The RealityGrid Computational Steering API

Version 1.2

27th January, 2006

Robert Haines, Stephen Pickles, Robin Pinning, Andrew Porter[1], Graham Riley, Rupert Ford, Ken Mayes[2], David Snelling[3], Jim Stanton[4], Steven Kenny[5], Shantenu Jha[6].

1. Manchester Computing, University of Manchester
2. CNC, Department of Computer Science, University of Manchester
3. Fujitsu Laboratories of Europe
4. Previously at Department of Computing, Imperial College
5. Department of Mathematical Sciences, Loughborough University
6. University College London

Change Log

Author	Comments	Date
Andrew Porter	As released with v.1.1 of the RealityGrid steering software.	29th April 2004
Andrew Porter	Corrections plus new routines: Add_checkpoint_file, En/Disable_IOType_acks	13th July 2004
Andrew Porter	Updated for v.1.2b of the steering library.	27th May 2005
Andrew Porter	Updated with bug fix for 1.2b	12th Dec 2005
Andrew Porter	Added Robert Haines as an author. Release as version 1.2	30th Jan 2006

Table of Contents

1 Introduction....3
1.1 Background assumptions and general considerations....3
2 An API for Computational Steering....4
2.1 General....4
2.2 Notation....4
2.3 Pre-defined steering commands....5
2.4 Application-side routines....6
2.4.1 Utility routines....6
2.4.2 Startup and shutdown....6
2.4.3 Parameters....7
2.4.4 IO control....11
2.4.5 Data output....15
2.4.6 Data input....17
2.4.7 Checkpoint control....20
2.4.8 Communication with steering client....23
2.5 Steering-client-side routines....26
2.5.1 Startup and shutdown....26
2.5.2 Discovery....26
2.5.3 Attach/detach....27
2.5.4 Communication with steered application....28
2.5.5 Query functions....30
2.5.6 Editing routines....35
2.5.7 Debugging routines....36
4....36
5 Acknowledgements....36
Appendix A: Return values....37
Appendix B: Example usage of Emit_data_slice....38
Appendix C: Coding scheme for data types....40
Interface index....41

1 Introduction

This document describes an Application Program Interface (API) to a computational steering framework. The term computational steering describes the process by which a scientist interacts with a running program (often some form of simulation of a physical system) in order to make the investigation/problem-solving task more efficient.
In the remainder of this section we describe the intended use of the API and some of the underlying design choices.

1.1 Background assumptions and general considerations

Application scientists have pre-existing codes (hereafter referred to as *simulations*), which are to be made steerable. These codes are written in Fortran, C, C++ or a mixture of these, and can be compiled using standards-compliant Fortran90 and C/C++ compilers. These codes will in general be parallel, but we avoid assuming or prescribing any particular paradigm (*e.g.* message passing or shared memory) or harness (*e.g.* MPI, PVM, SHMEM). As a consequence of this, the API presented here is designed to be *called on a single thread only*. In a parallel code therefore, the application programmer assumes the responsibility of communicating any changes resulting from steering activity to the other threads/processes.

We assume that the logical structure within the simulation is such that there exists a point (*breakpoint*) within a larger control loop at which it is feasible to insert new functionality intended to

(a) accept a change to one or more of the parameters of the simulation (*steerable parameters*);
(b) emit a consistent representation of the current state of both the steerable parameters and other variables (*monitored quantities*);
(c) emit a consistent representation (provisionally called *outsample)* of part of the system being simulated that may be required by a downstream component (eg. a visualisation system or another simulation).

We also assume that it is feasible, at the same point in the control loop, to

(d) output a consistent representation of the system (*checkpoint*) containing sufficient information to enable a subsequent *restart* of the simulation from its current state;
(e) (in the case that the steered component is itself downstream of another component), to accept a sample emitted by an upstream component (provisionally called *insample*).

As our immediate concern is with making simulations steerable, the language in this document reflects that pre-occupation. However, it is our expectation that the above will easily generalise to other components. An example is a visualisation component that consumes outsamples emitted by an upstream simulation component – it should be possible to steer the visualisation within the same framework.

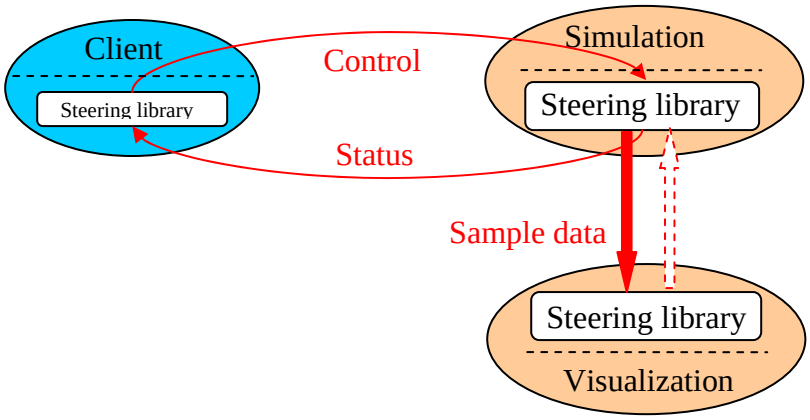

Figure 1: Computational steering architecture

Figure 1 shows an example use case for computational steering. The simulation component has been instrumented using the steering API. This enables it to receive control messages from the steering client and return status messages to it. In addition to interacting with the steering client, the simulation in Figure 1 also uses the API to transfer (potentially large) data sets to a second component. Although this second component is labelled "visualization," it could in fact be any other type of component including a second form of simulation.

2 An API for Computational Steering

2.1 General

The API has been designed with the assumption that the control and status links in Figure 1 will involve the exchange of small quantities of data. For cases where the steerable "parameter" is a large array of data, it will best be treated as a form of "sample data" with the steering client issuing instructions for it to be emitted and consumed as required.

All routines will provide integer return values with a return value of `REG_SUCCESS` (zero) indicating success and a return value of `REG_FAILURE` (unity) indicating failure. (These quantities are defined in `ReG_Steer_types.h`.) Some routines use this return value to specify more detailed information on the nature of the success/failure – see individual interface specifications for details. All return values utilised by one or more of the steering routines are listed in Appendix A.

All strings passed into the library must be of no more than `REG_MAX_STRING_LENGTH` (defined in `ReG_Steer_types.h`) characters in length.

2.2 Notation

The routine specifications given in the following sections are in C with the aim being to establish what information needs to be supplied to each routine and in what form. More information on each argument is supplied immediately below the routine specification. An argument labelled as "IN" is used as input by the routine and is not modified by it. An argument labelled as "OUT" is used only to return results to the

caller. An argument labelled "INOUT" is used as input by the routine and may have its value modified in order to return results to the user.

The API itself also provides F90 bindings for the simulation-side routines. These are given immediately after the C specification of each routine. All of the routines making up the F90 bindings are implemented as F90 *subroutines* and hence their return value is passed back to the caller by an additional 'status' argument (*c.f.* MPI).

Since F90 does not have the `void` type, where an argument to one of the F90 interfaces in this API can be of more than one type then its type is denoted by `<TYPE>`. Where an argument to an interface in the API is an array, the first element of the array should be supplied by the calling routine. The KIND parameters used here, `REG_SP_KIND` and `REG_DP_KIND`, are defined in `reg_steer_f90.inc` and correspond to KIND(1) and KIND(1.0D0), respectively.

2.3 Pre-defined steering commands

The API defines and uses four pre-defined commands: "stop", "pause", "resume" and "detach". An application that is steering enabled using the API supports "detach" by default. It is the responsibility of the application programmer to decide whether or not their application will support "stop" and "pause" and, if so, to implement this functionality. An application that supports "pause" is automatically assumed to support "resume." The following constants are defined in `ReG_Steer_types.h`:

Command	Encoding
REG_STR_STOP	1
REG_STR_PAUSE	2
REG_STR_RESUME	3
REG_STR_DETACH	4
REG_STR_EMIT_PARAM_LOG	5
REG_STR_PAUSE_INTERNAL	6

While `REG_STR_PAUSE` indicates that an application supports the pause command, it is up to the application programmer to implement this functionality. `REG_STR_PAUSE_INTERNAL` on the other hand indicates that the steering library will handle any pause command that is received internally (*i.e.* the call to `Steering_control` will block until a resume or stop command is received).

