QNT SOFTWARE DEVELOPMENT INC.

# ExLAF77 1.01

# A Variable- and Mixed-Precision Linear Algebra Library for Fortran-77 and Other Languages

## User Guide and Reference Manual


Printed in Canada
Document Number U1002-EN-100

# Table of Contents

# Section 1. Overview

## 1.1. What is ExLAF77

ExLAF77 has been designed and developed as an extended general-purpose mathematical library callable from applications that require error-free and/or variable-precision floating-point computations. Its first version supports:

- basic arithmetical operations on exact (i.e. signed integer and rational) and variable-precision real and complex floating-point numbers;
- guaranteed-accuracy variable-precision evaluation of a number of simplest transcendental functions included in the Fortran-77 standard;
- basic vector-vector, matrix-vector, and matrix-matrix algebraic operations for dense real and complex vectors and matrices, represented as uniform arrays of machine native or multi-precision floating-point numbers;
- arbitrarily accurate solution of systems of linear equations with dense real and complex square matrices including general, Hermitian positive-definite and indefinite ones;
- arbitrarily accurate solution of eigenvalue problems for dense real and complex square matrices of the same kinds.

ExLAF77 is intended mainly for applied computations rather then academic research. It does not support specific math operations implemented in computer algebra systems (primality tests, modular arithmetical procedures, etc.), nor does it use advanced algorithms for processing very long numbers, such as Karatsuba method, FFT, and others. Thus, it should not be treated as a new CAS.

Instead, ExLAF77 offers a number of features that provide extended automatism, flexibility, and reliability when being used as a low-level library called from scientific and engineering applications. In particular:

- it executes operations on objects of abstract types so that the user does not have to know and explicitly declare types of the computational results;
- it automatically recognizes types and precision of operands, converts them to a highest type, and selects an appropriate representation form for the result;
- it detects all computational anomalies, such as underflow, overflow, square root of negative value etc., and selects a proper representation form for the result to produce a correct output;
- it allows operations on mixed-type operands, when one of them has a machine native type, while another one is represented by an object of abstract type, or they have different precision;
- when evaluating transcendental functions it provides user-defined accuracy without any adjustment of system parameters or repeated calculations.

- for executing vector and matrix operations it uses a bitwise-optimized arithmetical engine that illustrates a quite reasonable speed of processing uniform arrays of moderately big floats;
- its external interfaces make it possible for the user's application to create and manipulate non-uniform (i.e. mixed-type) arrays of arbitrary objects;
- it includes interface tools for importing and exporting numerical data in machine native formats, supports unformatted I/O operations with user-defined binary files and formatted text I/O;
- finally, it does not require some specific programming environment, and can be easily integrated with any Fortran-77, Fortran-90/95, C, or C++ application.

Some of those features allow easy development of guaranteed-precision algorithms that check precision of computational results and repeat calculations with incremental increase of their working precision until required accuracy is reached.

The first ExLAF77 version is developed for X86 platform. It is a good workhorse for applied computations in the fields of computational geometry, stability analysis, numerical solution of ill-conditioned, ill-posed and other problems sensitive to round-off errors.

## 1.2. Why Fortran-77?

As it is widely known, a big percentage of currently used applied codes is written in Fortran-77 just due to a huge accumulated amount of highly optimized and exhaustively tested Fortran libraries that are compatible with virtually any OS and hardware platform. Fortran-77 code has simplest and most straightforward interfaces since it does not require any extra environment like headers, make-files, or preprocessors. In addition, it possesses one-way compatibility with codes written in modern languages. For example, it is not a problem to build a cross-language application that would include low-level Fortran-77 routines callable from high-level Fortran-90/95, C or C++ modules.

However, the inverse calling sequence, such as calling C++ library from a Fortran code, in general case cannot be implemented so easily. That is why Fortran-oriented interfaces seem to be the most convenient for supporting software development in different environments specified by programming language, OS, and hardware platform.

ExLAF77 offers a number of features of the most advanced languages, such as abstraction mechanism, exceptions handling, and dynamic memory allocation to software developers who write their codes in more conservative languages. As far as ExLAF77 is callable from Fortran-77 it can be easily embedded in any application written in Fortran-90/95, C, or C++ as well.

## 1.3. Handled Objects

ExLAF77 math objects accessible from external applications are identified by their unique "handles" represented by integer variables. In this manual they are called "**Handled Objects**" or just "**H-objects**". ExLAF77 executes operations on the following classes of H-objects:

- signed and unsigned infinities;
- short (4-byte) and arbitrarily long signed Integer numbers;
- arbitrarily long signed rational numbers;

- single (4-byte) and double (8-byte) precision real and complex floating-point numbers;
- arbitrarily long real and complex floating-point numbers;
- dense real and complex vectors represented as uniform arrays of single or double precision floating-point numbers;
- dense real and complex vectors represented as uniform arrays of arbitrarily long floating-point numbers;
- dense real and complex square and rectangular matrices represented as uniform arrays of floating-point numbers of the same kinds;
- dense real and complex Hermitian matrices represented as uniform arrays of floating-point numbers of the same kinds and stored in a packed form;
- complete triangular decompositions of general real and complex square matrices in the same representations;
- complete triangular decompositions of real and complex positive-definite and indefinite Hermitian matrices in the same representations;
- Hessenberg forms of general real and complex square matrices in the same representations;
- tridiagonal forms of real and complex positive-definite and indefinite Hermitian matrices in the same representations.

When executing operations ExLAF77 allows using abstract handles as operands. Therefore, the user's application can perform any meaningful operation without explicit type declarations for its operands and the result. For example, multiplication $\mathbf{a}\cdot\mathbf{A}$, where $\mathbf{a}$ is a finite number and $\mathbf{A}$ is a matrix, does not require extra specifications whether $\mathbf{a}$ is real or complex, whether it has an exact, machine native or multi-precision representation. Similarly, operand $\mathbf{A}$ can be referenced as an abstract matrix without knowing whether it is general or Hermitian, real or complex, etc.

## 1.4. *Create&Assign* and *Update* Operations

Operations supported by ExLAF77 can be divided into four main groups: a) arithmetical operations on numbers, b) algebraic vector/matrix operations, c) evaluation of math constants and functions, and d) system tools, I/O and miscellaneous operations.

Arithmetical operations on numbers include:

- operations of assigning finite values to floating-point numbers and real/imaginary parts of complex numbers with automatic type conversion;
- comparison operations similar to the Fortran `.EQ.`, `.LT.`, and `.GT.` logical operators (last two for real numbers);
- tests for zero, negative, and positive value (last two for real numbers);
- absolute and complex-conjugate values, extraction of real and imaginary parts similar to the Fortran `ABS`, `CONJ`, `REAL`, and `IMAG` generic intrinsic functions;
- unary "`+`" and "`-`" operations;
- four standard arithmetical binary operations (`+`, `-`, `*`, `/`);
- multiplication by an integer power of 2;

- integer quotient and remainder in division of two exact numbers;
- extraction of integer numerator and denominator of an exact number;
- test for parity of an integer number;
- minimum, maximum, and "machine epsilon" values for given sizes of mantissa and exponent fields of a multi-precision real float.

Arithmetical operations with floating-point and mixed-type operands are realized in two versions: so-called "***Create&Assign***" and "***Update***" ones. Each *Create&Assign* operation creates a new resulting H-object whose type is appropriately selected to represent a correct output. However, in cases of undefined result *Create&Assign* operations generate errors.

In contrast, *Update* operations try to assign the result to an existing H-object and generate errors in cases of type incompatibility, overflow, underflow etc. Arithmetical operations with exact operands are realized only in the *Create&Assign* version, i.e. exact numbers cannot be updated.

Algebraic vector/matrix operations include:

- assign operations with automatic type conversion;
- assigning finite values to selected elements or their imaginary/real parts;
- splitting into imaginary and real parts, and constructing complex conjugate vectors and matrices;
- multiplication and division by a finite number;
- addition, subtraction, left and right multiplication;
- linear combination of two vectors or matrices with a matrix factor;
- vector and matrix dot products;
- triangular decomposition of square matrices;
- multiple-RHS solution of linear algebraic systems with an option of transposed matrix;
- transformation of square matrices to Hessenberg or tridiagonal form;
- solution of linear eigenvalue problems.

Simplest algebraic vector/matrix operations are realized in both *Create&Assign* and *Update* versions. Triangular decompositions, transformations to Hessenberg and tridiagonal forms, and solvers of linear eigenvalue problems are represented by their *Update* versions only.

ExLAF77 evaluates the following math constants, algebraic and transcendental functions:

- constants $\pi$, e, and **ln2**;
- factorial of a natural number;
- square root (`SQRT`);
- exponential function (`EXP`) and natural logarithm (`LOG`);
- sine (`SIN`), cosine (`COS`), and tangent (`TAN`);
- arc sine (`ASIN`), arc cosine (`ACOS`), and arc tangent (`ATAN`);
- arc tangent of two real arguments (`ATAN2`);
- hyperbolic sine (`SINH`), cosine (`COSH`), and tangent (`TANH`);

- inverse hyperbolic sine (ASINH), cosine (ACOSH), and tangent (ATANH).

Square root and transcendental functions accept any abstract number as an argument and return result of user-defined bit accuracy. Generally, types of the output results of those functions are not known in advance since they depend on arithmetical values of arguments. For this reason all the functions are realized in *Create&Assign* versions only.

System tools, I/O and miscellaneous operations include:

- formatted decimal output of numbers and selected vector/matrix elements to a text string;
- text input of numerical data with an option of creating and initializing numbers whose type has to be automatically selected in accordance with format of the input string;
- binary I/O operations with user-defined files that read and write H-objects via user-supported callback subroutines;
- transformation of Fortran numbers and numerical arrays into H-objects and the inverse operations;
- operations of creating, initializing, and deleting H-objects;
- information on class membership and specific properties of H-objects;
- dynamic masking and unmasking of error messages;
- dynamic switching of floating-point underflow control (allowed/not allowed);
- opening and closing ExLAF77 working session.

## 1.5. User Interface

All the operations are executed via calling ExLAF77 interface subroutines described in this Manual. To open ExLAF77 working session the Fortran application should call subroutine **HINIT** that sets a number of user-defined parameters and initializes system data structures used by low-level subsystems for memory managing, exceptions handling, floating-point errors detecting, etc. Note that no one of ExLAF77 functions can work properly until the system is initialized. To close working session the user's application should call subroutine **HEXIT** that removes all the created H-objects and auxiliary data structures from computer memory, and closes system log file.

Therefore, all other ExLAF77 operations can be executed only between consecutive calls **HINIT** and **HEXIT**. Before closing working session the user's application should save all required data, i.e. output them to text string(s), write to binary file(s), or convert them to machine native types and store in respective Fortran variables and arrays. ExLAF77 working session can be repeatedly opened and closed as many times as necessary during program run.

Fortran program identifies each of H-objects by a unique `INTEGER` variable (handle) that stores an absolute address of the H-object in computer memory. Calling code can use handles like all other variables, i.e. declare arrays of handles, use them as elements of common blocks, parameters of subprograms, etc., but it should never modify their values.

Operations are executed by calling respective interface subroutines that accept handles as actual parameters. For example, let `INTEGER` variables `INUM, IVECT`, `IMATR` in user's code serve as handles of previously created finite number $\mathbf{a}$, vector $\mathbf{x}$, and square matrix $\mathbf{A}$. Then statements

```
CALL HUMHH( INUM,  INUM,  'L', *100 )
CALL HUMHH( IVECT, INUM,  'L', *200 )
CALL HUMHH( IMATR, INUM,  'L', *300 )
CALL HUMHH( IMATR, IVECT, 'R', *400 )
CALL HUMHH( IMATR, IMATR, 'L', *500 )
```

invoke *Update* multiplication operations **a = a·a, x = x·a, A = A·a, x = A·x, x = x·A,** and **A = A·A** respectively. The 3-rd parameter of **HUMHH** specifies the operand to be updated, and the 4-th one defines a label for the alternate (erroneous) return.

ExLAF77 *Update* operations modify existing H-objects identified by their handles, while *Create&Assign* operations create new H-objects and associate them with given `INTEGER` variables that serve as handles in succeeding operations. Interface subroutines intended specifically for creating new H-objects behave like *Create&Assign* operations, i.e. associate new H-objects with `INTEGER` variables. On deleting H-objects their handles are set to zero.

ExLAF77 interface subroutines realize exceptions handling via Fortran-77 alternate return mechanism. Lists of formal parameters of virtually all of those subroutines include asterisk and the `RETURN 1` statement is executed when an exception is caught. The calling Fortran code should specify statement labels as respective actual parameters and make provision for appropriate processing erroneous events. In particular, the Fortran code can inquire for numerical error code, analyze it, and try to recover the error in run time (e.g. by increasing accuracy of calculations).

Erroneous values of actual parameters detected by interface subroutines before calling ExLAF77 kernel math procedures are processed as if they would catch exceptions. Thus, any run-time error regardless of its nature results in execution of the alternate return statement. By default, detection of any error is accompanied with recording a brief message to the user-defined log file. However, if the user's code processes and recovers some "harmless" erroneous events it can mask selected kinds of errors or even all of them to suppress over-filling log file with multiple useless messages.

## 1.6. Limitations

ExLAF77 does not use advanced algorithms for executing arithmetical operations on very long numbers just because it is not intended for pure academic fields of researches such. as computational number theory. However, it illustrates a good performance for moderately long numbers. Note that the first ExLAF77 version has been developed without programming optimization.

Design of ExLAF77 internal data structures imply two limitations to the way of storing H-objects and sizes of extended numbers:

- all operands of any math operation should be stored in computer RAM (incore), and the result of operation is always placed incore as well.
- each of mantissa and exponent fields of multi-precision floating-point numbers cannot exceed $2^{31}$ - 1 bits (about 256 Mbytes) in size;

However, in applied computations those limitations are not too restrictive.

# Section 2. H-Object Classification

ExLAF77 is written in C++. Its architecture is based on a strict classification of mathematical objects and operations expressed in terms of C++ class hierarchies. Probably, not all of the software developers who write their codes in Fortran are quite familiar with C++ inheritance mechanism, so it might seem that this section cannot be useful for them. However, use of ExLAF77 implies a very general comprehension of the classification rather than C++ itself. Presented in this section hierarchy charts are understandable on an intuitive level that does not require deep immersion in programming details. They are particularly helpful for development of generalized algorithms that safely manipulate abstract handles while keeping compatibility of operations with types of operands.

**Note:** In this manual the "Fortran number", "Fortran operand", etc. mean a number represented in one of hardware-supported formats: `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, or `DOUBLE COMPLEX`.

## 2.1. Hierarchy ANumber

ExLAF77 executes arithmetical operations on numbers represented in different forms. All of them are united in hierarchy derived from the base abstract class ANumber (see Chart 2.1-1 below).

The abstract classes introduce operations valid for all their descendants. If, for example, an ExLAF77 interface subroutine performing a binary operation accepts arguments of the abstract type AFFloat, then any combination of six particular kinds of numbers (CFReal4, CFReal8, CFRealX, CFComplex4, CFComplex8, CFComplexX) can be processed by that subroutine. Thus, the hierarchy explicitly classifies different kinds of numbers by the criterion of applicability of math operations. Operations introduced by the abstract classes are listed in the Table 2.1-1 below.

**Table 2.1-1. Distribution of Operations over Hierarchy ANumber**

| Class Name and Abstraction Scope | Main Operations |
|---|---|
| ANumber<br>Generic number | ▪ `.EQ.` and test for zero<br>▪ *Create&Assign* unary operations `+`, `-`, `ABS`, `CONJ`, `REAL`, `IMAG`<br>▪ *Create&Assign* binary `+`, `-`, `*`, `/` with a Fortran number as the right operand, and multiplication by $2^N$<br>▪ *Create&Assign* binary `+`, `-`, `*`, `/` with the right operand ANumber<br>▪ Formatted I/O |
| AReal<br>Real number | ▪ `.LT.`, `.GT.`, and test for sign<br>▪ Integer quotient and remainder in division<br>▪ Conversion to a Fortran number |
| AComplex<br>Complex number | No extra operations |
| AFReal<br>Finite real number | ▪ Integer quotient and remainder in division<br>▪ Conversion to Fortran standard floating-point types |

| Class Name and Abstraction Scope | Main Operations |
|---|---|
| AFRealFloat<br>Real floating-point number | ▪ Creation of a real number with specified lengths of its mantissa and exponent fields<br>▪ Assignment of a real Fortran number or H-number AFReal<br>▪ *Update* unary operations `-`, `ABS`, `CONJ`, `SQRT`, and inverse<br>▪ *Update* binary `+`, `-`, `*`, `/` with a real Fortran number as the right operand, and multiplication by $2^N$<br>▪ *Update* binary `+`, `-`, `*`, `/` with the right operand AFReal<br>▪ Setting min, max, and "machine epsilon" values. |
| AFComplex<br>Finite complex number | ▪ Conversion to Fortran standard floating-point types |
| AFComplexFloat<br>Complex floating-point number | ▪ Creation of a complex number with specified lengths of the mantissa and exponent fields of its real and imaginary parts<br>▪ Assignment of a real or complex Fortran number or H-number AFinite<br>▪ Selective assignment of a real Fortran number or H-number AFReal to the real or imaginary part<br>▪ *Update* unary operations `-`, `ABS`, `CONJ`, `SQRT`, and inverse<br>▪ *Update* binary `+`, `-`, `*`, `/` with a real or complex Fortran as the right operand, and multiplication by $2^N$<br>▪ *Update* binary `+`, `-`, `*`, `/` with the right operand AFinite |
| AFRealExact<br>Real number in exact representation | ▪ Extraction of numerator and denominator |
| AFInteger<br>Integer number | ▪ Test for parity. |

However, the system of different kinds of numbers, their machine representations, and a variety of permissible operations cannot be described by a simple tree-structured scheme. It is often necessary to use concretization sequence based on alternative criteria. As the standard C++ multiple inheritance mechanism is unable to resolve this problem efficiently, hierarchy ANumber is supplemented with two switch classes that unify some important operations. When being used in ExLAF77 interfaces each of them should be treated as an ordinary abstract base class.

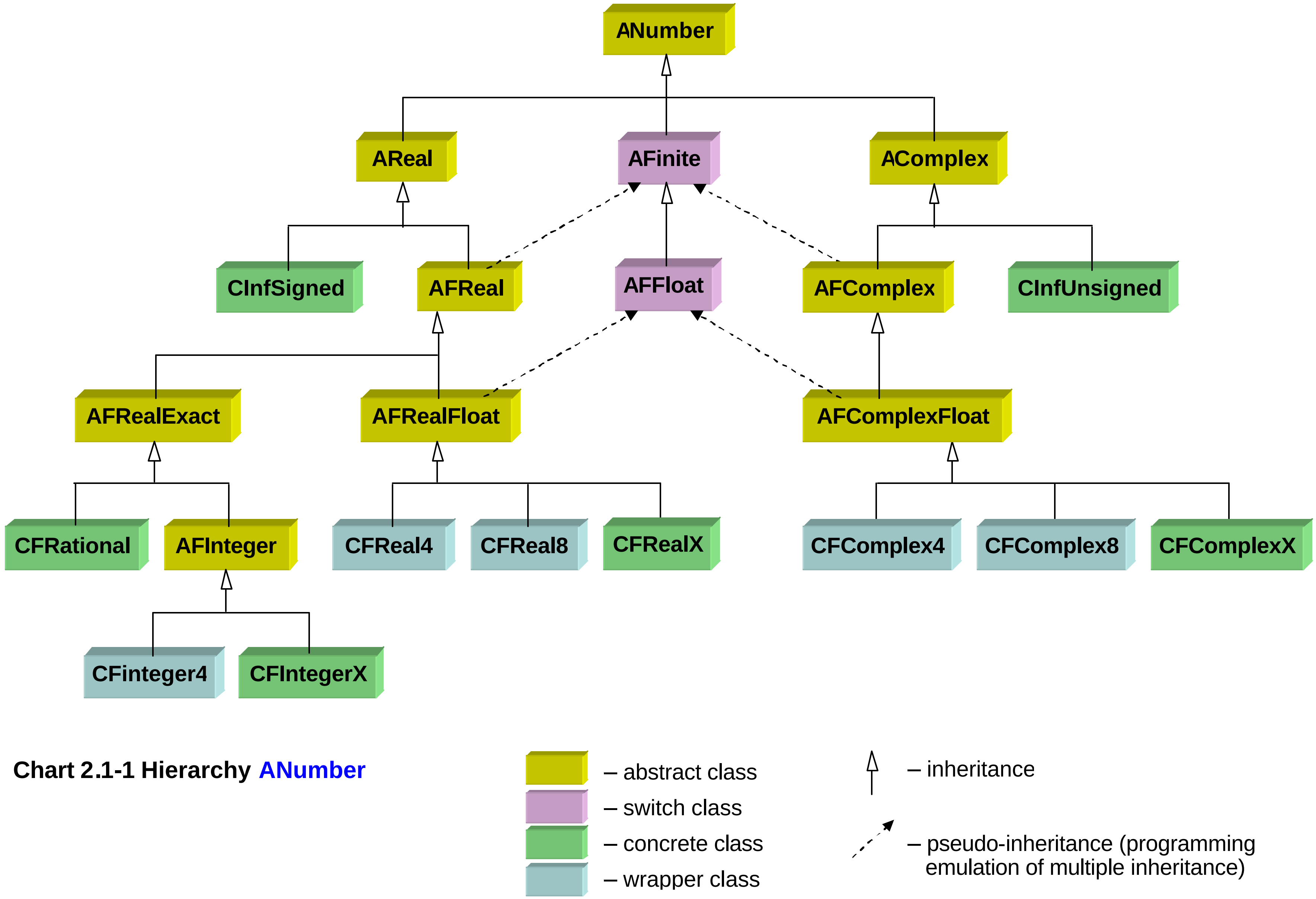



**Chart 2.1-1 Hierarchy ANumber**

**Table 2.1-2. Switch Classes Derived from ANumber**

| Class Name and Abstraction Scope | Unified Operations |
|---|---|
| AFinite<br>Generic finite number | ▪ Conversion to standard Fortran floating-point types<br>▪ *Create&Assign* multiplication by H-vector AVector and H-matrix AMatrix (see sections 2.1, 2.2 below) |
| AFFloat<br>Floating-point number | ▪ Assignment of a Fortran number or H-number AFinite<br>▪ *Update* versions of all the arithmetical operations |

Note that the mixed-type assignment and *Update* binary operations that combine real and complex operands are originally illegal if they imply assigning complex result to a real number. Since in these cases there is no way produce any meaningful result, ExLAF77 processes such operations as run-time errors and output message “ASSIGN COMPLEX TO REAL”. If there is a danger of arising errors of this kind the user’s code should check whether the respective operands of assignment or an *Update* binary operation are real or complex before calling respective ExLAF77 interface subroutine. This is specifically important if user’s algorithm manipulate abstract handles AFinite or AFFloat that do not make a difference between real and complex numbers.

Wrapper classes serve as containers for the Fortran numerical variables. They are intended for safe executing operations regardless of numerical values of operands. In contrast to hardware-supported arithmetic and math procedures included in system libraries, operations on H-objects of the wrapper classes never return invalid, erroneous or undefined values. Those operations in run-time verify intermediate data and properly process all detected anomalies, such as division by zero, underflow, overflow, square root of negative argument, and many others. On discovering invalid data *Create&Assign* operations appropriately change type of the resulting H-object, while *Update* operations generate errors.

**Table 2.1-3. Wrapper Classes Derived from ANumber**

| Class Name | Data Members | Respective Fortran-77 Type |
|---|---|---|
| CFInteger4 | 32-bit signed integer | `INTEGER` |
| CFReal4 | 32-bit IEEE floating-point number | `REAL` |
| CFReal8 | 64-bit IEEE floating-point number | `DOUBLE PRECISION` |
| CFComplex4 | A pair of 32-bit IEEE floating-point numbers | `COMPLEX` |
| CFComplex8 | A pair of 64-bit IEEE floating-point numbers | `DOUBLE COMPLEX` |

Concrete classes listed in the Table 2.1-2 below introduce types of numbers that do not have equivalent hardware-supported representations.

**Table 2.1-4. Concrete Classes Derived from ANumber**

| Class Name | Number Kind |
|---|---|
| CInfSigned | Signed (real) infinity |
| CInfUnsigned | Unsigned (complex) infinity |
| CFIntegerX | Extended signed integer number |
| CFRational | Extended signed rational number, represented as a pair of AFInteger |
| CFRealX | Extended floating-point real number |

| Class Name | Number Kind |
|---|---|
| CFComplexX | Extended floating-point complex number represented as a uniform pair of CFRealX with a common precision specifier |

## 2.2. Hierarchy AVector

Vectors and matrices have different sense in physics. For example, four different product operations are defined for physical vectors: dot, conjugate dot, Cartesian, and cross products. First three of them have obvious generalizations for matrix operands, while the last one does not have any sense for matrices. Furthermore, in contrast to linear algebra, tensor calculus typically does not require qualifying a vector as a "row" or "column". This makes the cause for using separate representations for vectors and matrices.

Currently ExLAF77 executes operations only on dense vectors stored as uniform arrays of real and complex numbers. Representations of vectors are united in the hierarchy derived from the base abstract class AVector (see Chart 2.2-1). Its nearest descendant AUVector is intended for deriving only floating-point, i.e. approximate vector representations that makes it possible to add in future a parallel inheritance branch for dense vectors in exact representations.

AVector and its abstract descendants introduce the following operations:

**Table 2.2-1. Distribution of Operations over Hierarchy AVector**

| Class Name and Abstraction Scope | Main Operations |
|---|---|
| AVector<br>Generic dense vector | ▪ `.EQ.` and test for zero<br>▪ *Create&Assign* vector unary operations `+`, `-`, `CONJ`, `REAL`, `IMAG`<br>▪ *Create&Assign* vector-vector binary `+` and `–`<br>▪ *Create&Assign* multiplication and division by a Fortran number or H-number AFinite<br>▪ *Create&Assign* dot and conjugate dot vector products<br>▪ *Create&Assign* multiplication by H-matrix AUMatrix<br>▪ Finding element of maximum or minimum norm<br>▪ Conversion to a Fortran array<br>▪ Extraction of selected element<br>▪ Formatted I/O of selected element |
| AUVector<br>Dense vector composed of uniform floating-point numbers | ▪ Creation a vector with specified lengths of the mantissa and exponent fields of its elements<br>▪ Initialization by a real Fortran array<br>▪ Assignment of a real Fortran number or H-number AFReal to selected element<br>▪ Assignment of H-vector AUVectorReal<br>▪ *Update* vector unary operations `-` and `CONJ`<br>▪ *Update* vector-vector binary `+` and `–` with the right operand AUVectorReal<br>▪ *Update* multiplication and division by a real Fortran number or H-number AFReal<br>▪ *Update* left and right multiplications by a real H-matrix AUMatrixSq |
| AUVectorReal<br>Dense uniform real vector | ▪ Finding maximum or minimum element |

| Class Name and Abstraction Scope | Main Operations |
|---|---|
| AUVectorCompl<br>Dense uniform complex vector | ▪ Initialization by a complex Fortran array<br>▪ Assignment of a complex Fortran number or H-number AFinite to selected element<br>▪ Selective assignment of a real Fortran number or H-number AFReal to the real or imaginary part of selected element<br>▪ Assignment of H-vector AUVector<br>▪ *Update* vector-vector binary + and – with the right operand AUVector<br>▪ *Update* multiplication and division by a complex Fortran number or H-number AFinite<br>▪ *Update* left and right multiplications by a complex H-matrix AUMatrixSq<br>▪ *Update* solution of a single-RHS linear algebraic complex system with an option of matrix transposition (left- and right multiplications by H-object AUCompleteLU) |

Mixed-type assignment and *Update* binary vector-number, vector-vector, and vector-matrix operations that combine real and complex operands are potentially dangerous. To avoid run-time errors resulted from attempts of assigning complex numbers to real ones, the user's code should check in advance whether the respective operands are real or complex.

