Reduced Precision Emulator Manual

Release 3.1.1

Andrew Dawson, Peter Dueben

January 05, 2016

Contents

This user guide provides documentation for using the reduced precision emulation Fortran library.

CHAPTER 1

Overview

The library contains two core types: *rpe_var* and *rpe_shadow*. Both of these types are subclasses of the abstract type *rpe_type*. These types can be used in place of real-valued variables to perform calculations with floating-point numbers represented with a reducednumber of bits in the floating-point significand.

1.1 Basic use of the reduced-precision types

The *rpe_var* type is a simple container for a double precision floating point value, whereas the *rpe_shadow* type acts as a memory-view onto an existing double precision floating point number defined outside the type itself. Changing the value of an *rpe_shadow* variable also changes the value of the floating point number it is shadowing and vice-versa, since they both refer to the same block of memory.

Using an *rpe_var* instance is as simple as declaring it and using it just as you would a real number:

```
TYPE(rpe_var) :: myvar

myvar = 12
myvar = myvar * 1.287   ! reduced-precision result is stored in `myvar`
```

An *rpe_shadow* instance is different, it must first be initialized to point at an existing real-valued variable before it can be used:

```
REAL(KIND=RPE_REAL_KIND) :: real_target
TYPE(rpe_shadow)         :: myvar

CALL init_shadow (myvar, target)
myvar = 12               ! both `myvar` and `real_target` contain the value 12
myvar = myvar * 1.287    ! reduced precision result is stored in both `myvar` and `real_target`
```

1.2 Controlling the precision

The precision used by reduced precision types can be controlled at two different levels. Each reduced precision variable has an `sbits` attribute which controls the number of explicit bits in its significand. This can be set independently for different variables, and comes into effect after it is explicitly set.

```
TYPE(rpe_var) :: myvar1
TYPE(rpe_var) :: myvar2

! Use 16 explicit bits in the significand of myvar1, but only 12 in the
```

```
! significand of myvar2.
myvar1%sbits = 16
myvar2%sbits = 12
```

For variables whose `sbits` attribute has not been explicitly set, there is a default global precision level, set by *RPE_DEFAULT_SBITS*. This will stand-in for the number of explicit significand bits in any variable where `sbits` has not been set. Setting *RPE_DEFAULT_SBITS* once to define the global default precision, and setting the precision of variables that require other precision manually using the `sbits` attribute is generally a good strategy. However, if you change *RPE_DEFAULT_SBITS* during execution, the document *Changing RPE_DEFAULT_SBITS during execution* lists some details you should be aware of.

The emulator can also be turned off completely by setting the module variable *RPE_ACTIVE* to `.FALSE.`.

CHAPTER 2

Accessing the Source Code

If you just want to access the source code, but you don't want to do any development for now, you can follow these instructions. The Developer Guide has instructions for those wishing to do development work with the code.

2.1 Get a local copy of the code

The source code of the emulator is sotred in a git repository, you therefore need to have git installed to access the code. Currently the repository is located on the ECMWF Stash server, to clone it use:

```
git clone git@gitlab.physics.ox.ac.uk:aopp-pred/rpe.git
```

This will ask you for your Stash username and password. You will now have a copy of the code in the new `rpe` directory.

2.2 Updating the code

from time to time you may want to pull down the latest code. Do this with:

```
cd rpe
git pull
```

The code in `rpe` will now have the latest changes from the initial repository.

CHAPTER 3

Building the Emulator

3.1 Build requirements

Building the emulator requires:

- GCC (gfortran) >= 4.8
- GNU Make

The emulator has been tested with GCC 4.8.4 and GCC 4.9.2 and is known to work correctly ith these compilers. No testing of other compilers has been done by us, it might work with other compilers, it might not.

3.2 Building

The library is built in two stages. Stage 1 constructs the full emulator source from various components. Stage 2 compiles the full source code into a library archive.

3.2.1 Stage 1 build

Stage 1 uses the C preprocessor to construct a single Fortran source file from a base module and extra included header files. The result of stage 1 is a self-contained source file that can be used directly in other projects if desired, without proceeding to build stage 2. The output of stage 1 is the file `src/rp_emulator.f90`. You can initiate a stage 1 build with:

```
make source
```

3.2.2 Stage 2 build

Stage 2 compiles the emulator source code into a library archive and a corresponding Fortran module file. The resulting library is the archive `lib/librpe.a` and the module file is `modules/rp_emulator.mod`. You can initiate a stage 2 build with:

```
make library
```

The Make target `all` will also build the stage 2 library, and is the default target.

3.3 Integration

Assuming you did a full (stage 2) build, integration with your project is fairly straightforward, requiring two files to be present, one at compile time and one at link time.

You must make sure that the module file `rp_emulator.mod` is available to the compiler when compiling any source file that uses the module. You can do this by pecifying an include flag at compile time: `-I/path/to/rp_emulator.mod`. Alternatively you could place the module file in the same directory as your source files, but it is recommended to store it separately and use an include flag.

At link time the `librpe.a` library will also need to be available to the linker. You can use a combination of linker path and library options to make sure this is the case: `-L/path/to/librpe.a -lrpe`. Alternatively, you can directly specify the full path to `librpe.a` as an object to include in linking.

3.3.1 Stage 1 builds

If you wish, you can just do a stage 1 build and include the source code of the emulator directly in your project. However, you need to use the correct options when compiling the emulator. The emulator source code may contain some lines longer than the default line-length limit (these lines are prodcued by the code generator). To make sure the library compiles properly you need to tell the compiler to ignore line-length restrictions. For gfortran the option is `-ffree-line-length-none`.

CHAPTER 4

Using the Emulator

4.1 Using the emulator in your code

In any subroutine, module or program where you want to use the emulator, you need to import the emulator's module. For example, to include the emulator in a particular subroutine:

```
SUBROUTINE some_routine (...)
    USE rp_emulator
    ...
END SUBROUTINE some_routine
```

You can then use any of the emulator features within the subroutine.

4.2 Reduced precision types

Two types are provided to represent reduced precision floating point numbers. The *rpe_var* type is used to contain an independent reduced precision floating point number, whereas the *rpe_shadow* contains a pointer to an actual real variable elsewhere in the program and ensures that precision is reduced when operating on the shadow.