2.4 Application-side routines

The part of the API described in this section is intended for instrumenting a simulation code for computational steering.

2.4.1 Utility routines

```
char** Alloc_string_array( int  String_len, int Array_len)

IN String_len    Length of strings, int
IN Array_len     Length of array of strings, int

RETURNS          Pointer to array of strings or NULL
```

Allocates memory for an array of `Array_len` strings, each of `String_len` characters in length. Intended to aid in the creation of the arrays needed by *e.g.* `Steering_control` (see later). The memory allocated by a call to this routine is managed and ultimately free'd by the steering library (during the call to `Steering_finalize`).

2.4.2 Startup and shutdown

```
void Steering_enable(const int EnableSteer)

IN EnableSteer    Enable (1) or disable (0) steering, int
```

```
SUBROUTINE steering_enable_f(EnableSteer)

INTEGER (KIND=REG_SP_KIND) :: EnableSteer
```

Enables/disables the whole steering library. The default value of the `EnableSteer` flag is `REG_FALSE` and therefore this routine *must* be called with the argument set to `REG_TRUE` if an application is to be steerable.

```
int Steering_initialize(char *AppName,
                        int   NumSupportedCmds,
                        int  *SupportedCmds)

IN AppName              Name & version of the user application, char*
IN NumSupportedCmds     No. of steering commands supported, int
IN SupportedCmds        Array of supported commands, array of ints

RETURNS                 REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE steering_initialize_f(AppName, NumSupportedCmds, &
                                 SupportedCmds, Status)

CHARACTER                   :: AppName
INTEGER (KIND=REG_SP_KIND) :: NumSupportedCmds
INTEGER (KIND=REG_SP_KIND) :: SupportedCmds
INTEGER (KIND=REG_SP_KIND) :: Status
```

Initialize the steering library and advertise the application as being steerable. `AppName` is a string (of less than `REG_MAX_STRING_LENGTH` characters) containing the name and version number of the application that is calling the library. This information is included in the log of checkpoints created and used to check consistency when restarts are requested. The `SupportedCmds` array specifies which of the pre-defined steering commands the application supports (*e.g.* `REG_STR_STOP`).

```
int Steering_finalize()

RETURNS     REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE steering_finalize_f(status)

INTEGER (KIND=REG_SP_KIND) :: Status
```

Close any existing connection to a steering client and clear the internal library tables of registered parameters and sample-data/checkpoint types. The steering library reverts to the state it had prior to the original call to `Steering_initialize`. The application cannot be steered following this call (unless a further call to `Steering_initialize` is made).

2.4.3 Parameters

```
int Register_params( int    NumParams,
                     char* *ParamLabels,
                     int   *ParamSteerable,
                     void **ParamPtrs,
                     int   *ParamTypes,
                     char* *ParamMinima,
                     char* *ParamMaxima)

IN NumParams        Number of params to register, int
IN ParamLabels      Label for each parameter, array of char*
IN ParamSteerable   1 if parameter steerable, array of int
IN ParamPtrs        Pointer to parameter value, array of void*
IN ParamTypes       Data type of parameter, array of integers
IN ParamMinima      Min. parameter value, array of char*
IN ParamMaxima      Max. parameter value, array of char*

RETURNS             REG_SUCCESS, REG_FAILURE
```

Register the provided list of parameters with the library – this allows a steering client to monitor or, if they are flagged as steerable, to modify them. The labels must be

unique as they are used to identify the parameters and must not match the following reserved strings:

- "CPU_TIME_PER_STEP";
- "SEQUENCE_NUM";
- "STEERING_INTERVAL".

The first two of these identify monitored parameters automatically generated by the steering library. The first exposes the amount of CPU time (in seconds) being used between calls of `Steering_control` (*e.g.* per simulation time step) and the second, the value of the "sequence number" supplied in the call to `Steering_control`. The third reserved label identifies a steerable parameter that allows the frequency with which `Steering_control` actually contacts the steering client to be adjusted. (*e.g.* A value of ten corresponds to the steering client being contacted once in every ten calls.)

The `ParamSteerable` array identifies whether each parameter is steerable (1) or not (0). Since the quantities being registered with the steering library are pointers, it is a requirement that the variables to which they refer remain in scope through multiple calls to `Steering_control` (*i.e.* until `Steering_finalize` is called).

This routine may be called more than once and at any time (whether or not `Steering_initialize` has successfully initiated a connection with a steering client). The steering library's internal tables of parameters remains unchanged by the (dis)connection of a steering client.

Multiple calls to this routine have a cumulative effect. That is, every successful call to this routine results in the *addition* of the specified parameters to the table of registered parameters within the library.

The pointers in `ParamPtrs` may be to variables of different types (since this routine is only available in C and C++). The type of each parameter is specified by the `ParamTypes` array using the coding scheme given in Appendix C. (This coding scheme is defined within the `ReG_Steer_types.h` header file, *c.f.* MPI.)

The user may specify a range of validity for steerable parameters consisting of one or both of a maximum and minimum value. These values are specified by the strings pointed to by the elements of the `ParamMinima` and `ParamMaxima` arrays. If any limit (minimum, maximum or both) is inapplicable, the user should supply a blank or empty string (" " or ""). If the parameter being registered is a (steerable) string then the corresponding `ParamMaxima` gives the maximum length that the string can take. This may then be enforced a steering client. The upper bound on the length of a string registered with the library is `REG_MAX_STRING_LENGTH` characters.

Although this routine is flexible, in practice it requires that a user set up arrays for the various quantities being passed to it. It is also not possible to provide a binding for this routine in F90. As a consequence of these issues, the API also provides a singular form of the parameter registration routine. This is described below.

```
int Register_param(  char* ParamLabel,
                     int   ParamSteerable,
                     void *ParamPtr,
```

```
                     int   ParamType,
                     char* ParamMin,
                     char* ParamMax)

IN ParamLabel       Label of parameter, char*
IN ParamSteerable   1 if parameter steerable, array of int
IN ParamPtr         Pointer to parameter value, array of void*
IN ParamType        Data type of parameter, array of integers
IN ParamMin         Min. parameter value, char*
IN ParamMax         Max. parameter value, char*

RETURNS             REG_SUCCESS, REG_FAILURE

SUBROUTINE register_param_f( ParamLabel, ParamSteerable, &
                             ParamPtr, ParamType, ParamMin, &
                             ParamMax, Status)
```

```
CHARACTER(LEN=*)             :: ParamLabel
INTEGER (KIND=REG_SP_KIND)   :: ParamSteerable
<TYPE>                       :: ParamPtr
INTEGER (KIND=REG_SP_KIND)   :: ParamType
CHARACTER(LEN=*)             :: ParamMin
CHARACTER(LEN=*)             :: ParamMax
INTEGER (KIND=REG_SP_KIND)   :: Status
```

These two routines provide the singular form of Register_params for both C/C++ and F90. The behaviour of the singular form is identical to that of the plural form described above with the obvious exception that only one parameter may be registered per call.

```
SUBROUTINE register_string_param_f(ParamLabel, ParamSteerable, &
                                   StringParam, Status)

IN  ParamLabel      Label of param. to register, CHARACTER
IN  ParamSteerable  Whether param is steerable,
                    INTEGER (KIND=REG_SP_KIND)
IN  StringParam     The FORTRAN variable to register, CHARACTER
OUT Status          Routine's return status, INTEGER
                    (KIND=REG_SP_KIND)
```

This routine is specific to the F90-binding of the API and is as for `register_param_f` but only for CHARACTER parameters. The maximum length of the string is automatically obtained and stored along with the registered parameter.

```
int Unregister_param(char *ParamLabel)

IN ParamLabel     Label of param to unregister, char*

RETURNS           REG_SUCCESS, REG_FAILURE
```

Not currently implemented. Removes the steered/monitored parameter identified by the label in `SteerParamLabel` from the steering library such that its value will no longer be reported to the steering client or adjusted from within the library.

```
int Enable_param_logging(char *ParamLabel)

IN ParamLabel     Label of param for which to enable logging, char*

RETURNS           REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE enable_param_logging_f(ParamLabel, Status)

CHARACTER(LEN=*)          :: ParamLabel
INTEGER(KIND=REG_SP_KIND) :: Status
```

Enable logging of the specified parameter (which must have been previously registered with the steering library). When logging of a parameter is enabled, all values of that parameter (as captured when `Steering_control` is called) are stored and can be accessed by a steering client. These values are cached to disk and are stored in `$REG_STEER_DIRECTORY/ReG_params_log.dat`. This file is moved to one side and stored as `ReG_params_log.dat.backup` by any subsequent steering-enabled program that is using the same `$REG_STEER_DIRECTORY`.