Table 2.2-2 below explains composition of the concrete descendant classes:

**Table 2.2-2. Concrete Classes Derived from AVector**

| Class Name | Internal Representation |
|---|---|
| CUVectorReal4 | Array of *n* 32-bit IEEE floating-point numbers |
| CUVectorReal8 | Array of *n* 64-bit IEEE floating-point numbers |
| CUVectorRealX | Array of *n* CFRealX with a common precision specifier |
| CUVectorCompl4 | Array of $2 \cdot n$ 32-bit IEEE floating-point numbers |
| CUVectorCompl8 | Array of $2 \cdot n$ 64-bit IEEE floating-point numbers |
| CUVectorComplX | Array of *n* CFComplX with a common precision specifier |

Here *n* denotes dimension of a vector.

## 2.3. Hierarchy AMatrix

ExLAF77 supports a number of basic linear algebra operations on dense matrices stored as uniform arrays of real and complex floating-point numbers. Their representations are united in hierarchy with the base abstract class AMatrix (see Chart 2.3-1 below). Just as AVector, the class AMatrix is reserved for future deriving a parallel inheritance branch for exact representations of dense matrices.

Class AUMatrixRect unites matrices with strictly different dimensions. i.e. square matrices have mandatory membership AUMatrixSq. Descendants of AUMatrixSqHerm have an internal signature specifier that indicates whether the matrix is positive-definite or indefinite. The signature specifier should be explicitly initialized at the stage of creating Hermitian H-matrix.

.Operations introduced by AMatrix and its abstract descendants are listed in the Tables 3.2-1 and 3.2-2 below. They are similar to AUVector operations with the exception of transferring operations specific for real or complex matrices from abstract to switch classes.

**Table 2.3-1. Distribution of Operations over Hierarchy AMatrix**

| Class Name and Abstraction Scope | Main Operations |
|---|---|
| AMatrix<br>Generic dense matrix | ▪ .EQ. and test for zero<br>▪ *Create&Assign* matrix unary operations +, -, CONJ, REAL, IMAG<br>▪ *Create&Assign* matrix-matrix binary + and -<br>*Create&Assign* multiplication and division by a Fortran number or H-number AFinite<br>▪ *Create&Assign* multiplication by H-vector AUVector<br>▪ *Create&Assign* multiplication by H-matrix AUMatrix<br>▪ *Create&Assign* generalized conjugate dot matrix product<br>▪ Conversion to a Fortran array<br>▪ Extraction of selected element, row, or column<br>▪ Formatted I/O of selected element |
| AUMatrix<br>Dense matrix composed of uniform floating-point numbers | ▪ Creation of a matrix with specified lengths of the mantissa and exponent fields of its elements<br>▪ Initialization of selected row, column, or entire matrix by a real Fortran array<br>▪ Assignment of a real Fortran number or H-number AFReal to a selected element<br>▪ Assignment of H-vector AUVectorReal to selected row or column<br>▪ Assignment of H-matrix AUMatrixReal<br>▪ *Update* matrix unary operations - and CONJ<br>▪ *Update* matrix-matrix binary + and – with the right operand AUMatrixReal<br>▪ *Update* multiplication and division by a real Fortran number or H-number AFReal<br>▪ *Update* left and right multiplications by a real H-matrix AUMatrixSq |
| AUMatrixRect<br>Dense uniform strictly rectangular matrix | No extra operations |
| AUMatrixSq<br>Dense uniform square matrix | No extra operations. H-objects of the class can participate in *Update* operations of complete LU-decomposition and transformation to Hessenberg form introduced by AUCompleteLU and AUHessenberg |
| AUMatrixSqGen<br>General dense uniform square matrix in the full storage format | No extra operations. H-objects of the class can participate in *Update* operations of complete LU-decomposition and transformation to Hessenberg form introduced by AUCompleteLUGen and AUHessenbergGen |
| AUMatrixSqHerm<br>Dense uniform Hermitian matrix in the packed storage format | No extra operations. H-objects of the class can participate in *Update* operations of complete LU-decomposition and transformation to Hessenberg form introduced by AUCompleteLUHerm and AUHessenbergHerm |

**Table 2.3-2. Switch Classes Derived from AMatrix**

| Class Name and Abstraction Scope | Unified Operations |
|---|---|
| AUMatrixReal<br>Generic real dense uniform matrix | ▪ No extra operations |

| Class Name and Abstraction Scope | Unified Operations |
|---|---|
| AUMatrixCompl<br>Generic complex dense uniform matrix | ▪ Initialization of selected row, column, or entire matrix by a complex Fortran array<br>▪ Assignment of a complex Fortran number or H-number AFinite to selected element<br>▪ Selective assignment of a real Fortran number or H-number AFReal to the real or imaginary part of selected element<br>▪ Assignment of H-vector AUVector to selected row or column<br>▪ Assignment of H-matrix AUMatrix<br>▪ *Update* matrix-matrix binary + and – with the right operand AUMatrix<br>▪ *Update* multiplication and division by a complex Fortran number or H-number AFinite<br>▪ *Update* left and right multiplications by a complex H-matrix AUMatrixSq<br>▪ *Update* solution of a multiple-RHS linear algebraic complex system with an option of matrix transposition (left- and right multiplications by H-object AUCompleteLU) |

Mixed-type assignment and *Update* binary matrix-number, matrix-vector, and matrix-matrix operations that combine real and complex operands are potentially dangerous. To avoid run-time errors resulted from attempts of assigning complex numbers to real ones, the user's code should check in advance whether the respective operands are real or complex.

Table 2.3-3 below explains composition of the concrete descendant classes:

**Table 2.3-3. Concrete Classes Derived from AMatrix**

| Class Name | Internal Representation |
|---|---|
| CUMatrixRectReal4 | Array of $n{\cdot}m$ 32-bit IEEE floating-point numbers |
| CUMatrixRectReal8 | Array of $n{\cdot}m$ 64-bit IEEE floating-point numbers |
| CUMatrixRectRealX | Array of $n{\cdot}m$ CFRealX with a common precision specifier |
| CUMatrixRectCompl4 | Array of $2{\cdot}n{\cdot}m$ 32-bit IEEE floating-point numbers |
| CUMatrixRectCompl8 | Array of $2{\cdot}n{\cdot}m$ 64-bit IEEE floating-point numbers |
| CUMatrixRectComplX | Array of $n{\cdot}m$ CFComplX with a common precision specifier |
| CUMatrixSqGenReal4 | Array of $n^2$ 32-bit IEEE floating-point numbers |
| CUMatrixSqGenReal8 | Array of $n^2$ 64-bit IEEE floating-point numbers |
| CUMatrixSqGenRealX | Array of $n^2$ CFRealX with a common precision specifier |
| CUMatrixSqGenCompl4 | Array of $2\ n^2$ 32-bit IEEE floating-point numbers |
| CUMatrixSqGenCompl8 | Array of $2{\cdot}n^2$ 64-bit IEEE floating-point numbers |
| CUMatrixSqGenComplX | Array of $n^2$ CFComplX with a common precision specifier |
| CUMatrixSqHermReal4 | Array of $n{\cdot}(n+1)/2$ 32-bit IEEE floating-point numbers |
| CUMatrixSqHermReal8 | Array of $n{\cdot}(n+1)/2$ 64-bit IEEE floating-point number |
| CUMatrixSqHermRealX | Array of $n{\cdot}(n+1)/2$ CFRealX with a common precision specifier |
| CUMatrixSqHermCompl4 | Array of $n{\cdot}(n+1)$ 32-bit IEEE floating-point numbers |
| CUMatrixSqHermCompl8 | Array of $n{\cdot}(n+1)$ 64-bit IEEE floating-point numbers |

| Class Name | Internal Representation |
|---|---|
| CUMatrixSqHermComplX | Array of $n \cdot (n+1)/2$ CFComplX with a common precision specifier |

Here *n* and *m* denote dimensions of a matrix.

## 2.4. Hierarchy ACompleteLU

ExLAF77 linear algebra operations include solving dense systems of linear equations of the forms: $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, $\mathbf{x} \cdot \mathbf{A} = \mathbf{b}$, $\mathbf{A} \cdot \mathbf{X} = \mathbf{B}$ and $\mathbf{X} \cdot \mathbf{A} = \mathbf{B}$, where $\mathbf{A}$ is an H-matrix AUMatrixSq, $\mathbf{b}$ and $\mathbf{x}$ are H-vectors AUVector, $\mathbf{B}$ and $\mathbf{X}$ are H-matrices AUMatrix. Depending on particular kind of the matrix $\mathbf{A}$ one of two standard complete triangular decomposition methods is used:

- Crout's LU-factorization with partial pivoting for general H-matrices AUMatrixSqGen and indefinite Hermitian H-matrices AUMatrixSqHerm;
- Cholesky's $U^T U$-factorization for positive definite Hermitian H-matrices AUMatrixSqHerm;

In addition to triangular factors the result of LU-decomposition includes permutation vector used when computing solution for a given RHS. Hierarchy with the base abstract class ACompleteLU illustrated by Chart 2.4-1 below holds respective data structures. ACompleteLU and Its abstract descendants introduce the following operations:

**Table 4.2-1. Distribution of Operations over Hierarchy ACompleteLU**

| Class Name and Abstraction Scope | Main Operations |
|---|---|
| ACompleteLU<br>Factored form of a generic dense square matrix | No extra operations. Reserved for deriving factored forms of exact matrices. |
| AUCompleteLU<br>Factored form of a uniform dense square matrix composed of floating-point numbers | ▪ *Update* complete LU decomposition of H-matrix AUMatrixSq<br>▪ *Create&Assign* solution of a single-RHS linear algebraic system with an option of matrix transposition (left and right *Create&Assign* multiplications by H-vector AUVector)<br>▪ *Update* solution of a single-RHS linear real algebraic system with an option of matrix transposition (*Update* left and right multiplications by H-vector AUVector)<br>▪ *Create&Assign* solution of a multiple-RHS linear algebraic system with an option of matrix transposition (*Create&Assign* left and right multiplications by H-matrix AUMatrix)<br>▪ *Update* solution of a multiple-RHS linear real algebraic system with an option of matrix transposition (*Update* left and right multiplications by H-matrix AUMatrix) |
| AUCompleteLUGen<br>Factored form of a general uniform dense square matrix in the full storage format | No extra operations. The class implements Crout's LU-decomposition method with partial pivoting |

| Class Name and Abstraction Scope | Main Operations |
|---|---|
| AUCompleteLUHerm<br>Factored form of a Hermitian uniform dense square matrix in the packed or full storage format | No extra operations. The class implements Cholesky's $U^TU$-decomposition for positive definite matrices, and LU-decomposition with partial pivoting for indefinite ones |

Note that all of LU-factorization methods are realized in the *Update* versions only, i.e. they store factored matrix on the place of the original one. Therefore, the original H-matrix appears to be overwritten during decomposition. Mixed-type *Update* operation of solving linear system with a complex matrix and a real RHS generates a run-time error at the stage of assigning complex solution to RHS.

Similarly to AUVector and AUMatrix concrete classes derived from AUCompleteLU are specified by binary representations of the internal floating-point data:

**Table 2.4-2. Concrete Classes Derived from ACompleteLU**

| Class Name | Internal Representation of Floating-Point Data |
|---|---|
| CUCompleteLUGenReal4<br>CUCompleteLUGenCompl4<br>CUCompleteLUHermReal4<br>CUCompleteLUHermCompl4 | 32-bit IEEE floating-point data |
| CUCompleteLUGenReal8<br>CUCompleteLUGenCompl8<br>CUCompleteLUHermReal8<br>CUCompleteLUHermCompl8 | 64-bit IEEE floating-point data |
| CUCompleteLUGenRealX<br>CUCompleteLUGenComplX<br>CUCompleteLUHermRealX<br>CUCompleteLUHermComplX | Extended floating-point numbers CFRealX or CFComplexX with a common precision specifier |

## 2.5. Hierarchy AHessenberg

ExLAF77 supports solution of linear eigenvalue problems $\mathbf{A}\cdot\mathbf{X} = \mathbf{X}\cdot\Lambda$ for dense square H-matrices AUMatrixSq. Its current version includes only algorithms for simultaneous computing all the eigenvalues $\Lambda$ and eigenvectors $\mathbf{X}$ stored as H-vector AUVector and H-matrix AUMatrix respectively. Depending on particular kind of the matrix $\mathbf{A}$ one of two following numerical procedures is used:

- Transformation of H-matrix AUMatrixSqGen to Hessenberg form by elementary stable non-orthogonal transformations and LR-algorithm for computing eigenvalues and eigenvectors of the Hessenberg matrix;
- Householder's transformation of H-matrix AUMatrixSqHerm to tridiagonal form and QL-algorithm for computing eigenvalues and eigenvectors of the tridiagonal matrix.

Hierarchy with the base abstract AHessenberg (see Chart 2.5-1 below) holds intermediate data structures composed of Hessenberg or tridiagonal matrix, triangular transformation matrix, and permutation vector. Its abstract classes introduce the following operations:

**Table 5.2-1. Distribution of Operations over Hierarchy AHessenberg**

| Class Name and Abstraction Scope | Main Operations |
|---|---|
| AHessenberg<br>Hessenberg form of a generic dense square matrix | No extra operations. Reserved for deriving Hessenberg forms of exact matrices. |
| AUHessenberg<br>Hessenberg form of a uniform dense square matrix composed of floating-point numbers | ▪ *Update* transformation of H-matrix AUMatrixSq to Hessenberg/tridiagonal form<br>▪ *Update* solution of a linear eigenvalue problem for a given Hessenberg/tridiagonal matrix form |
| AUHessenbergGen<br>Hessenberg form of a general uniform dense square matrix in the full storage format | No extra operations. The class implements elementary stable non-orthogonal transformations and LR-algorithm |
| AUHessenbergHerm<br>Tridiagonal form of a Hermitian uniform dense square matrix in the packed storage format | No extra operations. The class implements Householder's transformation and QL-algorithm |

Transformation procedures have only in *Update* versions since Hessenberg matrix form always overwrites the original H-matrix matrix. When solving an eigenvalue problem the output eigenvector matrix overwrites input Hessenberg form as well.

Like descendants of AUVector, AUMatrix, and AUCompleteLU concrete classes derived from AUHessenberg are specified in accordance with binary representations of the internal floating-point data:

**Table 2.5-2. Concrete Classes Derived from AHessenberg**

| Class Name | Internal Representation of Floating-Point Data |
|---|---|
| CUHessenbergGenReal4<br>CUHessenbergGenCompl4<br>CUHessenbergHermReal4<br>CUHessenbergHermCompl4 | 32-bit IEEE floating-point data |
| CUHessenbergGenReal8<br>CUHessenbergGenCompl8<br>CUHessenbergHermReal8<br>CUHessenbergHermCompl8 | 64-bit IEEE floating-point data |
| CUHessenbergGenRealX<br>CUHessenbergGenComplX<br>CUHessenbergHermRealX<br>CUHessenbergHermComplX | Extended floating-point numbers CFRealX or CFComplexX with a common precision specifier |

Descendants of AUHessenberg do not have independent significance in the current ExLAF77 configuration. They play part of “hidden” intermediate objects used only in the context of two-step procedure of solving linear eigenvalue problems. Actual purpose of splitting that procedure in two stages and introducing hierarchy AUHessenbers is keeping invariable interfaces when further extending functionality of ExLAF77.

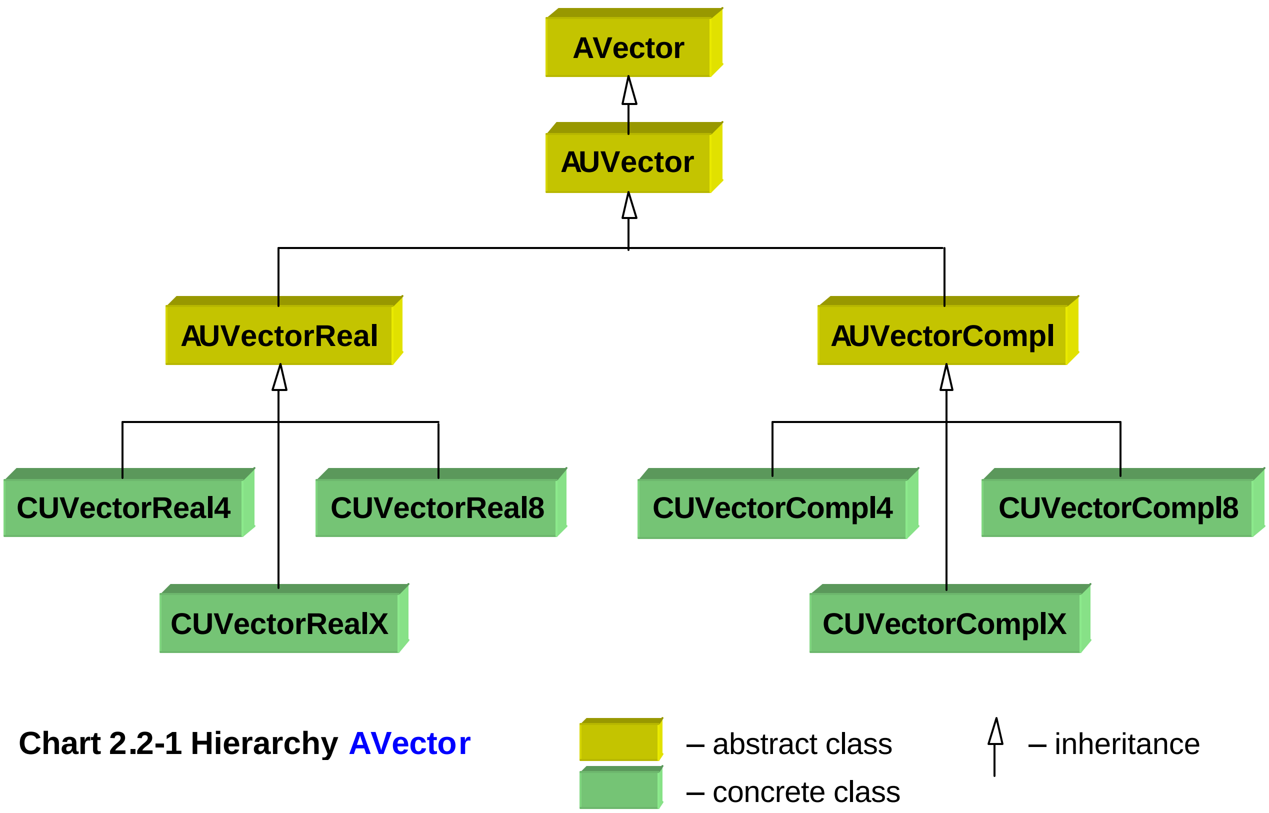



**Chart 2.2-1 Hierarchy AVector**

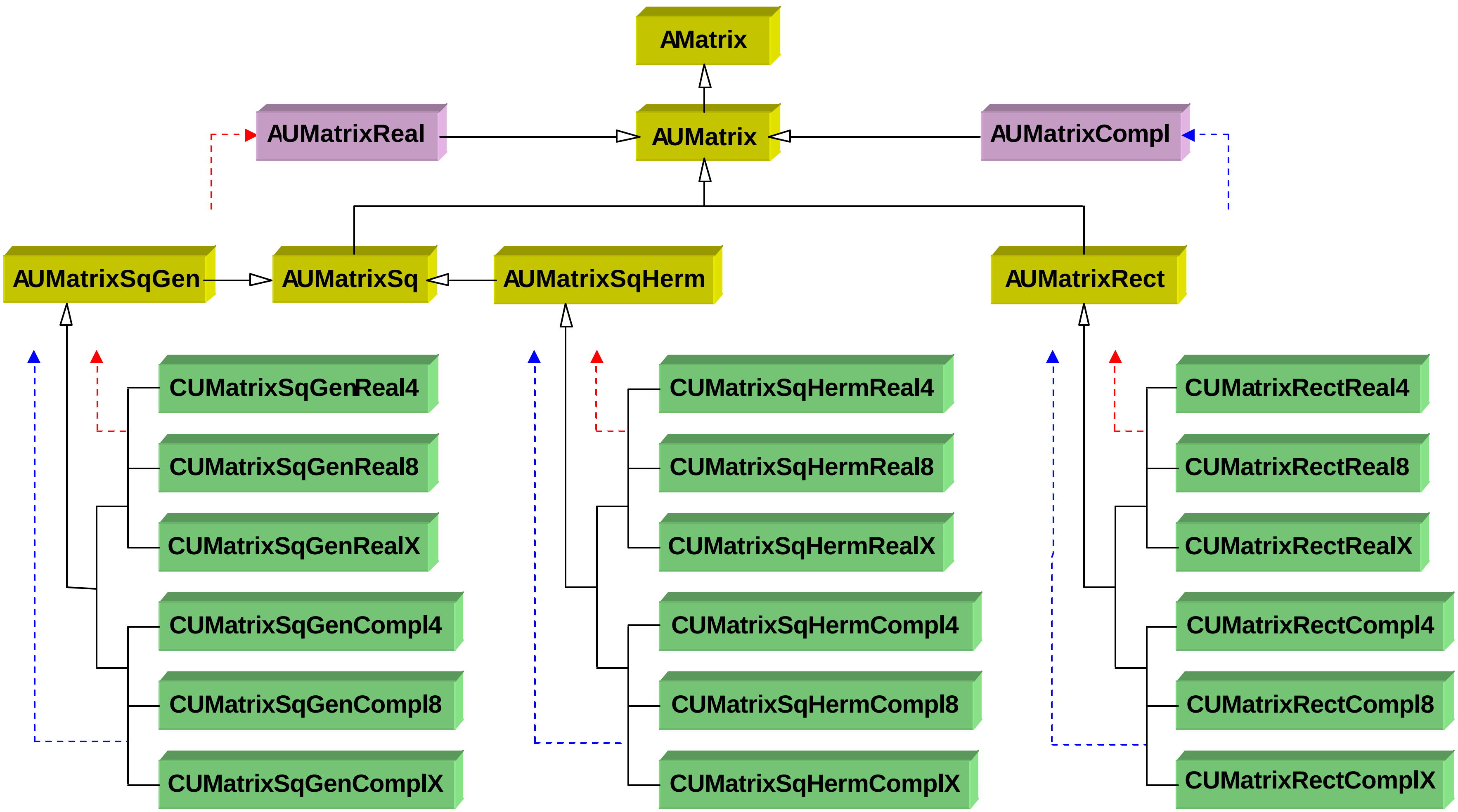
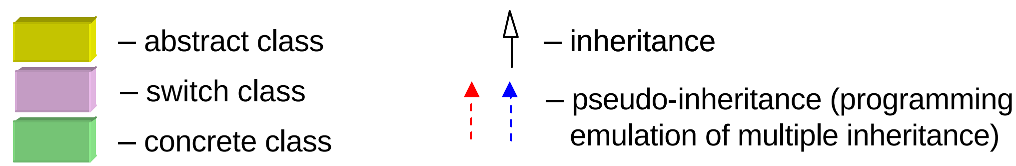



**Chart 2.3-1 Hierarchy AMatrix**



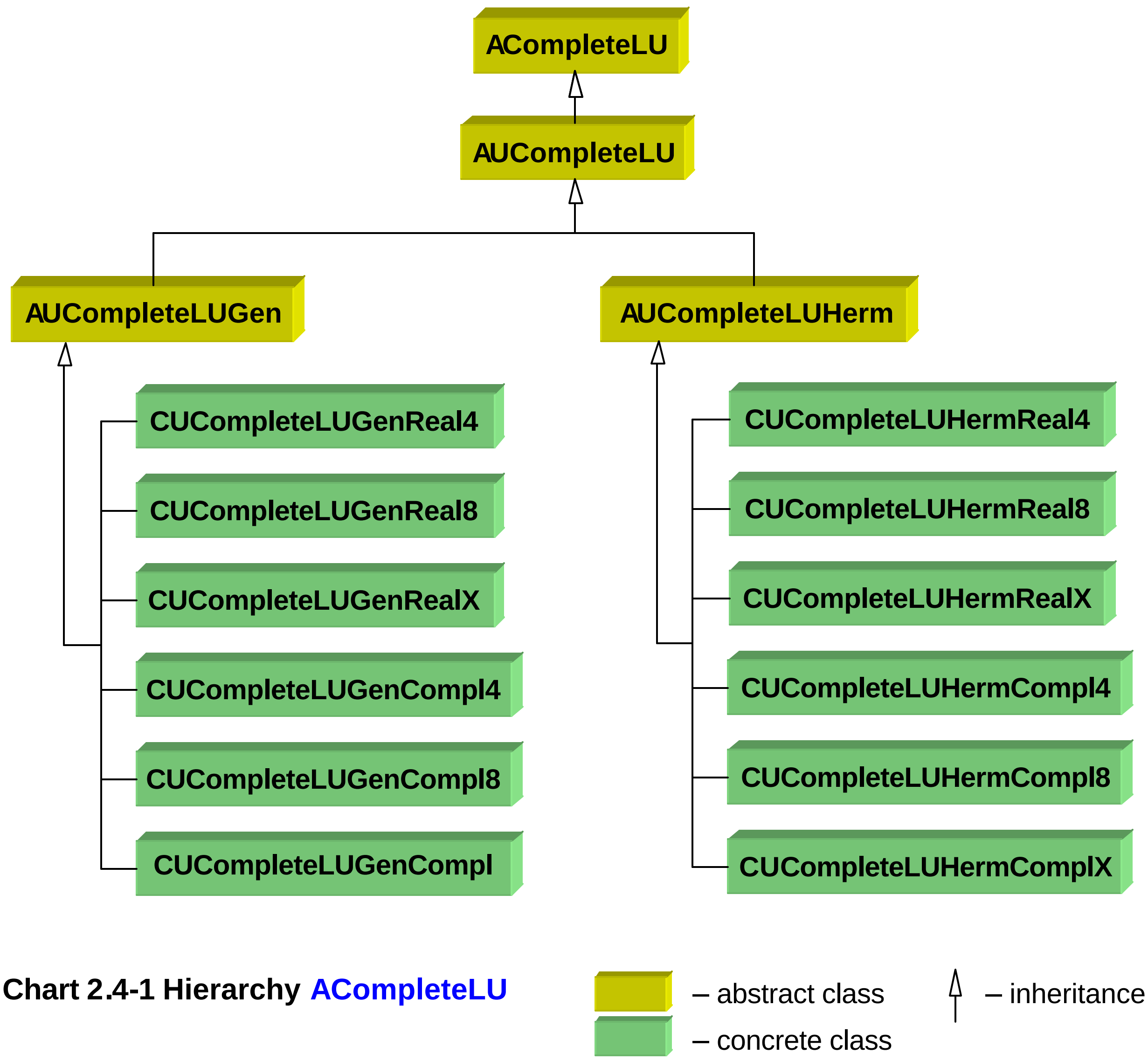



**Chart 2.4-1 Hierarchy ACompleteLU**

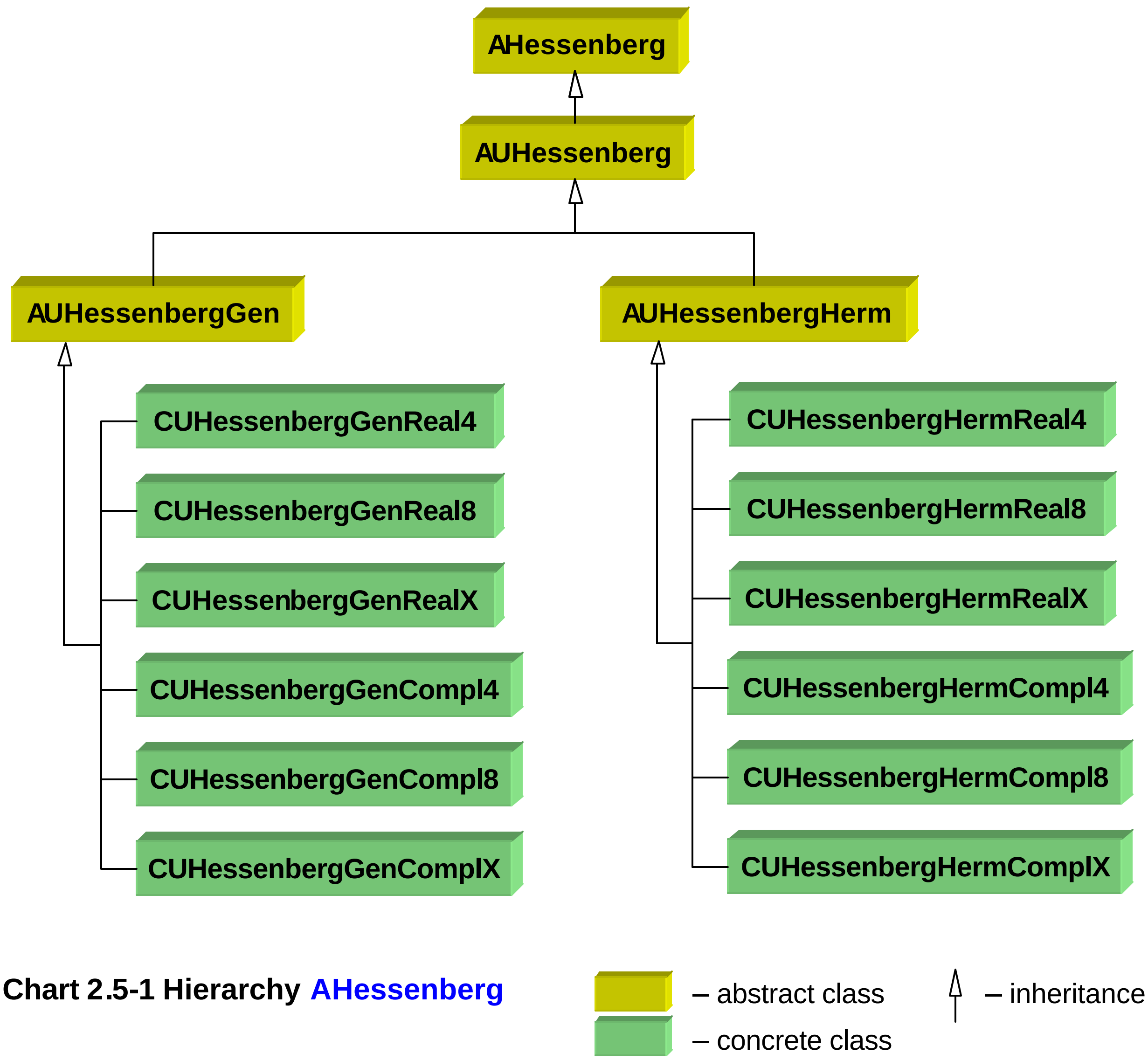



**Chart 2.5-1 Hierarchy AHessenberg**

## 2.6. Logical Class Indicators

As it was mentioned above, ExLAF77 *Create&Assign* operations automatically select the type of output H-object to provide a proper representation for the result. This feature of *Update* operations eliminates the necessity of explicit type declarations for intermediate and final data, and allows manipulating H-objects of unknown types via their abstract handles. However, in some circumstances run-time verification of class membership of an H-object is required to avoid incompatibilities of a subsequent operation with types of its operands.

