SUN/39.2

Starlink Project
Starlink User Note 39.2

R F Warren-Smith

28 February 1995

PRIMDAT — Processing of Primitive Numerical Data v1.6 Programmer's Guide

---

Abstract

The PRIMDAT package is a collection of Fortran functions and subroutines providing support for primitive data processing.

Contents

1 INTRODUCTION

This document describes version 1.0 of the PRIMDAT package, which is a collection of Fortran functions and subroutines providing support for "primitive data processing". Routines from this package may be used to perform arithmetic, mathematical operations, type conversion and inter-comparison of any of the primitive numerical data types supported by the Starlink Hierarchical Data System HDS (SUN/92).

It provides:

- **Processing facilities which are not normally available.**
  This package provides processing facilities for **all** the numerical HDS data types. In contrast, standard Fortran 77 only supports the _INTEGER, _REAL and _DOUBLE numerical types and even in VAX Fortran there are no operators for the unsigned types _UWORD and _UBYTE (those for _BYTE are also somewhat restricted).

- **A uniform interface.**
  A consistent naming scheme is used which facilitates the production of *generic* routines for processing data of any numerical type (SUN/7 describes the GENERIC compiler and contains further information on this subject).

- **Improved portability and efficiency.**
  By using this package, applications can process non-standard data types while making minimum explicit use of non-standard Fortran statements. In addition, PRIMDAT routines can exploit machine-specific features to enhance the efficiency of primitive data processing, so applications may take advantage of these features without compromising their own portability.

- **Facilities for processing *bad* data.**
  The *bad values* defined by Starlink for flagging undefined data may be processed and consistently propagated. Recognition of these *bad values* may easily be disabled when not required in order to save processing time.

- **Handling of numerical errors.**
  Numerical errors (such as division by zero or numerical overflow) may be handled automatically and converted into appropriate *bad values* for subsequent recognition. Error codes are returned to allow such occurrences to be reported, or recovery action taken, as desired.

- **A set of constants.**
  A set of machine-specific symbolic constants is provided to support the processing facilities in this package and to simplify the task of transporting software to other machines which may have different arithmetic capabilities.

Note that only the **numerical** HDS data types (_BYTE, _UBYTE, _WORD, _UWORD, _INTEGER, _REAL & _DOUBLE) are supported at present. There is currently no provision for the non-numerical _LOGICAL and _CHAR types, nor for any other Fortran (or VAX Fortran) types such as COMPLEX. The CHR facility should be used for processing character data and for converting between character and numerical data types.

Note added March 2004: This manual refers throughout to VAX floats. The package was ported to Unix in 1991, and accordingly most of the values now defined by it are those appropriate for IEEE floats instead; the manual has not been updated to match.

1.1 The VAL, VEC and NUM Facilities

The routines in this package have names of the form <fac>_<name>, where <fac> is a three character facility name and <name> specifies the operation which the routine performs.

The routines are divided into three facilities as follows:

<fac>	**Facility provides...**
VAL	Arithmetic, mathematical functions and type conversion on single (scalar) *values*. Handling of numerical errors and *bad value* propagation are incorporated.
VEC	Arithmetic, mathematical functions and type conversion on *vectorised arrays*, allowing more efficient processing of large numbers of data. Handling of numerical errors and *bad value* propagation are incorporated.
NUM	Lower level routines for arithmetic, mathematical functions, type conversion and inter-comparison of single *numbers*. Operations are performed without protection against numerical errors and without regard for *bad* values.

Note that a distinction is drawn between three classes of primitive data, which differ in their interpretation and in the algorithms best suited to processing them:

- **Values** (VAL facility) are scalar data which may take one of the Starlink-defined *bad values* (sometimes called "magic" values), whose presence signifies that the affected datum is undefined. Numerical errors which occur while processing *values* will result in the generation of a *bad value* as a result. Subsequent recognition of *bad values* on input may (or may not) occur, depending on the logical setting of the *BAD* routine argument (section 1.2).
- **Vectorised arrays** (VEC facility) are 1-dimensional arrays of *values* (or arrays treated as 1-dimensional) whose elements are processed in the way described above. The routines provided are optimised to perform processing in an efficient manner when the size of the array is large.
- **Numbers** (NUM facility) are always interpreted literally as scalar numerical quantities (*i.e. bad* values are not recognised on input and are not explicitly generated on output). Numerical errors which occur while processing *numbers* will cause the application to terminate (*i.e.* crash).

Italics are used throughout this document to refer to *values*, *vectorised arrays* and *numbers* when it is important to emphasise the particular meaning (and the implied processing rules) described here.

Type Code <T>	HDS Type		Fortran Type
UB	_UBYTE	(unsigned byte)	BYTE
B	_BYTE	(byte)	BYTE
UW	_UWORD	(unsigned word)	INTEGER∗2
W	_WORD	(word)	INTEGER∗2
I	_INTEGER	(integer)	INTEGER
R	_REAL	(real)	REAL
D	_DOUBLE	(double precision)	DOUBLE PRECISION

Table 1: The data type codes used in constants and routine names, and the HDS and VAX Fortran types to which they correspond.

1.2 Propagation of *Bad* Data

The handling of numerical errors by the VAL_ and VEC_ routines[1] depends on the generation, recognition and propagation of *bad values* in the data, and also on the presence of an integer *STATUS* argument carried by each routine.

Bad values are a set of special numerical constants (one for each data type) which are defined by Starlink to be used for flagging "bad" or "undefined" data. The philosophy and conventions surrounding their use are described in SGP/38. The Starlink "*bad* constants" which specify these values have symbolic names[2] of the form:

```
VAL__BAD<T>
```

where <T> is one of the data type codes in Table 1 (also see section 5.1). *Bad values* may be set explicitly by assigning the value of one of these constants to a variable. They are also set by VAL_ and VEC_ routines when numerical errors occur — implying that the result of the attempted operation is undefined.

Recognition of *bad values* involves comparing them for equality with the appropriate "*bad* constant", and may be used to control subsequent processing. Most frequently, this control takes the form of "propagation" — *i.e.* the outcome of any operation attempted on a *bad* operand is itself *bad*; so that the *bad value* is "propagated" to the result. After a sequence of such operations, the resulting *bad values* make it possible to identify all those results which have been rendered invalid by errors occurring *en route*.

The VAL_ and VEC_ routines have the ability to recognise and propagate *bad values* in this manner (the precise processing rules are described in sections 2.2 & 3.2). In some circumstances, however, appreciable amounts of processing time can be saved if it is known in advance that *bad values* are not present and that their recognition is not necessary. Also, it is occasionally

[1] The NUM_ functions do not have a built-in error handling capability and are not discussed here.
[2] These symbolic constants are defined in the include file PRM_PAR.

necessary to disable *bad value* recognition so that the numerical values specified by the "*bad* constants" may themselves be processed as if they were valid data.

