the TRANSFORM facility.[8] There are three of these, distinguished by their HDS type:

TYPE = TRN_TRANSFORM structures are used to hold data defining a complete (possibly multi-stage) *transformation* and are handled directly by user-level TRANSFORM routines.

TYPE = TRN_MODULE structures are used to hold *transformation module* information which represents a simple 1-stage transformation without associated classification information. These structures are not handled directly by user-level TRANSFORM routines.

TYPE = TRN_CLASS structures are used to hold *classification* information about the properties of a transformation. These structures are also not handled directly by the user-level TRANSFORM routines.

The following Sections describe these structures in detail.

C.1 The TRN_TRANSFORM Structure

An HDS structure of type TRN_TRANSFORM contains the complete specification of a (possibly multi-stage) transformation with its associated classification information. It is defined as follows:

Components of a TRN_TRANSFORM Structure

Component Name	*HDS Type*	*Typical Value*	*Optional?*
TRN_VERSION	_REAL	0.9	no
FORWARD	_CHAR	'DEFINED'	no
INVERSE	_CHAR	'UNDEFINED'	no
MODULE_ARRAY()	TRN_MODULE	<array of structures>	no
CLASSIFICATION	TRN_CLASS	<structure>	yes

The structure components have the following meanings and restrictions:

TRN_VERSION is a mandatory scalar _REAL component containing the version number of the TRANSFORM software which created or last modified the TRN_TRANSFORM structure or any of its components. This information allows programs linked with old versions of TRANSFORM software to detect if they are processing data structures created by more recent versions, with which they may not be compatible.

FORWARD is a mandatory scalar _CHAR component which must contain one of the two values 'DEFINED' or 'UNDEFINED' (case insensitive). It specifies whether the transformation's **forward** mapping is defined or undefined.

[8] The data structures described here may be subject to change in future. However, apart from minor changes in the way data precision is handled, the current structures will continue to be supported by future versions of user-level TRANSFORM routines.

INVERSE is a mandatory scalar _CHAR component which must contain one of the two values 'DEFINED' or 'UNDEFINED' (case insensitive). It specifies whether the transformation's **inverse** mapping is defined or undefined. It is not permitted for both the FORWARD and INVERSE components to have the value 'UNDEFINED'.

MODULE_ARRAY() is a mandatory 1-dimensional array of structures of type TRN_MODULE. It holds a sequence of transformation modules which are to be used in succession to define the overall transformation. If the FORWARD component has the value 'DEFINED', then all the modules in this array must have their **forward** mappings defined. Similarly, if the INVERSE component has the value 'DEFINED', then all the modules must have their **inverse** mappings defined. Adjacent modules in this array must have matching numbers of output and input variables. The number of input variables for the transformation as a whole is determined by the number of input variables for the first module in this array, while the number of output variables is determined by the number of output variables for the final module.

CLASSIFICATION is an optional scalar structure component of type TRN_CLASS. If present, it holds the classification information for the entire transformation. If absent, a default set of classification properties applies, which corresponds to all elements of the transformation's logical classification array being set to .FALSE..

C.2 The TRN_MODULE Structure

A structure of type TRN_MODULE contains transformation module information, which represents the simplest form of 1-stage transformation without associated classification information. More complex transformations may be built up by joining several of these modules together, as is done in the MODULE_ARRAY component of a TRN_TRANSFORM structure. The TRN_MODULE structure is defined as follows:

Components of a TRN_MODULE Structure

Component Name	*HDS Type*	*Typical Value*	*Optional?*
NVAR_IN	_INTEGER	2	no
NVAR_OUT	_INTEGER	2	no
COMMENT	_CHAR	'2-d Cartesian --> 2-d Polar'	yes
PRECISION	_CHAR	'_REAL:'	no
FORWARD_FUNC()	_CHAR	'R=SQRT(X*X+Y*Y)','THETA=A...'	no
INVERSE_FUNC()	_CHAR	'X=R*COS(THETA)','Y=R*SIN(...'	no

The structure components have the following meanings and restrictions:

NVAR_IN is a mandatory scalar _INTEGER component with a positive value specifying the number of **input** variables for the transformation which the module describes.