4.2.1 When to use each type

For new codes where you can write from the start by using reduced precision variables instead of real numbers, or for relatively small existing codes with few subroutine calls, it is suggested to stick to the *rpe_var* type, as in general you are less likely to encounter problems when using the *rpe_var* type.

The *rpe_shadow* is particularly useful when modifying only parts of an existing code, especially if the code structure has many subroutine calls. In some situations using an *rpe_shadow* provides a simpler solution than using and *rpe_var*, or it may be easy to automtically parse an existing code and introduce *rpe_shadow* instances than to modify the code to use *rpe_var* types.

The choice of which to use is up to you, we suggest using whichever seems easiest to implement in your given scenario. The two types can be mixed freely within a single code, so you don't have to choose one and stick to it, you can choose which type to use on a per-variable basis if you want. The next section of this guide contains a short brief on the differences between the two types: Differences between rpe_var and rpe_shadow.

CHAPTER 5

Differences between `rpe_var` and `rpe_shadow`

Although both the *rpe_var* and *rpe_shadow* types are designed to do the same thing (and are both extensions of the *rpe_type* type), there are some subtle differences in the way they are used.

5.1 Storage of full-precision values inside reduced precision types

The basic idea of the reduced-precision types is that they hold reduced-precision values. This is always true for the *rpe_var* (see *Changing RPE_DEFAULT_SBITS during execution* for caveats to this), but it is not necessarily the case for the *rpe_shadow* type. This is because the value stored in an *rpe_shadow* type can be set both by assigning to the *rpe_shadow* instance *and* by assigning to the real variable it shadows. When assigning directly to the *rpe_shadow* instance the precision of the stored value will be altered according to the `sbits` attribute, but when assigning to the real variable it shadows the assignment does not reduce the precision of the stored value. This is perhaps best illustrated with some examples.

5.1.1 Example 1

Assigning directly to a shadow, the precision is reduced to 8 bits as expected:

```
PROGRAM example1

    USE rp_emulator
    IMPLICIT NONE

    REAL(KIND=RPE_REAL_KIND) :: target
    TYPE(rpe_shadow)         :: shadow

    ! Use 8 bits of precision in the shadow's significand.
    shadow%sbits = 8
    CALL init_shadow (shadow, target)

    ! Assign a value directly to the shadow.
    shadow = 3.2812147

    ! Write the value contained within the shadow variable.
    WRITE (*, *) shadow%get_value()

END PROGRAM
```

Output:

```
 3.2812500000000000
```

5.1.2 Example 2

Assigning to the variable being shadowed, the precision is not reduced:

```
PROGRAM example2

    USE rp_emulator
    IMPLICIT NONE

    REAL(KIND=RPE_REAL_KIND) :: target
    TYPE(rpe_shadow)         :: shadow

    ! Use 8 bits of precision in the shadow's significand.
    shadow%sbits = 8
    CALL init_shadow (shadow, target)

    ! Assign a value to the variable being shadowed.
    target = 3.2812147

    ! Write the value contained within the shadow variable.
    WRITE (*, *) shadow%get_value()

END PROGRAM
```

Output:

```
 3.2812147140502930
```

5.1.3 Example 3

Shadowing a variable that is already assigned, the precision is not reduced:

```
PROGRAM example3

    USE rp_emulator
    IMPLICIT NONE

    REAL(KIND=RPE_REAL_KIND) :: target
    TYPE(rpe_shadow)         :: shadow

    ! Assign a value to the variable that will be shadowed later.
    target = 3.2812147

    ! Use 8 bits of precision in the shadow's significand.
    shadow%sbits = 8
    CALL init_shadow (shadow, target)

    ! Write the value contained within the shadow variable.
    WRITE (*, *) shadow%get_value()

END PROGRAM
```

Output:

```
 3.2812147140502930
```

5.2 Resolutions

The illustrated behaviour is by design. To ensure consistency between implementations using *rpe_var* and *rpe_shadow* you need to make some manual calls to *apply_truncation()*. A slightly modified version of example 3 is given below, with an extra *apply_truncation()* call inserted to ensure the value stored within the shadow has reduced-precision.

5.2.1 A simple fix

```
PROGRAM simplefix

    USE rp_emulator
    IMPLICIT NONE

    REAL(KIND=RPE_REAL_KIND) :: target
    TYPE(rpe_shadow)         :: shadow

    ! Assign a value to the variable that will be shadowed later.
    target = 3.2812147

    ! Use 8 bits of precision in the shadow's significand.
    shadow%sbits = 8
    CALL init_shadow (shadow, target)
    CALL apply_truncation (shadow)

    ! Write the value contained within the shadow variable.
    WRITE (*, *) shadow%get_value()

END PROGRAM
```

Output:

```
3.2812500000000000
```

5.3 Realistic examples

The *rpe_shadow* type is designed to make changing only parts of an existing code simpler. A good example of this is introducing reduced-precision types into a routine that then makes calls to subroutines that operate in full precision. Below are two equivalent programs, one using the *rpe_var* type and one using the *rpe_shadow* type. The programs are very similar, but the way calls to full-precision subroutines are made is different.

5.3.1 Using `rpe_var`

```
PROGRAM myprog_var
! Program with main variable replaced by a reduced precision variable.
!

    USE rp_emulator
    IMPLICIT NONE

    ! A reduced precision variable.
    TYPE(rpe_var)            :: reduced_value
    ! A temporary variable needed later.
```

```
    REAL(KIND=RPE_REAL_KIND) :: tmp_real_value

    ! Use 8 bits of significand precision for the reduced precision
    ! variable
    reduced_value%sbits = 8

    ! Assign a value to the reduced precision variable. In a real program
    ! there could be an arbitrary mount of computation here.
    reduced_value = 3.2812147

    ! Call the subroutine modify_value, this expects a real number rather
    ! than a reduced precision type, so we have to make a copy of our
    ! value to feed in and then copy the resulting modified value back into
    ! our reduced precision variable.
    tmp_real_value = reduced_value
    CALL modify_value (tmp_real_value)
    reduced_value = tmp_real_value

    ! Write the value contained within the shadow variable.
    WRITE (*, *) reduced_value%get_value()

CONTAINS

    SUBROUTINE modify_value (x)
    ! Performs some simple modification on a real number, in this case
    ! just squares it, but this could be any black-box operation performed
    ! in full precision.
    !
        REAL(KIND=RPE_REAL_KIND), INTENT(INOUT) :: x
        x = x * x
    END SUBROUTINE modify_value

END PROGRAM
```