Because of this, recognition and propagation of *bad values* by the VAL_ and VEC_ routines is controlled by a logical value *BAD*, which appears at the start of each routine's argument list. If *BAD* is set .TRUE., then *bad* input arguments are recognised and propagated to the result. Conversely, if *BAD* is set .FALSE., then input argument values are interpreted literally (there is no *bad value* recognition), although *bad values* may still be generated on output if numerical errors occur. Most of the routines execute more efficiently if *BAD* is .FALSE. (Appendix A).

It is up to the calling routine to ensure that the *BAD* argument has an appropriate logical value when VAL_ and VEC_ routines are invoked. This value will often depend on whether numerical errors have occurred in previous stages of processing, so each routine also carries a *STATUS* argument which may be used to detect the occurrence of such errors.

1.3 The *STATUS* Argument

Each VAL_ and VEC_ routine takes a standard integer *STATUS* variable as its final argument. In normal use, this *STATUS* variable should be set to SAI__OK when the routine is invoked, otherwise it will return immediately without action.[3]

On exit, the *STATUS* value will still be set to SAI__OK unless a numerical error has occurred, in which case it will be set to an appropriate error code (Appendix B). PRIMDAT error codes may be tested to detect when a particular numerical error has occurred, and may also be used to report these errors (using the ADAM ERR_ routines for instance).

Note that routines in this package do not report errors themselves. This is partly an efficiency consideration. In many circumstances, the automatic generation of a *bad value* will itself be an adequate response to a numerical error and, in such cases, the *STATUS* variable may simply be reset to SAI__OK and processing can continue.

2 THE VAL_ FUNCTIONS

The VAL_ functions are a set of Fortran functions which perform arithmetic, mathematical operations and type conversion on single (scalar) *values* stored using any numerical data type. These functions will handle numerical errors which occur during their evaluation (such as division by zero or overflow) and can also recognise and propagate the standard Starlink *bad values*.

They are normally invoked by statements of the form:

```
RESULT = VAL_<name>( BAD, ARG, STATUS )
```

or

```
RESULT = VAL_<name>( BAD, ARG1, ARG2, STATUS )
```

[3] In the case of the VAL_ functions, a *bad* function result will be returned.

where <name> specifies the operation to be performed and:

- *BAD* is a logical value specifying whether *bad* input arguments are to be recognised;
- *ARG*, *ARG1* and *ARG2* are the input arguments;
- *STATUS* is an integer error status variable.

2.1 Applicability & Efficiency

The VAL_ functions are designed primarily for ease of use and robustness, with efficiency being a secondary consideration. In fact, the overhead involved in invoking many of these functions exceeds the time required to actually perform the calculation. Consequently, VAL_ functions should **not** be used for processing large arrays of data. Rather, they are best used for relatively small numbers of calculations (up to a few thousand), when they can enhance the robustness and flexibility of applications by handling numerical errors and *bad values* with the minimum of programming effort.

The VEC_ routines (section 3) should normally be used for processing larger arrays of data.

2.2 VAL_ Error Handling

The error handling strategy used by the VAL_ routines consists of returning a *bad* function result under the following circumstances:

(1) If *STATUS* is not set to SAI__OK when the function is invoked.

(2) If the *BAD* argument is set to .TRUE. and any of the input arguments is *bad*.

(3) If any numerical error occurs during evaluation of the function — in this latter case an appropriate error code will also be returned via the *STATUS* argument.

2.3 VAL_ Arithmetic and Mathematical Functions

Arithmetic and mathematical operations may be performed on scalar *values* stored using any numerical data type by invoking appropriate VAL_ functions, using statements such as:

```
RESULT = VAL_<FUNC><T>( BAD, ARG, STATUS )
```

or

```
RESULT = VAL_<FUNC><T>( BAD, ARG1, ARG2, STATUS )
```

Here, the VAL_ function takes an argument (or arguments) of type <T> and returns a result which is also of type <T>, having performed the arithmetic or mathematical operation specified by <FUNC>. The data type code <T> may be any of those specified in Table 1 and <FUNC> may be any of the function codes specified in Tables 2 & 3. The number of input arguments (1 or 2) appropriate to each function is also indicated in these latter Tables.

Thus, for example, the function VAL_ADDUW adds two unsigned word arguments to produce an unsigned word result, while VAL_SQRTD evaluates the square root of a double precision argument, returning a double precision result.

Function Code <**FUNC**>	**Number of Arguments**	**Operation performed**
ADD	2	addition: ARG1 + ARG2
SUB	2	subtraction: ARG1 − ARG2
MUL	2	multiplication: ARG1 ∗ ARG2
DIV	2	*(floating) division: ARG1 / ARG2
IDV	2	**(integer) division: ARG1 / ARG2
PWR	2	raise to power: ARG1 ∗∗ ARG2
NEG	1	negate (change sign): −ARG
SQRT	1	square root: $\sqrt{\text{ARG}}$
LOG	1	natural logarithm: $\ln(\text{ARG})$
LG10	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
INT	1	Fortran AINT (truncation to integer) function
MAX	2	maximum: $\max(\text{ARG1}, \text{ARG2})$
MIN	2	minimum: $\min(\text{ARG1}, \text{ARG2})$
DIM	2	Fortran DIM (positive difference) function
MOD	2	Fortran MOD (remainder) function
SIGN	2	Fortran SIGN (transfer of sign) function
Notes:	*Equivalent to NINT(REAL(ARG1)/REAL(ARG2)) for non-floating quantities **Equivalent to AINT(ARG1/ARG2) for floating quantities	

Table 2: Function codes used in routine names and the operations to which they correspond. The functions shown here are implemented for **all** the numerical data types (type codes UB, B, UW, W, I, R & D).

2.4 VAL_ Type Conversion Functions

Conversion of scalar *values* between numerical data types may be performed by invoking the appropriate VAL_ type conversion functions, using statements such as:

```
RESULT = VAL_<T1>TO<T2>( BAD, ARG, STATUS )
```

Here, the VAL_ function takes an argument of type $<T_1>$ and returns a result of type $<T_2>$, having performed type conversion. The data type codes $<T_1>$ and $<T_2>$ may be any of those specified in Table 1.

Thus, for example, the function VAL_BTOUW converts a byte argument to an unsigned word result, while VAL_DTOI converts from double precision to integer.

Note that conversions from floating point types (_REAL or _DOUBLE) to non-floating point types result in rounding of the data (not truncation as would happen with a Fortran assignment statement).

2.5 Declaring the VAL_ Functions

Since the VAL_ functions return a result via the routine name, appropriate declarations of their Fortran type must be made before they are used. Thus, for instance, an application which used the routines VAL_ADDUB and VAL_DTOI would require the declarations:

```
BYTE VAL_ADDUB
INTEGER VAL_DTOI
```

A routine suitable for processing by the GENERIC compiler might require equivalent generic declarations:

```
<TYPE> VAL_ADD<T>
<TYPE> VAL_DTO<T>
```

3 THE VEC_ ROUTINES

The VEC_ routines are a set of subroutines which perform arithmetic, mathematical operations and type conversion on *vectorised arrays* of data stored using any numerical type. They incorporate handling of numerical errors and propagation of the standard *bad values* in a way that is fully compatible with the VAL_ functions.