NVAR_OUT is a mandatory scalar _INTEGER component with a positive value specifying the number of **output** variables for the transformation which the module describes.

COMMENT is an optional scalar _CHAR component of arbitrary length which contains a comment associated with the transformation module. This comment should describe (in English) the type of transformation the module performs. The contents of the COMMENT component are only intended to be human-readable and should not be machine-processed, except that the special three-character sequences '-->' and '<--' (if present) may be automatically interchanged to reflect the effect of inverting the transformation module.

PRECISION is a mandatory scalar _CHAR component of arbitrary length which contains a valid precision specification for the transformation module, according to the syntax described in Section 5.2.

FORWARD_FUNC() is a mandatory 1-dimensional _CHAR array, with elements of arbitrary length, containing transformation functions which define the **forward** mapping (according to the syntax in Appendix A). The number of array elements must match the value stored in the NVAR_OUT component.

INVERSE_FUNC() is a mandatory 1-dimensional _CHAR array, with elements of arbitrary length, containing transformation functions which define the **inverse** mapping (according to the syntax in Appendix A). The number of array elements must match the value stored in the NVAR_IN component.

C.3 The TRN_CLASS Structure

A structure of type TRN_CLASS contains information specifying the set of classification properties associated with a transformation. These properties are described in Appendix B. The TRN_CLASS structure is defined as follows:

Components of a TRN_CLASS Structure

Component Name	*HDS Type*	*Typical Value*	*Optional?*
LINEAR	_LOGICAL	TRUE	yes
INDEPENDENT	_LOGICAL	TRUE	yes
DIAGONAL	_LOGICAL	TRUE	yes
ISOTROPIC	_LOGICAL	FALSE	yes
POSITIVE_DET	_LOGICAL	TRUE	yes
NEGATIVE_DET	_LOGICAL	FALSE	yes
CONSTANT_DET	_LOGICAL	TRUE	yes
UNIT_DET	_LOGICAL	FALSE	yes

All the structure components are scalar _LOGICAL objects, whose presence is optional. Their names correspond with the classification properties described in Appendix B and their logical values indicate whether the transformation has the associated property. If any component is absent, its value defaults to .FALSE.

D Routine Descriptions

D.1 Routine List

The following is a complete list of user-level TRANSFORM routines with a brief description of their purpose. Full specifications are given in the next Section.

TRN_ANNUL(ID, STATUS)
Annul compiled mapping

TRN_APND(LOCTR1, LOCTR2, STATUS)
Append transformation

TRN_CLOSE(STATUS)
Close the TRANSFORM facility

TRN_COMP(LOCTR, FORWD, ID, STATUS)
Compile transformation

TRN_GTCL(LOCTR, FORWD, CLASS, STATUS)
Get classification

TRN_GTCLC(ID, CLASS, STATUS)
Get compiled classification

TRN_GTNV(LOCTR, NVIN, NVOUT, STATUS)
Get numbers of variables

TRN_GTNVC(ID, NVIN, NVOUT, STATUS)
Get numbers of compiled variables

TRN_INV(LOCTR, STATUS)
Invert transformation

TRN_JOIN(LOCTR1, LOCTR2, ELOC, NAME, LOCTR, STATUS)
Concatenate transformations

TRN_NEW(NVIN, NVOUT, FOR, INV, PREC, COMM, ELOC, NAME, LOCTR, STATUS)
Create new transformation

TRN_PRFX(LOCTR1, LOCTR2, STATUS)
Prefix transformation

TRN_PTCL(CLASS, LOCTR, STATUS)
Put classification

TRN_STOK[x](TOKEN, VALUE, TEXT, NSUBS, STATUS)
Substitute token

TRN_TR1x(BAD, NX, XIN, ID, XOUT, STATUS)
Transform 1-dimensional data

TRN_TR2x(BAD, NXY, XIN, YIN, ID, XOUT, YOUT, STATUS)
Transform 2-dimensional data

TRN_TRNx(BAD, ND1, NCIN, NDAT, DATA, ID, NR1, NCOUT, RESULT, STATUS)
Transform general data