Let us consider a simple example. Suppose a user's routine checks inequality Ö**x** > **y**, where **x**, **y** are H-numbers AFReal, and the square root is being computed by invoking *Create&Assign* subroutine **HASQRT**. If **x** is negative then output handle of the **HASQRT** will be associated with a new H-number AFComplexFloat representing principal value of the complex square root Ö**x**. Since comparison operations > and < are not defined for complex operands, subsequent calling subroutine **HLGNN** (.GT.) will result in run-time error #0303 "COMPARE COMPLEX NUMBERS", see Appendix A. Hence, the user's code should check whether the output H-number of **HASQRT** is real or complex before invoking **HLGNN.**

Necessity of run-time verifying some attribute of an H-object arises in many cases. It is particularly useful when executing mixed-type assignment and *Update* binary operations with a real numerical, vector, or matrix left operand. If the right operand appears to be complex then ExLAF77 generates run-time errors #0301 "ASSIGN COMPLEX TO REAL" and #404 "UPDATE OPERATION FAILURE" respectively. So, the user's code has to be responsible for checking types of the operands and appropriate processing incompatibilities.

To support retrieving general attributes of H-objects referenced by abstract handles ExLAF77 implements a set of logical indicators. For simplicity, in this manual they are called LISFIN, LISREAL, LISFLT, LISNUM, LISINT, LISVECT, LISMATR, LISSQR, LISHERM, LISCLU, and LISHES. Value of each indicator for a particular H-object can be inquired via calling respective interface subroutine. Table 2.6-1 below explains the meaning of indicators and their definition for different classes.

**Table 2.6-1. Definition of the Logical Indicators**

| Indicator Name and Meaning | Interface Subroutine Name | Definition |
|---|---|---|
| LISFIN - is H-object a finite number or composition of finite numbers? | **HLFIN** | =.FALSE. for CInfSigned and CInfUnsigned<br>=.TRUE. for all other classes of H-objects |
| LISREAL - is H-object a real number or composition of real numbers? | **HLREAL** | =.TRUE. for (pseudo)descendants of AReal, AUVectorReal, AUMatrixReal, and classes CUCompleteLUReal4,8,X, CUHessenbergReal4,8,X<br>=.FALSE. for all other classes of H-objects |
| LISFLT - is H-object inexact, i.e. floating-point number or composition of floating-point numbers? | **HLFLT** | =.TRUE. for (pseudo)descendants of AFFloat, AUVector, AUMatrix, AUCompleteLU, and AUHessenberg<br>=.FALSE. for descendants of AFRealExact and classes CInfSigned, CInfUnsigned |
| LISNUM - is H-object a number? | **HLNUM** | =.TRUE. for descendants of ANumber<br>=.FALSE. for all other classes of H-objects |

| Indicator Name and Meaning | Interface Subroutine Name | Definition |
|---|---|---|
| LISINT - is H-object an integer number? | **HLINT** | =.TRUE. for descendants of AFInteger<br>=.FALSE. for all other classes of H-objects |
| LISVECT - is H-object a vector? | **HLVECT** | =.TRUE. for descendants of AVector<br>=.FALSE. for all other classes of H-objects |
| LISMATR - is H-object a matrix? | **HLMATR** | =.TRUE. for descendants of AMatrix<br>=.FALSE. for all other classes of H-objects |
| LISSQR - is H-object a (transformed) square matrix? | **HLMSQR** | =.TRUE. for descendants of AUMatrixSq, AUCompleteLU, and AUHessenberg<br>=.FALSE. for all other classes of H-objects |
| LISHERM - is H-object a (transformed) Hermitian matrix? | **HLHERM** | =.TRUE. for descendants of AUMatrixSqHerm, AUCompleteLUHerm, and AUHessenbergHerm<br>=.FALSE. for all other classes of H-objects |
| LISCLU - is H-object complete LU-decomposition of a square matrix? | **HLCLU** | =.TRUE. for descendants of ACompleteLU<br>=.FALSE. for all other classes of H-objects |
| LISHES - is H-object Hessenberg form of a square matrix? | **HLHES** | =.TRUE. for descendants of AHessenberg<br>=.FALSE. for all other classes of H-objects |

Subroutines returning values of the listed class indicators are described in section 4.6.

# Section 3. Executing Operations

## 3.1. Working Session

Use of ExLAF77 requires performing some auxiliary procedures on starting and finishing the work. To start computations ExLAF77 has to open its log file (see section 3.2) and initialize memory allocation protocol (see section 3.3). No one operation can be executed properly without initializing the system. On finishing the work ExLAF77 has to close log file and remove allocation protocol from computer memory. In addition, to avoid memory leaks it has to remove all created H-objects on finishing the computations. Thus, the user's code should explicitly open and close interaction with ExLAF77 system subroutines **HSINIT** and **HSEXIT** described in section 4.3 support those procedures.

ExLAF77 executes the math operations during its working session, i.e. in the period between calling **HSINIT** and **HSEXIT**. Since it removes all the H-objects created during computations, the user's application has to output results and/or save required data using I/O and export subroutines:

- Write H-objects to unformatted file(s) by calling the binary output subroutine **HWRITE** described in section 4.19.
- Convert H-objects or their parts to text strings by calling the text output subroutines **HGTNX0**, **HGTEV0**, **HGTEM0** (unformatted output), **HGTNX**, **HGTEV**, **HGTEM** (formatted output) described in section 4.20.
- Convert H-objects or their parts to Fortran variables or arrays by calling the export subroutines **HEFNX**, **HEFEV**, **HEFV**, **HEFEM**, **HEFMR**, **HEFMC**, **HEFM** described in section 4.21.

ExLAF77 working session can be repeatedly opened and closed as many times as necessary during program run.

## 3.2. Errors Handling

ExLAF77 detects a number of run-time errors. Each error is associated with a unique numerical code and text error message. When discovering an error ExLAF77 output respective numerical code and text message to its text log file opened by system subroutine **HSINIT** on starting working session (see 3.1). Run-time errors are divided into the following categories:

**Resource Errors** that can arise due to insufficiency of hardware resources. Example: error #0001 "HEAP MEMORY ALLOCATION FAILURE".

**Interface Errors** are generated on detecting invalid values of input parameters. Example: error #0101 "INVALID OBJECT HANDLE".

**Floating Point Errors** are generated on discovering abnormal results of floating-point arithmetical operations. Example: error #0201 "FLOATING POINT UNDERFLOW".

**Illegal Operations** errors arise in response to attempts of performing algorithmically forbidden operations. Example: error #0301 "ASSIGN COMPLEX TO REAL".

**Calculus Errors** mean that the passed values of operands cannot be properly processed due to algorithmic or other restrictions. Example: error #0401 "TOO BIG ABS VALUE OF ARGUMENT".

**Matrix Operation Errors** are generated on detecting uncoordinated dimensions or other attributes of vector and matrix operands. Example: error #501 "`OPERANDS' DIMENSIONS MISMATCH`".

**Undefined Result** errors stand for mathematical uncertainty of results of operations. Example: #601 "`DIVIDE ZERO BY ZERO`".

**Programming Bugs** signify internal ExLAF77 errors that should be reported to QNT Software Development Inc.

Numerical codes and text messages of run-time errors are listed in Appendix A.

Note that ExLAF77 always treats undefined results and floating-point overflows as errors. Processing floating-point underflows depends on setting an internal underflow control flag. If underflow control is turned on then underflows are processed like all other run-time errors, otherwise respective denormalized values are set to zero without generating errors. However, regardless of current processing mode ExLAF77 internal representations of floating-point numbers always remain valid, i.e. they never contain "pathological" bit patterns such as denormalized values, ±INF, and `NaN`,. One can switch underflow control flag in run time by calling system subroutine **`HSUNDF`** described in section 4.4.

To make it possible for calling application to process erroneous events in run-time, ExLAF77 interface subroutines are supplied with an extra alternate return parameter that is always the last one in the argument list. When an error occurs during operation interface subroutine appends associated numerical code and text message to the log file and executes the alternate return statement `RETURN 1`. Therefore, the calling program can recognize the error by its numerical code and properly process it in run-time. To retrieve error codes one should use system subroutine **`HSERR`** described in section 4.4.

A variable-precision application can safely recover many typical computational anomalies arising due to accumulation of round-off errors, such as underflow, overflow, algorithmic matrix singularity, etc. The alternate return mechanism allows developing self-adjustable codes that automatically perform all the necessary recovering actions. However, if the user's code intensively uses alternate return and run-time error recovering it should be capable to suppress appending respective text messages to the log-file. Without that capability size of the log-file would progressively increase during program run due to multiple useless error messages.

ExLAF77 has a built-in tool for selective "masking" specified run-time errors. A masked error results in alternate return like any other one, but does not produce text output. On starting working session all errors are unmasked, i.e. every alternate return is accompanied with appending a corresponding text message to the log file. With using system subroutines **`HSEMSK`**, **`HSDMSK`** and **`HSMSKA`** (see section 4.4) the calling program can create a list of masked errors, dynamically modify it, and switch modes of error masking.

## 3.3. *Create&Assign* Operations and Memory Management

*Create&Assign* operations introduced in section 1.4 above provide one of the most important features of ExLAF77, namely, abstraction mechanism. When invoking such an operation the user has not to know type of the result since the operation selects it automatically. Resulting H-objects referenced by their abstract handles can be passed as operands to subsequent operations, and so on.

Thus, ExLAF77 makes it possible to develop generalized computational procedures that manipulate abstract handles and do not include explicit type declarations. In order to ensure compatibility of intermediate operations with types of operands sometimes it is necessary to check general properties of H-objects. However, it can be easily done via retrieving logical class indicators without exact specifying the types (see section 2.6).

Since intensive use of *Create&Assign* operations typically results in fast accumulating multiple H-objects in computer memory, development of generalized algorithms requires effective tools of memory release. The same tools appear to be very useful when developing self-adjustable computational procedures that incrementally increase working precision until required accuracy is reached.

ExLAF77 has a built-in memory manager based on tracking allocation and deallocation events. It stores information on created H-objects in a buffered dynamically extendable list called "**memory allocation protocol**". Each element of the protocol called "**allocation node**" corresponds to a single H-object located in system heap memory. Creating and deleting H-objects are accompanied with appropriate modifying the protocol. The simplest memory managing operations: deleting a single H-object and deleting all H-objects can be performed by calling system subroutines **HSDOBJ** and **HSDALL** described in section 4.5.

The memory allocation protocol can include void nodes called "**allocation marks**" or just **"marks**" that do not correspond to existing H-objects. Allocation marks serve as pointers to particular locations within the protocol intended for designating groups of subsequently created H-objects. Like regular H-objects, the marks are referenced by their unique handles. Hence, they can be treated just as empty H-objects. System subroutines **HSEMRK** and **HSDMRK** perform setting and removing allocation marks.

Manipulating marks allows single-call removing designated groups of H-objects from memory. Consider a fragment of computational procedure with intensive use of *Create&Assign* operations. When program running those operations create multiple temporary H-objects that should be deleted on exiting the fragment. In order to release memory allocated for temporary objects it is enough to set allocation marks immediately before and after the fragment, and remove all the objects by calling subroutine **HSDGRP** described in section 4.5.

After calling subroutines **HSDOBJ**, **HSDALL**, **HSDMRK**, or **HSDGRP** handles to the removed H-objects and allocation marks become invalid, i.e. they cannot be used as input parameters of ExLAF77 subroutines until they are associated with other H-objects or marks. Any attempt of using handle to deleted object as an input parameter results in run-time error #0101 "INVALID OBJECT HANDLE".

## 3.4. Creation and Initialization of H-Objects

### 3.4.1. Ways of Initialization

Ways of creating and initializing H-objects are closely connected with applicability of *Update* operations. It is convenient to consider separately two main groups of H-objects with different values of the logical indicator LISFLT (see section 2.6):

LISFLT=.TRUE. Floating-point numbers and H-objects composed of them: (pseudo)descendants of AFFloat, AUVector, AUMatrix, AUCompleteLU, and AUHessenberg.

LISFLT=.FALSE. Classes CInfSigned, CInfUnsigned and exact numbers - descendants of AFRealExact.

In general, H-objects of the first group allow modifying their values without change of type, i.e. they can appear as left operands of *Update* operations. Since values of those H-objects can be repeatedly updated during program run, there is no mandatory necessity to initialize them at the stage of creating. In many cases, it is more preferable to create "empty" variables of appropriate types and use them in further computations like regular Fortran variables.

ExLAF77 includes four subroutines for creating empty floating-point H-numbers AFFloat, H-vectors AUVector, general H-matrices AUMatrixSqGen and AUMatrixCompl, and Hermitian H-matrices AUMatrixSqHerm: **HMN**, **HMV**, **HMM**, and **HMMS** respectively (see section 4.7).

Elements of H-objects created with those subroutines are set to zero. Other four subroutines, **HANF**, **HAVF**, **HAMF**, and **HAMSF** described in section 4.8.2 create H-objects of the same kinds and initialize them with Fortran variables and arrays. Subroutine **HANXT** creates real and complex H-numbers AFFloat initialized with text strings (see sections 3.4.2 and 4.8.1).

In contrast, H-objects of the second group typically change their sizes or/and types during arithmetical operations. Implementation of *Update* operations for them would be unnatural since it results in encumbering the code with run-time type verifications, memory reallocations, and processing integer overflows. That is why ExLAF77 supports only *Create&Assign* operations for H-objects of the second group.

Thus, exact and infinite numbers must always be initialized at the stage of creating. They can participate in further *Create&Assign* and *Update* operations as *right* operands, but cannot change their values. Currently ExLAF77 allows creating exact and infinite numbers with initialization with text strings and floating-point H-numbers. To perform those operations one should use subroutines **HANXT** and **XAXN** described in sections 4.8.1 and 4.8.3 respectively.

## 3.4.2. Formats of Initializing Text Strings

Described in section 4.8.1 subroutine **HANXT** that create H-object ANumber and initialize it with input text string, automatically selects type of new object according to the string format. This section describes permissible formats of text representations of numbers and rules of determining their types.

Text representation of any number cannot contain intermediate blanks. Hence, the input string can contain only leading and trailing blanks. The following formal rules pre-determine type of the created H-number:

**1.** If the string is either **'INF'** or **'inf'** then H-number CInfUnsigned is created.

**2.** If the string is either **'+INF'**, **'+inf'**, **'-INF'**, or **'-inf'** then H-number CInfSigned is created.

**3.** If the string contains character **'/'** then H-number AFRealExact is created. In this case the initializing string should have one of the following two formats:

<numerator>**/**<denominator>
<sign><numerator>**/**<denominator>

where <sign> = {**'+'**|**'-'**}
<numerator> = <digit><digit>...<digit>
<denominator> = <digit><digit>...<digit>
<digit> = {**'0'**|**'1'**|**'2'**|**'3'**|**'4'**|**'5'**|**'6'**|**'7'**|**'8'**|**'9'**}

The substrings <numerator> and <denominator> cannot be empty, and <denominator> should contain at least one character different from **'0'**. The type and numerical value of created object is determined as result of respective division.

**4.** If the string contains character **','** (comma) then H-number AFComplexFloat is created. In this case the initializing string should have the following format:

**'('**<real part>**','**<imaginary part>**')'**

where both <real part> and <imaginary part> substrings may have any form permitted for text representation of real floating-point and integral numbers (see points **5** and **6** below). **HANXT** selects precision of new complex floating-point H-number in accordance with maximum number of significant digits in <real part> and <imaginary part>.

**5.** If the string contains neither substrings **'INF'**, **'inf'** nor characters **'/'**, **','**, but it includes character **'.'** (point) then H-number AFRealFloat is created; In this case the initializing string should have one of the following six formats:

<mantissa>
<sign><mantissa>
<mantissa><exponent prefix><exponent>
<mantissa><exponent prefix><sign><exponent>
<sign><mantissa><exponent prefix><exponent>
<sign><mantissa><exponent prefix><sign><exponent>

where <exponent prefix> = {**'E'**|**'e'**}
<mantissa> = <digit or point><digit or point >...<digit or point >
<exponent> = <digit><digit>...<digit>
<digit or point > = {**'0'**|**'1'**|**'2'**|**'3'**|**'4'**|**'5'**|**'6'**|**'7'**|**'8'**|**'9'**|**'.'**}

The substrings <mantissa> and <exponent> cannot be empty and <mantissa> can contain no more than one character **'.'** (point). **HANXT** selects precision of new floating-point H-number in accordance with number of significant digits in <mantissa>.

**6.** If the string contains none of substrings **'INF'**, **'inf'** and characters **'/'**, **','**, **'.'** then H-number AFInteger is created. In this case, the initializing string should have one of the following two forms:

<number>
<sign><number>

where <number> = <digit><digit>...<digit>. Substring <number> cannot be empty. **HANXT** creates H-number CFInteger4 or CFIntegerX depending on value of the integer number.

**Examples:**

| | |
|---|---|
| 'inf' | CInfUnsigned =INF |
| '-INF' | negative CInfSigned =-INF |
| '137' | CFInteger4 =137 |
| '-999999999999999/3' | CFIntegerX =-333333333333333 |
| '42/12' | CFRational =7/2 |
| '137.e-8' | CFReal4 =1.37×10^-6 |
| '3.14159265358979323846264 33' | CFReaX =3.14159265358979323846264 33 |
| '(1.0,999999999999999)' | CFComplex8 =1+**i**×9.99999999999999×10^14 |

Note that subroutines **NUNT**, **HUEVT**, and **HUEMT** performing update of floating-point H-numbers and selected elements of H-vectors and H-matrices with text strings (see section 4.9.1) accept string formats **3**, **4**, **5**, and **6** above

## 3.5. Output to Text Strings

ExLAF77 interface subroutines described in section 4.20 support formatted and unformatted output of H-numbers and selected elements of H-vectors and H-matrices to text strings. Subroutines **HETNX**, **HETEV**, and **HETEM** adjust format of output string in accordance with user-defined parameters, while subroutines **HETNX0**, **HETEV0**, and **HETEM0** provide text output with an automatic format selection.

Output text representations of H-numbers have generally the same formats as input strings of the subroutines **HANXT**, **HUNT**, **HUEVT**, and **HUEMT** (see section 3.4 above). Therefore, strings generated by subroutines **HETNX**, **HETNX0**, **HETEV**, **HETEV0**, **HETEM**, and **HETEM0** can be used for approximate reproducing respective H-numbers, H-vectors, and H-matrices with **HANXT**, **HUNT**, **HUEVT**, and **HUEMT**.

Unformatted output mode implies left text alignment, i.e. non-blank characters start from the beginning of text string, while unused right part of the string is padded with blanks. Automatic selecting sizes of the mantissa and exponent fields of the floating-point H-numbers is performed in such a way that guarantees output of all significant decimal digits encoded in their binary representations. If the text string is not long enough to hold the number, then the string is padded with asterisks.

Formatted output of real floating-point H-numbers and elements or real H-vectors and H-matrices uses the very last of six permissible formats described in section 3.4:

<sign><mantissa><exponent prefix><sign><exponent>

where positions and structures of the <mantissa> and <exponent> fields are specified by four user-defined integer parameters `IW`, `IP`, `IM`, and `IE`. The first one specifies full width of the output field that starts from the beginning of text string. Parameter `IP` defines position of decimal point within the <mantissa> field, or in other words, scaling factor for mantissa. Last two parameters `IM` and `IE` specify numbers of decimal digits in the <mantissa> and <exponent> fields. Thus, output text representation of a real floating-point H-number looks as follows:

```
□□□□□□□□□□□□±MMMMMMMMM.MMMMMMMMMMMMMMMMME±EEEEEEEEE
            ←———————————IM+1———————————→  ←——IE——→
←—————————————————————————IW—————————————————————————→
```

where characters □, *M*, and *E* denote blanks, decimal digits of mantissa and exponent respectively. One can see that the full width `IW` of output field should be equal to or greater than `IM`+`IE`+4 to hold all decimal digits and four auxiliary characters. If this condition is not satisfied, or `IW` exceeds total length of the string, the output field is padded with asterisks.

In order to clarify the meaning of scaling parameter `IP`, compare output text representations of the number $\pi$ =3.14159265358979323846264433... with `IW`=20, `IM`=10, `IE`=2 and different `IP`:

```
IP=-10:  '********************'
IP= -9:  '********************'
IP= -8:  '    +0.000000003E+09'
IP= -7:  '    +0.000000031E+08'
IP= -6:  '    +0.000000314E+07'
IP= -5:  '    +0.000003142E+06'
IP= -4:  '    +0.000031416E+05'
IP= -3:  '    +0.000314159E+04'
IP= -2:  '    +0.003141593E+03'
```

```
IP= -1:  '    +0.031415927E+02'
IP=  0:  '    +0.314159265E+01'
IP=  1:  '    +3.141592654E+00'
IP=  2:  '    +31.41592654E-01'
IP=  3:  '    +314.1592654E-02'
IP=  4:  '    +3141.592654E-03'
IP=  5:  '    +31415.92654E-04'
IP=  6:  '    +314159.2654E-05'
IP=  7:  '    +3141592.654E-06'
IP=  8:  '    +31415926.54E-07'
IP=  9:  '    +314159265.4E-08'
IP= 10:  '    +3141592654.E-09'
IP= 11:  '********************'
IP= 12:  '********************'
```

The same four integer parameters control output format for complex floating-point H-numbers and elements of complex H-vectors and H-matrices. Text representation of a complex floating-point number has the following form

'(' <real part> ',' <imaginary part> ')'

where both <real part> and <imaginary part> components are formatted like real floating-point H-numbers (see above) with the same values of parameters IW, IP, IM, and IE, and without leading blanks. Therefore, full width IW of the output field should be equal to or greater than 2*(IM+IE)+11 to hold all decimal digits of the real and imaginary parts and eleven auxiliary characters. If this condition is not satisfied, or IW exceeds total length of the string, the output field is padded with asterisks.

Subroutine **HETNX** provides a uniform interface for formatted text output of generic H-numbers referenced by abstract handles ANumber. The output field starts from the beginning of text string and has full width IW for any particular kind of number. **HETNX** keeps right alignment of non-blank characters, and pad unused left part of the output field with blanks.

In contrast to text output of floating-point H-numbers, parameters IM and IE are of no importance for text representations of infinite and exact H-numbers CInfSigned, CInfUnsigned, AFRealEact. As to parameter IP, it is not significant for representations of infinite and integer H-numbers CInfSigned, CInfUnsigned, AFInteger. However, when dealing with text output of rational H-numbers CFRational positive IP specifies position of the slash ('/') separating the <numerator> and <denominator> fields, while zero IP sets the standard right alignment mode. Compare output text representations of the rational number -130321/279841 with IW=15, and different IP.

```
IP= -1:  '***************'
IP=  0:  ' -130321/279841'
IP=  1:  '/279841        '
IP=  2:  '*/279841       '
IP=  3:  '**/279841      '
IP=  4:  '***/279841     '
IP=  5:  '****/279841    '
IP=  6:  '*****/279841   '
IP=  7:  '******/279841  '
IP=  8:  '-130321/279841 '
IP=  9:  ' -130321/279841'
```

```
IP= 10:  '  -130321/*****'
IP= 11:  '   -130321/****'
IP= 12:  '    -130321/***'
IP= 13:  '     -130321/**'
IP= 14:  '      -130321/*'
IP= 15:  '       -130321/'
IP= 16:  '***************'
```

## 3.6. Unformatted Binary I/O

Subroutines **HREAD** and **HWRITE** described in section 4.19 support unformatted I/O operation with user-defined binary files. Design of their interfaces allows communicating with binary files of arbitrary structures and mixing H-objects with any other data in one file.

OPEN and CLOSE statements are to be executed by the calling program that is solely responsible for appropriate definition of the file attributes. Calling statements for the subroutines **HREAD** and **HWRITE** have the following form:

CALL **HREAD** ( RCBACK, NSIZE, ILH, *ERROR ) and
CALL **HWRITE**( WCBACK, ILH, *ERROR )

where ILH (INTEGER) is a handle to H-object, NSIZE (INTEGER) is the size of that H-object expressed in 32-bit words, ERROR is a label for alternate return, see section 4.19. Finally, RCBACK and WCBACK are symbolic names of user-supported callback subroutines that execute respective READ and WRITE operations depending on specific properties of the binary file. Symbolic names RCBACK and WCBACK must appear in an EXTERNAL statement in the calling program.

Calling statements used for invoking callback subroutines from **HREAD** and **HWRITE** are equivalent to the following ones:

CALL RCBACK( NSIZE, IARRAY ) and
CALL WCBACK( NSIZE, IARRAY )

where IARRAY is an adjustable INTEGER array that serve as container for the transferred H-object, and NSIZE (INTEGER) is the size of that H-object expressed in 32-bit words.