N.B. logging is *on* by default.

```
int Disable_param_logging(char *ParamLabel)

IN ParamLabel     Label of param for which to disable logging, char*

RETURNS           REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE disable_param_logging_f(ParamLabel, Status)

CHARACTER(LEN=*)          :: ParamLabel
INTEGER(KIND=REG_SP_KIND) :: Status
```

Disables logging of the specified parameter. (Logging is on by default for all parameter types except `REG_BIN`.)

```
int Enable_all_param_logging(int toggle)

IN  toggle        Whether or not to enable logging, int

RETURNS           REG_SUCCESS
```

```
SUBROUTINE enable_all_param_logging_f(toggle, Status)

IN  toggle        Whether or not to enable logging, int

RETURNS           REG_SUCCESS
```

Toggle whether (`toggle=REG_TRUE`) or not (`toggle=REG_FALSE`) to log values of all registered parameters. Logging is on by default.

2.4.4 IO control

The routines described in this section are intended to allow an application to emit and consume potentially large quantities of data – the "Sample data" referred to in Figure 1. An application may emit or consume various sorts of data set and we term each of these an "IOType." The library is designed to cope with parallel applications that are unable to collect the complete data set to be emitted on a single processor. Such a data set must therefore be emitted piece by piece – this corresponds to several calls to `Emit_data_slice` in the steering library. Multiple calls to this routine must also be made if emitting data of different types.

```
int Register_IOTypes(int    NumTypes,
                     char* *IOLabel,
                     int   *type,
                     int   *IOFrequency,
                     int   *IOType)

IN  NumTypes        No. of IOTypes to register, integer
IN  IOLabel         Unique label for each IOType, array of char*
IN  type            IN or OUT, array of int
IN  IOFrequency     Initial value of frequency of emission/
                    consumption, array of int
OUT IOType          Identifier of registered IO type, array of int

RETURNS             REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE register_iotypes_f(NumTypes, IOLabel, IODirn,
                              IOFrequency, IOType, Status)

INTEGER (KIND=REG_SP_KIND)              :: NumTypes
CHARACTER(LEN=*), DIMENSION()           :: IOLabel
INTEGER (KIND=REG_SP_KIND), DIMENSION() :: IODirn
INTEGER (KIND=REG_SP_KIND), DIMENSION() :: IOFrequency
INTEGER (KIND=REG_SP_KIND), DIMENSION() :: IOType
INTEGER (KIND=REG_SP_KIND)              :: Status
```

In order to enable the generic steering client to control the emission and consumption of different types of sample data, the steered application must register the different sample-data types that it supports. `IOType` is then used as the identifier of this IOType. Frequencies are interpreted as the number of steps (calls of `Steering_control`) between IO activity. A frequency of zero for a given type thus disables the automatic consumption/emission of data for that IO type.

This routine is cumulative in its operation in that the IOTypes passed to it are appended to the internal list of any IOTypes registered in previous calls.

It is assumed that the mapping from `IOLabel` to a source/sink is performed by the steering library using some Grid framework. However, in the RealityGrid implementation, if file-based IOTypes are being used then the label is taken as the basis for the filenames to be produced: any spaces in the label are replaced by '_' characters and '_<sequence no.>' is appended (where <sequence no.> is the `SeqNum` argument supplied in the call to `Emit_start`, see below).

As with parameter registration, in practise it is sometimes more convenient to use a routine that simply registers one IOType at a time. The API also includes these:

```
int Register_IOType(  char *IOLabel,
                      int   type,
                      int   IOFrequency,
                      int  *IOType)

IN  IOLabel         Unique label for the IOType, char*
IN  type            IN or OUT, int
IN  IOFrequency     Initial value of frequency of emission/
                    consumption, int
OUT IOType          Identifier of registered IO type, int

RETURNS             REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE register_iotype_f( IOLabel, IODirn, IOFrequency,
                              IOType, Status)

CHARACTER(LEN=*)                          :: IOLabel
INTEGER (KIND=REG_SP_KIND)                :: IODirn
INTEGER (KIND=REG_SP_KIND)                :: IOFrequency
INTEGER (KIND=REG_SP_KIND)                :: IOType
INTEGER (KIND=REG_SP_KIND)                :: Status
```

These routines behave in exactly the same way as the plural forms but do not require arrays of variables as arguments.

```
int Enable_IOTypes_on_registration(int toggle)

IN  toggle    Whether to enable IOTypes when they are registered,
              TRUE/FALSE (default is TRUE)

RETURNS       REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE enable_iotypes_on_registrn_f(Toggle, Status)

INTEGER (KIND=REG_SP_KIND) :: Toggle
INTEGER (KIND=REG_SP_KIND) :: Status
```

Toggle whether or not to enable IOTypes during calls to the family of `Register_IOType` routines (the default behaviour is for them to be enabled). Intended for use with sockets-based IOTypes in which case "enabling" corresponds to the creation of the socket.

```
int Disable_IOType(int IOType)

IN  IOHandle  The IOType (as returned by Register_IOTypes) to
              disable

RETURNS       REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE disable_iotype_f(IOType, Status)

INTEGER (KIND=REG_SP_KIND) :: IOType
INTEGER (KIND=REG_SP_KIND) :: Status
```

Disable a previously registered (and enabled) IOType. If the steering library is built to use sockets-based communications for IOTypes then this corresponds to shutting-down the socket associated with the IOType.

```
int Enable_IOType(int IOType)

IN  IOType    The IOType (as returned by Register_IOTypes) to
              enable

RETURNS       REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE enable_iotype_f(IOType, Status)

INTEGER (KIND=REG_SP_KIND) :: IOType
INTEGER (KIND=REG_SP_KIND) :: Status
```

Enables a previously disabled IOType (such as one produced by `Register_IOTypes` after an earlier call of `Enable_IOTypes_on_registration(toggle)` with `toggle=FALSE`). For a socket-based IOType this corresponds to creation of the socket. Details of the IOType's endpoints are (re-)obtained from the underlying Grid framework during this call to allow for component creation and destruction.

```
int Disable_IOType_acks(int IOType)

IN  IOType    The IOType (as returned by Register_IOTypes) for which
              to turn off the use of acknowledgements.

RETURNS       REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE disable_iotype_acks_f(IOType, Status)

INTEGER (KIND=REG_SP_KIND) :: IOType
INTEGER (KIND=REG_SP_KIND) :: Status
```

Switches off the use of acknowledgements for the specified IOType (acknowledgements are on by default). When acknowledgements are disabled, the library will attempt to emit a data set, irrespective of whether or not the consumer has acknowledged the previous one. Note that when socket-based IO is used this can result in the emitter blocking if the consumer is unable to keep up (*e.g.* if the

consumer's processing of a data set is taking longer than the interval between the generation of data sets by the emitter).

```
int Enable_IOType_acks(int IOType)

IN  IOType    The IOType (as returned by Register_IOTypes) for which
              to turn on the use of acknowledgements.

RETURNS       REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE enable_iotype_acks_f(IOType, Status)

INTEGER (KIND=REG_SP_KIND) :: IOType
INTEGER (KIND=REG_SP_KIND) :: Status
```

Switches on the use of acknowledgements for the specified IOType. When acknowledgements are enabled, calls to `Emit_start` will return `REG_NOT_READY` until an acknowledgement of the last data set emitted has been received from the consumer.