D.2 Full Routine Specifications

This Section gives full specifications for the user-level TRANSFORM routines. The following notation is used to specify the types of routine arguments:

Notation	:	Fortran type
L	:	LOGICAL
I	:	INTEGER
R	:	REAL
C	:	CHARACTER$*$($*$)
?	:	Unspecified, dependent on the data type being processed.
()	:	A 1-dimensional array of one of the above types.
(,)	:	A 2-dimensional array of one of the above types.
$\mathbf{L}_{CLS}$	:	A 1-dimensional LOGICAL array, with TRN__MXCLS[9] elements, to contain classification information.
$\mathbf{C}_{PRC}$	:	A CHARACTER argument to contain a precision specification of maximum length TRN__SZPRC.[9]
$\mathbf{C}_{LOC}$	:	A CHARACTER$*$(DAT__SZLOC)[10] argument containing an HDS locator. In some routines where this is an input argument, a blank string (of any length) may be substituted to specify different routine behaviour; the individual routine descriptions give details.
$\mathbf{C}_{NAM}$	:	A CHARACTER argument specifying an HDS name string of maximum length DAT__SZNAM.[10]

[9]The symbolic constants TRN__MXCLS and TRN__SZPRC are defined in the include file TRN_PAR.
[10]The symbolic constants DAT__SZLOC and DAT__SZNAM are defined by HDS.

TRN_ANNUL
Annul compiled mapping

Description:

Annul the compiled mapping associated with the identifier supplied. Resources associated with the compiled mapping are released and the identifier is reset to TRN__NOID.

Invocation:

```
CALL TRN_ANNUL( ID, STATUS )
```

Arguments:

ID = I

Compiled mapping identifier to be annulled.

STATUS = I

The global status

Notes:

- The symbolic constant TRN__NOID is defined in the include file TRN_PAR.
- This routine attempts to execute even if *STATUS* is set on entry, although no error report will be made if it subsequently fails under these circumstances.

TRN_APND
Append transformation

Description:

Concatenate two transformations; the first transformation is altered by appending the second one to it. The second transformation is not altered.

Invocation:

```
CALL TRN_APND( LOCTR1, LOCTR2, STATUS )
```

Arguments:

LOCTR1 = C_{LOC}
Locator to the first transformation (to be altered).

LOCTR2 = C_{LOC}
Locator to the second transformation (to be appended).

STATUS = I
Inherited error status.

Notes:

Before two transformations may be concatenated, the following conditions must apply:

- The number of output variables from the first transformation must match the number of input variables to the second transformation.
- The transformations must have at least one mapping (forward or inverse) in common.

TRN_CLOSE
Close the TRANSFORM facility

Description:

Close the TRANSFORM facility down, annulling all compiled mappings and releasing all resources. No action is taken if the facility is already closed.

Invocation:

```
CALL TRN_CLOSE( STATUS )
```

Arguments:

STATUS = I

Inherited error status.

Notes:

This routine attempts to execute even if the *STATUS* value is set on entry, although no error report will be made if it subsequently fails under these circumstances.

TRN_COMP
Compile transformation

Description:

Compile a transformation and return an identifier for the resulting compiled mapping.

Invocation:

```
CALL TRN_COMP( LOCTR, FORWD, ID, STATUS )
```

Arguments:

LOCTR = C_{LOC}
Locator to the transformation.

FORWD = L
Specifies which mapping is to be compiled; .TRUE. for forward, .FALSE. for inverse.

ID = I
Identifier for the compiled mapping.

STATUS = I
Inherited error status.

TRN_GTCL
Get classification

Description:

Obtain an array of *logical* classification values from a transformation. The classification information is fully validated before it is returned.

Invocation:

```
CALL TRN_GTCL( LOCTR, FORWD, CLASS, STATUS )
```

Arguments:

LOCTR = C_{LOC}
Locator to the transformation.

FORWD = L
Specifies which mapping the information is required for; .TRUE. for forward, .FALSE. for inverse.

CLASS = L_{CLS}
Classification array.

STATUS = I
Inherited error status.

TRN_GTCLC
Get compiled classification

Description:

Obtain an array of *logical* classification values from a compiled mapping.