Before calling **HREAD** the user's code must retrieve a correct value of NSIZE to make it possible to allocate sufficient amount of memory for the H-object to be read. Probably, writing size of H-object to the immediately preceding record is the best way of saving and restoring NSIZE when performing binary I/O. The following are simplest examples of the callback subroutines:

```
SUBROUTINE RCB(NSIZE, IARRAY)
DIMENSION IARRAY(NSIZE)
READ(10) (IARRAY(I),I=1,NSIZE)
RETURN
END

SUBROUTINE WCB(NSIZE, IARRAY)
DIMENSION IARRAY(NSIZE)
WRITE(10) NSIZE
WRITE(10) (IARRAY(I),I=1,NSIZE)
```

```
RETURN
END
```

A fragment of the user's code that performs binary I/O using **HREAD**, **HWRITE**, RCB, and WCB should look as follows:

```
EXTERNAL RCB, WCB
INTEGER NSIZE, ILH
..................................
OPEN(10,...)
..................................
CALL HWRITE(WCB, ILH, *100)
..................................
CLOSE(10)
..................................
..................................
OPEN(10,...)
..................................
READ(10) NSIZE
CALL HREAD(RCB, NSIZE, ILH, *200)
..................................
CLOSE(10)
..................................
```

In more compound programming contexts the callback subroutines can read and write some extra data passed via COMMON blocks, thus allowing user's code to mix H-objects and other entities in one file.

## 3.7. Types of Automatically Created H-objects

Any of ExLAF77 *Create&Assign* arithmetical operations automatically selects the type of resulting H-object that depends on both types and numerical values of the operands. Therefore, type of the result is unpredictable in general case. An *Update* operation is equivalent to combination of the corresponding *Create&Assign* operation, converting its result to a required type, and updating the left hand side operand. This section documents the implemented formal rules of selecting and converting types of H-objects.

### 3.7.1. Operations on Infinities and Divisions by Zero

ExLAF77 permits using infinite H-numbers CInfSigned and CInfUnsigned as operands of unary and binary arithmetical operations. Some arithmetical operations and functions can output infinite resulting values as well. Thus, H-objects CInfSigned and CInfUnsigned play an important part in calculations since they allow using mathematically correct infinite values without generating run-time errors. However, one should keep in mind that manipulation infinite H-numbers is potentially dangerous because of the risk of producing indefinite results. The tables 3.7.1-1, 3.7.1-2, and 3.7.1-3 below list arithmetical operations that can accept infinite operands and/or produce infinite output values.

**Table 3.7.1-1. Unary *Create&Assign* Operations on Infinite H-numbers**

| Operation | Interface Subroutine | Operand | | |
|---|---|---|---|---|
| | | CInfUnsigned | Positive CInfSigned | Negative CInfSigned |
| Unary plus | **HACPYH** | CInfUnsigned | Positive CInfSigned | Negative CInfSigned |
| Unary minus | **HANEGH** | CInfUnsigned | Negative CInfSigned | Positive CInfSigned |
| Complex conjugate | **HACNJH** | CInfUnsigned | Positive CInfSigned | Negative CInfSigned |
| Magnitude | **HAABS** | Positive CInfSigned | Positive CInfSigned | Positive CInfSigned |
| Real part | **HERH** | Run-time error #608 "RE/IM PART OF UNSIGNED INFINITY" | Positive CInfSigned | Negative CInfSigned |
| Imaginary part | **HEIH** | Run-time error #608 "RE/IM PART OF UNSIGNED INFINITY" | Zero CFInteger4 | Zero CFInteger4 |

**Table 3.7.1-2. Binary *Create&Assign* Addition with Infinite Operands (Subroutine HAAHH)**

| First Summand | Second Summand | Result |
|---|---|---|
| CInfUnsigned | CInfUnsigned or CInfSigned | Run-time error #605 "SUBTRACT INFINITY FROM INFINITY" |
| | H-number AFinite | CInfUnsigned |
| Positive CInfSigned | CInfUnsigned or negative CInfSigned | Run-time error #605 "SUBTRACT INFINITY FROM INFINITY" |
| | H-number AFReal or positive CInfUnsigned | Positive CInfSigned |
| | H-number AFComplex | CInfUnsigned |
| Negative CInfSigned | CInfUnsigned or positive CInfSigned | Run-time error #605 "SUBTRACT INFINITY FROM INFINITY" |
| | H-number AFReal or negative CInfSigned | Negative CInfSigned |
| | H-number AFComplex | CInfUnsigned |
| AFReal | CInfUnsigned | CInfUnsigned |
| | Positive CInfSigned | Positive CInfSigned |
| | Negative CInfSigned | Negative CInfSigned |
| AFComplex | CInfUnsigned or CInfSigned | CInfUnsigned |

**Table 3.7.1-3. Binary *Create&Assign* Subtraction with Infinite Operands (Subroutine HASHH)**

| Minuend | Subtrahend | Result |
|---|---|---|
| CInfUnsigned | CInfUnsigned or CInfSigned | Run-time error #605 "SUBTRACT INFINITY FROM INFINITY" |
| | H-number AFinite | CInfUnsigned |
| Positive CInfSigned | CInfUnsigned or positive CInfSigned | Run-time error #605 "SUBTRACT INFINITY FROM INFINITY" |
| | H-number AFReal or negative CInfSigned | Positive CInfSigned |
| | H-number AFComplex | CInfUnsigned |

| Minuend | Subtrahend | Result |
|---|---|---|
| Negative CInfSigned | CInfUnsigned or negative CInfSigned | Run-time error #605 "SUBTRACT INFINITY FROM INFINITY" |
| | H-number AFReal or positive CInfSigned | Negative CInfSigned |
| | H-number AFComplex | CInfUnsigned |
| AFReal | CInfUnsigned | CInfUnsigned |
| | Positive CInfSigned | Negative CInfSigned |
| | Negative CInfSigned | Positive CInfSigned |
| AFComplex | CInfUnsigned or CInfSigned | CInfUnsigned |

**Table 3.7.1-4. Binary *Create&Assign* Multiplication with Infinite Operands (Subroutine `HAMHH`)**

| First Factor | Second Factor | Result |
|---|---|---|
| CInfUnsigned | CInfUnsigned, CInfSigned, or nonzero H-number AFinite | CInfUnsigned |
| | Zero H-number AFinite | Run-time error #603 "MULTIPLY INFINITY BY ZERO" |
| Positive CInfSigned | CInfUnsigned or nonzero H-number AFComplex | CInfUnsigned |
| | Positive H-number AFReal or positive CInfSigned | Positive CInfSigned |
| | Negative H-number AFReal or negative CInfSigned | Negative CInfSigned |
| | Zero H-number AFinite | Run-time error #603 "MULTIPLY INFINITY BY ZERO" |
| Negative CInfSigned | CInfUnsigned or nonzero H-number AFComplex | CInfUnsigned |
| | Positive H-number AFReal or positive CInfSigned | Negative CInfSigned |
| | Negative H-number AFReal or negative CInfSigned | Positive CInfSigned |
| | Zero H-number AFinite | Run-time error #603 "MULTIPLY INFINITY BY ZERO" |
| Positive AFReal | CInfUnsigned | CInfUnsigned |
| | Positive CInfSigned | Positive CInfSigned |
| | Negative CInfSigned | Negative CInfSigned |
| Negative AFReal | CInfUnsigned | CInfUnsigned |
| | Positive CInfSigned | Negative CInfSigned |
| | Negative CInfSigned | Positive CInfSigned |
| Nonzero AFComplex | CInfUnsigned or CInfSigned | CInfUnsigned |
| Zero AFinite | CInfUnsigned or CInfSigned | Run-time error #603 "MULTIPLY INFINITY BY ZERO" |

**Table 3.7.1-5. Binary *Create&Assign* Division with Infinite Operands (Subroutine `HADHH`)**

| Dividend | Divisor | Result |
|---|---|---|
| CInfUnsigned | CInfUnsigned or CInfSigned | Run-time error #602 "DIVIDE INFINITY BY INFINITY" |

| Dividend | Divisor | Result |
|---|---|---|
| | H-number AFinite | CInfUnsigned |
| Positive CInfSigned | CInfUnsigned or CInfSigned | Run-time error #602 "DIVIDE INFINITY BY INFINITY" |
| | Positive H-number AFReal | Positive CInfSigned |
| | Negative H-number AFReal | Negative CInfSigned |
| | H-number AFComplex or zero AFReal | CInfUnsigned |
| Negative CInfSigned | CInfUnsigned or CInfSigned | Run-time error #602 "DIVIDE INFINITY BY INFINITY" |
| | Positive H-number AFReal | Negative CInfSigned |
| | Negative H-number AFReal | Positive CInfSigned |
| | H-number AFComplex or zero AFReal | CInfUnsigned |
| AFRealExact | CInfUnsigned or CInfSigned | Zero CFInteger4 |
| AFRealFloat | CInfUnsigned | Zero AFComplexFloat of the same FP-kind as the dividend (see section 3.7.2) |
| | CInfSigned | Zero AFRealFloat of the same FP-kind as the dividend (see section 3.7.2) |
| AFComplexFloat | CInfUnsigned or CInfSigned | Zero AFComplexFloat of the same FP-kind as the dividend (see section 3.7.2) |

If divisor is zero then subroutine **HADHH** generates run-time error #601 "DIVIDE ZERO BY ZERO" or outputs CInfUnsigned depending on whether the dividend is zero or not.

### 3.7.2. Kinds of Floating-Point Numbers

Depending on the required *precision* of a floating-point number ExLAF77 uses one of binary representations implemented in concrete descendant classes of AFRealFloat and AFComplexFloat (see section 2.1 above):

- FLOAT_4: Standard 32-bit IEEE representation implemented in the classes CFReal4 and CFComplex4, which contains 24-bit mantissa and 8-bit exponent fields.
- FLOAT_8: Standard 64-bit IEEE representation implemented in the classes CFReal8 and CFComplex8, which contains 53-bit mantissa and 11-bit exponent fields.
- FLOAT_X(NEXP,NMNT): Extended binary representation implemented in the classes CFRealX and CFComplexX, which contains (32*NEXP)-bit exponent field and (32*NMNT)-bit mantissa field. Positive integer parameters NEXP, NMNT of the extended representations denote sizes of the respective fields expressed in 32-bit words.

A particular binary representation uniquely defined by the bit sizes of mantissa and exponent fields is referred to as **FP-kind** of a floating-point number regardless whether the number is real or complex.

### 3.7.3. Selecting Types of Resulting H-objects

This section describes the rules of selecting type of result when performing *Create&Assign* binary arithmetical operations on finite H-objects AFinite, AUVector, AUMatrix, and

AUCompleteLU. Results of operations are supposed to be finite as well. The cases of infinite operands and/or results of operations are considered in section 3.7.1 above.

Subroutines **HAAHH**, **HASHH**, **HAMHH**, **HADHH**, and **HADPHH** described in section 4.13 select the type of output result in accordance with the following rules:

- If both operands are H-numbers AFRealExact then the resulting H-object is also a descendant of AFRealExact. This is the only case when an operation produces no round-off errors, i.e. it is performed in the error-free mode.
- If at least one of the operands is a complex H-object AFComplexFloat, AUVectorCompl, AUMatrixCompl, or AUCompleteLUCompl then the resulting H-object belongs to the same generic subclass of complex floating-point H-objects.
- If one of the operands is an H-number AFRealExact while another one is AFFloat, AUVector, or AUMatrix then the resulting H-object has the same generic class membership as the floating-point operand. Selected FP-kind of the resulting H-object (see 3.7.2) depends on values of its floating-point components. If no under- or overflows occurred during operation then the result has exactly the same kind as the floating-point operand (the default FP-kind).
- If both operands are H-objects composed of floating-point numbers, i.e. they are descendants of AFFloat, AUVector, AUMatrix, or AUCompleteLU then the resulting H-object has the default floating-point kind defined by the following table:

**Table 3.7.3-1. Default Kinds of the Results of Binary *Create&Assign* Operations with Floating-Point Operands (HAAHH, HASHH, HAMHH, HADHH, and HADPHH)**

| FP-Kind of the First Operand | FP-Kind of the Second Operand | | |
|---|---|---|---|
| | FLOAT_4 | FLOAT_8 | FLOAT_X (NEXP2,NMNT2) |
| FLOAT_4 | FLOAT_4 | FLOAT_8 | FLOAT_X (NEXP2,NMNT2) |
| FLOAT_8 | FLOAT_8 | FLOAT_8 | FLOAT_X(NEXP2, max(2,NMNT2)) |
| FLOAT_X (NEXP1,NMNT1) | FLOAT_X (NEXP1,NMNT1) | FLOAT_X(NEXP1, max(NMNT1,2)) | FLOAT_X (max(NEXP1,NEXP2), max(NMNT1,NMNT2)) |

However, if the default FP-kind cannot hold result of an operation because of overflow or underflow, then the corresponding subroutine incrementally increases the size of exponent field until an appropriate result representation is reached. Table 3.7.3-2 below illustrates the sequence of stepwise extensions of the resulting FP-kind:

**Table 3.7.3-2. Sequence of Extensions of the Default FP-Kind Caused by Under- and Overflows (HAAHH, HASHH, HAMHH, HADHH, and HADPHH)**

| Default FP-Kind | First Step | Second Step |
|---|---|---|
| FLOAT_4 | FLOAT_8 | FLOAT_X(1,1) |
| FLOAT_8 | FLOAT_X(1,2) | FLOAT_X(2,2) |
| FLOAT_X(NEXP,NMNT) | FLOAT_X(NEXP+1,NMNT) | FLOAT_X(NEXP+2,NMNT) |

Note that in any case no one *Create&Assign* arithmetical operation requires more than two extensions of the resulting FP-kind to eliminate underflow or overflow. In particular, when transforming H-numbers AFRealExact to a floating-point representation, the latter can never exceed FLOAT_X(1,*) since the bit size of any AFInteger is limited by $2^{31}$-1 (see section 1.6).

## 3.8. Solving Systems of Linear Equations

ExLAF77 implements the standard two-stage numerical procedure of solving systems of algebraic linear equations. First, one should perform complete triangular decomposition of the matrix AUMatrixSq of linear system using subroutine **HUCLU** described in section 4.17. **HUCLU** automatically selects an appropriate numerical method depending on particular kind of the system's matrix. It creates a corresponding H-object AUCompleteLU that contains triangular factor(s) of the matrix and, for the cases of general and indefinite matrices, a permutation vector. The output H-object AUCompleteLU is stored on the place of the input matrix AUMatrixSq, i.e. on exiting **HUCLU** the original system's matrix appears to be overwritten with its factored form.

At the second stage the factored matrix is used for computing solution of the system for a given right-hand side vector (RHS). In the context of solving linear equations it is convenient to consider decomposed matrix just as a specific form of the inverse one. Therefore, there is no reason to make difference between finding single- or multiple-RHS solution of the system and multiplying H-object AUCompleteLU by the corresponding right-hand side H-vector or H-matrix. One should perform the latter procedure with using subroutines **HAMHH** and **HUMHH** described in section 4.13.

Subroutine **HAMHH** multiplies generic H-objects and stores resulting product in a new created H-object (*Create&Assign* multiplication). Its calling statement has the following form:

```
CALL HAMHH( IRH1, IRH2, ILH, *ERROR )
```

where IRH1 and IRH2 are handles to the left and right factors respectively, and ILH is a handle to the new H-object initialized with their product. If one of the input handles IRH1 or IRH2 is associated with an H-object AUCompleteLU while another one is associated with H-object AUVector or AUMatrix, then the output handle ILH identifies H-object that is a solution of the corresponding system of linear equations. Permissible combinations of the arguments IRH1, IRH2 are listed in the Table3.7-1 below.

**Table 3.7-1. *Create&Assign* Multiplications by H-objects AUCompleteLU**

| Argument IRH1 | Argument IRH2 | Result IH |
|---|---|---|
| AUCompleteLU – complete LU-decomposition of a square non-singular *n* by *n* H-matrix **A** | AUVector – *n*-vector **b** | AUVector – *n*-vector **x** that is a solution of the system of linear equations $\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$ |
| AUCompleteLU – complete LU-decomposition of a square non-singular *n* by *n* H-matrix **A** | AUMatrix – *n* by *m* matrix **B** | AUMatrix – *n* by *m* matrix **X** that is a solution of the system of linear equations $\mathbf{A}\cdot\mathbf{X} = \mathbf{B}$ |
| AUVector – *n*-vector **b** | AUCompleteLU – complete LU-decomposition of a square non-singular *n* by *n* H-matrix **A** | AUVector – *n*-vector **x** that is a solution of the system of linear equations $\mathbf{x}\cdot\mathbf{A} = \mathbf{b}$ |
| AUMatrix – *m* by *n* matrix **B** | AUCompleteLU – complete LU-decomposition of a square non-singular *n* by *n* H-matrix **A** | AUMatrix – *m* by *n* matrix **X** that is a solution of the system of linear equations $\mathbf{X}\cdot\mathbf{A} = \mathbf{B}$ |

Subroutine **HUMHH** performs multiplications of floating-point H-object by finite H-object and updates the floating-point operand with the resulting product (*Update* multiplication). Its calling statement has the following form:

```
CALL HUMHH( IRH1, IRH2, SIDE, *ERROR )
```

where IRH1 and IRH2 are handles to the left and right factors respectively, and SIDE is a single-character text descriptor pointing the operand to be updated. If one of the input handles IRH1 or IRH2 is associated with an H-object AUCompleteLU while another one is associated with H-object AUVector or AUMatrix, then **HUMHH** updates the latter object with a solution the corresponding system of linear equations. Permissible combinations of the arguments IRH1, IRH2, and the descriptor SIDE are listed in the Table 3.7-2 below.

**Table 3.7-2. *Update* Multiplications by H-objects AUCompleteLU**

| Argument **IRH1** | Argument **IRH2** | **SIDE** | **Operation** |
|---|---|---|---|
| AUCompleteLU – complete LU-decomposition of a square non-singular *n* by *n* H-matrix **A** | AUVector – *n*-vector **b** | 'R' | Vector **b** is updated with a solution **x** of the system of linear equations $\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$ |
| AUCompleteLU – complete LU-decomposition of a square non-singular *n* by *n* H-matrix **A** | AUMatrix – *n* by *m* matrix **B** | 'R' | Matrix **B** is updated with a solution **X** of the system of linear equations $\mathbf{A}\cdot\mathbf{X} = \mathbf{B}$ |
| AUVector – *n*-vector **b** | AUCompleteLU – complete LU-decomposition of a square non-singular *n* by *n* H-matrix **A** | 'L' | Vector **b** is updated with a solution **x** of the system of linear equations $\mathbf{x}\cdot\mathbf{A} = \mathbf{b}$ |
| AUMatrix – *m* by *n* matrix **B** | AUCompleteLU – complete LU-decomposition of a square non-singular *n* by *n* H-matrix **A** | 'L' | Matrix **B** is updated with a solution **X** of the system of linear equations $\mathbf{X}\cdot\mathbf{A} = \mathbf{B}$ |

# Section 4. Interface Subroutines

All the ExLAF77 interface subprograms callable from Fortran programs have `SUBROUTINE`-like interfaces since Fortran `FUNCTION`-s do not provide the alternate return option. So, they should be called via `CALL` statement like any other Fortran subroutine.

## 4.1. Routine Naming Conventions

Names of interface subroutines consist of no more than 6 upper-case characters for compliance with Fortran-77 standards, and start with the letter **H** that indicates belonging to the ExLAF77. (The leading letter is associated with "handle").

The names are divided into two kinds: a) those exactly predefined by meaning of standard operations and types of the operands, and b) all others names appointed for some specific or "nonstandard" procedures.

Routine names have the following structure:

**H** <code of operation> [{<modifier>|<unique name>}][<operand type>[<operand type>]].

The <code of operation> field is a single-character specifier of a standard operation.

**M** **M**ake: Create new H-object without initialization.

**A** Create&**A**ssign: Create new H-object initialized with result of an operation.

**U** **U**pdate: Update existing H-object.

**E** **E**xtract: Extract part of H-object in a text or numerical representation, or create new H-object initialized with a part of existing one.

**F** **F**unction: Create new H-number initialized with computed value of a function

**C** **C**onstant: Create new H-number initialized with computed value of a math constant.

**L** **L**ogical: Compare two H-numbers, or get logical class indicator.

**G** **G**et: Retrieve parameter of H-object or its element.

**S** **S**ystem: General-purpose system subroutine. The letter **S** may be followed by one of two extra single-character specifiers.

- **E** **E**nable or **E**stablish
- **D** **D**isable or **D**elete

Names with <code of operation> = **E**, **F**, **L**, **G**, and **S{E|D}** belong to the kind (b) mentioned above. The subsequent <unique name> field provides a specific name for each subroutine of the kind.

Names with <code of operation> = **M**, **A**, and **U** belong to the kind (a). The subsequent <modifier> field provides additional details of the operation.

**CPY** **Copy**: Make a copy of H-object (*Create&Assign* unary **+**)

**NEG** **Negate**: Change sign of H-object

**CNJ** **Conjugate**: Complex conjugate of H-object

**A** **Add**. Binary arithmetical operation **+**

**S** **Subtract**. Binary arithmetical operation **-**

**M** **Multiply**. Binary arithmetical operation *****

**D** **Divide**. Binary arithmetical operation **/**

**DP** **Dot Product** of H-vectors or H-matrices

The <operand type> field following the <modifier> or <unique name> specifies types of the operands.

**H** Any **H**-object

**X** **Exact** number AFRealExact

**N** Floating-point **Number** AFFloat

**NX** Any object ANumber including CInfSigned and CInfUnsigned

**V** **Vector** AUVector

**EV** **Element** of AUVector

**M** **Matrix** AUMatrix

**MS** Hermitian (**Symmetrical**) matrix AUHermitian

**EM** **Element** of AUMatrix

**MR** **Row** of AUMatrix

**MC** **Column** of AUMatrix

**F** **Fortran** data

**T** **Text** string

**R** **Real** part of H-object

**I** **Imaginary** part of H-object

**Examples:**

**HUEMF** Update element of existing AUMatrix with Fortran variable

**HANXT** Create new object ANumber and initialize it with a text string

**HAAHH** Create new H-object and initialize it with the sum of two existing H-objects

**HSINIT** System subroutine opening ExLAF77 working session

## 4.2. Specifying Fortran Data Types

Many of ExLAF77 operations accept native Fortran data as operands. Since interface subroutines are unable to recognize the types of actual arguments, the calling statements have to contain explicit descriptions of the data types. Specifying Fortran data types is supported by an auxiliary single-character (`CHARACTER*1`) descriptor that immediately precedes respective "Fortran operand" in the parameter list. Table 4.2.1 below summarizes permissible values of the type descriptors.

**Table 4.2-1. Descriptors of the Fortran Data Types**

| Type Descriptor | Fortran Type |
|---|---|
| **'I'** | INTEGER |
| **'S'** | REAL |
| **'D'** | DOUBLE PRECISION |
| **'C'** | COMPLEX |
| **'Z'** | DOUBLE COMPLEX |

If the passed actual value of type descriptor does not coincide with any of the listed ones then interface subroutine generate error #103 "UNRECOGNIZED TEXT DESCRIPTOR". Note that converting H-objects into Fortran `INTEGER` type currently is not allowed, i.e. respective export subroutines treat descriptor `'I'` as an illegal one.

When invoking ExLAF77 operations with "Fortran operands" it is critically important to ensure strict accordance of type descriptors with actual data types. Incorrect specifying Fortran types usually results in irregular computational errors hard to detect.

## 4.3. Opening and Closing Working Session

ExLAF77 working session should be opened and closed by calling system subroutines **HSINIT** and **HSEXIT** described below.

**SUBROUTINE HSINIT( FILENAME, HEAPSIZE, *ERROR )**

*Opens ExLAF77 working session*

### Input Parameters

FILENAME `CHARACTER*.` Name for the ExLAF77 log file or path with a name. **HSINIT** automatically adds extension `.LOG` to the file name. If path is not specified then the log file is created in the current directory. If the specified file already exists, it is opened in "append" mode, otherwise a new file is created. If empty string is passed as actual parameter then the default file `EXLAF77.LOG` in the current directory is created.

`HEAPSIZE` `INTEGER.` Maximum size of available heap memory in Mbytes. This parameter is introduced to restrict uncontrollable physical memory overflow that typically result in OS deadlock due to intensive swapping. Provided that the amount of memory used by other concurrently running applications is negligible compared with ExLAF77, one can increase `HEAPSIZE` up to 80-90% of total amount of computer RAM.

### Output Parameters

`ERROR` Alternate return argument.

### Remarks

**`HSINIT`** should be called on starting every ExLAF77 working session. Repeated calling **`HSINIT`** before closing current working session produce no effect. For details of the opening procedure see section 3.1 above.

---

**`SUBROUTINE HSEXIT`**

*Closes ExLAF77 working session*

### Remarks

Repeated calling **`HSEXIT`** before opening working session produce no effect. For details of the closing procedure see section 3.1 above.

## 4.4. Handling Run-Time Errors

ExLAF77 provides a mechanism for run-time processing errors that can arise during computations. For details of the error handling procedures see section 3.2.

---

**`SUBROUTINE HSERR( ICODE )`**

*Retrieves numerical code of last run-time error*

### Output Parameters

`ICODE` `INTEGER.` Numerical code of the most recent run-time error.

### Remarks

Numerical error code is stored as a global ExLAF77 internal variable that is set to zero when opening working session. Every run-time error resulting in an alternate return resets its value in accordance with Table A-1 of Appendix A.

## SUBROUTINE HSEMSK( ICODE )

*Masks text massages of run-time error*

### Input Parameters

ICODE    INTEGER. Numerical code of the run-time error to be masked.

### Remarks

On opening ExLAF77 working session all run-time errors are unmasked. Calling **HSEMSK** suppresses text messages of the specified error. If that error has already been masked or passed value of ICODE does not coincide with any code from Table A-1 of Appendix A, then **HSEMSK** produces no effect.

## SUBROUTINE HSDMSK( ICODE )

*Unmasks text massages of run-time error*

### Input Parameters

ICODE    INTEGER. Numerical code of the run-time error to be unmasked.

### Remarks

Calling **HSDMSK** resumes writing text messages of the specified error to ExLAF77 log file. If that error has already been unmasked or passed value of ICODE does not coincide with any code from Table A-1 of Appendix A, then **HSDMSK** produces no effect.

## SUBROUTINE HSMSKA( MODE )

*Sets mode of masking error messages*

### Input Parameters

MODE    INTEGER. Specifies global mode of masking error messages:
MODE = 0 - Unconditionally suppress all error messages;
MODE = 1 - Suppress only the messages explicitly masked by **HSEMSK**;
MODE = 2 - Unmask all run-time errors and resume writing all messages to the log file.

### Remarks

Initially, on opening ExLAF77 working session, all run-time errors are unmasked. Calling **HSMSKA** with MODE = 0 suppresses all the error messages while keeping list of errors that have been previously masked by **HSEMSK**. Setting MODE = 1 resumes selective masking in accordance with that list. Invoking **HSMSKA** with MODE = 2 restores the initial state, i.e. resumes writing all error messages to the log file and clears the list of masked errors.

If `MODE` ≠ `0`, `1`, or 2 then **`HSMSKA`** produce no effect.

---

**`SUBROUTINE HSUNDF( LFLAG )`**

*Switches mode of floating-point underflow control*

### Input Parameters

`LFLAG` `LOGICAL.` Specifies the mode of the floating-point underflow control:
`MODE=.TRUE.` - Enable underflow control;
`MODE=.FALSE.` - Disable underflow control.

### Remarks

On opening working session the underflow control is enabled, i.e. floating-point underflows are treated like all other run-time errors. After disabling the control, underflows do not indicate errors while resulting denormalized values are set to zero.