VEC_ routines are invoked by statements of the form:

```
CALL VEC_<name>( BAD, N, ARG, RESULT, IERR, NERR, STATUS )
```

or

Function Code <**FUNC**>	**Number of Arguments**	**Operation performed**
SIN	1	*sine function: $\sin(\mathrm{ARG})$
SIND	1	**sine function: $\sin(\mathrm{ARG})$
COS	1	*cosine function: $\cos(\mathrm{ARG})$
COSD	1	**cosine function: $\cos(\mathrm{ARG})$
TAN	1	*tangent function: $\tan(\mathrm{ARG})$
TAND	1	**tangent function: $\tan(\mathrm{ARG})$
ASIN	1	*inverse sine function: $\sin^{-1}(\mathrm{ARG})$
ASND	1	**inverse sine function: $\sin^{-1}(\mathrm{ARG})$
ACOS	1	*inverse cosine function: $\cos^{-1}(\mathrm{ARG})$
ACSD	1	**inverse cosine function: $\cos^{-1}(\mathrm{ARG})$
ATAN	1	*inverse tangent function: $\tan^{-1}(\mathrm{ARG})$
ATND	1	*inverse tangent function: $\tan^{-1}(\mathrm{ARG})$
ATN2	2	*Fortran ATAN2 (inverse tangent) function
AT2D	2	**VAX Fortran ATAN2D (inverse tangent) function
SINH	1	hyperbolic sine function: $\sinh(\mathrm{ARG})$
COSH	1	hyperbolic cosine function: $\cosh(\mathrm{ARG})$
TANH	1	hyperbolic tangent function: $\tanh(\mathrm{ARG})$
Notes:	*Argument(s)/result in radians **Argument(s)/result in degrees	

Table 3: Function codes used in routine names and the operations to which they correspond. The functions shown here are only implemented for the floating point data types (type codes R & D).

```
CALL VEC_<name>( BAD, N, ARG1, ARG2, RESULT, IERR, NERR, STATUS )
```

where <name> specifies the operation to be performed and:

- *BAD* is a logical value specifying whether *bad* input arguments are to be recognised;
- *N* is an integer value specifying the number of array elements to process;
- *ARG*, *ARG1* and *ARG2* are the input arguments (each is an *N*-element *vectorised array*);
- *RESULT* is an *N*-element *vectorised* output array;
- *IERR* is an integer output argument which identifies the first array element to generate a numerical error;
- *NERR* is an integer output argument which returns a count of the number of numerical errors which occur;
- *STATUS* is an integer error status variable.

3.1 Applicability & Efficiency

The VEC_ routines are designed for optimum efficiency when processing large arrays of data and are consequently **far** more efficient at performing this task than the corresponding VAL_ functions. When possible, VEC_ routines should always be used in preference to VAL_ functions unless the number of calculations to be performed is small (typically 1000 or less).

The main disadvantage in using VEC_ routines is likely to be the additional programming effort, because workspace arrays will probably be required and may have to be acquired dynamically — unless only a single operation is being performed. Also, certain operations cannot be performed using VEC_ routines (summing the pixels in an image for example) and, in such cases, routines from the NUM facility (section 4) may have to be used instead. Note, however, that if NUM_ routines are used, then security against numerical errors is more difficult to achieve and (especially in the case of arithmetic operations) VAX-specific programming techniques may be required. The resulting applications will not then be portable.

3.2 VEC_ Error Handling

The error handling strategy used by the VEC_ routines is as follows:

(1) If the *STATUS* argument is not set to SAI__OK on entry, then the routine returns immediately without action — the contents of the *RESULT* array remain undefined in this case.

(2) If the *BAD* argument is set to .TRUE. and any element of an input argument array is *bad*, then a *bad value* is placed in the corresponding element of the *RESULT* array.

(3) A *bad value* is also placed in the corresponding element of the *RESULT* array if a numerical error occurs when processing any element of an input argument array — in this instance an appropriate error code will also be returned via the *STATUS* argument.

On exit, the *IERR* argument will contain the index of the first array element which generated a numerical error (or zero if there were no errors) and *NERR* will contain a count of the number of numerical errors which occurred. If more than one error occurred, then the *STATUS* value returned will be appropriate to the **first** of these.

3.3 VEC_ Arithmetic and Mathematical Routines

Arithmetic and mathematical operations may be performed on *vectorised arrays* of data of any numerical type by invoking appropriate VEC_ routines, using statements such as:

```
CALL VEC_<FUNC><T> ( BAD, N, ARG, RESULT, IERR, NERR, STATUS )
```

or

```
CALL VEC_<FUNC><T> ( BAD, N, ARG1, ARG2, RESULT, IERR, NERR, STATUS )
```

Here, the VEC_ routine takes a *vectorised* input argument array (or arrays) with *N* elements of type <T> and returns a *vectorised* result array which is also of type <T>, having performed the arithmetic or mathematical operation specified by <FUNC> on each array element. The data type code <T> may be any of those specified in Table 1 and <FUNC> may be any of the function codes specified in Tables 2 & 3. The number of input argument arrays (1 or 2) appropriate to each function is also indicated in these latter Tables.

Thus, for example, the subroutine VEC_ADDUW adds two *vectorised* unsigned word arrays to produce an array of unsigned word results, while VEC_SQRTD evaluates the square root of a single *vectorised array* of double precision *values*, returning an array of double precision results.

3.4 VEC_ Type Conversion Routines

Conversion of *vectorised arrays* between numerical data types may be performed by invoking the appropriate VEC_ type conversion routines, using statements such as:

```
CALL VEC_<T1>TO<T2> ( BAD, N, ARG, RESULT, IERR, NERR, STATUS )
```

Here, the VEC_ routine takes a *vectorised* input argument array with *N* elements of type $<T_1>$ and returns a *vectorised* result array of type $<T_2>$, having performed type conversion on each array element. The data type codes $<T_1>$ and $<T_2>$ may be any of those specified in Table 1.

Thus, for example, the subroutine VEC_BTOUW converts a *vectorised array* of byte data into unsigned word data, while VEC_DTOI converts a *vectorised array* from double precision to integer.

Note that conversions from floating point types (_REAL or _DOUBLE) to non-floating point types result in rounding of the data (not truncation as would happen with a Fortran assignment statement).

4 THE NUM_ FUNCTIONS

The NUM_ functions perform arithmetic, mathematical operations, type conversion and inter-comparison on *numbers*, operating at a rather lower level than the VAL_ or VEC_ routines — in fact many of these latter routines are currently implemented using the NUM facility.

At present, all the NUM_ routines are implemented as Fortran statement functions whose definitions must be incorporated into an application with appropriate INCLUDE statements (section 4.5).

They are normally invoked by statements of the form:

```
RESULT = NUM_<name>( ARG )
```

or

```
RESULT = NUM_<name>( ARG1, ARG2 )
```

where <name> specifies the operation to be performed and *ARG*, *ARG1* and *ARG2* are input arguments.