Invocation:

```
CALL TRN_GTCLC( ID, CLASS, STATUS )
```

Arguments:

ID = I
Identifier for the compiled mapping.

CLASS = L_{CLS}
Classification array.

STATUS = I
Inherited error status.

TRN_GTNV
Get numbers of variables

Description:

Obtain the numbers of input and output variables for a transformation.

Invocation:

```
CALL TRN_GTNV( LOCTR, NVIN, NVOUT, STATUS )
```

Arguments:

LOCTR = C_{LOC}

Locator to the transformation.

NVIN = I

Number of input variables.

NVOUT = I

Number of output variables.

STATUS = I

Inherited error status.

TRN_GTNVC
Get numbers of compiled variables

Description:

Obtain the numbers of input and output variables for a compiled mapping.

Invocation:

```
CALL TRN_GTNVC( ID, NVIN, NVOUT, STATUS )
```

Arguments:

ID = I

Identifier for the compiled mapping.

NVIN = I

Number of input variables.

NVOUT = I

Number of output variables.

STATUS = I

Inherited error status.

TRN_INV
Invert transformation

Description:

Invert a transformation. This interchanges the definitions of the forward and inverse mappings.

Invocation:

```
CALL TRN_INV( LOCTR, STATUS )
```

Arguments:

LOCTR = C_{LOC}

Locator to the transformation to be inverted.

STATUS = I

Inherited error status.

TRN_JOIN
Concatenate transformations

Description:

Concatenate two transformations to produce a combined transformation which is stored as a new component in an existing HDS structure. The combined transformation may also be assigned to a temporary HDS object if required.

Invocation:

```
CALL TRN_JOIN( LOCTR1, LOCTR2, ELOC, NAME, LOCTR, STATUS )
```

Arguments:

LOCTR1 = C_{LOC}
Locator to the first transformation.

LOCTR2 = C_{LOC}
Locator to the second transformation.

ELOC = C_{LOC}
Locator to an enclosing structure to contain the new object.

NAME = C_{NAM}
HDS name of the new structure component to be created.

LOCTR = C_{LOC}
Locator to the newly created transformation.

STATUS = I
Inherited error status.

Notes:

- If the enclosing structure locator *ELOC* is supplied as a blank string, then a temporary transformation will be created. The *NAME* argument is then not used and may also be blank.
- Before two transformations may be concatenated, the following conditions must apply:
  - The number of output variables from the first transformation must match the number of input variables to the second transformation.
  - The transformations must have at least one mapping (forward or inverse) in common.

TRN_NEW
Create new transformation

Description:

Create a new transformation and return a locator to it. The transformation is stored as a new component in an existing HDS structure, or may be assigned to a temporary object if required.

Invocation:

```
CALL TRN_NEW( NVIN, NVOUT, FOR, INV, PREC, COMM, ELOC,
NAME, LOCTR, STATUS )
```

Arguments:

NVIN = I
Number of input variables.

NVOUT = I
Number of output variables.

FOR = C(NVOUT)
Array of forward transformation functions.

INV = C(NVIN)
Array of inverse transformation functions.

PREC = C_{PRC}
Precision specification.

COMM = C
Comment string.

ELOC = C_{LOC}
Locator to an enclosing structure to contain the new object.

NAME = C_{NAM}
HDS name of the new structure component.

LOCTR = C_{LOC}
Locator to the newly created transformation.

STATUS = I
Inherited error status.

Notes:

- The transformation functions will be fully validated before use.
- If the enclosing structure locator *ELOC* is supplied as a blank string, then a temporary transformation will be created. The *NAME* argument is then not used and may also be blank.

TRN_PRFX
Prefix transformation

Description:

Concatenate two transformations; the second transformation is altered by prefixing the first one to it. The first transformation is not altered.

Invocation:

```
CALL TRN_PRFX( LOCTR1, LOCTR2, STATUS )
```

Arguments:

LOCTR1 = C_{LOC}
Locator to the first transformation (to be prefixed).

LOCTR2 = C_{LOC}
Locator to the second transformation (to be altered).

STATUS = I
Inherited error status.