2.4.5 Data output

```
int Emit_start(  int  IOType,
                 int  SeqNum,
                 int *IOHandle)

IN  IOType    Which registered IOType to emit, integer
IN  SeqNum    Indication of application's progress, integer
OUT IOHandle  Handle of opened IO channel, integer

RETURN        REG_SUCCESS, REG_FAILURE, REG_NOT_READY
```

```
SUBROUTINE emit_start_f(IOType, SeqNum, IOHandle, Status)

INTEGER (KIND=REG_SP_KIND) :: IOType
INTEGER (KIND=REG_SP_KIND) :: SeqNum
INTEGER (KIND=REG_SP_KIND) :: IOHandle
INTEGER (KIND=REG_SP_KIND) :: Status
```

A call to "open" an IOType ready to emit a data set. The channel to open is identified by `IOType`, which must have been obtained from a prior call to `Register_IOTypes` and be of type OUT. It is assumed that `Emit_start` can determine the destination associated with this particular type of sample data, *e.g.* by interrogating the Grid framework. If a call to this routine returns `REG_FAILURE` then this signifies that the IO channel is not connected and therefore no data can be emitted. This will typically occur when sockets-based IO is being used and the component intended to receive this data has yet to establish a connection. A return value of `REG_NOT_READY` indicates that no acknowledgement of the last data set that was (successfully) emitted has been received from the consumer of this `IOType`. (The use of acknowledgements for a given `IOType` may be disabled by calling `Disable_IOType_acks(IOType)`.)

```
int Emit_start_blocking( int  IOType,
                         int  SeqNum,
                         int *IOHandle,
                         float TimeOut)

IN  IOType    Which registered IOType to emit, integer
IN  SeqNum    Indication of application's progress, integer
OUT IOHandle  Handle of opened IO channel, integer
IN  TimeOut   Time out of call in seconds, float

RETURN        REG_SUCCESS, REG_TIMED_OUT
```

```
SUBROUTINE emit_start_blocking_f(IOType, SeqNum, IOHandle, Status)

INTEGER (KIND=REG_SP_KIND) :: IOType
INTEGER (KIND=REG_SP_KIND) :: SeqNum
INTEGER (KIND=REG_SP_KIND) :: IOHandle
INTEGER (KIND=REG_SP_KIND) :: Status
```

Blocking version of `Emit_start`. Blocks until `IOType` is ready to send data OR the specified `TimeOut` (seconds) is exceeded.

```
int Emit_data_slice (int   IOHandle,
                     int   DataType,
                     int   Count,
                     void *pData)

IN IOHandle   IO channel to use for emission, integer
IN DataType   Type of data to emit (int, float etc.), integer
IN Count      No. of elements in array to emit, integer
IN pData      Ptr. to array to emit, array of void*

RETURNS       REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE emit_data_slice_f(IOHandle, DataType, Count, &
                             pData, Status)

INTEGER (KIND=REG_SP_KIND)  :: IOHandle
INTEGER (KIND=REG_SP_KIND)  :: DataType
INTEGER (KIND=REG_SP_KIND)  :: Count
<TYPE>, DIMENSION()         :: pData
INTEGER(KIND=REG_SP_KIND)   :: Status
```

Having called `Emit_start`, an application should emit the various pieces of a data set by successive calls to `Emit_data_slice`. This routine wraps the low-level IO necessary for emitting the sample data. It will BLOCK until either the requested amount of data has been emitted or an error occurs. Note that the FORTRAN version of the routine cannot accept CHARACTER data – use `emit_char_data_slice_f` for this.

The application programmer is responsible for collecting sample data (in portions if necessary) and passing it to this routine. They are also responsible for reconstructing/identifying it in the downstream component. An example of the proposed usage of this routine is given in Appendix B.

`IOHandle` is the handle returned by a prior call to `Emit_start` and `DataType` should be specified using the coding scheme given in Appendix C.

```
SUBROUTINE emit_char_data_slice_f(IOHandle, Count, pData, Status)

INTEGER (KIND=REG_SP_KIND)  :: IOHandle
INTEGER (KIND=REG_SP_KIND)  :: Count
CHARACTER, DIMENSION()      :: pData
INTEGER(KIND=REG_SP_KIND)   :: Status
```

As for `emit_data_slice_f` but for data of type CHARACTER only.

```
int Emit_stop(int *IOHandle)

INOUT IOHandle   Handle of IO channel to close, integer

RETURNS          REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE emit_stop_f(IOHandle, Status)

INTEGER (KIND=REG_SP_KIND) :: IOHandle
INTEGER (KIND=REG_SP_KIND) :: Status
```

Signal the end of the emission of the sample/data set referred to by `IOHandle`. The value held by `IOHandle` is not a valid handle once this call has completed. This signals the receiving end that the transmission is complete – *i.e.* that they have all of the 'slices' constituting the data set.

2.4.6 Data input

```
int Consume_start(  int  IOType,
                    int *IOHandle)

IN  IOType    Which registered IOType to consume, integer
OUT IOHandle  Handle of opened IO channel, REG_IOHandleType

RETURN        REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE consume_start_f(IOType, IOHandle, Status)

INTEGER (KIND=REG_SP_KIND) :: IOType
INTEGER (KIND=REG_SP_KIND) :: IOHandle
INTEGER (KIND=REG_SP_KIND) :: Status
```

The equivalent of `Emit_start` for input - a call to "open" an IO channel ready to receive a data set. The channel to open is identified by `IOType`, which must have been obtained from a prior call to `Register_IO_types` and be of type IN. It is assumed that `Consume_start` can determine the source associated with this particular type of sample data, *e.g.* by interrogating the Grid framework. If this call returns `REG_FAILURE` then there is currently no data available on the channel (which might be because the channel is not connected to a data source). Note that in calling this routine, the application signals the data source that it is ready for new data – *i.e.* if `Consume_stop` has been called successfully for a previous data set then a call to `Consume_start` results in an acknowledgement of that data set being sent to the emitter.

```
int Consume_start_blocking (int      IOType,
                            int     *IOTypeIndex,
                            float    TimeOut)

IN  IOType        Which registered IOType to consume, integer
OUT IOHandle      Handle of opened IO channel, REG_IOHandleType
IN  TimeOut       Time out of call in seconds, float

RETURN            REG_SUCCESS, REG_TIMED_OUT
```

```
SUBROUTINE consume_start_blocking_f(IOType, IOHandle, TimeOut, &
                                    Status)

INTEGER (KIND=REG_SP_KIND) :: IOType
INTEGER (KIND=REG_SP_KIND) :: IOHandle
REAL    (KIND=REG_SP_KIND) :: TimeOut
INTEGER (KIND=REG_SP_KIND) :: Status
```

Blocking version of the previous routine. Blocks until data is available to read OR `TimeOut` (seconds) is exceeded.

```
int Consume_data_slice_header(int   IOHandle,
                              int  *DataType
                              int  *Count)

IN  IOHandle      IO channel to use for consumption, integer
OUT  DataType     Data type of data to consume, integer
OUT  Count        No. of data items to consume, integer

RETURNS           REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE consume_data_slice_header_f(IOHandle, DataType, &
                                       Count, Status)

INTEGER (KIND=REG_SP_KIND) :: IOHandle
INTEGER (KIND=REG_SP_KIND) :: DataType
INTEGER (KIND=REG_SP_KIND) :: Count
INTEGER (KIND=REG_SP_KIND) :: Status
```