4.1 Applicability & Efficiency

The NUM_ functions are the most flexible routines in this package. Being implemented as statement functions, they will normally be expanded "in-line" by the VAX Fortran compiler and may therefore be used in any situation without incurring the overhead of an external function call. The expanded function definition can subsequently take part in the compiler's optimisation process and will therefore result in code which executes very efficiently (although some of the machine-specific features used by VEC_ routines cannot be accessed in this way).

However, the NUM_ functions are also the least robust routines in this package. This is because they incorporate no protection against numerical errors (such as division by zero or overflow), neither do they generate or recognise *bad* data — which they will always treat as valid. If a numerical error occurs when a NUM_ function is invoked, the application will immediately terminate (*i.e.* crash).

The robustness of NUM_ functions may be improved by incorporating explicit checks on the *numbers* being processed and by ensuring that the functions are not invoked if their arguments may be undefined (under error conditions, for instance). Such checks will always degrade the execution efficiency, however, and the equivalent VEC_ routine (section 3) is then likely to be at least as efficient. Consequently, use of the NUM_ functions is generally appropriate only when:

- It is known (from the nature of the data) that numerical errors cannot occur; or
- Efficiency considerations completely outweigh the need for robustness; or
- The problem is such that no alternative routine from this package is applicable.

For applications where VAX-specific code is acceptable, the NUM facility also includes a simple VAX condition handler which can be used to recover from numerical errors (section 4.6). This may be used with the NUM_ functions (and Fortran arithmetic expressions) to combine robustness with efficiency and flexibility in cases where other routines from this package are not suitable. However, the programming effort required is relatively high and the resulting applications will not be portable.

4.2 NUM_ Arithmetic and Mathematical Functions

Arithmetic and mathematical operations may be performed on *numbers* stored using any numerical data type by invoking appropriate NUM_ functions, using statements such as:

```
RESULT = NUM_<FUNC><T>( ARG )
```

or

```
RESULT = NUM_<FUNC><T>( ARG1, ARG2 )
```

Here, the NUM_ function takes an argument (or arguments) of type <T> and returns a result which is also of type <T>, having performed the arithmetic or mathematical operation specified by <FUNC>. The data type code <T> may be any of those specified in Table 1 and <FUNC> may be any of the function codes specified in Tables 2 & 3. The number of function arguments (1 or 2) appropriate to each function is also indicated in these latter Tables.

Thus, for example, the function NUM_ADDUW adds two unsigned word arguments to produce an unsigned word result, while NUM_SQRTD evaluates the square root of a double precision argument, returning a double precision result.

4.3 NUM_ Type Conversion Functions

Conversion of *numbers* between numerical data types may be performed by invoking the appropriate NUM_ type conversion functions, using statements such as:

```
RESULT = NUM_<T1>TO<T2>( ARG )
```

Here, the NUM_ function takes an argument of type $<T_1>$ and returns a result of type $<T_2>$, having performed type conversion. The data type codes $<T_1>$ and $<T_2>$ may be any of those specified in Table 1.

Thus, for example, the function NUM_BTOUW converts a byte argument to an unsigned word result, while NUM_DTOI converts from double precision to integer.

Note that conversions from floating point types (_REAL or _DOUBLE) to non-floating point types result in rounding of the data (not truncation as would happen with a Fortran assignment statement).

Relational Operation Code <ROPER>	**Inter-comparison**
EQ	ARG1 .EQ. ARG2
NE	ARG1 .NE. ARG2
GT	ARG1 .GT. ARG2
GE	ARG1 .GE. ARG2
LT	ARG1 .LT. ARG2
LE	ARG1 .LE. ARG2

Table 4: The relational operation codes used in the names of NUM_ functions and the inter-comparisons to which they correspond.

4.4 NUM_ Inter-comparison Functions

Inter-comparison of *numbers* stored using any numerical data type, to test for equality or inequality relations, may be performed by invoking the appropriate NUM_ inter-comparison functions, using statements such as:

```
RESULT = NUM_<ROPER><T>( ARG1, ARG2 )
```

Here, the NUM_ function takes two arguments of type <T> and returns a logical result, having compared its arguments according to the relational operation specified by <ROPER>. The data type code <T> may be any of those in Table 1 and <ROPER> may be any of the relational operation codes specified in Table 4.

Thus, for example, the logical function NUM_GTUB returns a .TRUE. result if the unsigned byte *number* ARG1 is greater than the unsigned byte *number* ARG2, while NUM_LED returns .TRUE. if the double precision *number* ARG1 is less than or equal to the double precision *number* ARG2.

Note that equivalent inter-comparison functions are not available in the VAL or VEC facilities.

4.5 Declaring and Defining the NUM_ Functions

The NUM facility is implemented as a set of Fortran statement functions. To use them, an application must:

(1) Declare the function names and dummy arguments, specifying their Fortran types;

(2) Define the functions.

Two files are provided which contain Fortran statements to perform these tasks and they may be included in an application with the statements:

```
INCLUDE 'NUM_DEC'
INCLUDE 'NUM_DEF'
```

which should appear (in this order) immediately after any local Fortran variables have been declared and before the executable code begins. These statements will declare and define the entire set of NUM_ functions.

In almost all cases, however, compilation time can be considerably reduced by defining only a sub-set of these functions. Files are therefore provided which allow the function definitions to be selected according to the data type of their arguments. When using this option, the following two statements should first be used to declare and define the type conversion functions and their arguments (since these are used by all the other NUM_ functions they must always be present):

```
INCLUDE 'NUM_DEC_CVT'
INCLUDE 'NUM_DEF_CVT'
```

If the application only uses the NUM_ functions to perform type conversion, then this is all that is required. However, to additionally declare and define the NUM_ functions which process a particular argument data type, the statements:

```
INCLUDE 'NUM_DEC_<T>'
INCLUDE 'NUM_DEF_<T>'
```

should then be used (here, the argument type code <T> is one of those specified in Table 1). Thus, to give a complete example, the NUM_ functions for processing unsigned word data would be declared and defined using the following four statements:

```
INCLUDE 'NUM_DEC_CVT'
INCLUDE 'NUM_DEC_UW'
INCLUDE 'NUM_DEF_CVT'
INCLUDE 'NUM_DEF_UW'
```

Note that to comply with Fortran restrictions on statement order, all the type declaration statements (NUM_DEC_ files) have to appear **before** the function definition statements (NUM_DEF_ files).

4.6 NUM Error Handling

Because the NUM_ functions do not have any built-in error handling capability, the NUM facility provides a simple VAX condition handler which may be used to explicitly implement error handling, both for the NUM_ functions and straightforward Fortran arithmetic expressions.

Note that using this capability involves calls to VAX Run Time Library (RTL) routines and will therefore result in applications which are not portable. Such applications will continue to operate, however, if the RTL calls are simply removed, although error handling will not then occur.