## 4.5. Releasing Memory

---

**`SUBROUTINE HSDOBJ( IH, *ERROR )`**

*Deletes H-object*

### Input Parameters

`IH` `INTEGER.` Handle to the H-object to be deleted.

### Output Parameters

`ERROR` Alternate return argument.

### Remarks

Handle `IH` becomes invalid after deleting H-object it is associated with. Henceforth `IH` cannot be used as an input parameter of any ExLAF77 subroutine until it is associated with another H-object.

---

**`SUBROUTINE HSDALL`**

*Deletes all H-objects*

### Remarks

**`HSDALL`** removes all the H-objects created during current working session without closing it.

## SUBROUTINE HSEMRK( IHMRK, *ERROR )

*Sets memory allocation mark*

### Output Parameters

IHMRK INTEGER. Handle to the new allocation mark.

ERROR Alternate return argument.

### Remarks

For details of using memory allocation marks see section 3.3.

## SUBROUTINE HSDMRK( IHMRK, *ERROR )

*Removes memory allocation mark*

### Input/Output Parameters

IHMRK INTEGER. Handle to the allocation mark to be removed.

### Output Parameters

ERROR Alternate return argument.

### Remarks

Handle IHMRK becomes invalid after removing the mark it is associated with. For details of using memory allocation marks see section 3.3.

## SUBROUTINE HSDGRP( IHMRK1, IHMRK2, *ERROR )

*Deletes designated group of H-objects*

### Input Parameters

IHMRK1 INTEGER. Handle to the starting allocation mark. If IHMRK1 = 0 then the designated group of H-objects starts with the very first one.

IHMRK2 INTEGER. Handle to the final allocation mark. If IHMRK2 = 0 then the designated group of H-objects concludes with the very last one.

### Output Parameters

ERROR Alternate return argument.

### Remarks

**HSDGRP** deletes all H-objects in the range between allocation marks IHMRK1 and IHMRK2, i.e. those created after setting mark IHMRK1, but before setting IHMRK2. It removes the final mark IHMRK2 as well. Handles to deleted H-objects and IHMRK2 become invalid. For details of using memory allocation marks see section 3.3.

## 4.6. Retrieving Information on H-Objects

**SUBROUTINE HLFIN( IH, LISFIN, *ERROR )**

*Is H-object finite?*

### Input Parameters

IH INTEGER. Handle to H-object.

### Output Parameters

LISFIN LOGICAL.
LISFIN=.FALSE. – for CInfSigned and CInfUnsigned;
LISFIN=.TRUE. – for all other classes of H-objects.

ERROR Alternate return argument.

**SUBROUTINE HLREAL( IH, LISREAL, *ERROR )**

*Is H-object real?*

### Input Parameters

IH INTEGER. Handle to H-object.

### Output Parameters

LISREAL LOGICAL.
LISREAL=.TRUE. – for (pseudo)descendants of AReal, AUVectorReal, AUMatrixReal, and classes CUCompleteLUReal4,8,X, CUHessenbergReal4,8,X.
LISREAL=.FALSE. – for all other classes of H-objects.

ERROR Alternate return argument.

**SUBROUTINE HLFLT( IH, LISFLT, *ERROR )**

*Is H-object composed of floating-point numbers?*

### Input Parameters

IH INTEGER. Handle to H-object.

### Output Parameters

LISFLT LOGICAL.
LISFLT=.TRUE. – for (pseudo)descendants of AFFloat, AUVector, AUMatrix, AUCompleteLU, and AUHessenberg;

LISFLT=.FALSE. – for descendants of AFRealExact and classes CInfSigned, CInfUnsigned.

ERROR Alternate return argument.

---

**SUBROUTINE HLNUM( IH, LISNUM, *ERROR )**

*Is H-object a number?*

### Input Parameters

IH INTEGER. Handle to H-object.

### Output Parameters

LISNUM LOGICAL.
LISNUM=.TRUE. – for descendants of ANumber;
LISNUM=.FALSE. – for all other classes of H-objects.

ERROR Alternate return argument.

---

**SUBROUTINE HLINT( IHX, LISINT, *ERROR )**

*Is H-number an integer number?*

### Input Parameters

IH INTEGER. Handle to H-object.

### Output Parameters

LISINT LOGICAL.
LISINT=.TRUE. – for descendants of AFInteger;
LISINT=.FALSE. – for all other classes of H-objects.

ERROR Alternate return argument.

---

**SUBROUTINE HLVECT( IH, LISVECT, *ERROR )**

*Is H-object a vector?*

### Input Parameters

IH INTEGER. Handle to H-object.

### Output Parameters

LISVECT LOGICAL.
LISVECT=.TRUE. – for descendants of AVector;
LISVECT=.FALSE. – for all other classes of H-objects.

ERROR Alternate return argument.

### SUBROUTINE HLMATR( IH, LISMATR, *ERROR )

*Is H-object a matrix?*

**Input Parameters**

IH INTEGER. Handle to H-object.

**Output Parameters**

LISMATR LOGICAL.
LISMATR=.TRUE. – for descendants of AMatrix;
LISMATR=.FALSE. – for all other classes of H-objects.

ERROR Alternate return argument.

### SUBROUTINE HLMSQR( IH, LISSQR, *ERROR )

*Is H-object a (transformed) square matrix?*

**Input Parameters**

IH INTEGER. Handle to H-object.

**Output Parameters**

LISSQRM LOGICAL.
LISSQRM=.TRUE. – for descendants of.AUMatrixSq, AUCompleteLU, and AUHessenberg;
LISSQRM=.FALSE. – for all other classes of H-objects.

ERROR Alternate return argument.

### SUBROUTINE HLHERM( IH, LISHERM, *ERROR )

*Is H-object a (transformed) Hermitian matrix?*

**Input Parameters**

IH INTEGER. Handle to H-object.

**Output Parameters**

LISHERM LOGICAL.
LISHERM=.TRUE. – for descendants of AUMatrixSqHerm, AUCompleteLUHerm, and AUHessenbergHerm;
LISHERM=.FALSE. – for all other classes of H-objects.

ERROR Alternate return argument.

## SUBROUTINE HLCLU( IH, LISCLU, *ERROR )

*Is H-object a complete LU decomposition of square matrix?*

### Input Parameters

IH INTEGER. Handle to H-object.

### Output Parameters

LISCLU LOGICAL.
LISCLU=.TRUE. – for descendants of.AUCompleteLU;
LISCLU=.FALSE. – for all other classes of H-objects.
ERROR Alternate return argument.

## SUBROUTINE HLHES( IH, LISHES, *ERROR )

*Is H-object a Hessenberg form of square matrix?*

### Input Parameters

IH INTEGER. Handle to the H-object.

### Output Parameters

LISHES LOGICAL.
LISHES=.TRUE. – for descendants of.AUHessenberg;
LISHES=.FALSE. – for all other classes of H-objects.

ERROR Alternate return argument.

## SUBROUTINE HLZERO( IH, LISZERO, *ERROR )

*Is H-object zero?*

### Input Parameters

IH INTEGER. Handle to H-object.

### Output Parameters

LISZERO LOGICAL.
LISZERO=.TRUE. – the H-object IH has zero value;
LISZERO=.FALSE. – the H-object IH has a non zero value.

ERROR Alternate return argument.

### Remarks

Output result .TRUE. for H-vector or H-matrix means that all its elements are equal to zero.

## SUBROUTINE HLNXPO( IHNX, LISPOS, *ERROR )

*Is real H-number positive?*

### Input Parameters

IHNX INTEGER. Handle to H-number AReal;

### Output Parameters

LISPOS LOGICAL.
LISPOS=.TRUE. – the H-number IHNX is positive;
LISPOS=.FALSE. – the H-number IHNX is zero or negative.

ERROR Alternate return argument.

## SUBROUTINE HLNXNE( IHNX, LISNEG, *ERROR )

*Is real H-number negative?*

### Input Parameters

IHNX INTEGER. Handle to H-number AReal.

### Output Parameters

LISNEG LOGICAL.
LISNEG=.TRUE. – the H-number IHNX is negative;
LISNEG=.FALSE. – the H-number IHNX is zero or positive.

ERROR Alternate return argument.

## SUBROUTINE HLEVPO( IHV, INDEX, LISPOS, *ERROR )

*Is element of real H-vector positive?*

### Input Parameters

IHV INTEGER. Handle to H-vector AUVectorReal.

INDEX INTEGER. Index of the selected element of the H-vector IHV (positive number).

### Output Parameters

LISPOS LOGICAL.
LISPOS=.TRUE. – the INDEX-th element of the H-vector IHV is positive;
LISPOS=.FALSE. – the INDEX-th element of the H-vector IHV is zero or negative.

ERROR Alternate return argument.

**SUBROUTINE HLEVNE( IHV, INDEX, LISNEG, *ERROR )**

*Is element of real H-vector negative?*

### Input Parameters

IHV — INTEGER. Handle to H-vector AUVectorReal.

INDEX — INTEGER. Index of the selected element of the H-vector IHV (positive number).

### Output Parameters

LISNEG — LOGICAL.
LISNEG=.TRUE. – the INDEX-th element of the H-vector IHV is negative;
LISNEG=.FALSE. – the INDEX-th element of the H-vector IHV is zero or positive.

ERROR — Alternate return argument.

**SUBROUTINE HLEMPO( IHM, IROW, ICOL, LISPOS, *ERROR )**

*Is element of real H-matrix positive?*

### Input Parameters

IHM — INTEGER. Handle to H-matrix AUMatrixReal.

IROW — INTEGER. Row index of the selected element of the H-matrix IHM (positive number).

ICOL — INTEGER. Column index of the selected element of the H-matrix IHM (positive number).

### Output Parameters

LISPOS — LOGICAL.
LISPOS=.TRUE. – the (IROW,ICOL)-th element of the H-matrix IHM is positive;
LISPOS=.FALSE. – the (IROW,ICOL)-th element of the H-matrix IHM is zero or negative.

ERROR — Alternate return argument.

**SUBROUTINE HLEMNE( IHM, IROW, ICOL, LISNEG, *ERROR )**

*Is element of real H-matrix negative?*

### Input Parameters

IHM — INTEGER. Handle to H-matrix AUMatrixReal.

IROW INTEGER. Row index of the selected element of the H-matrix IHM (positive number).

ICOL INTEGER. Column index of the selected element of the H-matrix IHM (positive number).

### Output Parameters

LISNEG LOGICAL.
LISNEG=.TRUE. – the (IROW,ICOL)-th element of the H-matrix IHM is negative;
LISNEG=.FALSE. – the (IROW,ICOL)-th element of the H-matrix IHM is zero or positive.

ERROR Alternate return argument.

---

## SUBROUTINE HGNAME( IH, NAME, *ERROR )

*Returns class name of H-object*

### Input Parameters

IH INTEGER. Handle to H-object.

### Output Parameters

NAME CHATACTER*. Concrete class name of the H-object IH.

ERROR Alternate return argument.

### Remarks

If the length of string NAME is less than required then the string is padded with asterisks.

---

## SUBROUTINE HGFLTS( IH, NEXP, NMNT, *ERROR )

*Returns sizes of exponent and mantissa fields*

### Input Parameters

IH INTEGER. Handle to H-object composed of floating-point numbers.

### Output Parameters

NEXP INTEGER. Exponent length in 32-bit words (non-negative number). NEXP=0 stands for single or double precision IEEE floating-point data.

NMNT INTEGER. Mantissa length in 32-bit words (positive number). If NEXP=0 then NMNT=1 and 2 imply single and double precision IEEE floating-point data respectively.

ERROR Alternate return argument.

### Remarks

The input H-object IH should be descendant of AFFloat, AUVector, AUMatrix, AUCompleteLU, or AUHessenberg.

---

**SUBROUTINE HGVDIM( IHV, NDIM, *ERROR )**

*Returns dimension of H-vector*

### Input Parameters

IHV INTEGER. Handle to H-vector AVector.

### Output Parameters

NDIM INTEGER. Dimension of the H-vector IHV (non-negative number).

ERROR Alternate return argument.

---

**SUBROUTINE HGMDIM( IHM, NROW, NCOL, *ERROR )**

*Returns dimensions of (transformed) H-matrix*

### Input Parameters

IHM INTEGER. Handle to (transformed) H-matrix.

### Output Parameters

NROW INTEGER. Number of rows of the H-matrix IHM (non-negative number).

NCOL INTEGER. Number of columns of the H-matrix IHM (non-negative number).

ERROR Alternate return argument.

### Remarks

The input H-matrix IHM should be descendant of AMatrix, ACompleteLU, or AHessenberg.

## 4.7. Creating Empty H-Objects

---

**SUBROUTINE HMN( NEXP, NMNT, LISREAL, IHN, *ERROR )**

***Create**s new floating-point H-number*

### Input Parameters

NEXP INTEGER. Exponent length in 32-bit words (non-negative number). NEXP=0 stands for single or double precision IEEE floating-point data.

NMNT INTEGER. Mantissa length in 32-bit words (positive number). If NEXP=0 then NMNT=1 and 2 imply single and double precision IEEE floating-point data respectively.

LISREAL LOGICAL.
LISREAL=.TRUE. – Create real H-number AFRealFloat;
LISREAL=.FALSE. – Create complex H-number AFComplexFloat.

### Output Parameters

IHN INTEGER. Handle to the created H-number AFFloat.

ERROR Alternate return argument.

### Remarks

The new H-number is initialized with zero.

---

## SUBROUTINE HMV( NEXP, NMNT, LISREAL, NDIM, IHV, *ERROR )

***Creates** new H-vector*

### Input Parameters

NEXP INTEGER. Exponent length in 32-bit words (non-negative number). NEXP=0 stands for single or double precision IEEE floating-point data.

NMNT INTEGER. Mantissa length in 32-bit words (positive number). If NEXP=0 then NMNT=1 and 2 imply single and double precision IEEE floating-point data respectively.

LISREAL LOGICAL.
LISREAL=.TRUE. – Create real H-vector AUVectorReal;
LISREAL=.FALSE. – Create complex H-vector AUVectorCompl.

NDIM INTEGER. Dimension of the new H-vector IHV (non-negative number).

### Output Parameters

IHV INTEGER. Handle to the created H-vector AUVector.

ERROR Alternate return argument.

### Remarks

The new H-vector IHV consists of NDIM real or complex floating-point elements with exponent size NEXP and mantissa size NMNT. All the elements are initialized with zeros.

## SUBROUTINE HMM( NEXP, NMNT, LISREAL, NROW, NCOL, IHM, *ERROR )

*Creates new general H-matrix*

### Input Parameters

NEXP INTEGER. Exponent length in 32-bit words (non-negative number). NEXP=0 stands for single or double precision IEEE floating-point data.

NMNT INTEGER. Mantissa length in 32-bit words (positive number). If NEXP=0 then NMNT=1 and 2 imply single and double precision IEEE floating-point data respectively.

LISREAL LOGICAL.
LISREAL=.TRUE. – Create real H-matrix AUMatrixReal;
LISREAL=.FALSE. – Create complex H- matrix AUMatrixCompl.

NROW INTEGER. Number of rows of the new H-matrix IHM (non-negative number).

NCOL INTEGER. Number of columns of the new H-matrix IHM (non-negative number).

### Output Parameters

IHM INTEGER. Handle to the created H-matrix AUMatrixSqGen or AUMatrixRect depending on NROW and NCOL.

ERROR Alternate return argument.

### Remarks

The new general H-matrix has the full storage format and consists of NROW*NCOL real or complex floating-point elements with exponent size NEXP and mantissa size NMNT. All the elements are initialized with zeros. In cases NROW=NCOL and NROW≠NCOL H-matrices AUMatrixSqGen and AUMatrixRect respectively are created

## SUBROUTINE HMMS( NEXP, NMNT, LISREAL, NDIM, ISIGN, IHMS, *ERROR )

*Creates new Hermitian H-matrix*

### Input Parameters

NEXP INTEGER. Exponent length in 32-bit words (non-negative number). NEXP=0 stands for single or double precision IEEE floating-point data.

NMNT INTEGER. Mantissa length in 32-bit words (positive number). If NEXP=0 then NMNT=1 and 2 imply single and double precision IEEE floating-point data respectively.

LISREAL LOGICAL.

LISREAL=.TRUE. – Create real H-matrix AUMatrixReal;
LISREAL=.FALSE. – Create complex H- matrix AUMatrixCompl.

NDIM INTEGER. Dimension of the new H-matrix IHMS (non-negative number).

ISIGN INTEGER. Signature of the new matrix. IHMS
ISIGN=1 – Create positive-definite Hermitian matrix;
ISIGN=0 – Create indefinite Hermitian matrix.

### Output Parameters

IHMS INTEGER. Handle to the created H-matrix AUMatrixSqHerm.

ERROR Alternate return argument.

### Remarks

The new Hermitian H-matrix has the packed storage format and consists of NROW*(NROW+1)/2 real or complex floating-point elements with exponent size NEXP and mantissa size NMNT. All the elements are initialized with zeros.

## 4.8. Creating H-Objects with Initialization

### 4.8.1. Initialization with Text String

**SUBROUTINE HANXT( STR, IHNX, *ERROR )**

***Create**s new H-number initialized with text string*

### Input Parameters

STR CHARACTER*. Initializing text string.

### Output Parameters

IHNX INTEGER. Handle to the created H-number ANumber.

ERROR Alternate return argument.

### Remarks

For the permissible formats of the input string STR, and rules of automatic selection of the number kind please refer to section 3.4.2.

## 4.8.2. Initialization with Fortran Data

**SUBROUTINE HAXF( INUMER, IDENOM, IHX, *ERROR )**

*Creates new exact H-number initialized with quotient of two integers*

### Input Parameters

INUMER    INTEGER. Numerator.

IDENOM    INTEGER. Denominator.

### Output Parameters

IHX    INTEGER. Handle to the created H-number AFRealExact or CInfUnsigned.

ERROR    Alternate return argument.

### Remarks

**HAXF** defines the type of new H-number depending on actual value of the quotient INUMER/IDENOM. If INUMER≠0 and IDENOM=0 then H-number CInfUnsigned is generated. If IDENOM≠0 appears to be an exact factor of INUMER, then **HAXF** creates H-number AFInteger and initializes it with the quotient, otherwise an appropriate H-number CFRational is created.

**SUBROUTINE HANF( FTYPE, FVAR, IHN, *ERROR )**

*Creates new floating-point H-number initialized with Fortran variable*

### Input Parameters

FTYPE    CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR    Fortran initializing variable.

### Output Parameters

IHN    INTEGER. Handle to the created H-number AFFloat.

ERROR    Alternate return argument.

### Remarks

If FVAR is an INTEGER variable with type descriptor FTYPE='I' then **HANF** converts it to DOUBLE PRECISION and creates H-number CFReal8.

## SUBROUTINE HAVF( FTYPE, FARRAY, NDIM, IHV, *ERROR )

*Creates new H-vector initialized with Fortran array*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FARRAY (see 4.2).

FARRAY Fortran initializing array. Size of the array should be equal to or greater than dimension NDIM of new H-vector IHV.

NDIM INTEGER. Dimension of the new H-vector IHV (positive number).

### Output Parameters

IHV INTEGER. Handle to the created H-vector AUVector.

ERROR Alternate return argument.

### Remarks

If FARRAY is an INTEGER array with type descriptor FTYPE='I' then **HAVF** converts all its elements to DOUBLE PRECISION and creates H-vector CUVectorReal8.

## SUBROUTINE HAMF( FTYPE, FARRAY, NROW, NCOL, IHM, *ERROR )

*Creates new general H-matrix initialized with Fortran array*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FARRAY (see 4.2).

FARRAY Fortran initializing array. Size of the array should be equal to or greater than total number of elements NROW*NCOL of new H-matrix IHM.

NROW INTEGER. Number of rows of the new H-matrix IHM (positive number).

NCOL INTEGER. Number of columns of the new H-matrix IHM (positive number).

### Output Parameters

IHM INTEGER. Handle to the created H-matrix AUMatrixSqGen or AUMatrixRect.

ERROR Alternate return argument.

### Remarks

The new general H-matrix has full storage format and consists of NROW*NCOL real or complex floating-point elements. In cases NROW=NCOL and NROW≠NCOL H-matrices AUMatrixSqGen and AUMatrixRect respectively are created. If FARRAY is an INTEGER array with type descriptor FTYPE='I' then **HAMF** converts all its elements to DOUBLE PRECISION and creates H-matrix AUMatrixSqGenReal8 or AUMatrixRectReal8.

**SUBROUTINE HAMSF( FTYPE, FARRAY, NDIM, ISIGN, LISPACK, IHMS, *ERROR )**

***Create**s new Hermitian H-matrix initialized with Fortran array*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FARRAY (see 4.2).

FARRAY Fortran initializing array. Size of the array should be equal to or greater than total number of elements of new H-matrix IHMS, i.e. NDIM**2 or NDIM*(NDIM+1)/2 depending on the input storage format LISPACK.

NDIM INTEGER. Dimension of the new H-matrix IHMS (positive number).

ISIGN INTEGER. Signature of new matrix. IHMS:
ISIGN=1 – Create positive-definite Hermitian matrix;
ISIGN=0 – Create indefinite Hermitian matrix.

LISPACK LOGICAL. Specifies storage format for the source matrix:
LISPACK=.TRUE. – FARRAY contains the upper triangle of a source Hermitian matrix stored in the packed format with total number of elements NDIM*(NDIM+1)/2.
LISPACK=.FALSE. – FARRAY contains a source Hermitian matrix stored in the full format with total number of elements NDIM**2.

### Output Parameters

IHMS INTEGER. Handle to the created H-matrix AUMatrixSqHerm.

ERROR Alternate return argument.

### Remarks

If FARRAY is an INTEGER array with type descriptor FTYPE='I' then **HAMSF** converts all its elements to DOUBLE PRECISION and creates H-matrix CUMatrixSqHermReal8.

## 4.8.3. Initialization with H-Object

**SUBROUTINE HAXN( IRHN, ILHX, *ERROR )**

***Create**s new exact H-number initialized with floating-point H-number*

### Input Parameters

IRHN INTEGER. Handle to the source H-number AFFloat.

### Output Parameters

ILHX INTEGER. Handle to the created H-number AFRealExact.

ERROR Alternate return argument.

**Remarks**

One should realize that converting H-numbers AFFloat to AFRealExact typically produces very long numbers that take the amount of memory approximately equal to the sum of the mantissa's bit size and exponent's binary value.

## 4.9. Updating Floating-Point H-Objects

### 4.9.1. Text Input

---

**SUBROUTINE HUNT( STR, IHN, *ERROR )**

***Update**s floating-point H-number with text string*

**Input Parameters**

STR CHARACTER*. Source text string.

**Input/Output Parameters**

IHN INTEGER. Handle to the destination H-number AFFloat .

**Output Parameters**

ERROR Alternate return argument.

**Remarks**

For the permissible formats of the source string STR, please refer to section 3.4.2.

---

**SUBROUTINE HUEVT( STR, INDEX, IHV, *ERROR )**

***Update**s element of H-vector with text string*

**Input Parameters**

STR CHARACTER*. Source text string.

INDEX INTEGER. Index of the selected element of the H-vector IHV (positive number).

**Input/Output Parameters**

IHV INTEGER. Handle to the destination H-vector AUVector.

**Output Parameters**

ERROR Alternate return argument.

### Remarks

For the permissible formats of the source string STR, please refer to section 3.4.2.

---

**SUBROUTINE HUEMT( STR, IROW, ICOL, IHM, *ERROR )**

***Update**s element of H-matrix with text string*

### Input Parameters

STR CHARACTER*. Input string.

IROW INTEGER. Row index of the selected element of the H-matrix IHM (positive number).

ICOL INTEGER. Column index of the selected element of the H-matrix IHM (positive number).

### Input/Output Parameters

IHM INTEGER. Handle to the destination H-matrix AUMatrix.

### Output Parameters

ERROR Alternate return argument.

### Remarks

For the permissible formats of the source string STR, please refer to section 3.4.2.

## 4.9.2. Import of Fortran Data

---

**SUBROUTINE HUNF( FTYPE, FVAR, IHN, *ERROR )**

***Update**s floating-point H-number with Fortran variable*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR Source Fortran variable.

### Input/Output Parameters

IHN INTEGER. Handle to the destination H-number AFFloat.

### Output Parameters

ERROR Alternate return argument.

## SUBROUTINE HURNF( FTYPE, FVAR, IHN, *ERROR )

*Updates real part of complex floating-point H-number with Fortran variable*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR Source Fortran variable.

### Input/Output Parameters

IHN INTEGER. Handle to the destination H-number AFComplexFloat.

### Output Parameters

ERROR Alternate return argument.

### Remarks

Permissible types of the variable FVAR are INTEGER (FTYPE='I'), REAL (='S'), and DOUBLE PRECISION (='D'). Input values FTYPE='C' and 'Z' are treated as illegal ones.

## SUBROUTINE HUINF( FTYPE, FVAR, IHN, *ERROR )

*Updates imaginary part of complex floating-point H-number with Fortran variable*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR Source Fortran variable.

### Input/Output Parameters

IHN INTEGER. Handle to the destination H-number AFComplexFloat.

### Output Parameters

ERROR Alternate return argument.

### Remarks

Permissible types of the variable FVAR are INTEGER (FTYPE='I'), REAL (='S'), and DOUBLE PRECISION (='D'). Input values FTYPE='C' and 'Z' are treated as illegal ones.

## SUBROUTINE HUEVF( FTYPE, FVAR, INDEX, IHV, *ERROR )

*Updates element of H-vector with Fortran variable*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR Source Fortran variable.

INDEX INTEGER. Index of the selected element of the H-vector IHV (positive number).

### Input/Output Parameters

IHV INTEGER. Handle to the destination H-vector AUVector.

### Output Parameters

ERROR Alternate return argument.

## SUBROUTINE HUREVF( FTYPE, FVAR, INDEX, IHV, *ERROR )

***Update**s real part of element of complex H-vector with Fortran variable*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR Source Fortran variable.

INDEX INTEGER. Index of the selected element of the H-vector IHV (positive number).

### Input/Output Parameters

IHV INTEGER. Handle to the destination H-vector AUVectorCompl.

### Output Parameters

ERROR Alternate return argument.

### Remarks

Permissible types of the variable FVAR are INTEGER (FTYPE=‘I’), REAL (=‘S’), and DOUBLE PRECISION (=‘D’). Input values FTYPE=‘C’ and ‘Z’ are treated as illegal ones.

**SUBROUTINE HUIEVF( FTYPE, FVAR, INDEX, IHV, *ERROR )**

***Updates** imaginary part of element of complex H-vector with Fortran variable.*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR Source Fortran variable.

INDEX INTEGER. Index of the selected element of the H-vector IHV (positive number).

### Input/Output Parameters

IHV INTEGER. Handle to the destination H-vector AUVectorCompl.

### Output Parameters

ERROR Alternate return argument.

### Remarks

Permissible types of the variable FVAR are INTEGER (FTYPE=‘I’), REAL (=‘S’), and DOUBLE PRECISION (=‘D’). Input values FTYPE=‘C’ and ‘Z’ are treated as illegal ones.

**SUBROUTINE HUVF( FTYPE, FARRAY, NDIM, IHV, *ERROR )**

***Updates** H-vector with Fortran array*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FARRAY (see section 4.2).

FARRAY Source Fortran array. Size of the array should be equal to or greater than dimension NDIM of the H-vector IHV.

NDIM INTEGER. Dimension of the H-vector IHV (positive number).

### Input/Output Parameters

IHV INTEGER. Handle to the destination H-vector AUVector.

### Output Parameters

ERROR Alternate return argument.

### SUBROUTINE HUEMF( FTYPE, FVAR, IROW, ICOL, IHM, *ERROR )

***Update**s element of H-matrix with Fortran variable*

#### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR Source Fortran variable.

IROW INTEGER. Row index of the selected element of the H-matrix IHM (positive number).

ICOL INTEGER. Column index of the selected element of the H-matrix IHM (positive number).

#### Input/Output Parameters

IHM INTEGER. Handle to the destination H-matrix AUMatrix.

