Dec. 30 2009

Multiple-Precision Arithmetic Library exflib (Fortran90/95)

FUJIWARA Hiroshi
fujiwara@acs.i.kyoto-u.ac.jp

## exflib

Exflib (extended precision floating-point arithmetic library) is a simple software for multiple-precision arithmetic in scientific numerical computation. Multiple-precision arithmetic is a method for representation and calculation of real numbers with arbitrary accuracy.



We shall abbreviate binary digit as “bit”, and decimal digit as “digit.” And we use the term “decimal” as the same mean of radix-10.

## 1. Introduction

- Exflib consists of two files: a library file (`libexfloat.a`) and a module file (`exflib.F90`).
  Though files have common names on all architectures, entities are different on architectures.
- Exflib is a module of Fortran90. At the beginning of your program unit, you must declare `USE` sentence.

```
USE exflib
```

- The name of multiple-precision type is `TYPE(exfloat)`. A declaration of variables is the following form:

```
TYPE(exfloat) :: x
```

- You specify your request precision with a module member `exflib_exfloat_precision10`, which appears at the beginning of the file exflib.F90. For example, if you need 1000 digits accuracy, you write as follows:

```
!-------------------------------------------------------------
! Your Requirement Digits (in decimal)
! 40 <= exflib_exfloat_precision10 <= 19700
!-------------------------------------------------------------
INTEGER*4, PARAMETER :: exflib_exfloat_precision10 = 1000
```

  Precision must be greater than or equal to 40, and be less than 19700. Instead of `exflib_exfloat_precision10`, you specify precision with an identifier `EXFLIB_EXFLOAT_PRECISION10` by preprocessor. (See following)

  When you change precision, you must re-compile the module.

You must link the file `libexfloat.a` finally. When you have program `sample.f90`, `exflib.F90`, and `libexfloat.a` in the same directory, a compile instruction is follows:

- Sun Studio10, Studio11 (AMD64, EM64T / UltraSPARC)

```
% f90 -xarch=amd64 -xO2 -free exflib.F90 sample.f90 libexfloat.a
% f90 -xarch=v9 -xO2 -free exflib.F90 sample.f90 libexfloat.a
```

- G95 (64-bit / 32-bit)

```
% g95 -m64 -O2 -ffree-form exflib.F90 sample.f90 libexfloat.a
% g95 -m32 -O2 -ffree-form exflib.F90 sample.f90 libexfloat.a
```

- GNU gfortran (4.1.2, 4.2.2, 4.4.1, 64-bit / 32-bit)

```
% gfortran -m64 -O2 -ffree-form exflib.F90 sample.f90 libexfloat.a
% gfortran -m32 -O2 -ffree-form exflib.F90 sample.f90 libexfloat.a
```

- Intel Fortran Compiler (Ver. 9)

```
% ifort -O2 -free exflib.F90 sample.f90 libexfloat.a
```

- PGI Compiler (Ver.5, Ver. 6)

```
% pgf90 -O2 -Mfree exflib.F90 sample.f90 libexfloat.a
% pgf95 -O2 -Mfree exflib.F90 sample.f90 libexfloat.a
```

- frt on HPC2500

```
% frt -KV9 -Free -O5 -Am exflib.F90 sample.f90 libexfloat.a
```

- Specification of precision by preprocessor (500 digit)

  % *f90 options* `-DEXFLIB_EXFLOAT_PRECISION10=500 exflib.F90 sample.f90 libexfloat.a`

## 2. Substitution

- You can set a `TYPE(exfloat)` type value with an integer.

```
TYPE(exfloat) :: x
INTEGER :: i

x = 1                              ! valid
x = -10                            ! valid
x = i                              ! valid
```

- Literal constants must have the string form (within quotations) in a substitution sentence.

```
TYPE(exfloat) :: x

x = '0.001'                        ! valid
x = '1/100'                        ! valid
x = '#PI/2'                        ! valid
x = '1e-10'                        ! valid

x = '12.34*(5/6)*(7e-8)*#E'        ! valid
```

In a string, you can put decimal numbers (with fixed point format or scientific format), mathematical constants ($\pi$ `#PI`, the natural logarithm base $e$ `#E`, or Euler's constant $\gamma$ `#G`), their arithmetics, and parenthesis(). Strings are parsed and calculated at run time. Syntax errors in the strings are detected at run time.