The technique is best explained by an example:

```
      SUBROUTINE EXAMPL( N, DATA )

*   Declare constants and variables.
      INCLUDE 'SAE_PAR'
      INCLUDE 'PRM_PAR'
      INTEGER N, I
      REAL DATA( N )

*   Declare the condition handler.
      INTEGER NUM_TRAP                          [1]
      EXTERNAL NUM_TRAP

*   Include the NUM_CMN common block definition, which defines
*   the variable NUM_ERROR.
      INCLUDE 'NUM_CMN'                         [2]

*   Establish the condition handler.
      CALL LIB$ESTABLISH( NUM_TRAP )            [3]

*   Initialise.
      NUM_ERROR = SAI__OK                       [4]

*   Perform the calculations.
      DO 1 I = 1, N
         DATA( I ) = 1.0 / ( DATA( I ) ** 2 )   [5]

*   Check if an error occurred.
         IF( NUM_ERROR .NE. SAI__OK ) THEN      [6]

*   If so, reset the NUM_ERROR flag and define the result.
            NUM_ERROR = SAI__OK                 [7]
            DATA( I ) = VAL__BADR
         ENDIF
    1 CONTINUE

*   Remove the condition handler and exit the routine.
      CALL LIB$REVERT                           [8]
      END
```

Programming notes:

(1) The integer function NUM_TRAP is provided as a standard condition handler which traps numerical errors. This routine must be declared in an EXTERNAL statement.

(2) The common block NUM_CMN is defined using an INCLUDE statement. It contains a single integer variable called NUM_ERROR which communicates with the condition handler.

(3) The VAX RTL routine LIB$ESTABLISH is called to establish the condition handler. From this point on, numerical errors will not cause the application to crash, but will be intercepted by the NUM_TRAP routine. This behaviour is local to the routine from which LIB$ESTABLISH is called. The behaviour of VAL_ and VEC_ routines and the handling of non-numerical errors is not affected.

(4) The common block variable NUM_ERROR is initialised to SAI__OK. This is important because numerical errors will not be detected otherwise.

(5) Calculations are performed which may potentially fail. These may include calls to NUM_ functions and need not be confined to a single expression or statement. In the above example floating point overflow or division by zero could occur, according to the contents of the *DATA* array.

(6) After a calculation, the status variable NUM_ERROR is tested. If it is not SAI__OK, then a numerical error has occurred. The value of NUM_ERROR may be used as a status code for reporting the error if required.

(7) If an error has occurred, suitable action is taken. This must include resetting the numerical error status NUM_ERROR to SAI__OK and defining the result of any calculations which failed. This will usually be done using a flag for later identification (normally one of the Starlink *bad values*).

(8) Finally, the condition handler is removed by calling the RTL routine LIB$REVERT. This is strictly only necessary if further calculations which do not require error handling are to be performed in the same routine.

It is important to note that when a calculation fails, more than one numerical error may occur. This is because the (erroneous) result of the first operation to fail can be re-used in subsequent parts of an arithmetic expression and may precipitate further errors. To ensure that NUM_ERROR indicates the true (first) cause of an error in such cases, this variable behaves as a latch; once set, it will not again change in response to an error until it has first been reset to SAI__OK.

Note also, that the result of a calculation which fails (or may have failed) must **not** be used **under any circumstances**. It may either contain an erroneous result or an invalid floating point number (a reserved operand) which will cause any application which processes it to crash. Such results **must** immediately be replaced with a well-defined value.

5 CONSTANTS

This section describes a set of constants which support the processing facilities in this package by specifying machine-specific quantities associated with each data type. These constants are identified by symbolic names which are defined (using Fortran PARAMETER statements) in the file PRM_PAR. The definitions may be incorporated into an application with the statement:

```
INCLUDE 'PRM_PAR'
```

Here, as elsewhere in this package, a distinction is drawn between *values* (which may be flagged as *bad*) and *numbers* (which are always interpreted literally). The constants accordingly have names with one of the two forms:

```
VAL__<const><T>
```

Constant	**Type**	**Quantity represented**
VAL__BAD<T>	*<T>	*Bad value*, used for flagging undefined data.
VAL__EPS<T>	<T>	Machine precision.
VAL__MAX<T>	<T>	Maximum (most positive) non-*bad value*.
NUM__MAX<T>	<T>	Maximum (most positive) *number*.
VAL__MIN<T>	<T>	Minimum (most negative) non-*bad value*.
NUM__MIN<T>	<T>	Minimum (most negative) *number*.
VAL__NB<T>	_INTEGER	Number of basic machine units (bytes) used by a value.
VAL__SML<T>	<T>	Smallest positive (non-zero) value.
VAL__SZ<T>	_INTEGER	Number of characters required to format a value as a decimal string.
*Type=<T> indicates that the constant's data type matches the type code <T>.		

Table 5: A summary of the constants available, showing their data types and the quantities they represent.

or

```
NUM__<const><T>
```

where <T> is one of the type codes in Table 1 and <const> specifies the quantity which the constant represents. In most cases, the two forms of the constant are necessarily identical and only the VAL__ form is provided.

The constants are also available in the C include file prm_par.h.

Note that many of these constants may need to change if transported to another machine.[4] When writing software which is intended to be portable you should therefore:

- Use the constants provided here, rather than defining your own;
- Always refer to constants using symbolic names (**never** explicit numerical values);
- Never make implicit assumptions about the values these constants might take.

A summary of the constants available is given in Table 5. The following sections describe them in more detail.

[4] The constants given here are appropriate to VAX machines using F- and D-type floating point number representations. Only a simple description of floating point arithmetic is given. Applications requiring a more complete description should use routines from the NAG library, with which the values given here are compatible.

5.1 Bad Data Values

Constants with names of the form VAL__BAD<T> represent the Starlink-defined *bad* (or "magic") *values* to be used for flagging *bad* or undefined data. The use of these constants is discussed in SGP/38, which also defines the values to be used on VAX machines. These are:

Constant	Type	Value	Hexadecimal Pattern
VAL__BADUB	_UBYTE	255	FF
VAL__BADB	_BYTE	−128	80
VAL__BADUW	_UWORD	65535	FFFF
VAL__BADW	_WORD	−32768	8000
VAL__BADI	_INTEGER	−2147483648	80000000
VAL__BADR	_REAL	−1.7014117E+38	FFFFFFFF
VAL__BADD	_DOUBLE	−1.701411834604923D+38	FFFFFFFFFFFFFFFF

Note that the operations permitted on *bad values* are restricted to:

- Assigning the *value* of a "*bad* constant" to a variable;
- Testing a variable for equality with a "*bad* constant";
- Assigning a non-*bad value* to a variable to replace one which was previously *bad*.

Software which aims to be portable may assume that the *bad values* described by these constants will always lie at the extreme end of the *number* range which can be represented by each data type. However, no assumptions should be made about **which** end of the range (upper or lower) this will be — an explicit test should be made if it is necessary to determine this.