Must only be called once a successful call to `Consume_start` has indicated that data is available for consumption (`IOHandle` is the handle returned from that call and identifies the IO channel being read). This routine blocks until it receives a header describing the next data slice to be read or end-of-message. If this header is received successfully then `REG_SUCCESS` is returned and `DataType` and `Count` hold the type and number, respectively, of data elements forming the next slice. If end-of-message is received then this routine returns `REG_FAILURE`. Note that `Count` is the number of elements (*e.g.* integers) to be consumed and NOT the number of bytes. The coding scheme obeyed by `DataType` is given in Appendix C.

```
int Consume_data_slice( int      IOHandle,
                        int      DataType,
                        int      Count,
                        void   **pData)

IN  IOHandle  IO channel to use for consumption, integer
IN  DataType  Data type of data being consumed, integer
IN  Count     No. of data items to consume, integer
OUT pData     Ptr. to array containing consumed data, array void*

RETURNS       REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE consume_data_slice_f(IOHandle, pData, Status)

INTEGER (KIND=REG_SP_KIND)  :: IOHandle
INTEGER (KIND=REG_SP_KIND)  :: DataType
INTEGER (KIND=REG_SP_KIND)  :: Count
<TYPE>, DIMENSION()         :: pData
INTEGER (KIND=REG_SP_KIND)  :: Status
```

Wraps the low-level IO for receiving sample data from another component. `IOHandle` is obtained from a call to `Consume_start` and thus identifies the IO channel being used. This routine BLOCKS until the requested amount of data has been consumed. The type of data to consume is specified using the coding scheme given in Appendix C. The amount and type of the data to consume is obtained from a prior call to `Consume_data_slice_header`.

```
int Consume_stop(int *IOHandle)

INOUT IOHandle   Handle of IO channel to close, integer

RETURNS          REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE consume_stop_f(IOHandle, Status)

INTEGER (KIND=REG_SP_KIND) :: IOHandle
INTEGER (KIND=REG_SP_KIND) :: Status
```

Should be called after a call to `Consume_data_slice_header` has returned `REG_FAILURE` indicating that there are no more ‘slices’ to read for the current data set. It signals the end of the consumption of the sample/data set referred to by `IOHandle`. The value held by `IOHandle` is not a valid handle once this call has completed. This routine should be called by the same thread that made the corresponding call to `Consume_start`.

2.4.7 Checkpoint control

```
int Register_ChkTypes( int    NumTypes,
                       char* *ChkLabel,
                       int   *direction,
                       int   *ChkFrequency,
                       int   *ChkType);

IN  NumTypes       No. of Checkpoint types to register, integer
IN  ChkLabel       Unique label for each Chk type, array of char*
IN  direction      IN, OUT or INOUT, array of int
IN  ChkFrequency   Initial value of frequency of checkpoint
                   creation (ignored for ChkTypes with direction
                   IN), array of int
OUT ChkType        Identifier of registered Chk type, array of int
```

```
SUBROUTINE register_chktypes_f(NumTypes, ChkLabel, ChkDirn,
                               Frequency, ChkType, Status)

INTEGER (KIND=REG_SP_KIND)                :: NumTypes
CHARACTER(LEN=*), DIMENSION()             :: ChkLabel
INTEGER (KIND=REG_SP_KIND), DIMENSION()   :: ChkDirn
INTEGER (KIND=REG_SP_KIND), DIMENSION()   :: Frequency
INTEGER (KIND=REG_SP_KIND), DIMENSION()   :: ChkType
INTEGER (KIND=REG_SP_KIND)                :: Status
```

Register the listed Checkpoint types for control and monitoring purposes - no reference is made to the actual location of the checkpoint data. The `direction` parameter specifies whether the application can create (OUT), restart from (IN) or both create and restart from (INOUT) the given checkpoint type. `ChkFrequency` is interpreted in the same way as for `Register_IOTypes` and is only applied to the creation of checkpoints (*i.e.* those with a direction of OUT or INOUT).
This routine is cumulative in its operation in that the `ChkTypes` passed to it are appended to the internal list of any `ChkTypes` registered in previous calls.

```
int Register_ChkType(  char *ChkLabel,
                       int   direction,
                       int   ChkFrequency,
                       int  *ChkType);

IN  ChkLabel       Unique label for the Chk type, char*
IN  direction      IN, OUT or INOUT, int
IN  ChkFrequency   Initial value of frequency of checkpoint
                   creation (ignored for ChkTypes with direction
                   IN), int
OUT ChkType        Identifier of registered Chk type, int
```

```
SUBROUTINE register_chktype_f(ChkLabel, ChkDirn,
                              Frequency, ChkType, Status)

CHARACTER(LEN=*)                          :: ChkLabel
INTEGER (KIND=REG_SP_KIND)                :: ChkDirn
INTEGER (KIND=REG_SP_KIND)                :: Frequency
INTEGER (KIND=REG_SP_KIND)                :: ChkType
INTEGER (KIND=REG_SP_KIND)                :: Status
```

These routines are identical in operation to the plural forms described above with the obvious exception that they only register one ChkType at a time and therefore do not require arrays of variables as arguments.

```
int Add_checkpoint_file(  int   ChkType,
                          char *filename)

IN ChkType    Which type of checkpoint the file belongs to, integer
IN filename   The name of a file making up the checkpoint, char*

RETURNS       REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE add_checkpoint_file_f(ChkType, filename, Status)

INTEGER (KIND=REG_SP_KIND)  :: ChkType
CHARACTER(LEN=*)            :: filename
INTEGER(KIND=REG_SP_KIND)   :: Status
```

Intended to tell the steering library about each file making up a checkpoint of type `ChkType` (the handle created by a call to `Register_ChkTypes`). The `filename` argument should give the name of one file to be added to the list of files making up the current checkpoint, *it should not contain any path information*. Each call to this routine adds the named file to an internal list of files for the current checkpoint. Once this list is complete, the application should call `Record_checkpoint_set` (where a path to the files may be specified if required). Calling `Record_checkpoint_set` resets the internal list of files for the specified `ChkType`, ready for the creation of the next checkpoint.

```
int Record_checkpoint_set( int   ChkType,
                           char *ChkTag,
                           char *Path)

IN ChkType    Which type of checkpoint we've just created, integer
IN ChkTag     Unique tag identifying this checkpoint, char*
IN Path       Path, relative to current working directory, to
              checkpoint files, char*

RETURNS       REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE record_checkpoint_set_f(ChkType, filename, Status)

INTEGER (KIND=REG_SP_KIND)  :: ChkType
CHARACTER(LEN=*)            :: filename
INTEGER(KIND=REG_SP_KIND)   :: Status
```

Record that the application has just created/written a checkpoint of type `ChkType` - the handle created by a call to `Register_ChkTypes`. The calling application must supply a string in `ChkTag` which uniquely identifies the files making up the checkpoint (*i.e.* the filenames all contain this unique string) and will allow it (the calling application) to retrieve the data associated with this checkpoint. If the same tag is supplied in multiple calls then the checkpoint is assumed to have been overwritten.

If no calls to `Add_checkpoint_file` have been since the last call of this routine then it is assumed that the combination of information supplied in the `Path` and `ChkTag` arguments is sufficient for the files making up the checkpoint set to be identified. (*i.e.* executing "`ls ./<Path>/*<ChkTag>*`" will return a list of all the files constituting the checkpoint.)

If `Add_checkpoint_file` has been called (one or more times) then the resulting list of files is taken to define the checkpoint being recorded. `Record_checkpoint_set` clears the list of filenames built up by the calls to `Add_checkpoint_file`. Therefore, before registering the next checkpoint, the application must make another series of calls to `Add_checkpoint_file`.

When requesting that the simulation perform a restart, `Steering_control` returns the `ChkType` from which to restart and a `ChkTag` identifying which checkpoint set. Currently, no information is supplied on where to find the checkpoint or on how the `ChkTag` should be used to construct the necessary filenames; this is all assumed to be carried out by the simulation. This situation is not satisfactory but is difficult to resolve since it impinges on the more general area of checkpoint management. It will be addressed in the next release of this API.