#### Output Parameters

ERROR Alternate return argument.

### SUBROUTINE HUREMF( FTYPE, FVAR, IROW, ICOL, IHM, *ERROR )

***Update**s real part of element of complex H-matrix with Fortran variable*

#### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR Source Fortran variable.

IROW INTEGER. Row index of the selected element of the H-matrix IHM (positive number).

ICOL INTEGER. Column index of the selected element of the H-matrix IHM (positive number).

#### Input/Output Parameters

IHM INTEGER. Handle to the destination H-matrix AUMatrixCompl.

#### Output Parameters

ERROR Alternate return argument.

### Remarks

Permissible types of the variable FVAR are INTEGER (FTYPE='I'), REAL (='S'), and DOUBLE PRECISION (='D'). Input values FTYPE='C' and 'Z' are treated as illegal ones.

## SUBROUTINE HUIEMF( FTYPE, FVAR, IROW, ICOL, IHM, *ERROR )

***Update**s imaginary part of element of complex H-matrix with Fortran variable*

### Input Parameters

FTYPE — CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR — Source Fortran variable.

IROW — INTEGER. Row index of the selected element of the H-matrix IHM (positive number).

ICOL — INTEGER. Column index of the selected element of the H-matrix IHM (positive number).

### Input/Output Parameters

IHM — INTEGER. Handle to the destination H-matrix AUMatrixCompl.

### Output Parameters

ERROR — Alternate return argument.

### Remarks

Permissible types of the variable FVAR are INTEGER (FTYPE='I'), REAL (='S'), and DOUBLE PRECISION (='D'). Input values FTYPE='C' and 'Z' are treated as illegal ones.

## SUBROUTINE HUMRF( FTYPE, FARRAY, IROW, NCOL, IHM, *ERROR )

***Update**s H-matrix row with Fortran array*

### Input Parameters

FTYPE — CHARACTER*1. Fortran type descriptor for FARRAY (see section 4.2).

FARRAY — Source Fortran array. Size of the array should be equal to or greater than number of columns NCOL of the H-matrix IHM.

IROW — INTEGER. Index of the selected row of the H-matrix IHM (positive number).

NCOL — INTEGER. Number of columns of the H-matrix IHM (positive number).

### Input/Output Parameters

IHM INTEGER. Handle to the destination H-matrix AUMatrix.

### Output Parameters

ERROR Alternate return argument.

---

**SUBROUTINE HUMCF( FTYPE, FARRAY, ICOL, NROW, IHM, *ERROR )**

*__Updates__ H-matrix column with Fortran array*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FARRAY (see section 4.2).

FARRAY Source Fortran array. Size of the array should be equal to or greater than number of rows NROW of the H-matrix IHM.

ICOL INTEGER. Index of the selected column in the H-matrix IHM (positive number).

NROW INTEGER. Number of rows of the H-matrix IHM (positive number).

### Input/Output Parameters

IHM INTEGER. Handle to the destination H-matrix AUMatrix.

### Output Parameters

ERROR Alternate return argument.

---

**SUBROUTINE HUMF( FTYPE, FARRAY, NROW, NCOL, IHM, *ERROR )**

*__Updates__ general H-matrix with Fortran array*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FARRAY (see section 4.2).

FARRAY Source Fortran array. Size of the array should be equal to or greater than total number of elements NROW*NCOL of the H-matrix IHM.

NROW INTEGER. Number of rows of the H-matrix IHM (positive number).

NCOL INTEGER. Number of columns of the H-matrix IHM (positive number).

### Input/Output Parameters

IHM INTEGER. Handle the destination H-matrix AUMatrixSqGen or AUMatrixRect.

### Output Parameters

ERROR Alternate return argument.

**SUBROUTINE HUMSF( FTYPE, FARRAY, NDIM, LISPACK, IHMS, *ERROR )**

***Update**s Hermitian H-matrix with Fortran array*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FARRAY (see section 4.2).

FARRAY Source Fortran array.

NDIM INTEGER. Dimension of the H-matrix IHMS (positive number).

LISPACK LOGICAL. Specifies storage format for the source matrix:
LISPACK=.TRUE. – FARRAY contains the upper triangle of a source Hermitian matrix stored in the packed format with total number of elements NDIM*(NDIM+1)/2.
LISPACK=.FALSE. – FARRAY contains a source Hermitian matrix stored in the full format with total number of elements NDIM**2.

### Input/Output Parameters

IHMS INTEGER. Handle to the destination H-matrix AUMatrixSqHerm.

### Output Parameters

ERROR Alternate return argument.

## 4.9.3. Updating with Another H-Object

**SUBROUTINE HUHH( IRH, ILH, *ERROR )**

***Update**s floating-point H-object with finite H-object*

### Input Parameters

IRH INTEGER. Handle to the source finite H-object AFinite, AVector, or AMatrix.

### Input/Output Parameters

ILH INTEGER. Handle to the destination floating-point H-object.

### Output Parameters

ERROR Alternate return argument.

### Remarks

H-objects ILH and IRH must belong to the same generic kind, i.e. be descendants of the same parent class ANumber, AVector, or AMatrix. Senseless cross-kind update operations result in run time error #102 “ILLEGAL TYPE OF OPERAND”. If ILH and IRH are associated with H-objects AVector, or AMatrix then their respective dimensions should coincide.

**SUBROUTINE HURNN( IRHN, ILHN, *ERROR )**

*Updates real part of complex H-number with finite real H-number*

### Input Parameters

IRHN INTEGER. Handle to the source H-number AFReal.

### Input/Output Parameters

ILHN INTEGER. Handle to the destination H-number AFComplexFloat.

### Output Parameters

ERROR Alternate return argument.

**SUBROUTINE HUINN( IRHN, ILHN, *ERROR )**

*Updates imaginary part of complex H-number with finite real H-number*

### Input Parameters

IRHN INTEGER. Handle to the source H-number AFReal.

### Input/Output Parameters

ILHN INTEGER. Handle to the destination H-number AFComplexFloat.

### Output Parameters

ERROR Alternate return argument.

**SUBROUTINE HUEVN( IRHN, INDEX, ILHV, *ERROR )**

*Updates element of H-vector with finite H-number*

### Input Parameters

IRHN INTEGER. Handle the source H-number AFinite.

INDEX INTEGER. Index of the selected element of the H-vector ILHV (positive number).

### Input/Output Parameters

ILHV INTEGER. Handle to the destination H-vector AUVector.

### Output Parameters

ERROR Alternate return argument.

## SUBROUTINE HUREVN( IRHN, INDEX, ILHV, *ERROR )

*Updates real part of element of complex H-vector with finite real H-number*

### Input Parameters

IRHN INTEGER. Handle to the source H-number AFReal.

INDEX INTEGER. Index of the selected element of the H-vector ILHV (positive number).

### Input/Output Parameters

ILHV INTEGER. Handle to the destination H-vector AUVectorCompl.

### Output Parameters

ERROR Alternate return argument.

## SUBROUTINE HUIEVN( IRHN, INDEX, ILHV, *ERROR )

*Updates imaginary part of element of complex H-vector with finite real H-number*

### Input Parameters

IRHN INTEGER. Handle to the source H-number AFReal.

INDEX INTEGER. Index of the selected element of the H-vector ILHV (positive number).

### Input/Output Parameters

ILHV INTEGER. Handle to the destination H-vector AUVectorCompl.

### Output Parameters

ERROR Alternate return argument.

## SUBROUTINE HUEMN( IHN, IROW, ICOL, IHM, *ERROR )

*Updates element of H-matrix with finite H-number*

### Input Parameters

IHN INTEGER. Handle to the source H-number AFinite.

IROW INTEGER. Row index of the selected element of the H-matrix IHM (positive number).

ICOL INTEGER. Column index of the selected element of the H-matrix IHM (positive number).

### Input/Output Parameters

`IHM` `INTEGER.` Handle to the destination H-matrix AUMatrix.

### Output Parameters

`ERROR` Alternate return argument.

---

**SUBROUTINE HUREMN( IHN, IROW, ICOL, IHM, *ERROR )**

***Updates** real part of element of complex H-matrix with finite real H-number*

### Input Parameters

`IHN` `INTEGER.` Handle to the source H-number AFReal.

`IROW` `INTEGER.` Row index of the selected element of the H-matrix `IHM` (positive number).

`ICOL` `INTEGER.` Column index of the selected element of the H-matrix `IHM` (positive number).

### Input/Output Parameters

`IHM` `INTEGER.` Handle to the destination H-matrix AUMatrixCompl.

### Output Parameters

`ERROR` Alternate return argument.

---

**SUBROUTINE HUIEMN( IHN, IROW, ICOL, IHM, *ERROR )**

***Updates** imaginary part of element of complex H-matrix with finite real H-number*

### Input Parameters

`IHN` `INTEGER.` Handle to the source H-number AFReal.

`IROW` `INTEGER.` Row index of the selected element of the H-matrix `IHM` (positive number).

`ICOL` `INTEGER.` Column index of the selected element of the H-matrix `IHM` (positive number).

### Input/Output Parameters

`IHM` `INTEGER.` Handle to the destination H-matrix AUMatrixCompl.

### Output Parameters

`ERROR` Alternate return argument.

## SUBROUTINE HUMRV( IHV, IROW, IHM, *ERROR )

*Updates H-matrix row with H-vector*

### Input Parameters

IHV INTEGER. Handle to the source H-vector AUVector.

IROW INTEGER. Index of the selected row of the H-matrix IHM (positive number).

### Input/Output Parameters

IHM INTEGER. Handle to the destination H-matrix AUMatrix.

### Output Parameters

ERROR Alternate return argument.

### Remarks

Dimension of the H-vector IHV should coincide with the number of columns of the H-matrix IHM.

## SUBROUTINE HUMCV( IHV, ICOL, IHM, *ERROR )

*Updates H-matrix column with H-vector*

### Input Parameters

IHV INTEGER. Handle to the source H-vector AUVector.

ICOL INTEGER. Index of the selected column of the H-matrix IHM (positive number).

### Input/Output Parameters

IHM INTEGER. Handle to the destination H-matrix AUMatrix.

### Output Parameters

ERROR Alternate return argument.

### Remarks

Dimension of the H-vector IHV should coincide with the number of rows of the H-matrix IHM.

## 4.10. Relational Operations

---

**SUBROUTINE HLEQL( ILH, IRH, LRES, *ERROR )**

*Logical **.EQ.** for generic H-objects*

### Input Parameters

ILH INTEGER. Handle to the left operand ANumber, AVector, or AMatrix.

IRH INTEGER. Handle to the right operand ANumber, AVector, or AMatrix.

### Output Parameters

LRES LOGICAL. Result of the operation.
LRES=.TRUE. – the H-object ILH is equal to H-object IRH.
LRES=.FALSE. – the H-object ILH is not equal to H-object IRH.

ERROR Alternate return argument.

### Remarks

Operands ILH and IRH must belong to the same generic kind, i.e. be descendants of the same parent class ANumber, AVector, or AMatrix. Senseless cross-kind comparisons result in run time error #102 “ILLEGAL TYPE OF OPERAND”.

---

**SUBROUTINE HLGNN( ILHN, IRHN, LRES, *ERROR )**

*Logical **.GT.** for real H-numbers*

### Input Parameters

ILHN INTEGER. Handle to the left operand AFReal.

IRHN INTEGER. Handle to the right operand AFReal.

### Output Parameters

LRES LOGICAL. Result of the operation.
LRES=.TRUE. – the H-number ILHN is greater than H-number IRHN.
LRES=.FALSE. – the H-number ILHN is less than or equal to H-number IRHN.

ERROR Alternate return argument.

**SUBROUTINE HLLNN( ILHN, IRHN, LRES, *ERROR )**

*Logical **.LT.** for real H-numbers*

### Input Parameters

ILHN INTEGER. Handle to the left operand AFReal.

IRHN INTEGER. Handle to the right operand AFReal.

### Output Parameters

LRES LOGICAL. Result of the operation.
LRES=.TRUE. – the H-number ILHN is less than H-number IRHN.
LRES=.FALSE. – the H-number ILHN is greater than or equal to H-number IRHN.

ERROR Alternate return argument.

## 4.11. Finding Maximum and Minimum Elements

**SUBROUTINE HGVG( IHV, INDEX, *ERROR )**

*Finds index of the greatest element of real H-vector*

### Input Parameters

IHV INTEGER. Handle to H-vector AUVectorReal.

### Output Parameters

INDEX INTEGER. Index of the greatest element of the H-vector IHV.

ERROR Alternate return argument.

**SUBROUTINE HGVL( IHV, INDEX, *ERROR )**

*Finds index of the lowest element of real H-vector*

### Input Parameters

IHV INTEGER. Handle to H-vector AUVectorReal.

### Output Parameters

INDEX INTEGER. Index of the lowest element of the H-vector IHV.

ERROR Alternate return argument.

## SUBROUTINE HGVG1( IHV, INDEX, *ERROR )

*Finds index of the greatest in octahedral norm element of H-vector*

### Input Parameters

IHV INTEGER. Handle to H-vector AUVector.

### Output Parameters

INDEX INTEGER. Index of the greatest in octahedral norm element of the H-vector IHV.

ERROR Alternate return argument.

### Remarks

The octahedral norm of a number $z$ is $|z|$ for real $z$, and $|\mathrm{Re}(z)|+|\mathrm{Im}(z)|$ for complex $z$.

## SUBROUTINE HGVL1( IHV, INDEX, *ERROR )

*Finds index of the lowest in octahedral norm element of H-vector*

### Input Parameters

IHV INTEGER. Handle to H-vector AUVector.

### Output Parameters

INDEX INTEGER. Index of the lowest in octahedral norm element of the H-vector IHV.

ERROR Alternate return argument.

### Remarks

The octahedral norm of a number $z$ is $|z|$ for real $z$, and $|\mathrm{Re}(z)|+|\mathrm{Im}(z)|$ for complex $z$.

## SUBROUTINE HGVG2( IHV, INDEX, *ERROR )

*Finds ndex of the greatest in Euclidian norm element of H-vector*

### Input Parameters

IHV INTEGER. Handle to H-vector AUVector.

### Output Parameters

INDEX INTEGER. Index of the greatest in Euclidian norm element of the H-vector IHV.

ERROR Alternate return argument.

### Remarks

The Euclidian norm of a number z is $|z|$ for real z, and $(\mathrm{Re}(z)^2+\mathrm{Im}(z)^2)^{1/2}$ for complex z.

---

## SUBROUTINE HGVL2( IHV, INDEX, *ERROR )

*Finds index of the lowest in Euclidian norm element of H-vector*

### Input Parameters

IHV INTEGER. Handle to H-vector AUVector.

### Output Parameters

INDEX INTEGER. Index of the lowest in Euclidian norm element of the H-vector IHV.

ERROR Alternate return argument.

### Remarks

The Euclidian norm of a number z is $|z|$ for real z, and $(\mathrm{Re}(z)^2+\mathrm{Im}(z)^2)^{1/2}$ for complex z.

---

## SUBROUTINE HGMRG( IHM, IROW, ICOL, *ERROR )

*Finds column index of the greatest element in row of real H-matrix*

### Input Parameters

IHM INTEGER. Handle to H-matrix AUMatrixReal.

IROW INTEGER. Index of the selected row of the H-matrix IHM (positive number).

### Output Parameters

ICOL INTEGER. Column index of the greatest element in the IROW-th row.

ERROR Alternate return argument.

---

## SUBROUTINE HGMRL( IHM, IROW, ICOL, *ERROR )

*Finds column index of the lowest element in row of real H-matrix*

### Input Parameters

IHM INTEGER. Handle to H-matrix AUMatrixReal.

IROW INTEGER. Index of the selected row of the H-matrix IHM (positive number).

### Output Parameters

ICOL INTEGER. Column index of the lowest element in the IROW-th row.

ERROR Alternate return argument.

## SUBROUTINE HGMRG1( IHM, IROW, ICOL, *ERROR )

*Finds column index of the greatest in octahedral norm element in H-matrix row*

### Input Parameters

IHM INTEGER. Handle to H-matrix AMatrix.

IROW INTEGER. Index of the selected row of the H-matrix IHM (positive number).

### Output Parameters

ICOL INTEGER. Column index of the greatest in octahedral norm element in the IROW-th row.

ERROR Alternate return argument.

### Remarks

The octahedral norm of a number $z$ is $|z|$ for real $z$, and $|\mathrm{Re}(z)|+|\mathrm{Im}(z)|$ for complex $z$.

## SUBROUTINE HGMRL1( IHM, IROW, ICOL, *ERROR )

*Finds column index of the lowest in octahedral norm element in H-matrix row*

### Input Parameters

IHM INTEGER. Handle to H-matrix AUMatrix.

IROW INTEGER. Index of the selected row of the H-matrix IHM (positive number).

### Output Parameters

ICOL INTEGER. Column index of the lowest in octahedral norm element in the IROW-th row.

ERROR Alternate return argument.

### Remarks

The octahedral norm of a number $z$ is $|z|$ for real $z$, and $|\mathrm{Re}(z)|+|\mathrm{Im}(z)|$ for complex $z$.

## SUBROUTINE HGMRG2( IHM, IROW, ICOL, *ERROR )

*Finds column index of the greatest in Euclidian norm element in H-matrix row*

### Input Parameters

IHM INTEGER. Handle to H-matrix AUMatrix.

IROW INTEGER. Index of the selected row of the H-matrix IHM (positive number).

### Output Parameters

ICOL INTEGER. Column index of the greatest in Euclidian norm element in the IROW-th row.

ERROR Alternate return argument.

### Remarks

The Euclidian norm of a number $z$ is $|z|$ for real $z$, and $(\mathrm{Re}(z)^2+\mathrm{Im}(z)^2)^{1/2}$ for complex $z$.

## SUBROUTINE HGMRL2( IHM, IROW, ICOL, *ERROR )

*Finds column index of the lowest in Euclidian norm element in H-matrix row*

### Input Parameters

IHM INTEGER. Handle to H-matrix AUMatrix.

IROW INTEGER. Index of the selected row of the H-matrix IHM (positive number).

### Output Parameters

ICOL INTEGER. Column index of the lowest in Euclidian norm element in the IROW-th row.

ERROR Alternate return argument.

### Remarks

The Euclidian norm of a number $z$ is $|z|$ for real $z$, and $(\mathrm{Re}(z)^2+\mathrm{Im}(z)^2)^{1/2}$ for complex $z$.

## SUBROUTINE HGMCG( IHM, ICOL, IROW, *ERROR )

*Finds row index of the greatest element in column of real H-matrix*

### Input Parameters

IHM INTEGER. Handle to H-matrix AUMatrixReal.

ICOL INTEGER. Index of the selected column of the H-matrix IHM (positive number).

### Output Parameters

IROW INTEGER. Row index of the greatest element in the ICOL-th column.

ERROR Alternate return argument.

### **SUBROUTINE HGMCL( IHM, ICOL, IROW, *ERROR )**

*Finds row index of the lowest element in column of real H-matrix*

#### Input Parameters

`IHM` `INTEGER.` Handle to H-matrix AUMatrixReal.

`ICOL` `INTEGER.` Index of the selected column of the H-matrix `IHM` (positive number).

#### Output Parameters

`IROW` `INTEGER.` Row index of the lowest element in the column `ICOL`.

`ERROR` Alternate return argument.

### **SUBROUTINE HGMCG1( IHM, ICOL, IROW, *ERROR )**

*Finds row index of the greatest in octahedral norm element in H-matrix column*

#### Input Parameters

`IHM` `INTEGER.` Handle to H-matrix AMatrix.

`ICOL` `INTEGER.` Index of the selected column of the H-matrix `IHM` (positive number).

#### Output Parameters

`IROW` `INTEGER.` Row index of the greatest in octahedral norm element in the `ICOL`-th column.

`ERROR` Alternate return argument.

#### Remarks

The octahedral norm of a number $z$ is $|z|$ for real $z$, and $|\mathrm{Re}(z)|+|\mathrm{Im}(z)|$ for complex $z$.

### **SUBROUTINE HGMCL1( IHM, ICOL, IROW, *ERROR )**

*Finds row index of the lowest in octahedral norm element in H-matrix column*

#### Input Parameters

`IHM` `INTEGER.` Handle to H-matrix AMatrix.

`ICOL` `INTEGER.` Index of the selected column of the H-matrix `IHM` (positive number).

#### Output Parameters

`IROW` `INTEGER.` Row index of the lowest in octahedral norm element in the `ICOL`-th column.

ERROR    Alternate return argument.

### Remarks

The octahedral norm of a number z is $|z|$ for real z, and $|\mathrm{Re}(z)|+|\mathrm{Im}(z)|$ for complex z.

## SUBROUTINE HGMCG2( IHM, ICOL, IROW, *ERROR )

*Finds row index of the greatest in Euclidian norm element in H-matrix column*

### Input Parameters

IHM    INTEGER. Handle to H-matrix AUMatrix.

ICOL    INTEGER. Index of the selected column of the H-matrix IHM (positive number).

### Output Parameters

IROW    INTEGER. Row index of the greatest in Euclidian norm element in the ICOL-th column.

ERROR    Alternate return argument.

### Remarks

The Euclidian norm of a number z is $|z|$ for real z, and $(\mathrm{Re}(z)^2+\mathrm{Im}(z)^2)^{1/2}$ for complex z.

## SUBROUTINE HGMCL2( IHM, ICOL, IROW, *ERROR )

*Finds row index of the lowest in Euclidian norm element in H-matrix column*

### Input Parameters

IHM    INTEGER. Handle to H-matrix AUMatrix.

ICOL    INTEGER. Index of the selected column of the H-matrix IHM (positive number).

### Output Parameters

IROW    INTEGER. Row index of the lowest in Euclidian norm element in the ICOL-th column.

ERROR    Alternate return argument.

### Remarks

The Euclidian norm of a number z is $|z|$ for real z, and $(\mathrm{Re}(z)^2+\mathrm{Im}(z)^2)^{1/2}$ for complex z.

## SUBROUTINE HGMG( IHM, IROW, ICOL, *ERROR )

*Finds indices of the greatest element of real H-matrix*

### Input Parameters

IHM INTEGER. Handle to H-matrix AUMatrixReal.

### Output Parameters

IROW INTEGER. Row index of the greatest element of the H-matrix IHM.

ICOL INTEGER. Column index of the greatest element of the H-matrix IHM.

ERROR Alternate return argument.

## SUBROUTINE HGML( IHM, IROW, ICOL, *ERROR )

*Finds indices of the lowest element of real H-matrix*

### Input Parameters

IHM INTEGER. Handle to H-matrix AUMatrixReal.

### Output Parameters

IROW INTEGER. Row index of the lowest element of the H-matrix IHM.

ICOL INTEGER. Column index of the lowest element of the H-matrix IHM.

ERROR Alternate return argument.

## SUBROUTINE HGMG1( IHM, IROW, ICOL, *ERROR )

*Finds indices of the greatest in octahedral norm element of H-matrix*

### Input Parameters

IHM INTEGER. Handle to H-matrix AUMatrix.

### Output Parameters

IROW INTEGER. Row index of the greatest in octahedral norm element of the H-matrix IHM.

ICOL INTEGER. Column index of the greatest in octahedral norm element of the H-matrix IHM.

ERROR Alternate return argument.

### Remarks

The octahedral norm of a number z is $|z|$ for real z, and $|\mathrm{Re}(z)|+|\mathrm{Im}(z)|$ for complex z.

## SUBROUTINE HGML1( IHM, IROW, ICOL, *ERROR )

*Finds indices of the lowest in octahedral norm element of H-matrix*

### Input Parameters

IHM INTEGER. Handle to H-matrix AUMatrix.

### Output Parameters

IROW INTEGER. Row index of the lowest in octahedral norm element of the H-matrix IHM.

ICOL INTEGER. Column index of the lowest in octahedral norm element of the H-matrix IHM.

ERROR Alternate return argument.

### Remarks

The octahedral norm of a number z is $|z|$ for real z, and $|\mathrm{Re}(z)|+|\mathrm{Im}(z)|$ for complex z.

## SUBROUTINE HGMG2( IHM, IROW, ICOL, *ERROR )

*Finds indices of the greatest in Euclidian norm element of H-matrix*

### Input Parameters

IHM INTEGER. Handle to H-matrix AUMatrix.

### Output Parameters

IROW INTEGER. Row index of the greatest in Euclidian norm element of the H-matrix IHM.

ICOL INTEGER. Column index of the greatest in Euclidian norm element of the H-matrix IHM.

ERROR Alternate return argument.

### Remarks

The Euclidian norm of a number z is $|z|$ for real z, and $(\mathrm{Re}(z)^2+\mathrm{Im}(z)^2)^{1/2}$ for complex z.

**SUBROUTINE HGML2( IHM, IROW, ICOL, *ERROR )**

*Finds indices of the lowest in Euclidian norm element of H-matrix*

**Input Parameters**

IHM INTEGER. Handle to H-matrix AUMatrix.

**Output Parameters**

IROW INTEGER. Row index of the lowest in Euclidian norm element of the H-matrix IHM.

ICOL INTEGER. Column index of the lowest in Euclidian norm element of the H-matrix IHM.

ERROR Alternate return argument.

**Remarks**

The Euclidian norm of a number $z$ is $|z|$ for real $z$, and $(\mathrm{Re}(z)^2+\mathrm{Im}(z)^2)^{1/2}$ for complex $z$.

## 4.12. Extracting Elements of H-Objects

**SUBROUTINE HERH( IH, IHRE, *ERROR )**

***Create&Assign** real part of H-object*

**Input Parameters**

IH INTEGER. Handle to H-object ANumber, AVector, or AMatrix.

**Output Parameters**

IHRE INTEGER. Handle to the new real H-object initialized with real part of the H-object IH.

ERROR Alternate return argument.

**Remarks**

Created object IHRE belongs to the same generic kind (ANumber, AVector, or AMatrix) as the input object IH.

## SUBROUTINE HEIH( IH, IHIM, *ERROR )

*__Create&Assign__ imaginary part of H-object*

### Input Parameters

IH INTEGER. Handle to H-object ANumber, AVector, or AMatrix.

### Output Parameters

IHIM INTEGER. Handle to the new real H-object initialized with imaginary part of the H-object IH.

ERROR Alternate return argument.

### Remarks

Created object IHIM belongs to the same generic kind (ANumber, AVector, or AMatrix) as the input object IH. If H-object IH is a descendant of AUMatrixSqHerm then **HEIH** represents its imaginary part IHIM as a corresponding descendant of AUMatrixSqGen.

## SUBROUTINE HENUMX( IHX, IHNUM, *ERROR )

*__Create&Assign__ integer numerator of exact H-number*

### Input Parameters

IHX INTEGER. Handle to H-number AFRealExact.

### Output Parameters

IHNUM INTEGER. Handle to the new H-number AFInteger initialized with numerator of the H-number IHX.

ERROR Alternate return argument.

## SUBROUTINE HEDENX( IHX, IHDEN, *ERROR )

*__Create&Assign__ integer denominator of exact H-number*

### Input Parameters

IHX INTEGER. Handle to H-number AFRealExact.

### Output Parameters

IHDEN INTEGER. Handle to the new H-number AFInteger initialized with denominator of the H-number IHX.

ERROR Alternate return argument.

## SUBROUTINE HEEV( IHV, INDEX, IHEV, *ERROR )

***Create&Assign** element of H-vector*

### Input Parameters

IHV INTEGER. Handle to H-vector AVector.

INDEX INTEGER. Index of the selected element of the H-vector IHV (positive number).

### Output Parameters

IHEV INTEGER. Handle to the new H-number AFFloat initialized with the INDEX-th element of the H-vector IHV.

ERROR Alternate return argument.

## SUBROUTINE HEREV( IHV, INDEX, IHEVRE, *ERROR )

***Create&Assign** real part of element of H-vector*

### Input Parameters

IHV INTEGER. Handle to H-vector AVector.

INDEX INTEGER. Index of the selected element of the H-vector IHV (positive number).