- You can not substitute a sigle-precision type (`REAL*4`) or a double precision type (`REAL*8`) to a `TYPE(exfloat)` type. You need an explicit type conversion.

```
REAL*4 :: s
REAL*8 :: d
TYPE(exfloat) :: x

x = s                              ! invalid
x = exflib_cast(s)                 ! valid

x = d                              ! invalid
x = exflib_cast(d)                 ! valid

x = 0.1                            ! invalid
x = exflib_cast(0.1)               ! valid, but not accurate (about 7 digits)

x = 0.1d                           ! invalid
x = exflib_cast(0.1d)              ! valid, but not accurate (about 15 digits)
```

Values set with a single- or double-precision type have the same accuracy as single- or double-precision type respectively. (`REAL*4` has about 7 digits, and `REAL*8` has about 15 digits)

```
x = '0.1'                          ! valid, accurate
x = '1e-10'                        ! valid, accurate
x = '1/10'                         ! valid, accurate
x = '0.1d'                         ! invalid
```

## 3. Output

There are two ways to output a `TYPE(exfloat)` value in decimal:

1. Convert `TYPE(exfloat)` to built-in floating-point value, then output the built-in value. (fast, low-accurate)
2. Directly output `TYPE(exfloat)` (slow, high- and arbitrary accurate)

- Convert `TYPE(exfloat)` to built-in floating-point value, then output the built-in value

```
TYPE(exfloat) :: x
REAL*8 :: tmp

tmp = x
WRITE(*,*) tmp
```

or

```
TYPE(exfloat) :: x

WRITE(*,*) DBLE(x)                ! DBLE() is implemented,
                                  ! but it does not work well with some compilers.
```

`DBLE()` sometimes prints 'NaN'. Please see 'Known Bugs'.

- Directly output; convert to a string then output the string

```
TYPE(exfloat) :: x
CHARACTER(100) :: str

str = exflib_format('F.20', x)
WRITE(*,*) TRIM(str)
```

or

```
TYPE(exfloat) :: x
WRITE(*,*) TRIM(exflib_format('E100.200', x))
```

The first argument of the subroutine `exflib_format` is a format specification, which has the form $Fw.p$.

| | values | description |
|---|---|---|
| $F$ | `F` | fixed-point format (decimal) |
| | `E` | floating-point format (scientific format) |
| | `H` | hexadecimal format (internal representatin of `TYPE(exfloat)` type) |
| $w$ | decimal | width of an output string |
| $p$ | decimal | precision after the decimal point |

The multiple-precision value in the second argument is stored after the white space within the specified width. When the format specification $Fw.p$ is incorrect, it is treated as `H` conversion.

- You can abbreviate the specifications $w$ or $p$. Default values are $w = 6, p = 6$. When $w < p$, $w$ is automatically extended. In `H` format, $w$ and $p$ have no meanings.
- `G`, `ES`, `EN` formats are not supported.

(Examples)

- `'F.20'` : 20 digits after the decimal point with fixed-point format (width is automatically decided)
- `'E200.100'` : 100 digits after the decimal point with floating-point format, with 200 character widths.
- `'H'` : Internal representation of `TYPE(exfloat)`
- `'I20.10'` : Incorrect format. Internal representation of `TYPE(exfloat)` is returnd. (As `H` format)

## 4. Four Basic Rules

A `TYPE(exfloat)` type has the four basic rules (`+`, `-`, `*`, `/`) with `TYPE(exfloat)` type, and with built-in integers.

```
TYPE(exfloat) :: x, y, z
INTEGER :: i

z = x + y                     ! valid
z = x / y                     ! valid

z = x + i                     ! valid
z = x * i                     ! valid

z = (x + y) * i               ! valid
```