2.4.8 Communication with steering client

```
int Steering_control(int      SeqNum,
                     int     *NumSteerParams,
                     char*   *SteerParamLabels,
                     int     *NumSteerCommands,
                     int     *SteerCommands,
                     char*   *SteerCmdParams)

IN  SeqNum              Indicator of application's progress, int
OUT NumSteerParams      No. of modified steering params, int
OUT SteerParamLabels    Labels of modified params, array of char*
OUT NumSteerCommands    No. of commands returned to application, int
OUT SteerCommands       List of steering commands, array of int
OUT SteerCmdParams      Params associated with each cmd (if any),
                        array of char*

RETURNS                 REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE steering_control_f(SeqNum, NumSteerParams, &
                              SteerParamLabels, NumSteerCommands, &
                              SteerCommands, SteerCommandParams, &
                              Status)

INTEGER (KIND=REG_SP_KIND)     :: SeqNum
INTEGER (KIND=REG_SP_KIND)     :: NumSteerParams
CHARACTER(LEN=*), DIMENSION()  :: SteerParamLabels
INTEGER (KIND=REG_SP_KIND)     :: NumSteerCommands
INTEGER (KIND=REG_SP_KIND)     :: SteerCommands
CHARACTER(LEN=*), DIMENSION()  :: SteerCommandParams
INTEGER (KIND=REG_SP_KIND)     :: Status
```

This routine should be called at the breakpoint referred to in the background assumptions described earlier.

It performs the following actions:

- checks for and enables a steering client to connect if the application is not already being steered;
- retrieves any (new) values for the steerable parameters from an attached steering client and updates the associated simulation variables via the pointers supplied at their registration;
- retrieves any commands from an attached steering client – those that must be handled by the application itself are returned as a list (encoded as integers) in `SteerCommands`. It is the application's responsibility to deal with these commands in the order in which they occur in the list;
- reports the current values of all registered parameters (both monitored and steered) to the steering client.

This routine (internally) handles attach and detach requests from the steering client. Since there may be performance implications associated with checking whether a client is attempting to attach, the frequency with which this is done is configurable by means of the `REG_APP_POLL_INTERVAL` environment variable. This may be used to override the default minimum time interval (in seconds) between checks on

whether a steering client has connected. If it is set to zero then a check is performed on every call to `Steering_control`. If the environment variable is not set then a default value (set in `ReG_Steer_types.h`) is used.

`Steering_control` returns a list of labels identifying which of the steered parameters have been changed by the steering client. (The values themselves will have been updated using the pointers previously registered with the library.)

The commands returned by this routine may include pre-defined commands (such as `stop') as well as commands to emit/consume a sample (of type `IOType`) or create/restart from a checkpoint (of type `ChkType`). The latter are indicated by returning the value of an `IOType` or `Chktype` (as returned by a previous call to `Register_IOTypes` or `Register_ChkTypes`, respectively) in the `SteerCommands` array.

Certain commands may require the supply of additional information in order to enable the application to execute them. This information is supplied in the form of a character string for each command. The only command that makes use of this is the instruction to checkpoint – the associated parameter string then contains either "IN" plus the tag of the checkpoint (see `Record_checkpoint_set`) or "OUT" according to whether a checkpoint is to be created or a restart performed.

```
int Steering_pause( int   *NumSteerParams,
                    char **SteerParamLabels,
                    int   *NumCommands,
                    int   *SteerCommands,
                    char **SteerCmdParams);

OUT NumSteerParams     No. of parameters edited by steerer, integer
OUT SteerParamLabels   Labels of those params, array of char*
OUT NumCommands        No. of commands sent by steerer, integer
OUT SteerCommands      Commands sent by steerer, array of integer
OUT SteerCmdParams     Params associated with each cmd (if any),
                       array of char*

RETURNS                REG_SUCCESS, REG_FAILURE
```

```
SUBROUTINE steering_pause_f( NumSteerParams, SteerParamLabels, &
                             NumCommands, SteerCommands, &
                             SteerCommandParams, Status)

INTEGER (KIND=REG_SP_KIND)               :: NumSteerParams
CHARACTER(LEN=*), DIMENSION()            :: SteerParamLabels
INTEGER (KIND=REG_SP_KIND)               :: NumSteerCommands
INTEGER (KIND=REG_SP_KIND), DIMENSION()  :: SteerCommands
CHARACTER(LEN=*), DIMENSION()            :: SteerCommandParams
INTEGER (KIND=REG_SP_KIND)               :: Status
```

Intended to be called by the application in response to a "pause" command from the steerer. Blocks until a "resume" or "stop" command is received. Returns all commands that immediately followed a "resume" command as well as the labels of any parameters edited while the application has been paused. (Parameters can be

edited while the application is paused because this routine continually calls `Consume_control` until it receives “resume” or “stop”.)

The application programmer is free to provide their own version of this routine, should they need to take actions while the simulation is paused (as may be the case with an experiment). Any “pause” routine must periodically call `Steering_control` in order to check for a “resume” command.

2.5 Steering-client-side routines

These routines are intended to support the implementation of a generic steering client.

2.5.1 Startup and shutdown

```
int Steerer_initialize();

RETURNS     REG_SUCCESS, REG_FAILURE
```

Initialise the internal tables *etc*. used by the steering library on the steering client side. This must be called before all other steering library routines.

```
int Steerer_finalize();

RETURNS     REG_SUCCESS
```

Cleans up the internal tables *etc*. used by the client-side steering library. This should be called after all steering activity is complete.

2.5.2 Discovery

```
int Get_sim_list( int   *nsims,
                  char **simName,
                  char **simGSH)

OUT nsims         No. of steerable applications available, int
OUT simName       Name of each steerable application, array of char*
OUT simGSH        Grid Service Handle of each app., array of char*

RETURNS           REG_SUCCESS, REG_FAILURE
```

Queries available middleware for a list of steerable applications. Returns `REG_FAILURE` and `nsims`=0 if middleware unavailable. If the query is successful and a list obtained then the names and Grid Service Handles of the available applications are returned. The Grid Service Handle of the (user-) selected application must be passed as the `SimID` to `Sim_attach`.
`simName` and `simGSH` should point to arrays of strings, each of `REG_MAX_STRING_LENGTH` characters. This routine will return details for a maximum of `REG_MAX_NUM_STEERED_SIM` simulations.

2.5.3 Attach/detach

```
int Sim_attach(char  *SimID,
               int   *SimHandle)

IN  SimID         ID of sim. to attach to, char*
OUT SimHandle     Handle of simulation, integer

RETURNS           REG_SUCCESS, REG_FAILURE, REG_MEM_FAIL
```

Attach to the specified simulation. Returns `SimHandle` for use in future calls (since the steering client may be steering more than one component).

In the RealityGrid implementation, this routine first attempts to contact the SteeringGridService with the handle given by `SimID`. If this fails then it attempts to revert to steering via the local file system. In this case, the library first tries interpreting `SimID` as a directory on the local filesystem (*i.e*. it assumes that `SimID` gives the directory that was passed to the running simulation in the `REG_STEER_DIRECTORY` environment variable). If this too fails then the library attempts to get a valid steering directory from the `REG_STEER_DIRECTORY` environment variable. If all of these attempts fail then the routine returns `REG_FAILURE`. `SimID` should point to a string of no more than `REG_MAX_STRING_LENGTH` characters.

```
int Sim_detach(int *SimHandle)

INOUT SimHandle  Handle of simulation from which to detach, integer

RETURNS     REG_SUCCESS
```

Detaches from the specified simulation and cleans up associated table entries and communication infrastructure. `SimHandle` will not be valid following the call.

```
int Delete_sim_table_entry(int *SimHandle);

INOUT  SimHandle  Handle of sim. to delete, int

RETURNS           REG_SUCCESS, REG_FAILURE
```

Deletes all data associated with the simulation with handle `SimHandle`. Used when a simulation detaches. Supplied as a separate interface because also required when the simulation initiates the detach (*e.g.* when it has completed its run).