Output:

```
10.781250000000000
```

5.3.2 Using `rpe_shadow`

```
PROGRAM myprog_shadow
! Program with main variable replaced by a reduced precision variable.
!

    USE rp_emulator
    IMPLICIT NONE

    ! A real variable (the main variable) and a reduced precision shadow
    ! variable.
    REAL(KIND=RPE_REAL_KIND) :: real_value
    TYPE(rpe_shadow)         :: reduced_value

    ! Use 8 bits of significand precision for the reduced precision
    ! variable
    reduced_value%sbits = 8
    CALL init_shadow (reduced_value, real_value)
```

```
    ! Assign a value to the reduced precision variable. In a real program
    ! there could be an arbitrary mount of computation here.
    reduced_value = 3.2812147

    ! Call the subroutine modify_value, this expects a real number rather
    ! than a reduced precision type. Since the variables 'real_value' and
    ! 'reduced_value' refer to the same block of memory we can just input
    ! the 'real_value' variable and modifications to it will also appear in
    ! 'reduced_value'. However, since the modification is done in double
    ! precision, we'll need to call apply_truncation to ensure the resulting
    ! value is stored at reduced precision.
    CALL modify_value (real_value)
    CALL apply_truncation (reduced_value)

    ! Write the value contained within the shadow variable.
    WRITE (*, *) reduced_value%get_value()

CONTAINS

    SUBROUTINE modify_value (x)
    ! Performs some simple modification on a real number, in this case
    ! just squares it, but this could be any black-box operation performed
    ! in full precision.
    !
        REAL(KIND=RPE_REAL_KIND), INTENT(INOUT) :: x
        x = x * x
    END SUBROUTINE modify_value

END PROGRAM
```

Output:

```
10.781250000000000
```

CHAPTER 6

Potential Sources of Error

There are a few dark corners of both the library and Fortran itself which may catch out unsuspecting users.

6.1 Initialization on assignment

The emulator overloads the assignment operator allowing you to assign integers, real numbers, and other *rpe_type* instances to any *rpe_type* instance:

```
PROGRAM assign_other_types

    USE rp_emulator
    IMPLICIT NONE

    TYPE(rpe_var) :: x

    ! Assign an integer value to x.
    x = 4

    ! Assign a single-precision real value to x.
    x = 4.0

    ! Assign a double-precision real value to x.
    x = 4.0d0

END PROGRAM
```

However, you cannot perform the same assignments at the time the *rpe_type* is defined. This is not allowed:

```
PROGRAM assign_other_types

    USE rp_emulator
    IMPLICIT NONE

    TYPE(rpe_var) :: x = 4

END PROGRAM
```

Compiling this code with *gfortran* will give the following compilation error:

```
    TYPE(rpe_var) :: x = 4
                         1
Error: Can't convert INTEGER(4) to TYPE(rpe_var) at (1)
```

This is because this form of assignment does not actually call the overloaded assignment operator defined for *rpe_var* instances, instead it calls the default constructor for the derived type, which only allows the right-hand-side of the assignment to be another *rpe_var* instance.

6.2 Changing RPE_DEFAULT_SBITS during execution

You are able to change the value of the module variable *RPE_DEFAULT_SBITS* to anything you like at any time you like. However, you need to be aware of the potential inconsistencies you might introduce by doing so.

```
PROGRAM change_default_bits1

    USE rp_emulator
    IMPLICIT NONE

    TYPE(rpe_var) :: pi

    RPE_DEFAULT_SBITS = 16

    ! This value of Pi will be stored with only 16 bits in the mantissa.
    pi = 3.1415926535897932d0
    WRITE (*, '("RPE_DEFAULT_SBITS=16, pi=", F20.18)') pi%get_value()

    ! Doing this means that whilst any operations on Pi following will assume
    ! 4 bits of significand precision, the value currently stored still has 16
    ! bits of significand precision
    RPE_DEFAULT_SBITS = 4
    WRITE (*, '("RPE_DEFAULT_SBITS=4,  pi=", F20.18)') pi%get_value()

END PROGRAM
```

Output:

```
RPE_DEFAULT_SBITS=16, pi=3.141601562500000000
RPE_DEFAULT_SBITS=4,  pi=3.141601562500000000
```

To avoid any issues you may want to insert manual calls to *apply_truncation()* to ensure every variable used within the scope of the changed default is represented at the required precision.

In other circumstances this may not be a problem at all, for example around encapsulted subroutine calls. In the example below the procedure `some_routine()` takes no reduced precision types as arguments, but does work with reduced precision types internally, and in this case setting the default number of bits around the subroutine call is a useful way to set the default precision of all reduced precision variables within the subroutine (and within any routines it calls):

```
RPE_DEFAULT_SBITS = 4
CALL some_routine (a, b, c)
RPE_DEFAULT_SBITS = 16
```

Whatever you choose to do, you need to make sure you have considered this potential issue before you change the value of *RPE_DEFAULT_SBITS* at run-time.

6.3 Parallel and thread safety

The default number of bits for any reduced precision type is controlled by a module variable *RPE_DEFAULT_SBITS*. This is a mutable variable that can be changed dynamically during program execution if desired. If the application

using the emulator is parallelised then the behaviour of the default bits setting needs to be considered.

For MPI parallelism, each MPI task will get its own separate instance of the *RPE_DEFAULT_SBITS* variable, and modifying it within a task will only affect the default precision within that task (unless programmed otherwise using message passing).

For threaded parallelism (e.g. OpenMP) the behaviour is less clear. Depending on the threading type used, the variable may be shared by threads, or each may have its own copy.

For parallel applications, care must be taken when changing the value of *RPE_DEFAULT_SBITS* at run time to make sure the implementation is safe.

6.4 Non-equivalence of single and compound operations

One would normally expect the following operations to yield identical results:

```
a * a * a
```

and

```
a ** 3
```

However, due to the way the emulator works by doing individual computations in full precision and reducing the precision of the result, these two would not necessarily yield the same result if `a` were a reduced precision variable. In the first example the compound multiplication would be done in two parts, the first part computes `a * a` and the precision of the temporary result is reduced, then the second part multiplies this reduced precision result by `a` and once again reduces the precision of the final result. However, in the second example a single operation is used to express the computation, this coimputation will be performed in full precision and the result will have its precision reduced. Whereas the first example uses reduced precision to store intermediate results, the second does not.