### Output Parameters

IHEVRE INTEGER. Handle to the new H-number AFRealFloat initialized with real part of the INDEX-th element of the H-vector IHV.

ERROR Alternate return argument.

## SUBROUTINE HEIEV( IHV, INDEX, IHEVIM, *ERROR )

***Create&Assign** imaginary part of element of H-vector*

### Input Parameters

IHV INTEGER. Handle to H-vector AVector.

INDEX INTEGER. Index of the selected element of the H-vector IHV (positive number).

### Output Parameters

IHEVIM INTEGER. Handle to the new H-number AFRealFloat initialized with imaginary part of the INDEX-th element of the H-vector IHV.

ERROR Alternate return argument.

### SUBROUTINE HEEM( IHM, IROW, ICOL, IHEM, *ERROR )

***Create&Assign** element of H-matrix*

#### Input Parameters

IHM — INTEGER. Handle to H-matrix AMatrix.

IROW — INTEGER. Row index of the selected element of the H-matrix IHM (positive number).

ICOL — INTEGER. Column index of the selected element of the H-matrix IHM (positive number).

#### Output Parameters

IHEM — INTEGER. Handle to the new H-number AFFloat initialized with the (IROW,ICOL)-th element of the H-matrix IHM.

ERROR — Alternate return argument.

### SUBROUTINE HEREM( IHM, IROW, ICOL, IHEMRE, *ERROR )

***Create&Assign** real part of element of H-matrix*

#### Input Parameters

IHM — INTEGER. Handle to H-matrix AMatrix.

IROW — INTEGER. Row index of the selected element of the H-matrix IHM (positive number).

ICOL — INTEGER. Column index of the selected element of the H-matrix IHM (positive number).

#### Output Parameters

IHEMRE — INTEGER. Handle to the new H-number AFRealFloat initialized with real part of the (IROW,ICOL)-th element of the H-matrix IHM.

ERROR — Alternate return argument.

### SUBROUTINE HEIEM( IHM, IROW, ICOL, IHEMIM, *ERROR )

***Create&Assign** imaginary part of element of H-matrix*

#### Input Parameters

IHM — INTEGER. Handle to H-matrix AMatrix.

IROW — INTEGER. Row index of the selected element of the H-matrix IHM (positive number).

ICOL INTEGER. Column index of the selected element of the H-matrix IHM (positive number).

### Output Parameters

IHEMIM INTEGER. Handle to the new H-number AFRealFloat initialized with imaginary part of the (IROW,ICOL)-th element of the H-matrix IHM.

ERROR Alternate return argument.

---

**SUBROUTINE HEVMR( IHM, IROW, IHV, *ERROR )**

***Create&Assign*** *H-matrix row*

### Input Parameters

IHV INTEGER. Handle to H-matrix AMatrix

IROW INTEGER. Index of the selected row of the H-matrix IHM (positive number).

### Output Parameters

IHV INTEGER. Handle to the new H-vector AUVector initialized with IROW-th row of the H-matrix IHM.

ERROR Alternate return argument.

---

**SUBROUTINE HEVMC( IHM, ICOL, IHV, *ERROR )**

***Create&Assign*** *H-matrix column*

### Input Parameters

IHV INTEGER. Handle to H-matrix AMatrix

ICOL INTEGER. Index of the selected column of the H-matrix IHM (positive number).

### Output Parameters

IHV INTEGER. Handle to the new H-vector AUVector initialized with ICOL-th column of the H-matrix IHM.

ERROR Alternate return argument.

## 4.13. Arithmetical Operations on H-objects

**SUBROUTINE HACPYH( IRH, ILH, *ERROR )**

***Create&Assign*** *copy of H-object (unary plus)*

### Input Parameters

IRH INTEGER. Handle to the initial H-object.

### Output Parameters

ILH INTEGER. Handle to the new copy of the H-object IRH.

ERROR Alternate return argument.

**SUBROUTINE HANEGH( IRH, ILH, *ERROR )**

***Create&Assign*** *negative of H-object (unary minus)*

### Input Parameters

IRH INTEGER. Handle to the initial H-object.

### Output Parameters

ILH INTEGER. Handle to the new H-object initialized with the negative of H-object IRH.

ERROR Alternate return argument.

**SUBROUTINE HACNJH( IRH, ILH, *ERROR )**

***Create&Assign*** *complex conjugate of H-object*

### Input Parameters

IRH INTEGER. Handle to the initial H-object.

### Output Parameters

ILH INTEGER. Handle to the new H-object initialized with the complex conjugate of H-object IRH.

ERROR Alternate return argument.

### SUBROUTINE HAABS( IRHN, ILHNX, *ERROR )

***Create&Assign** magnitude of H-number*

#### Input Parameters

IRHNX INTEGER. Handle to H-number ANumber.

#### Output Parameters

ILHNX INTEGER. Handle to the new positive H-number AReal initialized with absolute value of H-number IRHNX.

ERROR Alternate return argument.

### SUBROUTINE HAAHH( IRH1, IRH2, ILH, *ERROR )

***Create&Assign** addition of H-objects*

#### Input Parameters

IRH1 INTEGER. Handle to the first summand.

IRH2 INTEGER. Handle to the second summand.

#### Output Parameters

ILH INTEGER. Handle to the new H-object initialized with the result of the addition IRH1 and IRH2.

ERROR Alternate return argument.

#### Remarks

Operands IRH1 and IRH2 must belong to the same generic kind, i.e. be descendants of the same parent class ANumber, AVector, or AMatrix. Senseless cross-kind additions result in run-time error #102 “ILLEGAL TYPE OF OPERAND”. For the rules of selecting type of the resulting H-object ILH please refer to section 3.7.

### SUBROUTINE HASHH( IRH1, IRH2, ILH, *ERROR )

***Create&Assign** subtraction of H-objects*

#### Input Parameters

IRH1 INTEGER. Handle to the first operand (minuend).

IRH2 INTEGER. Handle to the second operand (subtrahend).

### Output Parameters

ILH INTEGER. Handle to the new H-object initialized with the result of the subtraction IRH2 from IRH1.

ERROR Alternate return argument.

### Remarks

Operands IRH1 and IRH2 must belong to the same generic kind, i.e. be descendants of the same parent class ANumber, AVector, or AMatrix. Senseless cross-kind subtractions result in run-time error #102 "ILLEGAL TYPE OF OPERAND". For the rules of selecting type of the resulting H-object ILH please refer to section 3.7.

---

## SUBROUTINE HAMHH( IRH1, IRH2, ILH, *ERROR )

***Create&Assign** multiplication of H-objects*

### Input Parameters

IRH1 INTEGER. Handle to the first factor.

IRH2 INTEGER. Handle to the second factor.

### Output Parameters

ILH INTEGER. Handle to the new H-object initialized with the result of multiplication.

ERROR Alternate return argument.

### Remarks

The table below represents the permissible combinations of types of the operands IRH1, IRH2 and the corresponding type of the resulting H-object ILH. Any other combinations of types result in run-time error #102 "ILLEGAL TYPE OF OPERAND".

| IRH1 | IRH2 | ILH |
|---|---|---|
| ANumber | ANumber | ANumber |
| AFinite | AUVector | AUVector |
| | AUMatrix | AUMatrix |
| AUVector | AFinite | AUVector |
| | AUVector | AFFloat |
| | AUMatrix | AUVector |
| | AUCompleteLU | AUVector |
| AUMatrix | AFinite | AUMatrix |
| | AUVector | AUVector |
| | AUMatrix | AUMatrix |
| | AUCompleteLU | AUMatrix |
| AUCompleteLU | AUVector | AUVector |
| | AUMatrix | AUMatrix |

For the rules of selecting type of the resulting H-object ILH please refer to section 3.7. Meaning of the left and right multiplications of H-vectors and H-matrices by H-objects AUCompleteLU is explained in section 3.8.

**SUBROUTINE HADHH( IRH1, IRH2, ILH, *ERROR )**

***Create&Assign** division of H-objects*

### Input Parameters

IRH1 INTEGER. Handle to the first operand (dividend).

IRH2 INTEGER. Handle to the second operand (divisor).

### Output Parameters

ILH INTEGER. Handle to the new H-object initialized with the result of division.

ERROR Alternate return argument.

### Remarks

A table below represents the permissible combinations of types of the operands IRH1, IRH2 and the corresponding type of resulting H-object ILH. Any other combinations of types of the operands result in run-time error #102 "ILLEGAL TYPE OF OPERAND".

| IRH1 | IRH2 | ILH |
|---|---|---|
| ANumber | ANumber | ANumber |
| AUVector | AFinite | AUVector |
| AUMatrix | AFinite | AUMatrix |

For the rules of selecting type of the resulting H-object ILH please refer to section 3.7.

**SUBROUTINE HADPHH( IRH1, IRH2, ILH, *ERROR )**

***Create&Assign** generalized conjugate dot product of H-objects*

### Input Parameters

IRH1 INTEGER. Handle to the first operand (factor).

IRH2 INTEGER. Handle to the second operand (factor).

### Output Parameters

ILH INTEGER. Handle to the new H-object initialized with result of the generalized conjugate dot product of H-objects IRH1 and IRH2.

ERROR Alternate return argument.

### Remarks

Generalized conjugate dot product implies that the first factor is to be transposed and complex conjugated when performing multiplication.

- For numbers a (the first operand) and b (the second operand) the result is a·b.
- For vectors **a** (the first operand) and **b** (the second operand) the result is **(a,b) = ? $\mathbf{a}_i$·$\mathbf{b}_i$**.
- For matrices **A** (the first operand) and **B** (the second operand) the result is **A**¯·**B**, where ¯ denotes Hermitian conjugation.

Operands IRH1 and IRH2 must belong to the same generic kind, i.e. be descendants of the same parent class ANumber, AVector, or AMatrix. Senseless cross-kind operations result in run-time error #102 "ILLEGAL TYPE OF OPERAND". For the rules of selecting type of the resulting H-object ILH please refer to section 3.7.

---

## SUBROUTINE HUAHH( IRH, ILH, *ERROR )

***Update** addition of H-objects*

### Input Parameters

IRH INTEGER. Handle to the unchangeable summand.

### Input/Output Parameters

ILH INTEGER. Handle to the updated floating-point summand.

ERROR Alternate return argument.

### Remarks

The table below represents the permissible combination of types of the operands ILH and IRH. Any other combinations of types result in run-time error #102 "ILLEGAL TYPE OF OPERAND".

| ILH | IRH |
|---|---|
| AFFloat | AFinite |
| AUVector | AUVector |
| AUMatrix | AUMatrix |

---

## SUBROUTINE HUSHH( IRH, ILH, *ERROR )

***Update** subtraction of H-objects*

### Input Parameters

IRH INTEGER. Handle to the unchangeable subtrahend.

### Input/Output Parameters

ILH INTEGER. Handle to the floating-point minuend.

ERROR Alternate return argument.

### Remarks

The table below represents the permissible combinations of operands ILH and IRH. Any other combinations of types result in run-time error #102 "ILLEGAL TYPE OF OPERAND".

| ILH | IRH |
|---|---|
| AFFloat | AFinite |
| AUVector | AUVector |
| AUMatrix | AUMatrix |

## SUBROUTINE HUMHH( ILH, IRH, SIDE, *ERROR )

***Update** multiplication of H-objects*

### Input Parameters

ILH INTEGER. Handle to the first factor.

IRH INTEGER. Handle to the second factor.

SIDE CHARACTER*1. The text descriptor that defines which operand is updated:
SIDE = 'L' - H-object ILH is to be updated with the product;.
SIDE = 'R' - H-object IRH is to be updated with the product.

### Input/Output Parameters

ILH or IRH INTEGER. Handle to the updated floating-point factor.

ERROR Alternate return argument.

### Remarks

The table below represents the permissible combinations of types of the operands ILH and IRH. Any other combinations of types result in run-time error #102 "ILLEGAL TYPE OF OPERAND".

| Updated Operand | Unchangeable Operand |
|---|---|
| AFFloat | AFinite |
| AUVector | AFinite |
| | AUMatrixSq |
| | AUCompleteLU |
| AUMatrix | AFinite |
| | AUMatrixSq |
| | AUCompleteLU |

Meaning of the left and right multiplications of H-vectors and H-matrices by H-objects AUCompleteLU is explained in section 3.8.

## SUBROUTINE HUDHH( IRH, ILH, *ERROR )

***Update** division of H-objects*

### Input Parameters

IRH INTEGER. Handle to the unchangeable dividend.

### Input/Output Parameters

ILH INTEGER. Handle to the updated floating-point divisor.

ERROR Alternate return argument.

### Remarks

The table below represents the permissible combinations of types of the operands ILH and IRH. Any other combinations of types result in run-time error #102 "ILLEGAL TYPE OF OPERAND".

| ILH | IRH |
|---|---|
| AFFloat | AFinite |
| AUVector | AFinite |
| AUMatrix | AFinite |

## SUBROUTINE HUNEGH( IH, *ERROR )

***Update** with negative of H-object (unary minus)*

### Input/Output Parameters

IH INTEGER. Handle the H-object AFFloat, AUVector, or AUMatrix that takes negative of its initial value.

ERROR Alternate return argument.

## SUBROUTINE HUCNJH( IH, *ERROR )

***Update** with complex conjugate of H-object*

### Input/Output Parameters

IH INTEGER. Handle to the H-object AFFloat, AUVector, or AUMatrix that takes complex conjugate of its initial value.

ERROR Alternate return argument.

## 4.14. Mixed-Type Operations with Fortran Operands

**SUBROUTINE HAANF( FTYPE, FVAR, IRH, ILH, *ERROR )**

***Create&Assign** addition of Fortran variable to H-number*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR The summand represented by Fortran variable.

IRH INTEGER. Handle to the summand ANumber.

### Output Parameters

ILH INTEGER. Handle to the new H-number ANumber initialized with sum of the H-number IRH and the variable FVAR.

ERROR Alternate return argument.

**SUBROUTINE HASNF( FTYPE, FVAR, IRH, ILH, *ERROR )**

***Create&Assign** subtraction of Fortran variable from H-number*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR The subtrahend represented by Fortran variable.

IRH INTEGER. Handle to the minuend ANumber.

### Output Parameters

ILH INTEGER. Handle to the new H-number ANumber initialized with difference of the H-number IRH and the variable FVAR.

ERROR Alternate return argument.

**SUBROUTINE HAMHF( FTYPE, FVAR, IRH, ILH, *ERROR )**

***Create&Assign** multiplication of H-object by Fortran variable*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR The factor represented by Fortran variable.

IRH INTEGER. Handle to the factor H-object ANumber, AUVector, or AUMatrix.

### Output Parameters

ILH INTEGER. Handle to the new H-object initialized with product of the H-object IRH by the variable FVAR

ERROR Alternate return argument.

### Remarks

The resulting H-object ILH belongs to the same generic kind as the input H-object IRH, i.e. it is a descendant of the same parent class ANumber, AUVector, or AUMatrix.

---

## SUBROUTINE HADHF( FTYPE, FVAR, IRH, ILH, *ERROR )

***Create&Assign*** *division of H-object by Fortran variable*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR The divisor represented by Fortran variable.

IRH INTEGER. Handle to the dividend H-object ANumber, AUVector, or AUMatrix.

### Output Parameters

ILH INTEGER. Handle to the new H-object initialized with quotient of division of the H-object IRH by the variable FVAR.

ERROR Alternate return argument.

### Remarks

The resulting H-object ILH belongs to the same generic kind as the input H-object IRH, i.e. it is a descendant of the same parent class ANumber, AUVector, or AUMatrix.

---

## SUBROUTINE HUANF( FTYPE, FVAR, IH, *ERROR )

***Update*** *addition of Fortran variable to floating-point H-number*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR The unchangeable summand represented by Fortran variable.

### Input/Output Parameters

IH INTEGER. Handle to the H-number AFFloat that takes value of sum of its initial value and the variable FVAR.

### Output Parameters

ERROR Alternate return argument.

---

**SUBROUTINE HUSNF( FTYPE, FVAR, IH, *ERROR )**

***Update** subtraction of Fortran variable from floating-point H-number*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR The unchangeable subtrahend represented by Fortran variable.

### Input/Output Parameters

IH INTEGER. Handle to the H-number AFFloat that takes value of difference of its initial value and the variable FVAR.

### Output Parameters

ERROR Alternate return argument.

---

**SUBROUTINE HUMHF( FTYPE, FVAR, IH, *ERROR )**

***Update** multiplication of floating-point H-object by Fortran variable*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR The unchangeable factor represented by Fortran variable.

### Input/Output Parameters

IH INTEGER. Handle to the H-object AFFloat, AUVector, or AUMatrix that takes value of product of its initial value by the variable FVAR.

### Output Parameters

ERROR Alternate return argument.

---

**SUBROUTINE HUDHF( FTYPE, FVAR, IH, *ERROR )**

***Update** division of floating-point H-object by Fortran variable*

### Input Parameters

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

FVAR The unchangeable divisor represented by Fortran variable.

### Input/Output Parameters

IH INTEGER. Handle to the H-object AFFloat, AUVector, or AUMatrix that takes value of quotient of division of its initial value by the variable FVAR.

### Output Parameters

ERROR Alternate return argument.

.

## 4.15. Math Constants and Functions

**SUBROUTINE HCPI( NBIT, IHPI, *ERROR )**

***Create&Assign*** *constant p*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

### Output Parameters

IHPI INTEGER. Handle to the new H-number AFRealFloat initialized with the NBIT-accurate floating-point approximation of p = 3.1415926535897932…

ERROR Alternate return argument.

**SUBROUTINE HCE( NBIT, IHE, *ERROR )**

***Create&Assign*** *constant **e***

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

### Output Parameters

IHE INTEGER. Handle to the new H-number AFRealFloat initialized with the NBIT-accurate floating-point approximation of **e** = 2.718281828459045…

ERROR Alternate return argument.

## SUBROUTINE HCLN2( NBIT, IHLN2, *ERROR )

***Create&Assign** constant **ln2***

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

### Output Parameters

IH INTEGER. Handle to the new H-number AFRealFloat initialized with the NBIT-accurate floating-point approximation of **ln2** = 0.69314718055994531…

ERROR Alternate return argument.

## SUBROUTINE HFSQRT( NBIT, IHNX, IH, *ERROR )

***Create&Assign** square root of H-number*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

IHNX INTEGER. Handle to argument ANumber.

### Output Parameters

IH INTEGER. Handle to the new H-number AFFloat initialized with the NBIT-accurate value of square root of the H-number IHNX.

ERROR Alternate return argument.

### Exceptions

If an infinite H-number is passed as the input parameter IHNX then **HFSQRT** produces the following results:

| Argument IHNX | Result IH |
|---|---|
| CInfUnsigned = INF | CInfUnsigned = INF |
| Positive CInfSigned = +INF | Positive CInfSigned = +INF |
| Negative CInfSigned = -INF | CInfUnsigned = INF |

### Remarks

The branch cut is on the real axis less than 0.

## SUBROUTINE HFEXP( NBIT, IHNX, IH, *ERROR )

***Create&Assign** exponential function of H-number*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

IHNX INTEGER. Handle to argument ANumber.

### Output Parameters

IH INTEGER. Handle to the new H-number AFFloat initialized with the NBIT-accurate value of exponential function of the H-number IHNX.

ERROR Alternate return argument.

### Exceptions

If an infinite H-number is passed as the input parameter IHNX then **HFEXP** produces the following results:

| Argument IHNX | Result IH |
|---|---|
| CInfUnsigned = INF | Run-time error #0609 "FUNCTION DOES NOT HAVE A LIMIT" |
| Positive CInfSigned = +INF | Positive CInfSigned = +INF |
| Negative CInfSigned = -INF | Zero AFRealFloat = 0 |

natural

## SUBROUTINE HFLN( NBIT, IHNX, IH, *ERROR )

***Create&Assign** natural logarithm of H-number)*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

IHNX INTEGER. Handle to argument ANumber.

### Output Parameters

IH INTEGER. Handle to the new H-number AFFloat initialized with the NBIT-accurate value of natural logarithm of the H-number IHNX.

ERROR Alternate return argument.

### Exceptions

If a zero or infinite H-number is passed as the input parameter IHNX then **HFLN** produces the following results:

| Argument **IHNX** | Result **IH** |
|---|---|
| Zero AFinite = 0 | CInfUnsigned = INF |
| CInfUnsigned = INF | CInfUnsigned = INF |
| Positive CInfSigned = +INF | Positive CInfSigned = +INF |
| Negative CInfSigned = -INF | CInfUnsigned = INF |

### Remarks

The branch cut is on the real axis less than 0.

---

## SUBROUTINE HFSIN( NBIT, IHNX, IH, *ERROR )

***Create&Assign** sine of H-number*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

IHNX INTEGER. Handle to argument ANumber.

### Output Parameters

IH INTEGER. Handle to the new H-number AFFloat initialized with the NBIT-accurate value of sine of the H-number IHNX.

ERROR Alternate return argument.

### Exceptions

If an infinite H-number is passed as the input parameter IHNX then **HFSIN** generates run-time error #0609 “FUNCTION DOES NOT HAVE A LIMIT”.

---

## SUBROUTINE HFCOS( NBIT, IHNX, IH, *ERROR )

***Create&Assign** cosine of H-number*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

IHNX INTEGER. Handle to argument ANumber.

### Output Parameters

IH INTEGER. Handle to the new H-number AFFloat initialized with the NBIT-accurate value of cosine of the H-number IHNX.

ERROR Alternate return argument.

### Exceptions

If an infinite H-number is passed as the input parameter IHNX then **HFCOS** generates run-time error #0609 “FUNCTION DOES NOT HAVE A LIMIT”.

---

## SUBROUTINE HFTAN( NBIT, IHNX, IH, *ERROR )

***Create&Assign** tangent of H-number*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

IHNX INTEGER. Handle to argument ANumber.

### Output Parameters

IH INTEGER. Handle to the new H-number AFFloat initialized with the NBIT-accurate value of tangent of the H-number IHNX.

ERROR Alternate return argument.

### Exceptions

If an infinite H-number is passed as the input parameter IHNX then **HFTAN** generates run-time error #0609 “FUNCTION DOES NOT HAVE A LIMIT”.

---

## SUBROUTINE HFSINH( NBIT, IHNX, IH, *ERROR )

***Create&Assign** hyperbolic sine of H-number*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

IHNX INTEGER. Handle to argument ANumber.

### Output Parameters

IH INTEGER. Handle to the new H-number AFFloat initialized with the NBIT-accurate value of hyperbolic sine of the H-number IHNX.

ERROR Alternate return argument.

### Exceptions

If an infinite H-number is passed as the input parameter IHNX then **HFSINH** produces the following results:

| Argument IHNX | Result IH |
|---|---|
| CInfUnsigned = INF | Run-time error #0609 "FUNCTION DOES NOT HAVE A LIMIT" |
| Positive CInfSigned = +INF | Positive CInfSigned = +INF |
| Negative CInfSigned = -INF | Negative CInfSigned = -INF |

## SUBROUTINE HFCOSH( NBIT, IHNX, IH, *ERROR )

***Create&Assign** hyperbolic cosine of H-number*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

IHNX INTEGER. Handle to argument ANumber.

### Output Parameters

IH INTEGER. Handle to the new H-number AFFloat initialized with the NBIT-accurate value of hyperbolic cosine of the H-number IHNX.

ERROR Alternate return argument.

### Exceptions

If an infinite H-number is passed as the input parameter IHNX then **HFCOSH** produces the following results:

| Argument IHNX | Result IH |
|---|---|
| CInfUnsigned = INF | Run-time error #0609 "FUNCTION DOES NOT HAVE A LIMIT" |
| Positive CInfSigned = +INF | Positive CInfSigned = +INF |
| Negative CInfSigned = -INF | Positive CInfSigned = +INF |

## SUBROUTINE HFTANH( NBIT, IHNX, IH, *ERROR )

***Create&Assign** hyperbolic tangent of H-number*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

IHNX INTEGER. Handle to argument ANumber.

### Output Parameters

IH INTEGER. Handle to the new H-number AFFloat initialized with the NBIT-accurate value of hyperbolic tangent of the H-number IHNX.

ERROR Alternate return argument.

### Exceptions

If an infinite H-number is passed as the input parameter IHNX then **HFTANH** produces the following results:

| Argument IHNX | Result IH |
|---|---|
| CInfUnsigned = INF | Run-time error #0609 "FUNCTION DOES NOT HAVE A LIMIT" |
| Positive CInfSigned = +INF | AFRealFloat = +1 |
| Negative CInfSigned = -INF | AFRealFloat = -1 |

## SUBROUTINE HFASIN( NBIT, IHNX, IH, *ERROR )

***Create&Assign*** *arcsine of H-number*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

IHNX INTEGER. Handle to argument ANumber.

### Parameters

IH INTEGER. Handle to the new H-number AFFloat initialized with the NBIT-accurate value of arcsine of the H-number IHNX.

ERROR Alternate return argument.

### Exceptions

If an infinite H-number is passed as the input parameter IHNX then **HFASIN** produces the following results:

| Argument IHNX | Result IH |
|---|---|
| CInfUnsigned = INF | CInfUnsigned = INF |
| Positive CInfSigned = +INF | CInfUnsigned = INF |
| Negative CInfSigned = -INF | CInfUnsigned = INF |

### Remarks

The branch cuts are on the real axis, less than -1 and greater than +1.

## SUBROUTINE HFACOS( NBIT, IHNX, IH, *ERROR )

***Create&Assign*** *arc-cosine of H-number*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

IHNX INTEGER. Handle to argument ANumber.

### Output Parameters

IH INTEGER. Handle to the new H-number AFFloat initialized with the NBIT-accurate value of arc-cosine of the H-number IHNX.

ERROR Alternate return argument.

### Exceptions

If an infinite H-number is passed as the input parameter IHNX then **HFACOS** produces the following results:

| Argument **IHNX** | Result **IH** |
|---|---|
| CInfUnsigned = INF | Run-time error #0609 "FUNCTION DOES NOT HAVE A LIMIT" |
| Positive CInfSigned = +INF | CInfUnsigned = INF |
| Negative CInfSigned = -INF | CInfUnsigned = INF |

### Remarks

The branch cuts are on the real axis, less than -1 and greater than +1.

## SUBROUTINE HFATAN( NBIT, IHNX, IH, *ERROR )

***Create&Assign*** *arc-tangent of H-number*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

IHNX INTEGER. Handle to argument ANumber.

### Output Parameters

IH INTEGER. Handle to the new H-number AFFloat initialized with the NBIT-accurate value of arc-tangent of the H-number IHNX.

ERROR Alternate return argument.

### Exceptions

If an infinite H-number is passed as the input parameter IHNX then **HFATAN** produces the following results:

| Argument IHNX | Result IH |
|---|---|
| CInfUnsigned = INF | Run-time error #0609 "FUNCTION DOES NOT HAVE A LIMIT" |
| Positive CInfSigned = +INF | AFRealFloat = $\pi/2$ |
| Negative CInfSigned = -INF | AFRealFloat = $-\pi/2$ |

### Remarks

The branch cuts are on the imaginary axis, below $-\boldsymbol{i}$ and above $+\boldsymbol{i}$.

## SUBROUTINE HFATN2( NBIT, IHNX1, IHNX2, IH, *ERROR )

***Create&Assign** arc-tangent of two real H-number arguments*

### Input Parameters

NBIT — INTEGER. Required number of correct significant bits in the result (positive number).

IHNX1 — INTEGER. Handle to the first argument AReal.

IHNX2 — INTEGER. Handle to the second argument AReal.

### Output Parameters

IH — INTEGER. Handle to the new H-number AFReal initialized with the NBIT-accurate value of arc-tangent of the arguments IHNX1,. IHNX2.

ERROR — Alternate return argument.