Notes:

Before two transformations may be concatenated, the following conditions must apply:

- The number of output variables from the first transformation must match the number of input variables to the second transformation.
- The transformations must have at least one mapping (forward or inverse) in common.

TRN_PTCL
Put classification

Description:

Enter classification information into a transformation. The information is supplied as an array of *logical* values and is validated before use. If the transformation already contains such information, it is over-written by this routine.

Invocation:

```
CALL TRN_PTCL( CLASS, LOCTR, STATUS )
```

Arguments:

CLASS = L$_{CLS}$

Array containing the classification information.

LOCTR = C$_{LOC}$

Locator to the transformation.

STATUS = I

Inherited error status.

TRN_STOK[x]
Substitute token

Description:

Substitute a value for a token in a text string. The *VALUE* supplied is formatted as a character string (if necessary) and used to replace all valid occurrences of the *TOKEN* sub-string in the *TEXT* supplied. The substitution is not recursive.

Invocation:

```
CALL TRN_STOK[x]( TOKEN, VALUE, TEXT, NSUBS, STATUS )
```

Arguments:

TOKEN = C

The token string.

VALUE = ?

The value to substitute.

TEXT = C

The text to be processed.

NSUBS = I

Number of substitutions made.

STATUS = I

Inherited error status.

Notes:

- There is a routine for each standard type x. Replace x by **I** (*integer*), **R** (*real*) or **D** (*double precision*). Omit x altogether if a character string is being supplied for the *VALUE* argument.
- When *VALUE* is a numerical quantity, the token replacement will be enclosed in parentheses if it is negative.
- To be replaced, a token sub-string within *TEXT* must be correctly delimited (*i.e.* surrounded by non-alphanumeric characters). It may also be enclosed in angle brackets (*e.g.* <a_token>), in which case the brackets will be regarded as part of the token and will also be replaced.
- To be valid, a token must begin with an alphabetic character and contain only alphanumeric characters (including underscore). It may be of any length. No embedded blanks are allowed.

TRN_TR1x
Transform 1-dimensional data

Description:

Apply a compiled $\{1 \rightarrow 1\}$ mapping to a set of 1-dimensional data points specified by an array of X_{IN} coordinates.

Invocation:

```
CALL TRN_TR1x( BAD, NX, XIN, ID, XOUT, STATUS )
```

Arguments:

BAD = L

Whether the input coordinates may be *bad*.

NX = I

The number of data points to transform.

XIN = ?(NX)

Array of input X_{IN} coordinates.

ID = I

Identifier for the compiled $\{1 \rightarrow 1\}$ mapping to be applied.

XOUT = ?(NX)

Array to receive the transformed X_{OUT} coordinates.

STATUS = I

Inherited error status.

Notes:

- There is a routine for each standard numerical data type x. Replace x by **I** (*integer*), **R** (*real*) or **D** (*double precision*).
- The number of input and output variables for the compiled mapping must both be equal to 1.

TRN_TR2x
Transform 2-dimensional data

Description:

Apply a compiled {2 → 2} mapping to a set of 2-dimensional data points specified by separate arrays of X_{IN} and Y_{IN} coordinates.

Invocation:

```
CALL TRN_TR2x( BAD, NXY, XIN, YIN, ID, XOUT, YOUT, STATUS )
```

Arguments:

BAD = L

Whether the input coordinates may be *bad*.

NXY = I

The number of data points to transform.

XIN = ?(NXY)

Array of input X_{IN} coordinates.

YIN = ?(NXY)

Array of input Y_{IN} coordinates.

ID = I

Identifier for the compiled {2 → 2} mapping to be applied.

XOUT = ?(NXY)

Array to receive the transformed X_{OUT} coordinates.

YOUT = ?(NXY)

Array to receive the transformed Y_{OUT} coordinates.

STATUS = I

Inherited error status.

Notes:

- There is a routine for each standard numerical data type x. Replace x by **I** (*integer*), **R** (*real*) or **D** (*double precision*).
- The number of input and output variables for the compiled mapping must both be equal to 2.

TRN_TRNx
Transform general data

Description:

Apply a general compiled mapping (with an arbitrary number of input/output variables) to a set of data points.