2.5.4 Communication with steered application

```
int Get_next_message(int   *SimHandle,
                     int   *msg_type);

OUT      SimHandle  Handle of sim. that emitted message, integer.
OUT      msg_type   Type of message received, integer.

RETURNS  REG_SUCCESS, REG_FAILURE
```

Looks for the next message from any attached simulations. If it finds one then it returns the handle of the originating simulation and the type of the message. If it fails to find a message then it returns `REG_FAILURE`.

```
int Consume_param_defs(int  SimHandle)

IN  SimHandle    Simul'n from which to recv param definitions, int

RETURNS          REG_SUCCESS, REG_FAILURE
```

Consume the parameter definitions emitted by the steered application. The internal table of parameters is updated ready for querying (*e.g.* in order to update a GUI).

```
int Consume_IOType_defs(int  SimHandle)

IN  SimHandle  Handle of sim. from which to consume IOType defs, int.

RETURNS        REG_SUCCESS, REG_FAILURE
```

Consume the IOType descriptions emitted by the steered application. These descriptions provide information to be displayed on the GUI in order to allow the user to request sample data to be emitted and consumed.

```
int Emit_control( int     SimHandle,
                  int     NumCommands,
                  int    *SysCommands,
                  char  **SysCmdParams)

IN SimHandle       Handle of sim'n to send control data to, int
IN NumCommands     No. of commands to send, integer
IN SysCommands     Commands to send to sim'n, array of int
IN SysCmdParams    Params associated with each cmd, array of char*

RETURNS            REG_SUCCESS, REG_FAILURE
```

Transfer the steering commands and associated parameters (if any) to the simulation. This routine also checks to see whether any of the steerable parameters have been edited since it was last called and, if so, sends the new values to the application. `SimHandle` identifies the simulation as it is conceivable that we could be steering more than one component. Note that `NumCommands` cannot exceed `REG_MAX_NUM_STR_CMDS` (defined in `ReG_Steer_types.h`).

```
int Emit_retrieve_param_log_cmd(int SimHandle, int ParamHandle)

IN SimHandle       Handle of sim'n to send command to, int
IN ParamHandle     Handle of parameter for which to retrieve log, int

RETURNS            REG_SUCCESS, REG_FAILURE
```

Emit a command to instruct the steered application to emit all of the logged values for the specified simulation. The log itself is stored internally and is accessed via `Get_param_log`.

```
int Emit_pause_cmd(int SimHandle)

IN SimHandle      Handle of sim'n to send command to, int

RETURNS           REG_SUCCESS, REG_FAILURE
```

Wrapper for generating pause command and sending it to the attached application.

```
int Emit_resume_cmd(int SimHandle)

IN SimHandle      Handle of sim'n to send command to, int

RETURNS           REG_SUCCESS, REG_FAILURE
```

Wrapper for generating resume command and sending it to the attached application.

```
int Emit_restart_cmd(  int SimHandle,
                       char *chkGSH)

IN SimHandle      Handle of sim'n to send command to, int
IN chkGSH         Grid Service Handle of checkpoint to restart from,
                  char*

RETURNS           REG_SUCCESS, REG_FAILURE
```

SOAP (OGSI)- specific wrapper for generating restart command and sending it to the attached application. Note that the steering library does not concern itself with the physical location of checkpoint files and therefore this must be handled externally if the specified checkpoint does not exist on the machine running the steered application.

```
int Consume_status(int     SimHandle,
                   int    *SeqNum,
                   int    *NumCommands,
                   int    *SysCommands)

IN  SimHandle     Sim'n from which to consume status, int
OUT SeqNum        Measure of progress of sim'n, integer
```

```
The following variables list any commands sent to the steerer from
the application (e.g. ‘detach’).

OUT NumCommands   No. of system commands recvd, integer
OUT SysCommands   List of system commands recvd, array of int

RETURNS           REG_SUCCESS, REG_FAILURE, REG_MEM_FAIL
```

Retrieve the status of the simulation being steered, along with any commands that the simulation has sent to the steerer (*e.g.* notification that it has finished).

Any parameter values received by this routine are automatically used to update the internal library table of parameters.

```
int Consume_log(int   SimHandle)

IN  SimHandle    Sim’n from which to consume log entries, int

RETURNS          REG_SUCCESS, REG_FAILURE
```

Consumes any logging messages (containing details on checkpoints taken) emitted by the simulation associated with `SimHandle`. The contents of the message are stored in the log for that simulation. This log may be accessed by the `Get_chk_log_*` functions described later in this document. This functionality is only used in local (file-based) steering – in remote steering using Grid Services the checkpoint logging is carried out using a Checkpoint Tree. See the ReG_Steering_Grid_Service.doc document for more details.

2.5.5 Query functions

```
int Get_param_number(int  sim_handle,
                     int  steerable,
                     int *num_params);

IN  sim_handle   Specify which simulation entry to query, integer
IN  steerable    Specify whether steerable (1) or monitored (0), int
OUT num_params   The no. of <steerable> params registered, int

RETURNS          REG_SUCCESS, REG_FAILURE
```

This routine reports the number of steerable or monitored parameters registered by the simulation with handle `sim_handle`. That is, if `steerable == TRUE`, then this returns the number of steerable parameters, otherwise it returns the number of monitored parameters.

```
int Get_param_values(int                   sim_handle,
                     int                   steerable,
                     int                   num_params,
                     Param_details_struct  *param_details );

IN  sim_handle    Sim. handle for which to retrieve param. vals, int
IN  steerable     Whether to get steerable (1) or monitored (0), int
```

```
IN  num_params    No. of param vals to get, integer
OUT param_details List of details params, array of struct

RETURNS           REG_SUCCESS, REG_FAILURE
```

If successful, this routine returns details of the first `num_params`, `steerable` (*i.e.* either steerable or monitored) parameters associated with the application with handle `sim_handle`. `param_details` must point to a chunk of memory large enough to receive `num_params` entries.

The `Param_details_struct` is defined as follows:

```
typedef struct {

  char  label[REG_MAX_STRING_LENGTH];
  int   type;
  int   handle;
  char  value[REG_MAX_STRING_LENGTH];
  char  min_val[REG_MAX_STRING_LENGTH];
  char  max_val[REG_MAX_STRING_LENGTH];

} Param_details_struct;
```

where `type` uses the coding scheme defined in Appendix C.

```
int Get_param_log (     int      sim_handle,
                   int      handle,
                   double **buf,
                   int     *num_entries)

IN  sim_handle     Handle of sim. table entry to query, int
IN  handle         Handle of parameter to get log for, int
OUT buf            Pointer to array holding log values, **double
OUT num_entries    Number of entries being returned, int
```

Retrieve pointer to internal buffer holding previous values of the parameter with handle 'handle' belonging to 'sim_handle.' num_entries holds the no. of entries in the log.

```
int Get_iotype_number( int  sim_handle,
                       int *num_iotypes)

IN  sim_handle    Handle of sim. table entry to query, int
OUT num_iotypes   No. of IO types registered, integer

RETURNS     REG_SUCCESS, REG_FAILURE
```

Gets the number of IOTypes registered by the simulation with handle `sim_handle`.

```
int Get_chktype_number( int sim_handle,
                        int *num_chktypes)
```

As for `Get_iotype_number`.

```
int Get_iotypes( int    sim_handle,
                 int    num_iotypes,
                 int   *handles,
                 char* *labels,
                 int   *types,
                 int   *io_freqs)

IN  sim_handle   Handle of sim. to get IO types for, int
IN  num_iotypes  No. of IO types to retrieve, int
OUT handles      List of handles of IO types, array of int
OUT labels       List of labels of IO types, array of char*
OUT types        List of 'types' of IO types, array of int
OUT io_freqs     Frequency associated with each IO type, array of int

RETURNS          REG_SUCCESS, REG_FAILURE
```

Gets the details of the first `num_iotypes` IO types associated with the simulation with handle `sim_handle`. Returns the handle, label, "type" (whether IN or OUT) and frequency associated with each IOType.

```
int Get_chktypes( int    sim_handle,
                  int    num_chktypes,
                  int   *handles,
                  char* *labels,
                  int   *types,
                  int   *chk_freqs)
```