This is true for any operator that can be expressed as multiple invocations of other operators.

CHAPTER 7

Developer Guide

This developer guide provides extra documentation required for doing development work on the reduced precision emulator.

7.1 Getting Started

7.1.1 Requirements

In addition to the user requirements, there are extra dependencies for developers:

Core

- GCC (gfortran) >= 4.8
- GNU Make
- Git

Test suites

To run the test suites the following software is required:

- Python 2.7
- pFUnit (tested with version 3.1)

Code generator

Running the code generator requires:

- Python 2.7
- Jinja2

Documentation

If you want to build the documentation you will need:

- Python 2.7
- Sphinx >= 1.3.1

7.1.2 Get the source code

Follow the instructions in Git for Development to set up your source code repository and version control tools.

7.2 Software Structure

The emulator library itself is a Fortran library that can be included in other programs to introduce reduced precision variables. However, the code repository contains several distinct components which are used to generate the final Fortran library.

7.2.1 The core library

The core of the emulator is written in Fortran, and is found in the file `rawsrc/rp_emulator.F90`. The emulator library provides two core derived types used to represent reduced precision floating point numbers. These types have accompanying overloaded operator definitions, allowing them to be used in the same way as the builtin real number type. For maximum ease of use, many of the builtin Fortran intrinsic functions are also overloaded to accept these reduced precision types.

The `rawsrc/rp_emulator.F90` file contains the module definition, the definition of the derived types, and the core functions and subroutines that control the emulator's functionality. However, it does not contain most of the overloaded operator and intrinsic function definitions. Instead there are a handful of C preprocessor directives in the file that pull in these definitions from external files in `rawsrc/include`. The way in which these external file are generated is described below.

7.2.2 The code generator

A lot of the code required to complete the full emulator is repetetive boiler-plate code; code which is largely the same for a whole group of functions/subroutines. Because of the repetetive nature of the required Fortran implementations for all overloaded operators and intrinsic functions, it is more efficient to provide only a template of what the code should look like, and let the computer actually write the code. The code generator for the emulator is written in Python and is included in the `generator/` subdirectory.

Using a generator is a big time saver, if you need to change 1 line of every intrinsic function implementation, you only need to modify that line in a few templates and let the code generator write all the actual Fortran code for you. Not only does this approach save time, it also has positive implications for code correctness. For example if 50 function implementations are produced by the generator, it is not possible for one of them to contain an error that the others don't, as would often be the case when hand-writing sucha a large number of essentially similar routines. You write the implementation carefully once, and the boring stuff is done automatically.

Due to its relative complexity, the Code Generator is documented separately.

7.2.3 Extra definitions

As well as code that is produced by the code generator, there are two more files in `rawsrc/include` that can be edited manually: `interface_extras.i` and `implementation_extras.f90`. You can add arbitrary functions/subroutines to the `implementation_extras.i` file, with any required interface definitions in `interface_extras.i`, and they will be included in the main library source file automatically. Just make sure you make your interface public, as the default is private.

7.2.4 Tests

The emulator is provided with a suite of basic tests, in the `tests/` subdirectory. It is expected that any change you make will not cause any of the existing tests to fail, and ideally new features should be accompanied with new tests to make sure they work, and continue to work in the future.

The tests are split into two categories, units tests and integration tests. Unit tests should be relatively self-contained tests of the functionality of a particular small element of the code (code unit). Ideally these tests are for a single component in isolation from the rest of the system, although the nature of the library means this sometimes is not possible. Integration tests are more like realistic tests of the software in use, and should tests multiple aspects of the library together (in integration).

7.3 Git for Development

This section describes the typical git workflow for development of the project. It covers both how to use git to achieve the required results, and the general version control strategy for the project.

7.3.1 Setting up a development fork

Create your own forked copy of the emulator

1. Log in to your Gitlab account.
2. Go to the rpe repository page.
3. Click the fork button at the centre of the page, and on the next page click your Gitlab user name.

Clone your fork

1. Clone your fork to the local computer with:

```
git clone git@gitlab.physics.ox.ac.uk:your-user-name/rpe.git
```

2. Investigate. Change directory to your new repo: `cd rpe`. Then `git branch -a` to show you all branches. You'll get something like:

```
* master
remotes/origin/master
```

   This tells you that you are currently on the `master` branch, and that you also have a `remote` connection to `origin/master`. What remote repository is `remote/origin`? Try `git remote -v` to see the URLs for the remote. They will point to your Gitlab fork.

   Now you want to connect to the upstream rpe repository, so you can merge in changes from trunk.

Linking your repository to the upstream repository

```
cd rpe
git remote add upstream git@gitlab.physics.ox.ac.uk:aopp-pred/rpe.git
```

`upstream` here is just the arbitrary name we're using to refer to the main rpe repository.

The upstream repository is read-only access, which means you can't accidentally (or deliberately) write to the upstream repository, you only use it to merge into your own code.

You can list your remote repositories with `git remote -v`, which gives you something like this:

```
upstream    git@gitlab.physics.ox.ac.uk:aopp-pred/rpe.git (fetch)
upstream    git@gitlab.physics.ox.ac.uk:aopp-pred/rpe.git (push)
origin      git@gitlab.physics.ox.ac.uk:your-user-name/rpe.git (fetch)
origin      git@gitlab.physics.ox.ac.uk:your-user-name/rpe.git (push)
```

Tell git who you are

It is good practice to tell git who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

This will write the settings into your git configuration file. Make sure that the email address you use is one that Gitlab knows about. You can either use the default address (your @physics.ox.ac.uk address) or add other email addresses you want to use via the Gitlab settings page.

7.3.2 Development workflow

Summary

- Don't use your master branch for anything.
- When you are starting a new set of changes, fetch any changes from upstream trunk, and start a new feature branch from that.
- Make a new branch for each separable set of changes - "one task, one branch".
- Name your branch for the purpose of the changes - e.g. bugfix-for-issue-14 or refactor-database-code.
- If you can possibly avoid it, avoid merging trunk or any other branches into your feature branch while you are working.
- If you do find yourself needing to merge from trunk, consider Rebasing on trunk instead.
- Ask for code review by submitting a pull request!