5.2 Machine Precision

Constants with names of the form VAL__EPS<T> represent the machine precision when performing calculations using each data type. The machine precision ϵ is the smallest positive value such that $(1 + \epsilon)$ can be represented on the machine and is distinguishable from 1 using the data type in question. The data types of these constants match the quantities they describe, as follows:

Constant	Type	Value	Hexadecimal Pattern
VAL__EPSUB	_UBYTE	1	01
VAL__EPSB	_BYTE	1	01
VAL__EPSUW	_UWORD	1	0001
VAL__EPSW	_WORD	1	0001
VAL__EPSI	_INTEGER	1	00000001
VAL__EPSR	_REAL	1.1920929E−7	00003500
VAL__EPSD	_DOUBLE	2.7755575615628914E−17	0000000000002500

5.3 Maximum Data Values

Constants with names of the form VAL__MAX<T> represent the maximum (most positive) non-*bad values* which can be represented on the machine. The data types of these constants match the quantities they describe, as follows:

Constant	Type	Value	Hexadecimal Pattern
VAL__MAXUB	_UBYTE	254	FE
VAL__MAXB	_BYTE	127	7F
VAL__MAXUW	_UWORD	65534	FFFE
VAL__MAXW	_WORD	32767	7FFF
VAL__MAXI	_INTEGER	2147483647	7FFFFFFF
VAL__MAXR	_REAL	1.7014117E+38	FFFF7FFF
VAL__MAXD	_DOUBLE	1.701411834604923D+38	FFFFFFFFFFFF7FFF

Constants with names of the form NUM__MAX<T> represent the maximum (most positive) *numbers* which can be represented on the machine. The data types of these constants also match the quantities they describe, as follows:

Constant	Type	Value	Hexadecimal Pattern
NUM__MAXUB	_UBYTE	255	FF
NUM__MAXB	_BYTE	127	7F
NUM__MAXUW	_UWORD	65535	FFFF
NUM__MAXW	_WORD	32767	7FFF
NUM__MAXI	_INTEGER	2147483647	7FFFFFFF
NUM__MAXR	_REAL	1.7014117E+38	FFFF7FFF
NUM__MAXD	_DOUBLE	1.701411834604923D+38	FFFFFFFFFFFF7FFF

5.4 Minimum Data Values

Constants with names of the form VAL__MIN<T> represent the minimum (most negative) non-*bad values* which can be represented on the machine. The data types of these constants match the quantities they describe, as follows:

Constant	Type	Value	Hexadecimal Pattern
VAL__MINUB	_UBYTE	0	00
VAL__MINB	_BYTE	−127	81
VAL__MINUW	_UWORD	0	0000
VAL__MINW	_WORD	−32767	8001
VAL__MINI	_INTEGER	−2147483647	80000001
VAL__MINR	_REAL	−1.7014117E+38	FFFEFFFF
VAL__MIND	_DOUBLE	−1.701411834604923D+38	FFFEFFFFFFFFFFFF

Constants with names of the form NUM__MIN<T> represent the minimum (most negative) *numbers* which can be represented on the machine. The data types of these constants also match the quantities they describe, as follows:

Constant	Type	Value	Hexadecimal Pattern
NUM__MINUB	_UBYTE	0	00
NUM__MINB	_BYTE	−128	80
NUM__MINUW	_UWORD	0	0000
NUM__MINW	_WORD	−32768	8000
NUM__MINI	_INTEGER	−2147483648	80000000
NUM__MINR	_REAL	−1.7014117E+38	FFFFFFFF
NUM__MIND	_DOUBLE	−1.701411834604923D+38	FFFFFFFFFFFFFFFF

5.5 Data Storage Requirements

Constants with names of the form VAL__NB<T> represent the number of basic machine units (bytes) required to hold one value of each data type. All these constants are of type _INTEGER, with values as follows:

Constant	Type	Value
VAL__NBUB	_INTEGER	1
VAL__NBB	_INTEGER	1
VAL__NBUW	_INTEGER	2
VAL__NBW	_INTEGER	2
VAL__NBI	_INTEGER	4
VAL__NBR	_INTEGER	4
VAL__NBD	_INTEGER	8

5.6 Smallest Value

Constants with names of the form VAL__SML<T> represent the smallest positive (non-zero) value which can be represented on the machine. The data types of these constants match the quantities they describe, as follows:

Constant	Type	Value	Hexadecimal Pattern
VAL__SMLUB	_UBYTE	1	01
VAL__SMLB	_BYTE	1	01
VAL__SMLUW	_UWORD	1	0001
VAL__SMLW	_WORD	1	0001
VAL__SMLI	_INTEGER	1	00000001
VAL__SMLR	_REAL	2.9387359E−39	00000080
VAL__SMLD	_DOUBLE	2.9387358770557188E−39	0000000000000080

5.7 Character String Sizes

Constants with names of the form VAL__SZ<T> specify the maximum number of characters required to represent each numerical data type when formatted as a decimal string. All these constants are of type _INTEGER and should be used when declaring character variables to hold formatted numerical data. Their values are:

Constant	Type	Value
VAL__SZUB	_INTEGER	3
VAL__SZB	_INTEGER	4
VAL__SZUW	_INTEGER	5
VAL__SZW	_INTEGER	6
VAL__SZI	_INTEGER	11
VAL__SZR	_INTEGER	14
VAL__SZD	_INTEGER	22

6 COMPILING AND LINKING

6.1 Unix

Before attempting to compile or link applications, the Starlink executables directory (normally /star/bin) should be added to your PATH. Links to the installed include files for PRIMDAT may then be set up in your current working directory by executing the command:

```
% prm_dev
```

and applications containing INCLUDE file names in upper case, in the normal way, may be compiled and linked with the commands:

```
% f77 -o prog prog.f ‘prm_link‘
```

for stand-alone applications, or:

```
% alink prog.f ‘prm_link_adam‘
```

for ADAM applications.

6.2 VMS

The files required for compilation and linking with the routines in this package reside in a directory with logical name PRIMDAT_DIR. Before attempting to compile or link applications, logical names must be defined for the required files by executing the DCL command:

```
$ @PRIMDAT_DIR:START
```

Applications which contain Fortran INCLUDE statements referring to files in this directory may then be compiled.

Applications should be linked with routines from this package using the link options file PRM_LINK. For instance, to link an application called PROG, the DCL command:

```
$ LINK PROG,PRM_LINK/OPT
```

might be used. To link an ADAM application, the $LINK command would be replaced by the ADAM command $ALINK.

7 CHANGES IN THIS DOCUMENT

Although PRIMDAT is now available for Unix platforms, the major part of this document has not been revised and is still VMS oriented. The use of PRIMDAT is similar for both systems but note that exception handling is not available for all platforms (in particular, not for Alpha OSF/1).

Section 6.1 has been added for Unix users.

A PERFORMANCE STATISTICS

This Appendix gives performance statistics for the PRIMDAT routines in the form of typical execution times. These are provided primarily to assist in the choice of routine for a particular purpose, but they also provide a benchmark against which alternative algorithms and future improvements can be judged.

In practice, execution times depend on many factors, such as the particular processor being used, the number of page faults generated and (in many cases) the routine argument values. Consequently, great care should be exercised if comparing the figures given here with the performance of a real application. Nevertheless, the figures do give a good indication of the **relative** efficiency of the routines, and are generally repeatable within 5 or 10 per cent. Users who know of more efficient algorithms are encouraged to contact the author so that they may be tested under identical conditions and, if appropriate, incorporated into this package.