Invocation:

```
CALL TRN_TRNx( BAD, ND1, NCIN, NDAT, DATA, ID, NR1, NCOUT,
RESULT, STATUS )
```

Arguments:

BAD = L

Whether the input coordinates may be *bad*.

ND1 = I

First dimension of the *DATA* array (as declared in the calling routine).

NCIN = I

Number of coordinates for each input data point.

NDAT = I

Number of data points to transform.

DATA = ?(ND1,NCIN)

Array of coordinates for the input data points; they need not fill the entire array.

ID = I

Identifier for the compiled $\{NCIN \rightarrow NCOUT\}$ mapping to be applied.

NR1 = I

First dimension of the *RESULT* array (as declared in the calling routine).

NCOUT = I

Number of coordinates for each output data point.

RESULT = ?(NR1,NCOUT)

Array to receive the (transformed) coordinates of the output data points.

STATUS = I

Inherited error status.

Notes:

- There is a routine for each standard numerical data type x. Replace x by **I** (*integer*), **R** (*real*) or **D** (*double precision*).
- *ND1* and *NR1* must both be at least equal to the number of data points *NDAT*.
- *DATA(I,J)* should contain the value of the *J'th* coordinate for the *I'th* input data point. Coordinate values are returned in the *RESULT* array in the same order.
- The values of *NCIN* and *NCOUT* must match the numbers of input and output variables for the compiled mapping being applied.

E Error Handling

E.1 The *STATUS* Argument and Error Reporting

All TRANSFORM routines carry a final *integer STATUS* argument and adhere to the ADAM inherited error handling strategy. Unless otherwise indicated, a routine which finds that *STATUS* is not set to SAI__OK[11] on entry will assume an error has occurred in a previous routine and will return immediately without action and without accessing other arguments (which may not be defined under error conditions). The value of the *STATUS* argument will not be changed. This behaviour usually allows tests of the *STATUS* value to be deferred until after several routine calls have been made.

If a TRANSFORM routine is called with *STATUS* set to SAI__OK, then it will attempt to execute. If it subsequently encounters an error, it will first perform any "cleaning up" which is possible and will then exit with its *STATUS* argument set to one of the error codes described in the next Section. When a *STATUS* value is set by a TRANSFORM routine, an associated error report will always be made by calling the ERR routine ERR_REP. The report will normally contain additional information about the circumstances of the error. Transmission of the report to the user will be deferred by the ERR facility pending action by the caller of the TRANSFORM routine (such as a call to ERR_FLUSH to output the error, or ERR_ANNUL to ignore it). The documentation for the ERR facility should be consulted for further details.

Exceptions to the rule. The routines TRN_ANNUL and TRN_CLOSE are exceptions to the above rules. Both of these perform "cleaning up" operations and will therefore attempt to execute regardless of the *STATUS* value set on entry. If either of these routines fail, they will only set a new *STATUS* value and make an error report if the value of *STATUS* was SAI__OK on entry. If *STATUS* was not set to SAI__OK when they were called, then these two routines will assume that a previous error has occurred and will fail "silently" so that the initial *STATUS* value and error report are preserved.

E.2 Error Codes

The value returned via the *STATUS* argument of TRANSFORM routines under error conditions may be:

- Any of the error codes returned by HDS routines.
- Any of the error codes described below.

If it is necessary to test for specific error conditions, then symbolic names (defined by Fortran PARAMETER statements) should be used to identify the associated error codes. Symbolic names for the TRANSFORM error codes are defined in the include file TRN_ERR and may be incorporated into an application with the statement:

```
INCLUDE 'TRN_ERR'
```

The following list gives the names of these error codes, the associated error messages and an explanation of each error condition:

TRN__CLSIN, classification information invalid
The classification information associated with a transformation is invalid. This may be because conflicting classification properties are being specified (*e.g.* POSITIVE_DET and NEGATIVE_DET) or because a property is being specified in circumstances where it cannot be adequately defined (because the numbers of input and output variables are unequal, for instance).

[11] SAI__OK is a symbolic constant defined in the include file SAE_PAR.

TRN__CMPER, compilation error
An error has been detected during compilation of the right hand side of a transformation function. The associated error report will give further diagnostic information.