Update your mirror of trunk

First make sure you have done *Linking your repository to the upstream repository*.

From time to time, and always before creating a new feature branch, you should fetch the upstream (trunk):

```
git fetch upstream
```

This will pull down any commits you don't have, and set the remote branches to point to the right commit. For example, 'trunk' is the branch referred to by (remote/branchname) upstream/master - and if there have been commits since you last checked, upstream/master will change after you do the fetch.

Make a new feature branch

When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called 'feature branches'.

Making an new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. For example `add-ability-to-fly`, or `buxfix-for-issue-42`.

```
# Update the mirror of trunk
git fetch upstream
# Make a new feature branch starting at current trunk
git checkout -b my-new-feature upstream/master
```

Generally, you will want to keep your feature branches on your fork of rpe. To do this, you git push this new branch up to your Gitlab repository. Generally (if you followed the instructions in these pages, and by default), git will have a link to your Gitlab repository, called `origin`. You push up to your own repo on Gitlab with:

```
git push origin my-new-feature
```

In git >= 1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in the Gitlab repository.

Making changes

1. Make some changes on your feature branch.
2. See which files have changed with `git status`. You'll see a listing like this one:

```
# On branch my-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#  modified:   rawsrc/rp_emulator.F90
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  tests/unit/new-suite
no changes added to commit (use "git add" and/or "git commit -a")
```

3. Check what the actual changes are with `git diff` (git diff).
4. Add any new files to version control `git add new_file_name` (see git add).
5. To commit all modified files into the local copy of your repository, do `git commit -a`. Note the `-a` option to `commit`, which automatically performs *git add* on all modified files. You may prefer to manually add the

files you wish to commit and then just use plain `git commit`. The latter method is useful if you want to make a commit out of only some of the modified files (this is quite normal).

6. To push the changes up to your forked repository on Gitlab, do a `git push` (see git push).

Ask for your changes to be reviewed or merged

When you are ready to ask for someone to review your code and consider a merge:

1. Go to the URL of your forked repo, say `https://gitlab.physics.ox.ac.uk/your-user-name/rpe.git`.
2. Click the plus sign button on the row of action buttons under the repository name on the main page, then click *New merge request* in the menu that pops up.
3. Use the drop-down boxes to select the branch you wish to be reviewed under the *Source branch* section, and the branch that this feature should be merged into (usually master) under the *Target branch* section.
4. Click *Compare branches*.
5. Enter a title for the set of changes, and some explanation of what you've done. Say if there is anything you'd like particular attention for - like a complicated change or some code you are not happy with or are unsure about.

   If you don't think your request is ready to be merged, just say so in your pull request message. This is still a good way of getting some preliminary code review.

   You can also select reviewers for your change. Doing so will send them a notification that you have submitted a pull request, so it is a good idea to name at least 1 reviewer in order to get your request reviewed in a timely manner.
6. When you are happy that you have entered all the required information you can click the *Submit new merge request* button.

Note: Before submitting a pull request you should first check that all the tests pass locally yourself. This will save the reviewer from having to point out that your shiny new feature breaks something!

7.3.3 Some other things you might want to do

Delete a branch on Gitlab

```
git checkout master
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on Gitlab
git push origin :my-unwanted-branch
```

Note the colon `:` before `my-unwanted-branch`.

Explore your repository

To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

Rebasing on trunk

Let's say you thought of some work you'd like to do. You *Update your mirror of trunk* and *Make a new feature branch* called `cool-feature`. At this stage trunk is at some commit, let's call it E. Now you make some new commits on your `cool-feature` branch, let's call them A, B, C. Maybe your changes take a while, or you come back to them after a while. In the meantime, trunk has progressed from commit E to commit (say) G:

```
      A---B---C cool-feature
     /
D---E---F---G trunk
```

At this stage you consider merging trunk into your feature branch, and you remember that this here page sternly advises you not to do that, because the history will get messy. Most of the time you can just ask for a review, and not worry that trunk has got a little ahead. But sometimes, the chan But sometimes, the changes in trunk might affect your changes, and you need to harmonize them. In this situation you may prefer to do a rebase.

Rebase takes your changes (A, B, C) and replays them as if they had been made to the current state of `trunk`. In other words, in this case, it takes the changes represented by A, B, C and replays them on top of G. After the rebase, your history will look like this:

```
              A'--B'--C' cool-feature
             /
D---E---F---G trunk
```

See rebase without tears for more detail.

To do a rebase on trunk:

```
# Update the mirror of trunk
git fetch upstream
# go to the feature branch
git checkout cool-feature
# make a backup in case you mess up
git branch tmp cool-feature
# rebase cool-feature onto trunk
git rebase --onto upstream/master upstream/master cool-feature
```

In this situation, where you are already on branch `cool-feature`, the last command can be written more succinctly as:

```
git rebase upstream/master
```

When all looks good you can delete your backup branch:

```
git branch -D tmp
```

If it doesn't look good you may need to have a look at *Recovering from mess-ups*.

If you have made changes to files that have also changed in trunk, this may generate merge conflicts that you need to resolve - see the git rebase man page for some instructions at the end of the "Description" section. There is some related help on merging in the git user manual - see resolving a merge.

Recovering from mess-ups

Sometimes, you mess up merges or rebases. Luckily, in git it is relatively straightforward to recover from such mistakes.

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

```
# reset branch back to the saved point
git reset --hard tmp
```

If you forgot to make a backup branch:

```
# look at the reflog of the branch
git reflog show cool-feature

8630830 cool-feature@{0}: commit: BUG: io: close file handles immediately
278dd2a cool-feature@{1}: rebase finished: refs/heads/my-feature-branch onto 11ee694744f2552d
26aa21a cool-feature@{2}: commit: BUG: lib: make seek_gzip_factory not leak gzip obj
...

# reset the branch to where it was before the botched rebase
git reset --hard cool-feature@{2}
```

Rewriting commit history

Note: Do this only for your own feature branches.

There's an embarassing typo in a commit you made? Or perhaps the you made several false starts you would like the posterity not to see.

This can be done via *interactive rebasing*.