The four rules with built-in floating-point type `REAL` is not supported. If you need, first you convert the bulit-in type to a temporary `TYPE(exfloat)` type with an explicit type conversion, then you can operate with the temporary `TYPE(exfloat)` instead of the built-in types.

```
TYPE(exfloat) :: x, y, z
REAL :: s

z = x + s                     ! invalid

y = exflib_cast(s)            ! valid
z = x + y                     ! valid

z = x * 1.2                   ! invalid
z = (x * 12) / 10             ! valid
```

## 5. Comparison

Comparison between a `TYPE(exfloat)` type with a `TYPE(exfloat)` type, a built-in floating-point type, or an integer are defined.

```
TYPE(exfloat) :: x, y
REAL :: s
INTEGER :: i

IF ( x == y ) THEN            ! valid
  ....
ENDIF

IF ( x /= 1 ) THEN            ! valid

IF ( x < s ) THEN             ! valid

IF ( x <= 1e-3 ) THEN         ! valid

IF ( x > y + 1 ) THEN         ! valid

IF ( x > y + i ) THEN         ! valid

IF ( x >= y + 1e-3 ) THEN     ! invalid
```

## 6. Auxiliary Utility Functions and Members

- `INTEGER*4, PARAMETER :: exflib_exfloat_precision10`
  Return the user request precision in digits.

- `REAL*4, PARAMETER :: exflib_exfloat_precision`
  Return precision in digits actually used in computation. (The library `exflib` uses more digits than user request digits `exflib_exfloat_precision10`. See followings.)

- `INTEGER*4, PARAMETER :: exflib_exfloat_size`
  Return the size of elements in the internal array `num` which holds the multiple-precision floating-point value. The entity of a multiple-precision value of `TYPE(exfloat)` is an `INTEGER*8` integer array `num(0:exflib_exfloat_size-1)`. The index start with 0.

- `INTEGER*4, PARAMETER :: exflib_exfloat_byte`
  Return the memory quantity in byte a `TYPE(exfloat)` spends. The sum of sign, exponent and fractional part.

- `exflib_version()`
  Return the date and time when the `libexfloat.a` was compiled.

- `INTEGER :: DIGITS(x)`
  Returns the number of significant digits of the internal model representation of `TYPE(exfloat) :: x`.

- `INTEGER :: PRECISION(x)`
  Return decimal precision of `TYPE(exfloat) :: x`.

- `TYPE(exfloat) :: EPSILON(x)`, where `TYPE(exfloat) :: x`
  Returns a nearly negligible value in the type `TYPE(exfloat)` relative to 1.

Example

```
WRITE(*,*) 'Required precision (decimal digits):', exflib_exfloat_precision10
WRITE(*,*) 'Current precision (decimal digits) :', DIGITS(x)
WRITE(*,*) 'Current precision (bits, including the hidden one):', PRECISION(x)
WRITE(*,*) 'Machine Epsilon (gap between 1 and the next value) : ', &
           TRIM(exflib_format('e.6', EPSILON(x)))

WRITE(*,*) 'An exfloat number consists of', exflib_exfloat_size, 'INTEGER*8 elements.'
WRITE(*,'(a,i0,a)') ' Entity is x%num(0:', exflib_exfloat_size-1, ')'
WRITE(*,*) 'The size of exfloat is', exflib_exfloat_byte, 'bytes.'
WRITE(*,*) 'Current libexfloat.a is ', TRIM(exflib_version())
```

Results of the example

```
Required precision (decimal digits): 1000 digits.
Current precision (decimal digits) : 1002
Current precision (bits, including the hidden one): 3329
Machine Epsilon (gap between 1 and the next value) : 1.486533e-1002

An exfloat number consists of 53 INTEGER*8 elements.
Entity is x%num(0:52)
The size of exfloat is 424 bytes.
Current libexfloat.a is compiled at Apr 17 2006 09:14:32
```

Following built-in floating-point operations are supported as generic functions. The common argument is `TYPE(exfloat) :: x`.