The statistics presented here have been gathered on the Durham MicroVAX II (DUMV1), using a program which processes a sequence of data obtained from an input array (or arrays) and returns the results to a separate output array. For instance, in the case of the NUM_ADDR function, it is equivalent to the simple loop:

```
      DO 1 I = 1, N
        A( I ) = NUM_ADDR( B( I ) , C( I ) )
    1 CONTINUE
```

The figures given represent the CPU time used to calculate one result (*i.e.* the total CPU time used by this loop divided by N), each being the median of five separate determinations. Note that the cost of executing the loop and of accessing the arguments and assigning the result is included, since these overheads will typically feature in most real applications.

The cost of page faults is not included, however. These will usually occur when data arrays are accessed for the first time (and subsequently if the array is large or has not been accessed for a while). The cost of page faults is difficult to quantify, as it may be installation-dependent and will also depend on the level of system activity at the time. However, as a very rough guide, page faulting on the Durham MicroVAX II adds a broad average of around 7μs to the CPU time figure for routines which access a single _REAL argument array and around 11μs for routines accessing two _REAL argument arrays. Page faults also impose a larger **elapsed** time overhead, so that their effect can be substantial and can often make small apparent differences in routine performance seem insignificant in practice.

VAL_ *Function*	*Data Type Code <T>*													
	D		**R**		**I**		**W**		**UW**		**B**		**UB**	
ADD<T>	100	82	94	78	89	76	89	77	95	83	89	76	97	85
SUB<T>	100	82	94	79	89	76	89	77	95	83	88	76	97	85
MUL<T>	102	83	95	79	93	81	94	82	99	88	92	79	101	91
DIV<T>	105	85	96	81	110	98	111	99	109	97	105	93	111	98
IDV<T>	151	134	139	122	103	91	98	85	103	90	94	82	105	94
PWR<T>	410	395	287	270	140	127	139	127	140	128	135	124	140	129
NEG<T>	51	44	48	42	47	48	47	48	59	57	48	48	52	51
SQRT<T>	120	115	111	107	114	113	115	113	122	119	115	114	113	111
LOG<T>	186	178	134	130	137	136	140	136	149	146	140	137	141	139
LG10<T>	194	190	141	137	146	143	146	143	157	154	146	143	149	147
EXP<T>	198	188	163	153	167	160	168	160	173	167	168	158	182	175
ABS<T>	53	47	48	42	49	48	54	55	45	41	49	50	43	43
NINT<T>	121	112	108	101	45	42	45	42	45	42	45	42	44	43
INT<T>	93	87	82	79	45	41	45	42	45	42	45	41	43	43
MAX<T>	68	58	66	59	58	53	56	52	64	60	60	56	66	62
MIN<T>	68	57	66	59	56	52	57	53	65	61	61	56	66	62
DIM<T>	103	83	95	80	97	85	88	76	65	61	95	82	66	63
MOD<T>	242	222	183	165	108	99	112	103	114	101	112	100	117	106
SIGN<T>	115	95	108	93	137	124	132	120	55	42	139	125	53	44
SIN<T>	258	247	196	186										
SIND<T>	222	210	172	163										
COS<T>	198	194	151	147										
COSD<T>	245	232	202	192										
TAN<T>	370	360	273	265										
TAND<T>	343	333	265	258										
ASIN<T>	255	248	206	202										
ASND<T>	258	253	206	202										
ACOS<T>	270	265	216	210										
ACSD<T>	273	270	218	212										
ATAN<T>	167	162	125	121										
ATND<T>	170	163	125	120										
ATN2<T>	232	218	194	180										
AT2D<T>	235	222	192	178										
SINH<T>	196	182	167	155										
COSH<T>	255	238	212	198										
TANH<T>	247	237	170	163										
	T	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**
	BAD argument set?													

Table 6: Approximate median execution times (μs per operation) for the VAL_ arithmetic and mathematical functions.

VEC_ *Routine*	*Data Type Code <T>*													
	D		**R**		**I**		**W**		**UW**		**B**		**UB**	
ADD<T>	21	10.6	16.6	8.6	11.2	6.6	11.7	6.3	17.0	12.8	11.4	6.8	19.4	14.8
SUB<T>	21	10.6	16.6	8.6	11.2	6.6	11.7	6.3	17.1	12.8	11.5	6.8	19.3	14.8
MUL<T>	23	12.2	17.4	9.3	15.2	10.6	16.5	11.1	21	17.0	15.1	10.1	25	20
DIV<T>	25	14.5	18.4	10.4	25	21	26	22	32	28	26	22	32	28
IDV<T>	77	68	66	56	18.7	13.9	20	14.9	24	20	16.7	12.1	28	23
PWR<T>	343	337	224	208	66	60	74	63	70	65	66	63	69	69
NEG<T>	10.7	6.1	8.3	4.8	8.1	7.5	8.1	7.5	13.0	10.8	8.1	7.4	13.8	11.8
SQRT<T>	78	72	72	67	78	73	77	74	76	74	80	74	76	77
LOG<T>	143	137	93	90	100	95	98	96	100	97	101	96	101	98
LG10<T>	155	148	100	97	108	102	107	104	107	104	108	103	108	106
EXP<T>	130	125	96	93	101	100	102	101	108	106	102	101	108	107
ABS<T>	11.0	6.5	7.7	4.4	9.2	8.6	9.1	8.6	5.4	3.7	11.0	10.4	5.3	3.8
NINT<T>	80	74	72	63	5.4	3.8	5.4	3.8	5.4	3.8	5.4	3.8	5.4	3.7
INT<T>	53	48	47	43	5.4	3.8	5.4	3.8	5.4	3.8	5.4	3.8	5.4	3.8
MAX<T>	20	11.8	14.3	8.3	9.0	6.2	8.9	6.3	15.1	12.6	11.1	8.5	17.0	14.4
MIN<T>	20	11.6	14.9	8.5	9.6	6.5	9.5	6.5	15.4	13.4	11.3	9.1	17.6	14.8
DIM<T>	22	13.8	16.8	10.6	12.3	9.3	12.8	8.8	15.7	13.2	13.6	12.1	17.8	15.0
MOD<T>	185	172	120	110	27	24	28	26	33	29	31	29	38	35
SIGN<T>	77	66	72	55	71	66	60	56	7.5	3.8	73	69	7.5	3.8
SIN<T>	182	178	122	126										
SIND<T>	147	143	101	103										
COS<T>	158	155	112	109										
COSD<T>	170	165	130	133										
TAN<T>	297	293	200	202										
TAND<T>	267	263	192	198										
ASIN<T>	212	206	165	165										
ASND<T>	213	210	165	165										
ACOS<T>	225	222	173	172										
ACSD<T>	228	225	175	175										
ATAN<T>	124	121	85	82										
ATND<T>	129	124	85	82										
ATN2<T>	192	178	150	134										
AT2D<T>	194	180	149	134										
SINH<T>	126	129	104	105										
COSH<T>	190	183	149	140										
TANH<T>	206	194	137	137										
	T	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**
	BAD argument set?													

Table 7: Approximate median execution times (μs per operation) for the VEC_ arithmetic and mathematical routines.