Suppose that the commit history looks like this:

```
git log --oneline
eadc391 Fix some remaining bugs
a815645 Modify it so that it works
2dec1ac Fix a few bugs + disable
13d7934 First implementation
6ad92e5 * masked is now an instance of a new object, MaskedConstant
29001ed Add pre-nep for a copule of structured_array_extensions.
...
```

and `6ad92e5` is the last commit in the `cool-feature` branch. Suppose we want to make the following changes:

- Rewrite the commit message for `13d7934` to something more sensible.
- Combine the commits `2dec1ac`, `a815645`, `eadc391` into a single one.

We do as follows:

```
# make a backup of the current state
git branch tmp HEAD
# interactive rebase
git rebase -i 6ad92e5
```

This will open an editor with the following text in it:

```
pick 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
pick a815645 Modify it so that it works
pick eadc391 Fix some remaining bugs
```

```
# Rebase 6ad92e5..eadc391 onto 6ad92e5
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

To achieve what we want, we will make the following changes to it:

```
r 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
f a815645 Modify it so that it works
f eadc391 Fix some remaining bugs
```

This means that (i) we want to edit the commit message for `13d7934`, and (ii) collapse the last three commits into one. Now we save and quit the editor.

Git will then immediately bring up an editor for editing the commit message. After revising it, we get the output:

```
[detached HEAD 721fc64] FOO: First implementation
 2 files changed, 199 insertions(+), 66 deletions(-)
[detached HEAD 0f22701] Fix a few bugs + disable
 1 files changed, 79 insertions(+), 61 deletions(-)
Successfully rebased and updated refs/heads/my-feature-branch.
```

and the history looks now like this:

```
0f22701 Fix a few bugs + disable
721fc64 ENH: Sophisticated feature
6ad92e5 * masked is now an instance of a new object, MaskedConstant
```

If it went wrong, recovery is again possible as explained *above*.

7.4 Code Generator

A large amount of the code included in the Fortran library is constructed programmatically by a code generator. The code generator is a small library written in Python, that uses templates to generate fortran code, as well as front-ends to generate the required functions, subroutines and interface blocks for all overloaded operators and instrinsic functions.

7.4.1 Structure of the Code Generator

The code generator consists of a small Python library for code generation, 2 configuation files for determining what code is generated, and two driver programs to build the code. The code generator itself resides within the top-level `generator/` directory. At the top level is a Makefile used to run the code generator, the directory containing generated code

Configuration files

The code generator driver programs require input files that define which overloaded operators and intrinsic functions need to be constructed. These files can be found in the `generator/configs` directory. The file `operators.json` defines overloaded operators and the file `intrinsics.json` defines the overloaded intrinsic functions; the files are in JSON format.

Python components

The python components are located in the subdirectory `generator/python`. In this directory are two Python programs `generate_intrinsics.py` and `generate_operators.py`. These programs take as input a JSON configuration file and output Fortran code files.

Also in this directory is another directory named `rpgen/`, this is the Python library for code generation.

7.4.2 Using the Code Generator

The basic workflow for using the code generator is detailed below, and can be summarised as:

1. Make some change to the code to be generated, either by modifying the JSON configuration files or by modifying templates or even the generator library itself.
2. Run the generator.
3. Integrate the generated source code into the main Fortran library.

Defining the code to be generated

What the code generator produces can be controlled either by configuration files, or modifying the generator itself. Modifying configuration files is covered in Adding New Operators or Intrinsic Functions.

Running the generator

The generator can be run from the top-level `generator/` directory using the Makefile:

```
make -C generator
```

This will produce 4 files in the directory `generator/generated`:

- `interface_operators.i`: interface blocks for overloaded operators.
- `implementation_operators.f90`: overloaded operator implementations.
- `interface_intrinsics.i`: interface blocks for overloaded intrinsic functions.
- `implementation_intrinsics.f90`: overloaded intrinsic function implementations.

Despite the file extensions, all 4 files contain Fortran code, the extensions are simply part of a naming convention.

Integrating generated code into the Fortran library

Once you have generated new files, you will want to integrate the generated code into the main Fortran library.

Warning: Make sure you carefully check the generated code is correct before proceeding. It is worthwhile looking at both the generator output, and the difference between each output and the existing file in `rawsrc/include`, for example:

```
diff generated/implementation_operators.f90 rawsrc/include/implementation_operators.f90
```

This way you can be sure that you have achieved the change you wanted without changing something else you didn't expect to change.

To integrate the generated code all you need to do is copy/move the generated files from `generator/generated` to `rawsrc/include`:

```
cp generator/generated/* rawsrc/include
```

Once you have done this you should rebuild the library:

```
make fullclean
make all
```

Now you can run the tests and verify that the new code works as expected:

```
cd tests
make test
```

7.4.3 Adding New Operators or Intrinsic Functions

Extending the emulator by adding new overloaded operators or intrinsic functions is fairly straightforward, and usually requires only editing configuration files.

Adding a new intrinsic function

In this example we'll add the GAMMA gfortran intrinsic to the emulator, allowing us to evaluate the Γ function with a reduced precision input.

Overview of existing definitions

We'll need to add a new entry to the `intrinsics.json` file in `generator/configs` to define this new function, but first let's take a look at some of the existing function definitions:

```
{"intrinsics":
    [
        ...
        {"epsilon":
            {
                "return_type": "rpe_var",
                "interface_types": ["1argscalar"]
            }
        },
        ...
        {"floor":
            {
                "return_type": "integer",
                "interface_types": ["1argelemental"]
            }
```

```
        },
        ...
    ]
}
```

Each function is defined as a JSON object, with a name that corresponds to the name of the intrinsic function, and two attributes that determine the return type of the function and the types of interface the function has.

In the above snippet you can see that the function `epsilon` has a return type of `"rpe_var"`, which corresponds to an *rpe_var* instance. The function `floor` has a return type of `"integer"`, which just corresponds to a normal Fortran integer type. A complete list of types that can be used in configuartion files can be found in *Type names and variables*.

The interface type is a concept used within the code generator to work out what kind of code it should produce. You can see that the function `epsilon` has interface type `"1argscalar"` which corresponds to a function that takes one scalar as an argument. The function `floor` has interface type `"1argelemental"` which corresponds again to a function that takes one argument, but this time the function is elemental meaning it can take a scalar or an array as input, and operate element-wise on array inputs returning an array output. A function can have more than one interface type if it has multiple interfaces (e.g. `atan`). See *Intrinsic function interface types* for a list of all intrinsic interface types.

Writing the new definition

From the gfortran GAMMA documentation we can see that our new function should accept a single input of a reduced precision number, and return a single output which will also be a reduced precision number. We can also see that the function should be elemental, meaning it can be applied element-wise to an array of input values. Therefore our definition in `intrinsics.json` should look like this:

```
{"intrinsics":
    [
        ...
        {"gamma":
            {
                "return_type": "rpe_var"
                "interface_types": ["1argelemental"]
            }
        }
    ]
}
```

Warning: Never use `"rpe_shadow"` as the return type of a function as it is not possible to properly initialize an *rpe_shadow* instance within the limited scope of a function and have it work correctly in the external scope of the function's return value. If you want to return a reduced precision value from a function always use an *rpe_var* type by specifying `"rpe_var"`.

Generating the code

Now that you have created the definition for GAMMA you need to use the generator to actually write the code. The simplest way to do this is by using the Makefile in the `generator/` directory:

```
cd generator/
make
```

This command will generatre a new set of files in the `generated/` subdirectory, and you can inspect these to verify that correct code was written for a GAMMA imnplementation on reduced precision types. First lets look at `interface_intrinsics.i`, it now has these extra lines:

```
PUBLIC :: gamma
INTERFACE gamma
    MODULE PROCEDURE gamma_rpe
END INTERFACE gamma
```

These lines define a public interface for a function `gamma`, with one member function called `gamma_rpe`. Now let's look in the newly generated `implementation_intrinsics.f90` to see the implementation of `gamma_rpe`:

```
!-----------------------------------------------------------------
! Overloaded definitions for 'gamma':
!

ELEMENTAL FUNCTION gamma_rpe (a) RESULT (x)
    CLASS(rpe_type), INTENT(IN) :: a
    TYPE(rpe_var) :: x
    x%sbits = significand_bits(a)
    x = GAMMA(a%get_value())
END FUNCTION gamma_rpe
```

The generated implementation consists of a single elemental function definition accepting any *rpe_type* type or a type that extends it (either *rpe_var* or *rpe_shadow*) and returns an *rpe_var* type. The body of the function is simple, it simply sets the nmumber of bits in the significand of the return value to match the input, then calls the normal Fortran `GAMMA` intrinsic with the real value cointained by the reduced precision number as input and stores the result in the output variable `x`. The precision of the return value `x` is reduced by the assignment operation.

To include this code in a build of the library simply follow the instructions in *Integrating generated code into the Fortran library*.

Adding a new operator

The process of adding a new operator proceeds much like *Adding a new intrinsic function*, except with a different configuration file and different JSON attributes. In this example we'll pretend that we don't already have a `**` operator and implement one.

The JSON configuration for operators is the `operators.json` file in `generator/configs`. An operator definition looks like this:

```
{<name>:
    {
        "operator": <operator-symbol>,
        "return_type": <return-type>,
        "operator_categories": [<categories>]
    }
}
```

In this example `<name>` is the name of the operator, in our case this will be `"pow"`; `<operator-symbol` is the symbol used to represent the operator, which in our case will be `"**"`; `<return-type>` is just the type that will be returned by the operator, in this case we want to return a reduced precision value so we will use `"rpe_var"` as the return type. The value supplied for `"operator_categories"` is a list of the categories this operator falls into. There are only 2 categories available, `"unary"` for unary operators and `"binary"` for binary operators. The list of categories can contain one or both of these values if appropriate, but is our case exponentiation is a binary operator so we'll supply the one value `["binary"]`.

Warning: Never use "rpe_shadow" as the return type of an operator as it is not possible to properly initialize an *rpe_shadow* instance within the limited scope of an operator and have it work correctly in the external scope of the operator's return value. If you want to return a reduced precision value from an operator always use an *rpe_var* type by specifying "rpe_var".

Generating the code for the new operator just requires running the Makefile in generator/:

```
cd generator/
make
```

Let's take a look at what was generated in the generated/ subdirectory, firstly in the interface_operators.i file:

```
PUBLIC :: OPERATOR(**)
INTERFACE OPERATOR(**)
    MODULE PROCEDURE pow_rpe_rpe
    MODULE PROCEDURE pow_rpe_integer
    MODULE PROCEDURE pow_rpe_long
    MODULE PROCEDURE pow_rpe_real
    MODULE PROCEDURE pow_rpe_realalt
    MODULE PROCEDURE pow_integer_rpe
    MODULE PROCEDURE pow_long_rpe
    MODULE PROCEDURE pow_real_rpe
    MODULE PROCEDURE pow_realalt_rpe
END INTERFACE OPERATOR(**)
```

This defines a public interface for the ** operator, which contains 9 member functions. These functions deal with all possible input combinations for the operator. Now let's look at how these operators are defined in the generated implementation_operators.f90 file, we'll just show a few of the 9 definitions to get a feel for what is generated:

```
!-----------------------------------------------------------------------
! Overloaded definitions for (**):
!

ELEMENTAL FUNCTION pow_rpe_rpe (x, y) RESULT (z)
    CLASS(rpe_type), INTENT(IN) :: x
    CLASS(rpe_type), INTENT(IN) :: y
    TYPE(rpe_var) :: z
    z%sbits = MAX(significand_bits(x), significand_bits(y))
    z = x%get_value() ** y%get_value()
END FUNCTION pow_rpe_rpe

...

ELEMENTAL FUNCTION pow_rpe_real (x, y) RESULT (z)
    CLASS(rpe_type), INTENT(IN) :: x
    REAL(KIND=RPE_REAL_KIND), INTENT(IN) :: y
    TYPE(rpe_var) :: z
    z%sbits = MAX(significand_bits(x), significand_bits(y))
    z = x%get_value() ** y
END FUNCTION pow_rpe_real

...