- `INTEGER        :: RADIX(x)`
- `TYPE(exfloat) :: HUGE(x)`
- `TYPE(exfloat) :: TINY(x)`
- `INTEGER*8      :: MAXEXPONENT(x)`
- `INTEGER*8      :: MINEXPONENT(x)`
- `INTEGER*8      :: RANGE(x)`
- `TYPE(exfloat) :: SIGN(x,s)` with `TYPE(exfloat) :: s`
- `INTEGER*8      :: EXPONENT(x)`
- `TYPE(exfloat) :: FRACTION(x)`
- `TYPE(exfloat) :: NEAREST(x,s)` with `TYPE(exfloat) :: s`
- `TYPE(exfloat) :: SET_EXPONENT(x,e)` with `INTEGER :: e`
- `TYPE(exfloat) :: SPACING(x)`
- `TYPE(exfloat) :: RRSPACING(x)`

See '`float.f90`' in the sample-f90 directory.

## 7. Built-in Mathematical Functions

Following functions are defined for `TYPE(exfloat)`.

| Generics | Interface | |
|---|---|---|
| ABS | `ABS(x)` | absolute value $\|x\|$ |
| SQRT | `SQRT(x)` | square root $\sqrt{x}, x > 0$ |
| | | |
| SIN | `SIN(x)` | $\sin(x)$, $x$ is radian |
| COS | `COS(x)` | $\cos(x)$, $x$ is radian |
| TAN | `TAN(x)` | $\tan(x)$, $x$ is radian |
| | | |
| EXP | `EXP(x)` | base-$e$ exponential $e^x$ |
| POW | `POW(x,y)` | power function $x^y$ |
| ** | `x**y` | power function $x^y$ |
| | | (A domain error occurs if $x = 0$ and $y \leq 0$, or $x < 0$ and $y$ is not an integer) |
| | | |
| LOG | `LOG(x)` | natural logarithmic function $\ln(x), x > 0$ |
| LOG10 | `LOG10(x)` | base-10 logarithmic function $\log_{10}(x), x > 0$ |
| | | |
| ASIN | `ASIN(x)` | $\sin^{-1}(x)$, with $\sin^{-1}(x) \in [-\pi/2, \pi/2], x \in [-1, 1]$ |
| ACOS | `ACOS(x)` | $\cos^{-1}(x)$, with $\cos^{-1}(x) \in [0, \pi], x \in [-1, 1]$ |
| ATAN | `ATAN(x)` | $\tan^{-1}(x)$ with $\tan^{-1}(x) \in [-\pi/2, \pi/2]$ |
| | | |
| SINH | `SINH(x)` | $\sinh(x)$, $x$ is radian |
| COSH | `COSH(x)` | $\cosh(x)$, $x$ is radian |
| TANH | `TANH(x)` | $\tanh(x)$, $x$ is radian |
| | | |
| MOD | `MOD(x,y)` | remainder of $y$ modulo $x$ |
| FRACTION | `FRACTION(x)` | fractional part of $x$ |
| INT | `INT(x)` | integer prat of $x$ |
| FLOOR | `FLOOR(x)` | largest integer not greater than $x$ |
| CEILING | `CEILING(x)` | smallest integer not less than $x$ |

If a domain error occurs, the computation is terminated.

(atan2, ldexp, Bessel functions, Gamma function, cubic root, degree trigonometric functions like sind are not implemented; future work)

## 8. Complex Number Type

### Declaration

```
USE exflib
TYPE(ezfloat) :: z
```

Precision of `z` is defined by `EXFLIB_EXFLOAT_PRECISION10`. Compilation with `-DEXFLIB_EXFLOAT_PRECISION10=100` means that both the real part and imaginary part `z` have 100 decimal digits accuracy.