### Exceptions

If zero or infinite H-number are passed as one of or both input parameters IHNX1, IHNX2 then **HFATN2** produces the following results:

| Argument IHNX1 | Argument IHNX2 | Result IH |
|---|---|---|
| Zero AFReal = 0 | Zero AFReal = 0 | Run-time error #0609 "FUNCTION DOES NOT HAVE A LIMIT" |
| CInfSigned = ±INF | CInfSigned = ±INF | Run-time error #0609 "FUNCTION DOES NOT HAVE A LIMIT" |
| Positive CInfSigned = +INF | Any AFReal | AFRealFloat = $\pi/2$ |
| Negative CInfSigned = -INF | Any AFReal | AFRealFloat = $-\pi/2$ |
| Any AFReal | Positive CInfSigned = +INF | AFRealFloat = 0 |
| Any AFReal $\geq 0$ | Negative CInfSigned = -INF | AFRealFloat = $\pi$ |

| Argument IHNX1 | Argument IHNX2 | Result IH |
|---|---|---|
| Any AFReal < 0 | Negative CInfSigned = -INF | AFRealFloat = -π |

### Remarks

**HFATN2** has exactly the same mathematical sense as the Fortran intrinsic function ATAN2(Y,X)= arctan(Y/X), whose resulting values belong to the half-interval (-π,π].

## SUBROUTINE HFASNH( NBIT, IHNX, IH, *ERROR )

***Create&Assign** hyperbolic arcsine of H-number*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

IHNX INTEGER. Handle to argument ANumber.

### Output Parameters

IH INTEGER. Handle to the new H-number AFFloat initialized with the NBIT-accurate value of hyperbolic arcsine of the H-number IHNX.

ERROR Alternate return argument.

### Exceptions

If an infinite H-number is passed as the input parameter IHNX then **HFASNH** produces the following results:

| Argument IHNX | Result IH |
|---|---|
| CInfUnsigned = INF | CInfUnsigned = INF |
| Positive CInfSigned = +INF | Positive CInfSigned = +INF |
| Negative CInfSigned = -INF | Negative CInfSigned = -INF |

### Remarks

The branch cuts are on the imaginary axis, below $-i$ and above $+i$.

## SUBROUTINE HFACSH( NBIT, IHNX, IH, *ERROR )

***Create&Assign** hyperbolic arc-cosine of H-number*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

IHNX INTEGER. Handle to argument ANumber.

### Output Parameters

IH INTEGER. Handle to the new H-number AFFloat initialized with the NBIT-accurate value of hyperbolic arc-cosine of the H-number IHNX.

ERROR Alternate return argument.

### Exceptions

If an infinite H-number is passed as the input parameter IHNX then **HFACSH** produces the following results:

| Argument IHNX | Result IH |
|---|---|
| CInfUnsigned = INF | CInfUnsigned = INF |
| Positive CInfSigned = +INF | Positive CInfSigned = +INF |
| Negative CInfSigned = -INF | CInfUnsigned = INF |

### Remarks

The branch cut is on the real axis less than +1.

---

**SUBROUTINE HFATNH( NBIT, IHNX, IH, *ERROR )**

***Create&Assign** hyperbolic arc-tangent of H-number*

### Input Parameters

NBIT INTEGER. Required number of correct significant bits in the result (positive number).

IHNX INTEGER. Handle to argument ANumber.

### Output Parameters

IH INTEGER. Handle to the new H-number AFFloat initialized with the NBIT-accurate value of hyperbolic arc-tangent t of the H-number IHNX.

ERROR Alternate return argument.

### Exceptions

If ±1 or an ±infinite H-number is passed as the input parameter IHNX then **HFATNH** produces the following results:

| Argument IHNX | Result IH |
|---|---|
| AFinite = +1 | Positive CInfSigned = +INF |
| AFinite = -1 | Negative CInfSigned = -INF |
| CInfUnsigned = INF | Run-time error #0609 “FUNCTION DOES NOT HAVE A LIMIT” |

| Argument IHNX | Result IH |
|---|---|
| Positive CInfSigned = +INF | Run-time error #0609 “FUNCTION DOES NOT HAVE A LIMIT” |
| Negative CInfSigned = -INF | Run-time error #0609 “FUNCTION DOES NOT HAVE A LIMIT” |

### Remarks

The branch cuts are on the real axis, less than -1 and greater than +1.

## 4.16. Miscellaneous Numerical Operations

**SUBROUTINE HUMAXN( IHN, *ERROR )**

*Updates floating-point H-number with maximum value*

### Input/Output Parameters

IHN INTEGER. Handle to the H-number AFRealFloat that takes the maximum representable value.

ERROR Alternate return argument.

### Remarks

The maximum representable value depends on the sizes of mantissa and exponent fields for the particular kind of the floating-point H-number IHN. Maximum values of H-numbers CFReal4 and CFReal8 are equal to FLT_MAX= 3.402823466E+38 and .DBL_MAX= 1.7976931348623158D+308 respectively.

**SUBROUTINE HUMINN( IHN, *ERROR )**

*Updates floating-point H-number with minimum value*

### Input/Output Parameters

IHN INTEGER. Handle to the H-number AFRealFloat that takes the minimum representable positive value.

ERROR Alternate return argument.

### Remarks

The minimum representable positive value depends on the sizes of mantissa and exponent fields for the particular kind of the floating-point H-number IHN. Minimum values of H-numbers CFReal4 and CFReal8 are equal to FLT_MIN= 1.175494351E-38 and .DBL_MIN= 2.2250738585072014D-308 respectively.

**SUBROUTINE HUEPSN( IHN, *ERROR )**

***Updates** floating-point H-number with "machine epsilon" value*

### Input/Output Parameters

IHN — INTEGER. Handle to the H-number AFRealFloat that takes the "machine epsilon" value, i.e. the smallest positive value EPS such that 1.0 + EPS is not equal to 1.0.

ERROR — Alternate return argument.

### Remarks

The "machine epsilon" value depends on the sizes of mantissa and exponent fields for the particular kind of the floating-point H-number IHN. Machine epsilons for H-numbers CFReal4 and CFReal8 are equal to FLT_EPSILON= 1.192092896E-07 and .DBL_EPSILON= 2.2204460492503131D-016 respectively.

**SUBROUTINE HFQTXX( IRHX1, IRHX2, IHX, *ERROR )**

***Create&Assign** integer quotient of division of exact H-numbers*

### Input Parameters

IRHX1 — INTEGER. Handle to dividend AFRealExact.

IRHX2 — INTEGER. Handle to divisor AFRealExact.

### Output Parameters

IHX — INTEGER. Handle to the new H-number AFInteger initialized with integer quotient of division of the H-number IRHX1 by IRHX2.

ERROR — Alternate return argument.

### Remarks

**HFQTXX** computes the nearest to zero integer approximation of the quotient, thus implying that the reminder of division has the same sign as numerator IRHX1.

**SUBROUTINE HFRMXX( IRHX1, IRHX2, IHX, *ERROR )**

***Create&Assign** reminder of division of exact H-numbers*

### Input Parameters

IRHX1 — INTEGER. Handle to dividend AFRealExact.

IRHX2 — INTEGER. Handle to divisor AFRealExact.

### Output Parameters

IHX INTEGER. Handle to the new H-number AFRealExact initialized with reminder of division of the H-number IRHX1 by IRHX2.

ERROR Alternate return argument.

### Remarks

Reminder of division computed by **HFRMXX** has the same sign as numerator IRHX1.

---

**SUBROUTINE HFFACT( IHX, IH, *ERROR )**

***Create&Assign** factorial of integer H-number*

### Input Parameters

IHX INTEGER. Handle to H-number CFInteger4.

### Output Parameters

IH INTEGER. Handle to the new H-number ANumber initialized with factorial of the H-number IHX.

ERROR Alternate return argument.

### Remarks

For negative values of the argument IHX **HFFACT** outputs H-objects CInfUnsigned.

---

**SUBROUTINE HAPWR2( IRH, POWER, ILH, *ERROR )**

***Create&Assign** product of H-number by integer power of 2*

### Input Parameters

IRH INTEGER. Handle to H-number ANumber.

POWER INTEGER. The power of 2.

### Output Parameters

ILH INTEGER. Handle to the new H-number ANumber initialized with the product of H-number IRH by 2**POWER.

ERROR Alternate return argument.

---

**SUBROUTINE HUPWR2( POWER, IH, *ERROR )**

***Update**s floating-point H-number with its product by integer power of 2*

### Input Parameters

POWER INTEGER. The power of 2.

### Input/Output Parameters

IH INTEGER. Handle to the H-number AFFloat that takes the value of product of its initial value by 2**POWER.

### Output Parameters

ERROR Alternate return argument.

---

**SUBROUTINE HUSQRT( IHN, *ERROR )**

*Updates floating-point H-number with its square root*

### Input/Output Parameters

IHN INTEGER. Handle to the H-number AFFloat that takes the value of square root of its initial value.

### Output Parameters

ERROR Alternate return argument.

### Remarks

If a negative H-number AFRealFloat is passed .as the input parameter IHN then **HUSQRT** generates run-time error #404 "UPDATE OPERATION FAILURE".

## 4.17. Linear Equations

---

**SUBROUTINE HUCLU( IH, *ERROR )**

*Performs complete LU decomposition of square H-matrix*

### Input/Output Parameters

IH INTEGER. Input value of IH should be a handle to H-matrix AUMatrixSq. Its output value is a handle to the corresponding H-object AUCompleteLU that contains triangular factor(s) of the original matrix.

### Output Parameters

ERROR Alternate return argument.

### Remarks

.To solve system of algebraic linear equations with a given RHS H-vector or H-matrix one should perform *Create&Assign* or *Update* multiplication of the RHS H-object by H-object AUCompleteLU using subroutines **HAMHH** or **HUMM** respectively. For details of the procedure, please refer to the section 3.8.

## 4.18. Linear Eigenvalue Problems

---

**SUBROUTINE HUHES( IH, *ERROR )**

*Transforms square H-matrix to Hessenberg form*

### Input/Output Parameters

IH — INTEGER. Input value of IH should be a handle to H-matrix AUMatrixSq. Its output value is a handle to the corresponding H-object AUHessenberg that contains Hessenberg form of the original matrix, matrix of transformation, and permutation vector.

### Output Parameters

ERROR — Alternate return argument.

### Remarks

**HUHES** overwrites the original matrix with its Hessenberg form. To solve a linear eigenvalue problem one should use the described below subroutine **HUEIG** that accepts an H-object AUHessenberg as input parameter.

---

**SUBROUTINE HUEIG( IH, IHV, *ERROR )**

*Solves linear eigenvalue problem*

### Input/Output Parameters

IH — INTEGER. Input value of IH should be a handle to H-object AUHessenberg. Its output value is a handle to the corresponding H-matrix AUMatrix composed of the computed column eigenvectors.

### Output Parameters

IH — INTEGER. Handle to the new H-vector AUVector composed of the computed eigenvalues.

ERROR — Alternate return argument.

### Remarks

**HUEIG** overwrites the original Hessenberg matrix with the computed matrix of eigenvectors. To transform a square H-matrix to its Hessenberg form one should use the described above subroutine **HUHES**.

## 4.19. I/O Binary Operations

---

**SUBROUTINE HWRITE( WCBACK, IH, *ERROR )**

*Writes H-object to binary file*

### Input Parameters

| | |
|---|---|
| `WCBACK` | Name of the Fortran callback subroutine. |
| `IH` | Handle to the H-object to be written. |

### Output Parameters

| | |
|---|---|
| `ERROR` | Alternate return argument. |

### Remarks

See section 3.6 for details of binary I/O operations and specifications of the callback subroutine `WCBACK`.

---

**SUBROUTINE HREAD( RCBACK, NSIZE, IH, *ERROR )**

*Reads H-object from binary file*

### Input Parameters

| | |
|---|---|
| `RCBACK` | Name of the Fortran callback subroutine. |
| `NSIZE` | `INTEGER.` The size of H-object in 32-bit words (positive number). |

### Output Parameters

| | |
|---|---|
| `IH` | `INTEGER.` Handle to the new H-object. |
| `ERROR` | Alternate return argument. |

### Remarks

See section 3.6 for details of binary I/O operations and specifications of the callback subroutine `RCBACK`.

## 4.20. Text Output

**SUBROUTINE HETNX0( IHNX, STR, *ERROR )**

*Converts H-number to unformatted text string*

### Input Parameters

IHNX INTEGER. Handle to H-number ANumber.

### Output Parameters

STR CHARACTER*. Unformatted text representation of the H-number IHNX.

ERROR Alternate return argument.

### Remarks

See section 3.5 for details of unformatted text output.

**SUBROUTINE HETEV0( IHV, INDEX, STR, *ERROR )**

*Converts element of H-vector to unformatted text string*

### Input Parameters

IHV INTEGER. Handle to H-vector AVector.

INDEX INTEGER. Index of the selected element of H-vector IHV (positive number).

### Output Parameters

STR CHARACTER*. Unformatted text representation of the INDEX-th element of H-vector IHV.

ERROR Alternate return argument.

### Remarks

See section 3.5 for details of unformatted text output.

**SUBROUTINE HETEM0( IHM, IROW, ICOL, STR, *ERROR )**

*Converts element of H-matrix to unformatted text string*

### Input Parameters

IHM INTEGER. Handle to H-matrix AMatrix.

IROW INTEGER. Row index of the selected element of H-matrix IHM (positive number).

ICOL INTEGER. Column index of the selected element of H-matrix IHM (positive number).

### Output Parameters

STR CHARACTER*. Unformatted text representation of the (IROW,ICOL)-th element of H-matrix IHM.

ERROR Alternate return argument.

### Remarks

See section 3.5 for details of unformatted text output.

---

## SUBROUTINE HETNX( IHNX, IW, IP, IM, IE, STR, *ERROR )

*Converts H-number to formatted text string*

### Input Parameters

IHNX INTEGER. Handle to H-number ANumber.

IW INTEGER. Full width of the output field (positive number).

IP INTEGER. Format parameter that specifies position of decimal point in the floating-point H-numbers AFFloat, or position of separating slash in the rational H-numbers CFRational.

IM INTEGER. Number of decimal digits of mantissa of the floating-point H-numbers AFFloat (positive number).

IE INTEGER. Number of decimal digits of exponent of the floating-point H-numbers AFFloat (positive number).

### Output Parameters

STR CHARACTER*. Formatted text representation of the H-number IHNX.

ERROR Alternate return argument.

### Remarks

Parameter IW should not be less than IM+IE+4 for the real H-numbers AFRealFloat and less than 2*(IM+IE)+11 for the complex H-numbers AFComplexFloat. When formatting exact and infinite H-numbers parameters IM and IE are ignored. Parameter IP makes a difference only for the floating-point and rational H-numbers AFFloat and CFRational. See section 3.5 for the specifications of output formats used for different kinds of numbers.

## SUBROUTINE HETEV( IHV, INDEX, IW, IP, IM, IE, STR, *ERROR )

*Converts element of H-vector to formatted text string*

### Input Parameters

| | |
|---|---|
| IHV | INTEGER. Handle to H-vector AUVector. |
| INDEX | INTEGER. Index of the selected element of H-vector IHV (positive number). |
| IW | INTEGER. Full width of the output field (positive number). |
| IP | INTEGER. Format parameter that specifies position of decimal point. |
| IM | INTEGER. Number of decimal digits of mantissa (positive number). |
| IE | INTEGER. Number of decimal digits of exponent (positive number). |

### Output Parameters

| | |
|---|---|
| STR | CHARACTER*. Formatted text representation of the INDEX-th element of H-vector IHV. |
| ERROR | Alternate return argument. |

### Remarks

Parameter IW should not be less than IM+IE+4 for real H-vectors AUVectorReal and less than 2*(IM+IE)+11 for complex H-vectors AUVectorCompl. See section 3.5 for the specifications of output formats.

## SUBROUTINE HETEM( IHM, IROW, ICOL, IW, IP, IM, IE, STR, *ERROR )

*Converts element of H-matrix to formatted text string*

### Input Parameters

| | |
|---|---|
| IHM | INTEGER. Handle to H-matrix AUMatrix. |
| IROW | INTEGER. Row index of the selected element of H-matrix IHM (positive number). |
| ICOL | INTEGER. Column index of the selected element of H-matrix IHM (positive number). |
| IW | INTEGER. Full width of the output field (positive number). |
| IP | INTEGER. Format parameter that specifies position of decimal point. |
| IM | INTEGER. Number of decimal digits of mantissa (positive number). |
| IE | INTEGER. Number of decimal digits of exponent (positive number). |

### Output Parameters

STR CHARACTER*. Formatted text representation of the (IROW, ICOL)-th element of H-matrix. IHM

ERROR Alternate return argument.

### Remarks

Parameter IW should not be less than IM+IE+4 for real H-matrices AUMatrixReal and less than 2*(IM+IE)+11 for complex H-matrices AUMatrixCompl. See section 3.5 for the specifications of output formats.
.

## 4.21. Conversion to Fortran Data

**SUBROUTINE HEFNX( IHN, FTYPE, FVAR, *ERROR )**

*Converts finite H-number to Fortran variable*

### Input Parameters

IHN INTEGER. Handle to H-number AFinite.

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

### Output Parameters

FVAR Fortran variable that takes converted value of the H-number IHN.

ERROR Alternate return argument.

### Remarks

Conversion to the INTEGER type is not allowed, i.e. input value FTYPE='I' is treated as an illegal one. If underflow or overflow occurs during conversion then FVAR is set to zero or ±INF respectively.

**SUBROUTINE HEFEV( IHV, INDEX, FTYPE, FVAR, *ERROR )**

*Converts element of H-vector to Fortran variable*

### Input Parameters

IHV INTEGER. Handle to H-vector AUVector.

INDEX INTEGER. Index of the selected element of the H-vector IHV (positive number).

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

### Output Parameters

FVAR Fortran variable that takes converted value of the INDEX-th element of H-vector IHV.

ERROR Alternate return argument.

### Remarks

Conversion to the INTEGER type is not allowed, i.e. input value FTYPE='I' is treated as an illegal one. If underflow or overflow occurs during conversion then FVAR is set to zero or ±INF respectively.

---

## SUBROUTINE HEFV( IHV, NDIM, FTYPE, FARRAY, *ERROR )

*Converts H-vector to Fortran array*

### Input Parameters

IHV INTEGER. Handle to H-vector AUVector.

NDIM INTEGER. Dimension of the output array FARRAY that should be equal to or greater than Dimension of the H-vector IHV.

FTYPE CHARACTER*1. Fortran type descriptor for FARRAY (see section 4.2).

### Output Parameters

FARRAY Fortran array that takes converted representation of the H-vector IHV.

ERROR Alternate return argument.

### Remarks

Conversion to the INTEGER type is not allowed, i.e. input value FTYPE='I' is treated as an illegal one. If underflow or overflow occurs during conversion then the corresponding element of FARRAY is set to zero or ±INF respectively.

---

## SUBROUTINE HEFEM( IHM, IROW, ICOL, FTYPE, FVAR, *ERROR )

*Converts element of H-matrix to Fortran variable*

### Input Parameters

IHM INTEGER. Handle to H-matrix AUMatrix.

IROW INTEGER. Row index of the selected element of the H-matrix IHM (positive number).

ICOL INTEGER. Column index of the selected element of the H-matrix IHM (positive number).

FTYPE CHARACTER*1. Fortran type descriptor for FVAR (see section 4.2).

### Output Parameters

FVAR Fortran variable that takes converted value of the (IROW,ICOL)-th element of H-matrix IHM.

ERROR Alternate return argument.

### Remarks

Conversion to the INTEGER type is not allowed, i.e. input value FTYPE='I' is treated as an illegal one. If underflow or overflow occurs during conversion then FVAR is set to zero or ±INF respectively.

## SUBROUTINE HEFMR( IHM, IROW, NDIM, FTYPE, FARRAY, *ERROR )

*Converts H-matrix row to Fortran array*

### Input Parameters

IHM INTEGER. Handle to H-matrix AUMatrix.

IROW INTEGER. Index of the selected row of the H-matrix IHM (positive number).

NDIM INTEGER. Dimension of the output array FARRAY that should be equal to or greater than number of columns of H-matrix IHV.

FTYPE CHARACTER*1. Fortran type descriptor for FARRAY (see section 4.2).

### Output Parameters

FARRAY Fortran array that takes converted representation of the IROW-th row of H-matrix IHM.

ERROR Alternate return argument.

### Remarks

Conversion to the INTEGER type is not allowed, i.e. input value FTYPE='I' is treated as an illegal one If underflow or overflow occurs during conversion then the corresponding element of FARRAY is set to zero or ±INF respectively.

## SUBROUTINE HEFMC( IHM, ICOL, NDIM, FTYPE, FARRAY, *ERROR )

*Converts H-matrix column to Fortran array*

### Input Parameters

IHM INTEGER. Handle to H-matrix AUMatrix.

ICOL INTEGER. Index of the selected column of the H-matrix IHM (positive number).

NDIM INTEGER. Dimension of the output array FARRAY that should be equal to or greater than number of rows of H-matrix IHV.

FTYPE CHARACTER*1. Fortran type descriptor for FARRAY (see section 4.2).

### Output Parameters

FARRAY Fortran array that takes converted representation of the ICOL-th column of H-matrix IHM.

ERROR Alternate return argument.

### Remarks

Conversion to the INTEGER type is not allowed, i.e. input value FTYPE='I' is treated as an illegal one. If underflow or overflow occurs during conversion then the corresponding element of FARRAY is set to zero or ±INF respectively.

---

## SUBROUTINE HEFM( IHM, NROW, NCOL, FTYPE, FARRAY, *ERROR )

*Converts H-matrix to Fortran array*

### Input Parameters

IHM INTEGER. Handle to H-matrix AUMatrix.

NROW INTEGER. Number of rows of the H-matrix IHM (positive number).

NCOL INTEGER. Number of columns of the H-matrix IHM (positive number).

FTYPE CHARACTER*1. Fortran type descriptor for FARRAY (see section 4.2).

### Output Parameters

FARRAY Fortran array that takes converted representation of the H-matrix IHM. Size of the array should be equal to or greater than total number of elements of H-matrix IHM, i.e. NROW*(NROW+1)/2 for Hermitian matrices stored in the packed form (in this case NROW = NCOL), or NROW*NCOL for all other kinds of matrices.

ERROR Alternate return argument.

### Remarks

Conversion to the INTEGER type is not allowed, i.e. input value FTYPE='I' is treated as an illegal one. If underflow or overflow occurs during conversion then the corresponding element of FARRAY is set to zero or ±INF respectively.

# Appendix A. Error Codes

**Table A-1. Numerical Error Codes**

| **Resource Errors** | | |
|---|---|---|
| **Code** | **Text Message** | **Comment** |
| 0001 | HEAP MEMORY ALLOCATION FAILURE | OS-level dynamic memory allocation failure. |
| 0002 | MAX HEAP SIZE OVERFLOW | User-defined maximum size of heap memory is exceeded. |
| 0003 | MEMORY POOL OVERFLOW | ExLAF77 internal memory pool overflow. Memory pools are not implemented in the present version though. |
| **Interface Errors** | | |
| **Code** | **Text Message** | **Comment** |
| 0101 | INVALID OBJECT HANDLE | Input H-object is not created, or it is deleted, or its handle is corrupted by the calling program |
| 0102 | ILLEGAL TYPE OF OPERAND | Called function does not accept input H-object of the present type as an operand. |
| 0103 | UNRECOGNIZED TEXT DEESCRIPTOR | Input CHARACTER descriptor does not coincide with any character or string recognizable by called function. |
| 0104 | ILLEGAL FORMAT OF INPUT STRING | Input text string has an illegal format, or it is empty, or its length is incorrectly defined. |
| 0105 | INVALID FLOATING POINT DATA | Input single or double precision floating-point data contain denormalized values, INFs, or NaNs. |
| 0106 | INDEX IS OUT OF RANGE | Present index value is not positive, or it exceeds respective dimension of the vector or matrix, or passed actual parameter is not an INTEGER. |
| 0107 | IMPROPER ARRAY DIMENSION | Dimension of input array is not equal to the respective dimension of target vector, matrix row, matrix column, or entire matrix. |
| 0108 | IMPROPER PARAMETER VALUE | Illegal or senseless numerical value of input parameter, or passed actual parameter has a wrong Fortran type. |
| **Floating-Point Errors** | | |
| **Code** | **Text Message** | **Comment** |
| 0201 | FLOATING POINT UNDERFLOW | Unrecoverable floating-point underflow during "update" operation resulted in denormalized value. |
| 0202 | FLOATING POINT OVERFLOW | Unrecoverable floating-point overflow during "update" operation resulted in the INF value. |
| 0203 | FLOATING POINT DIVISION ZERO BY ZERO | Unrecoverable floating-point division zero by zero resulted in the NaN value. |
| **Illegal operations** | | |
| **Code** | **Text Message** | **Comment** |

| | | |
|---|---|---|
| 0301 | ASSIGN COMPLEX TO REAL | Attempt of assigning complex value to a real number or to element of a real vector or matrix. |
| 0302 | ASSIGN TO IMAGINARY PART OF REAL | Attempt of assigning a value to imaginary part of a real number, or to imaginary part of element of a real vector or matrix. |
| 0303 | COMPARE COMPLEX NUMBERS | Attempt to compare two complex numbers by value (equivalent to the LT or GT operators). |
| **Calculus Errors** | | |
| **Code** | **Text Message** | **Comment** |
| 0401 | TOO BIG ABS VALUE OF ARGUMENT | Absolute value of a function argument is so big that the result length exceeds the internal ExLAF77 limit. |
| 0402 | TOO BIG ABS VALUE OF EXPONENT | Absolute value of exponent of a function argument is so big that the result length exceeds the internal ExLAF77 limit. |
| 0403 | ARGUMENT IS OUT OF RANGE | Argument value does not belong to the domain of algorithm applicability. |
| 0404 | UPDATE OPERATION FAILURE | Result of an *Update* operation cannot be assigned to variable due to incompatibility of types. Example: X = SQRT(X), where X is a negative real floating-point number. |
| **Matrix Operation Errors** | | |
| **Code** | **Text Message** | **Comment** |
| 0501 | OPERANDS' DIMENSIONS MISMATCH | Disparity of operands' dimensions of binary vector / matrix operations. |
| 0502 | SINGULAR MATRIX | The matrix appeared to be algorithmically singular during triangular decomposition. |
| 0503 | INDEFINITE HERMITIAN MATRIX | The Hermitian matrix declared as positive-definite appeared to be indefinite during decomposition. |
| **Undefined Result** | | |
| **Code** | **Text Message** | **Comment** |
| 0601 | DIVIDE ZERO BY ZERO | 0/0 |
| 0602 | DIVIDE INFINITY BY INFINITY | INF/INF, (±INF)/INF, INF/(±INF), or (±INF)/(±INF) |
| 0603 | MULTIPLY INFINITY BY ZERO | INF*0, or (±INF)*0 |
| 0604 | RAISE INFINITY TO ZEROTH POWER | INF**0, or (±INF)**0 |
| 0605 | SUBTRACT INFINITY FROM INFINITY | INF±INF, INF±(±INF), (±INF)±INF, (+INF)+(-INF), (+INF)-(+INF), or (-INF)-(-INF) |
| 0606 | REMAINDER WITH ZERO DENOMINATOR | MOD(N,0) where N is a finite number |
| 0607 | INT QUOTIENT WITH ZERO DENOMINATOR | INT(N/0) where N is a finite number |
| 0608 | RE/IM PART OF UNSIGNED INFINITY | REAL(INF), or IMAG(INF) |

| | | |
|---|---|---|
| 0609 | FUNCTION DOES NOT HAVE A LIMIT | SIN(INF), SIN(±INF), COS(INF), COS(±INF), TAN(INF), TAN(±INF), SINH(INF), COSH(INF), EXP(INF), ATAN2(0,0), or ATAN2(±INF,±INF) |
| **Programming Bugs** | | |
| **Code** | **Text Message** | **Comment** |
| >10000 | About 10 different messages | ExLAF77 internal bugs that should be reported to QNT Software Development Inc. |

# Appendix B. Routines Reference