	Data Type Code <T>						
NUM_ *Function*	**D**	**R**	**I**	**W**	**UW**	**B**	**UB**
ADD<T>	8.5	6.4	4.5	4.4	10.6	4.0	12.1
SUB<T>	8.4	6.5	4.5	4.4	10.6	3.9	12.1
MUL<T>	10.3	7.2	8.6	9.2	14.8	7.4	17.2
DIV<T>	12.6	8.4	18.6	19.2	25	18.8	25
IDV<T>	65	55	11.5	13.2	17.8	9.2	21
PWR<T>	335	213	60	62	66	60	67
NEG<T>	5.2	4.4	3.5	3.5	9.4	3.1	10.1
SQRT<T>	73	68	73	74	75	72	73
LOG<T>	143	90	96	97	96	94	95
LG10<T>	154	97	102	103	104	102	103
EXP<T>	122	92	97	98	102	93	100
ABS<T>	6.6	4.2	4.4	4.4	0.6	5.6	0.3
NINT<T>	86	72	1.1	0.5	0.5	0.3	0.3
INT<T>	51	44	1.1	0.5	0.5	0.3	0.3
MAX<T>	11.2	8.1	6.3	6.1	11.6	7.0	13.3
MIN<T>	10.5	8.1	6.3	6.3	12.1	7.2	13.7
DIM<T>	10.4	7.9	6.4	6.1	12.4	7.5	13.9
MOD<T>	178	112	20	22	27	25	31
SIGN<T>	74	72	73	69	18.4	69	18.4
SIN<T>	176	118					
SIND<T>	140	95					
COS<T>	154	110					
COSD<T>	162	124					
TAN<T>	291	195					
TAND<T>	264	186					
ASIN<T>	198	158					
ASND<T>	201	157					
ACOS<T>	214	166					
ACSD<T>	218	167					
ATAN<T>	120	84					
ATND<T>	124	84					
ATN2<T>	184	142					
AT2D<T>	185	140					
SINH<T>	127	98					
COSH<T>	180	140					
TANH<T>	210	130					

Table 8: Approximate median execution times (μs per operation) for the NUM_ arithmetic and mathematical functions.

	Result Type Code <T>													
VAL_ *Function*	**D**		**R**		**I**		**W**		**UW**		**B**		**UB**	
DTO<T>	35	45	83	72	86	73	87	75	92	79	88	75	97	86
RTO<T>	50	49	33	33	84	74	84	74	93	86	84	74	93	83
ITO<T>	49	51	49	44	33	33	80	71	87	79	79	72	87	79
WTO<T>	49	51	48	44	45	42	33	33	90	84	80	71	86	77
UWTO<T>	50	51	48	45	45	41	84	79	33	33	79	72	87	79
BTO<T>	50	46	48	44	45	42	45	42	89	83	32	33	87	80
UBTO<T>	50	46	48	45	45	41	46	42	45	41	81	72	26	26
	T	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**
	BAD argument set?													

Table 9: Approximate median execution times (μs per operation) for the VAL_ type conversion functions.

	Result Type Code <T>													
VEC_ *Routine*	**D**		**R**		**I**		**W**		**UW**		**B**		**UB**	
DTO<T>	2.4	2.2	12.6	8.0	14.7	9.6	15.6	9.8	21	15.8	15.6	10.0	21	16.4
RTO<T>	9.1	5.2	1.2	1.2	12.4	8.4	13.3	9.3	18.5	14.1	13.4	9.2	18.8	15.6
ITO<T>	9.0	6.8	8.5	6.3	1.2	1.2	8.2	6.8	13.4	11.6	8.2	6.5	13.9	12.0
WTO<T>	9.1	6.8	8.4	6.3	5.8	3.6	0.6	0.7	14.2	11.8	8.0	6.5	13.4	11.1
UWTO<T>	9.5	7.3	9.0	6.6	5.4	3.2	8.5	6.6	0.7	0.6	8.9	6.3	14.3	12.6
BTO<T>	9.1	6.8	8.5	6.1	5.8	3.6	5.8	3.5	14.2	11.6	0.4	0.4	13.6	11.8
UBTO<T>	9.5	7.3	9.0	6.7	5.4	3.2	6.5	4.2	5.3	3.2	8.9	6.3	0.4	0.4
	T	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**
	BAD argument set?													

Table 10: Approximate median execution times (μs per operation) for the VEC_ type conversion routines.

	Result Type Code <T>						
NUM_ *Function*	**D**	**R**	**I**	**W**	**UW**	**B**	**UB**
DTO<T>	2.2	5.4	6.6	7.6	12.6	7.6	13.3
RTO<T>	5.1	1.1	6.2	6.9	12.0	7.0	12.7
ITO<T>	6.9	6.4	1.1	3.7	9.1	3.7	9.4
WTO<T>	6.7	6.2	3.5	0.5	9.7	3.7	8.8
UWTO<T>	7.1	6.7	3.1	4.0	0.5	4.1	9.9
BTO<T>	6.9	6.1	3.5	3.5	9.4	0.3	9.4
UBTO<T>	7.1	6.7	3.1	4.1	3.1	3.9	0.3

Table 11: Approximate median execution times (μs per operation) for the NUM_ type conversion functions.

B ERROR CODES & MESSAGES

This Appendix describes the error conditions recognised by this package and the associated codes and messages. If it is necessary to test for any of these error conditions, then symbolic constants (defined by Fortran PARAMETER statements) should be used to identify the error code in question.

The names of PRIMDAT error codes are of the form:

```
PRM__<code>
```

where <code> specifies the error condition. Their numerical values are defined in the file PRM_ERR, and may be incorporated into an application with the statement:

```
INCLUDE 'PRM_ERR'
```

The following gives the names of the symbolic constants defined in this file, together with the associated error messages and an explanation of each error condition:

PRM__ARGIN, function argument invalid
An invalid argument was supplied to a mathematical function.

PRM__FLTDZ, floating point divide by zero
An attempt was made to perform floating point division by zero.

PRM__FLTOF, floating point overflow
The absolute value of the result of a floating point calculation or type conversion exceeded the largest floating point number which can be represented on the machine.

PRM__FLTUF, floating point underflow
The absolute value of the result of a floating point calculation or type conversion was smaller than the smallest non-zero floating point number which can be represented on the machine. This error condition will not normally be used on a VAX unless underflow detection has been explicitly enabled.

PRM__INTDZ, integer divide by zero
An attempt was made to perform integer division by zero.

PRM__INTOF, integer overflow
The result of an integer (non-floating point) calculation or type conversion was outside the number range which can be represented by the data type in question.

PRM__LOGZN, logarithm of zero or negative number
An attempt was made to take the logarithm of zero or a negative number.

PRM__SLOST, numerical significance lost
Numerical significance was lost during evaluation of a mathematical function; a result could not be calculated with any meaningful precision.

PRM__SQRNG, square root of negative number
An attempt was made to take the square root of a negative number.

PRM__UDEXP, undefined exponentiation
An attempt was made to raise the number zero to a negative power, or to raise a negative number to a non-integer power.