### Assignment

```
TYPE(ezfloat) :: z
TYPE(exfloat) :: x, y

z = 1                   ! INTEGER, 1 + 0 i
z = '#PI'               ! CHARACTER, \pi + 0 i
z = x                   ! TYPE(exfloat), x + 0 i

z = CMPLX(2, 3)         ! 2 + 3 i
z = CMPLX(x, y)         ! x + y i
z = CMPLX(x, '0.1')     ! valid, x + 0.1 i, CHARACTER is acceptable
z = CMPLX(0, '2*#PI')   ! 2 \pi i
z = polar(4, '#PI/3')   ! 4 e^{\pi/3 i}, polar form

z = 0.1                 ! not valid
z = CMPLX(x, 0.1)       ! not valid
z = CMPLX(x, 0.1d0)     ! not valid

COMPLEX*16 :: dz
z = dz                  ! not valid
z = exflib_cast(dz)     ! valid, not accurate (approx 15 digits)

REAL*8 :: d
z = d                   ! not valid
z = exflib_cast(d)      ! valid, not accurate (approx 15 digits), d + 0 i
```

Two constructors are available: `CMPLX` and `polar`; `z = CMPLX(x,y)` means $z = x + \sqrt{-1}y$, and `w = polar(r,t)` means $w = re^{\sqrt{-1}t}$.

Assignment by built-in types `COMPLEX` or `REAL` requires an explicit type conversion with `exflib_cast()`.

### Output

```
TYPE(ezfloat) :: z
COMPLEX*16 :: dz
dz = z                                   ! z is converted to COMPLEX*16
WRITE(*,*) dz                            ! double precision output
WRITE(*,*) TRIM(exflib_format('f.100', z))  ! fixed-point format, 100 digits
WRITE(*,*) TRIM(exflib_format('e.100', z))  ! floating-point format, 100 digits
! WRITE(*,*) CMPLX(z)                    ! This does not work well with some compilers.
```

Output with conversion to `COMPLEX*16` is fast.

**Arithmetic**

Basic for rules `+`,`-`,`*`,`/` with `TYPE(ezfloat)`, `TYPE(exfloat)`, `INTEGER` and literal integers are available. Arithmetic with `REAL` or `COMPLEX` are prohibited.

**Comparison**

Comparisons `==`, `/=` with `TYPE(ezfloat)`, `TYPE(exfloat)`, `INTEGER` `REAL`, `COMPLEX`, and literals are available.

**Built-in Functions**

| | | |
|---|---|---|
| CONJG | `CONJG(z)` | complex conjugate |
| REAL | `REAL(z)` | real part of $z$ |
| AIMAG | `AIMAG(z)` | imaginary part of $z$ |
| | | |
| ABS | `ABS(z)` | $\|z\|$, absolute value of $z$ |
| ARG | `ARG(z)` | $\mathrm{Arg}(z)$, argument of $z$, $-\pi < \mathrm{Arg}(z) \leq \pi$ |
| | | |
| SQRT | `SQRT(z)` | square root of $z$, $-\pi < \arg \sqrt{z} < \pi$ |
| | | |
| SIN | `SIN(z)` | $\sin(z)$, sine function |
| COS | `COS(z)` | $\cos(z)$, cosine function |
| TAN | `TAN(z)` | $\tan(z)$, tangent function |
| | | |
| EXP | `EXP(z)` | $e^z$, base-$e$ exponential |
| POW | `POW(z,a)` | $z^a$, power function |
| ** | `z**a` | $z^a$, power function |
| | | |
| LOG | `LOG(z)` | $\mathrm{Log}(z)$, natural logarithmic function, $-\pi < \arg \mathrm{Log}(z) < \pi$ |
| LOG10 | `LOG10(z)` | $\mathrm{Log}_{10}(z)$, base-10 logarithmic function, $-\pi < \arg \mathrm{Log}_{10}(z) < \pi$ |
| | | |
| SINH | `SINH(z)` | $\sinh(z)$, hyperbolic sine function |
| COSH | `COSH(z)` | $\cosh(z)$, hyperbolic cosine function |
| TANH | `TANH(z)` | $\tanh(z)$, hyperbolic tangent function |

## 9. Array Operations

Array operation is implemented in `exflib_array` module. One or two dimensional array and its operations are supported. Substitution, cast, operations (`+, -, *, /, **`) with an array or a scalar, comparison with an array or a scalar, and the following functions are supported.