TRN__CMTOF, compiled mapping table overflow
The maximum number of simultaneously active compiled mappings has been exceeded. This error should not be encountered because the permitted number of compiled mappings is very large.

TRN__CONIN, constant syntax invalid
The right hand side of a transformation function contains a numerical constant whose syntax is invalid.

TRN__DELIN, delimiting comma invalid
The right hand side of a transformation function contains an invalid comma (commas are only used to separate the arguments of built-in functions).

TRN__DIMIN, dimensions invalid
An HDS object has inappropriate dimensions for its purpose.

TRN__DSTIN, definition status invalid
The FORWARD or INVERSE component of an HDS transformation structure contains an invalid value. Only the values 'DEFINED' and 'UNDEFINED' (case insensitive) are allowed.

TRN__DUVAR, duplicate variable name
A variable name is defined (*i.e.* appears on the left hand side of a transformation function) more than once.

TRN__EXPUD, expression undefined
The expression on the right hand side of a transformation function is missing.

TRN__ICDIR, incompatible transformation directions
An attempt to concatenate two transformations has failed because only the forward mapping was defined within one of them and only the inverse mapping was defined within the other.

TRN__MAPUD, mapping undefined
The mapping requested from a transformation (either forward or inverse) has not been defined.

TRN__MIDIN, compiled mapping identifier invalid
A compiled mapping identifier is not valid (*i.e.* it is not currently associated with a compiled mapping).

TRN__MIOPA, missing or invalid operand
During compilation of the right hand side of a transformation function, an operand (*i.e.* a constant, variable or expression) was expected but was not found.

TRN__MIOPR, missing or invalid operator
During compilation of the right hand side of a transformation function, an operator (or delimiter) was expected but was not found.

TRN__MISVN, missing variable name
The variable name is missing from the left hand side of a transformation function.

TRN__MLPAR, missing left parenthesis
The right hand side of a transformation function has a left parenthesis missing.

TRN__MRPAR, missing right parenthesis
The right hand side of a transformation function has a right parenthesis missing.

TRN__NDCMM, number of data coordinates mis-matched
The number of input or output coordinates specified for a set of data points does not match the corresponding number of variables associated with the compiled mapping being used to transform the points.

TRN__NMVMM, number of module variables mis-matched
Two adjacent transformation modules in the MODULE_ARRAY component of an HDS transformation structure are mis-matched because the number of output variables from one module is not equal to the number of input variables for the one which follows.

TRN__NTVMM, number of transformation variables mis-matched
The number of output variables from the first of two transformations being concatenated is not equal to the number of input variables to the second transformation.

TRN__NVRIN, number of variables invalid
The number of variables specified in the NVAR_IN or NVAR_OUT component of an HDS transformation module (TRN_MODULE) structure is invalid (*i.e.* it is not positive).

TRN__OPCIN, operation code invalid
An invalid internal arithmetic operation code has been encountered while evaluating a transformation function which is part of a compiled mapping. This is a serious error indicating internal inconsistency within the TRANSFORM software. It should be reported immediately.

TRN__PRCIN, precision invalid
An invalid precision specification has been supplied when creating a new transformation.

TRN__TOKIN, token name invalid
An attempt to substitute a value for a token in a character string has failed because an invalid token name has been specified.

TRN__TRNUD, transformation undefined
A set of transformation functions is incomplete or is otherwise insufficient to fully define a transformation.

TRN__TRUNC, character string truncated
A character string has been truncated because the CHARACTER variable supplied is not of sufficient length to accommodate it.

TRN__TYPIN, type invalid
An HDS object has a type inappropriate for its purpose.

TRN__VARIN, variable name invalid
A variable used in a transformation function has an invalid name.

TRN__VARUD, variable name undefined
A variable used on the right hand side of a transformation function is undefined because it does not appear on the left hand side of one of the transformation functions which define the complementary mapping.

TRN__VERMM, software version mis-match
The version number of the TRANSFORM software used to create or last modify an HDS transformation structure exceeds the version number of the software with which the current application is linked.

TRN__WRNFA, wrong number of function arguments
An invalid number of arguments has been supplied for a built-in function invoked from the right hand side of a transformation function.