ELEMENTAL FUNCTION pow_real_rpe (x, y) RESULT (z)
    REAL(KIND=RPE_REAL_KIND), INTENT(IN) :: x
    CLASS(rpe_type), INTENT(IN) :: y
```

```
    TYPE(rpe_var) :: z
    z%sbits = MAX(significand_bits(x), significand_bits(y))
    z = x ** y%get_value()
END FUNCTION pow_real_rpe
```

The first definition defines how the `**` operator can be applied to two *rpe_type* instances. It can operate on either *rpe_var* or *rpe_shadow* types for each argument and returns an *rpe_var* instance. The number of bits in the significand of the result is set to the larger of the number of bits in the significands of the inputs, the calculation is then done in full precision and reduced to the specified precision on assignment to the return value `z`.

The other two functions do something very similar, except they operate on inputs of one reduced precision type and one real number type, the first raising a reduced precision number to the power of a real number, and the second raising a real number to the power of a reduced precision number.

Now that the code for the new operator has been generated and checked it can be included in a build of the library by following the instructions in *Integrating generated code into the Fortran library*.

7.4.4 Reference

Type names and variables

Type name (JSON)	Generator Type variable	Fortran type
`"logical"`	`rpgen.types.LOGICAL`	`LOGICAL`
`"integer"`	`rpgen.types.INTEGER`	`INTEGER`
`"long"`	`rpgen.types.LONG`	`INTEGER(KIND=8)`
`"real"`	`rpgen.types.REAL`	`REAL(KIND=RPE_REAL_KIND)`
`"realalt"`	`rpgen.types.REALALT`	`REAL(KIND=RPE_ALTERNATE_KIND)`
`"rpe_type"`	`rpgen.types.RPE_TYPE`	`CLASS(rpe_type)`
`"rpe_var"`	`rpgen.types.RPE_VAR`	`TYPE(rpe_var)`
`"rpe_shadow"`	`rpgen.types.RPE_SHADOW`	`TYPE(rpe_shadow)`

Operator categories

Operator category (JSON)	Definition
`"unary"`	A unary operator with one input and one output.
`"binary"`	A binary operator with two inputs and one output.

Intrinsic function interface types

Interface name (JSON)	Definition
`"1argscalar"`	A function with one scalar argument.
`"1argelemental"`	An elemental function with one scalar or array argument.
`"2argelemental"`	An elemental function with two scalar or array arguments.
`"1arrayarg"`	A function with one array argument and a scalalr return value.
`"multiarg"`	An elemental function with multiple arguments.

CHAPTER 8

API Reference

API documentation for the Fortran library:

8.1 RPE API: `rp_emulator.mod`

8.1.1 Derived types

type `rpe_type` *[abstract]*
An abstract type defining a reduced-precision floating-point number. You cannot construct an instance of this type, but you can construct an instance of any type that extends this type.

Type fields

- **`% sbits`** *[INTEGER]* :: Number of bits in the significand of the floating-point number

function `rpe_type%`**`get_value`**`()`
Retrieve the real value stored within an *`rpe_type`* instance.

Return value *[REAL,KIND=RPE_REAL_KIND]* :: Reduced-precision value stored in a built-in real-typed variable.

type `rpe_var` *[extends(rpe_type)]*
A type extending *`rpe_type`* representing a reduced-precision floating-point number. This type stores the number it represents internally.

type `rpe_shadow` *[extends(rpe_type)]*
A type extending *`rpe_type`* representing a reduced-precision floating-point number. The *`rpe_shadow`* provides a memory-view onto an existing double precision floating point number defined outside the type itself. Changing the value of the *`rpe_shadow`* also changes the value of the floating point number it is shadowing and vice-versa, since they both refer to the same block of memory. However, when values are assigned to the *`rpe_shadow`* their precision is reduced, which is not the case when assigning to the variable being shadowed.

8.1.2 User-callable procedures

subroutine `init_shadow`(*shadow*, *target*)
Initialize an *`rpe_shadow`* instance by associating it with a real-valued variable.

Parameters

- **shadow** *[*rpe_shadow,*INOUT]* :: The *`rpe_shadow`* instance to initialize.

- **target** *[REAL,KIND=RPE_REAL_KIND,IN]* :: A floating-point variable to shadow. This must be a variable defined within the scope of the *init_shadow()* call otherwise invalid memory references will occur.

subroutine apply_truncation(*rpe*) *[elemental]*

Apply the required truncation to the value stored within an *rpe_type* instance. Operates on the input in-place, modifying its value. The truncation is determined by the `sbits` attribute of the *rpe_type* instance, if this is not set then the value of *RPE_DEFAULT_SBITS*.

Parameters **rpe** *[*rpe_type*,INOUT]* :: The *rpe_type* instance to alter the precision of.

8.1.3 Variables

RPE_ACTIVE *[LOGICAL,default=.TRUE.]*

Logical value determining whether emulation is on or off. If set to `.FALSE.` then calls to *apply_truncation()* will have no effect and all operations will be carried out at full precision.

RPE_DEFAULT_SBITS *[INTEGER,default=23]*

The default number of bits used in the significand of an *rpe_type* instance when not explicitly specified. This takes effect internally when determining precision levels, but does not bind an *rpe_type* instance to a particular precision level (doesn't set `rpe_type%sbits`).

RPE_IEEE_HALF *[LOGICAL,default=.FALSE.]*

Logical value determining if IEEE half-precision emulation is turned on. If set to `.TRUE.` and a 10-bit significand is being emulated the emulator will additionally impose range constraints when applying truncation:

- Values that overflow IEEE half-precision will lead to real overflows with a corresponding floating-point overflow exception.
- Values out of the lower range of IEEE half-precision will be denormalised.

This option only affects the emulation when emulating a 10-bit significand.

8.1.4 Parameters

RPE_DOUBLE_KIND *[INTEGER]*

The kind number for double precision real types.

RPE_SINGLE_KIND *[INTEGER]*

The kind number for single precision real types.

RPE_REAL_KIND *[INTEGER]*

The kind number of the real-values held by reduced precision types. This is a reference to *RPE_DOUBLE_KIND*, but could be changed (in source) to be *RPE_SINGLE_KIND*.

RPE_ALTERNATE_KIND *[INTEGER]*

The kind number of an alternate type of real-value. This is a reference to *RPE_SINGLE_KIND*, but can be changed (in source) if the value referenced by *RPE_REAL_KIND* is changed.

CHAPTER 9

Reduced Precision Emulator

This software provides a Fortran library for emulating the effect of low-precision real numbers. It is intended to be used as an experimental tool to understand the effect of reduced precision within numerical simulations.

A

G

I

R