`SUM`, `PRODUCT`, `MAXVAL`, `MINVAL`, `MAXLOC`, `MINLOC` (one and two dimensional)

`DOT_PRODUCT` (one dimensional)

`MATMUL`, `TRANSPOSE` (two dimensional)

Compilation: (slow!)

```
% f90 -DEXFLIB_EXFLOAT_PRECISION10=100 exflib.F90 exflib_array.F90 user.f90 libexfloat.
```

Example:

```
      USE exflib
      USE exflib_array

      INTEGER*8 :: ix(5)
      TYPE(exfloat) :: x(5), y(5), s
      TYPE(ezfloat) :: z(5)

      TYPE(exfloat) :: a(3,5), at(5,3), b(3,3), a2(3,5), a3(3,5)
      TYPE(ezfloat) :: c(3,5)

      !TYPE(exfloat) :: d(1,2,3)          ! not implenented

      ix = (/1,2,3,4,5/)
      x = ix                              ! x is set to (/1.0, 2.0, 3.0, 4.0, 5.0/)
      y = 1                               ! y is set to (/1.0, 1.0, 1.0, 1.0, 1.0/)
      s = 1

      y = x + ix                          ! y is (/2,4,6,8,10/)
      y = x + 1                           ! y is (/2,3,4,5,6/)
      y = x + s                           ! same as above
      y = x + y                           ! same as above
      y = x * 2                           ! y is (/2,4,6,8,10/)
      y = 2 * x                           ! same as above
      y = x * x                           ! y is (/1,4,9,16,25/)

      s = SUM( x*x )                      ! s = x1*x1 + x2*x2 + ... + x5*x5
      y = x / SQRT( SUM( x*x ) )          ! normalization in the Euclid norm
      s = DOT_PRODUCT( x, y )             ! s = x1*y1 + x2*y2 + ... + x5*y5

      z = ix                              ! z is (/1+0i, 2+0i, 3+0i, 4+0i, 5+0i/)
      z = x                               ! z is (/1+0i, 2+0i, 3+0i, 4+0i, 5+0i/)

      a = RESHAPE( (/11,21,31, 12,22,32, 13,23,33, 14,24,34/), (/3,4/) )

      a2 = -a                             ! a2(i,j) = - a(i,j) for each (i,j)
      a3 = a + a2                         ! a3(i,j) = a(i,j) + a2(i,j)
      a3 = a * a2                         ! a3(i,j) = a(i,j) * a2(i,j)
                                          ! '*' is not matrix multiplication.

      at = TRANSPOSE( a )                 ! at = a^T
      b = MATMUL(at, a)                   ! b = A^T * A (matrix multiplication, not optimized)

      WRITE(*,*) DBLE( x )            ! valid, but see 'Knwon Bugs' section
      WRITE(*,*) CMPLX( z )           ! valid, but see 'Knwon Bugs' section
      WRITE(*,*) DBLE( a )            ! valid, but see 'Knwon Bugs' section
```

See '`array_aux.f90`' in sample-f90 directory for other functions.

## 10. Multiple-Precision Programming with exflib – How to Convert your Program

1. Change the declaration of real valued variables `REAL`.

   (Original)

```
REAL :: x                       ! Real
REAL*4 :: x                     ! single precision, floating-point type
REAL*8 :: x                     ! double precision, floating-point type
```

   (Revised)

```
TYPE(exfloat) :: x              ! multiple-precision, floating-point type
```

2. Add quotations to all literal constants in substitution to `TYPE(exfloat)` variables.

```
x = 0.1      →     x = '0.1'
x = 0.5d     →     x = '0.5'        ! Delete the suffix d
```

3. Changing all statements which contain arithmetic or built-in functions of literal floating-point constants.

   (Original)

```
z = w + 3.4
y = x*1.2
```

   (Revised)

```
TYPE(exfloat) :: tmp
tmp = '3.4'
z = w + tmp

y = x * 12 / 10
```

4. Change output statements.

   (Original)

```
WRITE(*,*) x
```

   (Revised)

```
WRITE(*,*) TRIM(exflib_format('F', x))
or
WRITE(*,*) DBLE(x)
```

   `DBLE()` sometimes prints 'NaN'. Please see 'Known Bugs'.

- A library which strongly depends on floating-point types, like LAPACK, does not work with exflib. Fortran does not have the generic programming method, you must convert existing programs using the multiple-precision type.

- OpenMP works under exflib with limitations in the use of REDUCTION directives.

- MPI works under exflib, with some changes of floating-point data transfer sentences.

- The entity of `TYPE(exfloat)` is an array of unsigned 8-byte long integers (64-bit width integer, `INTEGER*8`). The name of the array is `num`, the number of elements is `exflib_exfloat_size`, and the index of the array starts from 0.

  More concretely, the entity of the `TYPE(exfloat)` multiple-precision variable `x` is

  ```
  INTEGER*8 :: x%num(0:exflib_exfloat_size-1)
  ```

  In parallel computations user should send and receive the array to exchange `TYPE(exfloat)` values.

- The type `TYPE(ezfloat)` consists of two `TYPE(exfloat)` numbers, namely,

  ```
  TYPE(exfloat) :: re, im
  ```

  In parallel computation, user syould send and recieve two members of a variable `TYPE(ezfloat) :: z` ;

  `z%re%num(0:exflib_exfloat_size-1)` and `z%im%num(0:exflib_exfloat_size-1)`

- The following identifiers are reserved for use as keywords.

  ```
  exfloat exflib exflib_* EXFLIB_*
  ```

## 11. Known Bugs, Remarks and Limitations

- `DBLE()` and `CMPLX()` sometimes do not work well (gfortran (4.3.0 or before) PGI Fortran, and others).

  See the sample '`bug_dble.f90`' in `sample-f90` directory.

- Implementation of following is a future work.

  `AINT`, `ANINT`, `DIM`, `INT`, `MODULO`, `NINT`,
  `ATAN2`,
  `MERGE`, `PACK`, `SPREAD`, `UNPACK`, `CSHIFT`, `EOSHIFT`

  `DPROD`, `MAX`, `MIN`, `KIND`

- You must specify precision more than or equal to 40 digits. Some computations fail because of implementation when you requirement precision under 40 digits.

- You can not use the built-in floating-point types and `TYPE(exfloat)` in one arithmetic sentence togeter. You need an explicit type conversion.

- Multiple-precision complex variable and interval arithmetic are now under developped.

- Rounding to the nearest in the built-in mathematical functions are sometimes not exact.

- Precision of `TYPE(exfloat)` must be defined at a compiling time. You can not change precision in user programs. (future work)

- Actual computation precision is differ from the user request precision.

  Actual precision is higher than request precision. The member `exflib_exfloat_precision` has actual precision, and the member `exflib_exfloat_precision10` has request precision in digits. You do not care the difference in usual.

- For example, actual computation precision is same for requests `exflib_exfloat_precision10 = 100` and `exflib_exfloat_precision10 = 110`, for example. This means that the member `exflib_exfloat_precision` is same for these two descriptions.

  The next table shows user request precisions and actual computation precisions.

| User Request Precision `exflib_exfloat_precision10` | Actual Computation Precision in `exflib` `exflib_exfloat_precision` |
|---|---|
| 39 – 57 | 57.80 |
| 58 – 77 | 77.06 |
| 78 – 96 | 96.33 |
| 97 – 115 | 115.60 |
| 116 – 134 | 134.87 |
| 135 – 154 | 154.13 |
| 155 – 173 | 173.39 |
| 174 – 192 | 192.66 |
| ⋮ | ⋮ |

  A step of actual computation precision is about 19.3, more precisely, it is $64 \times \log_{10} 2 \approx 19.2659$. The step comes from the reason that `exflib` holds a multiple-precision value as an array of `INTEGER*8` types (radix-$2^{64}$ integer), and $2^{64} \approx 10^{19.3}$.

- Limiting to four basic rules, there are no limitations of computation precision in used algorithms.
- Precision of built-in constants ($\pi, \log_{10} 2$) is about $19,700$ digits. If you need more precision, the library `libexfloat.a` must be compiled again.

  This means that the upper limit precision of built-in functions and decimal output is about $19,700$ digits. This corresponds to 1023 elements in an array of `INTEGER*8` integers. ($2^{64\times 1023} \approx 10^{19709.035}$)

## 12. Copyright and Disclaimer