As for `Get_iotypes` with the exception that a checkpoint can be of "type" IN, OUT or INOUT.

```
int Get_supp_cmd_number(int  sim_handle,
                        int *num_cmds)

IN  sim_handle     Handle of sim. of interest, int
OUT num_cmds       No. of cmds that sim supports, int

RETURNS            REG_SUCCESS, REG_FAILURE
```

Gets the number of supported commands registered by the simulation with handle `sim_handle`.

```
int Get_supp_cmds(int  sim_handle,)
                  int  num_cmds,
                  int *cmd_ids)

IN  sim_handle    Handle of sim. to query, int
IN  num_cmds      No. of cmds to return info on, int
OUT cmd_ids       List of cmds supported, array of int

RETURNS           REG_SUCCESS, REG_FAILURE
```

Gets the first `num_cmds` registered for the simulation with handle `sim_handle`. Returns the ID of each command in the array pointed to by `cmd_ids`. This array must be large enough to hold `num_cmds` integers. The commands returned by this routine are those that the application has undertaken to support in its call to `Steering_initialize` (plus those that are supported by default such as detach – see Section 2.3).

```
int Get_chk_log_number( int  sim_handle,
                        int  chk_handle,
                        int *num_entries);

IN  sim_handle    Handle of sim. to query, int
IN  chk_handle    Handle of checkpoint type, int
OUT num_entries   No. of entries in log for this checkpoint type, int

RETURNS           REG_SUCCESS, REG_FAILURE
```

Query function that returns the number of entries currently held in the checkpoint log for the specified checkpoint type.
Note that in the RealityGrid implementation this routine is only used for local (file-based) steering. When the grid-service framework is used, checkpoint logging is performed using a tree of grid services and querying this tree is outside the scope of this API.

```
int Get_chk_log_entries( int                sim_handle,
                         int                chk_handle,
                         int                num_entries,
                         Output_log_struct *entries)

IN  sim_handle    Handle of sim. to query, int
IN  chk_handle    Handle of checkpoint type, int
IN  num_entries   No. of entries to retrieve, int
OUT entries       Returned log entries, ptr to log struct

RETURNS           REG_SUCCESS, REG_FAILURE
```

Returns the FIRST `num_entries` checkpoint entries for checkpoints with handle `chk_handle` (as returned in the `handles` array by a call to `Get_chktypes`) from the logs.

The definition of the `Output_log_struct` is as follows:

```
typedef struct {

  /* Tag associated with this checkpoint */
  char   chk_tag[REG_MAX_STRING_LENGTH];
  /* No. of parameters for which we have details at this chkpt */
  int    num_param;
  /* Associated parameter labels and values at this chkpt */
  char   param_labels[REG_MAX_NUM_STR_PARAMS][REG_MAX_STRING_LENGTH];
  char   param_values[REG_MAX_NUM_STR_PARAMS][REG_MAX_STRING_LENGTH];

} Output_log_struct;
```

```
int Get_chk_log_entries_reverse( int                sim_handle,
                                 int                chk_handle,
                                 int                num_entries,
                                 Output_log_struct *entries)
```

As for `Get_chk_log_entries` but returns the LAST `num_entries` checkpoint entries for checkpoints with handle `chk_handle` in reverse chronological order (*i.e.* most recent first).

2.5.6 Editing routines

```
int Set_param_values(int    sim_handle,
                     int    num_params,
                     int   *handles,
                     char* *vals)

IN  sim_handle    Handle of sim. whose params to update, int
IN  num_params    No. of params to update, int
IN  handles       List of param handles, array of int
IN  vals          List of associated vals as strings, array of char*

RETURNS           REG_SUCCESS, REG_FAILURE
```

Sets the values of the parameters with the specified handles for the simulation with handle `sim_handle`. The new values are sent to the simulation on the next call of `Emit_control`.

```
int Set_iotype_freq( int  sim_handle,
                     int  num_iotypes,
                     int *iotype_handles,
                     int *freqs)

IN  sim_handle      Handle of sim. to set IOtype freqs for, int
IN  num_iotypes     No. of IO types to adjust, int
IN  iotype_handles  Handles of IOtypes to be adjusted, int array
IN  freqs           New frequency values, int array

RETURNS             REG_SUCCESS, REG_FAILURE
```

A utility function that allows a steering client to update the emit/consume frequency associated with a given IOType. The new values are sent to the simulation on the next call of `Emit_control`.

```
int Set_chktype_freq(  int  sim_handle,
                       int  num_chktypes,
                       int *chktype_handles,
                       int *freqs)
```

As for `Set_iotype_freq`.

2.5.7 Debugging routines

```
int Dump_sim_table();

RETURNS     REG_SUCCESS, REG_FAILURE
```

A debugging routine - writes the complete contents of the internal table holding information on all connected simulations to `./sim_table.txt`

4

5 Acknowledgements

This work was funded by the Engineering and Physical Sciences Research Council (http://www.epsrc.ac.uk) as part of the RealityGrid project (http://www.realitygrid.org). More documentation and the library itself are available from http://www.sve.man.ac.uk/Research/AtoZ/RealityGrid/.

Appendix A: Return values

These are defined in `ReG_Steer_types.h` and have the following values:

Return label	Value
REG_SUCCESS	0
REG_FAILURE	1
REG_EOD	2
REG_MEM_FAIL	3
REG_TIMED_OUT	4
REG_NOT_READY	5
REG_UNFINISHED	6

Appendix B: Example usage of Emit_data_slice

An example of the structure of a routine to emit sample data as might be written by the application scientist is given below. This routine is called by the application in response to instructions returned by `Steering_control`. It is assumed that the MPI process with rank 0 is the process that calls `Steering_initialize`, `Steering_control`, *etc.*

```
int Emit_sample(int IOType, int SeqNum
                <, + any other required arguments >){

  int irank;
  int count;
  int nprocs;
  int *data_ptr;
  REG_SampleHandleType SampleHandle;

  MPI_Comm_rank(MPI_COMM_WORLD, &irank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  if(irank == 0){

      /* IOType corresponds to a previously registered sample
         type */
      Emit_start(IOType, SeqNum, &SampleHandle);
  }

  for(iproc=0; iproc<nprocs; iproc++){

      MPI_Barrier(MPI_COMM_WORLD);

      if(irank != 0 && irank == iproc){

        /* Some code to get data_ptr and the quantity of data
           (count) goes here... */

        MPI_Send(&count, 1, MPI_INT, 0, Some_Tag, MPI_COMM_WORLD);
        MPI_Send(data_ptr, count, MPI_DOUBLE, 0, Some_Tag,
                 MPI_COMM_WORLD);
      }
      else if(irank == 0){

        /* Collect data on master process and then emit it... */

        if(iproc != 0){

          MPI_Recv(&count, 1, MPI_INT, iproc, MPI_ANY_TAG,
                   MPI_COMM_WORLD, &status);

          /* Allocate memory if required */

          MPI_Recv(pdata, count, MPI_DOUBLE, iproc, MPI_ANY_TAG,
                  MPI_COMM_WORLD, &status);
        }
        else{

            /* Some code to get pdata and quantity of data (count)
             on process 0 goes here */
```

```
        }

        Emit_data_slice(sampleHandle, REG_DBL, count,
                        pdata);
      }

  }


  if(irank == 0){
      Emit_stop(sampleHandle);
  }

  return 0;
}
```

Appendix C: Coding scheme for data types

Various routines in the steering API require or return information on the *type* (whether integer, float *etc*.) of data being handled. This type information is encoded as integer values according to the following table:

C data type	F90 data type	Constant name	Value in coding scheme
int	INTEGER(KIND=REG_SP_KIND)	REG_INT	0
float	REAL(KIND=REG_SP_KIND)	REG_FLOAT	1
double	REAL(KIND=REG_DP_KIND)	REG_DBL	2
char	CHARACTER	REG_CHAR	3

Interface index

Add_checkpoint_file 21
add_checkpoint_file_f 21
Alloc_string_array 6
Consume_data_slice 18
consume_data_slice_f 18
Consume_data_slice_header 18
consume_data_slice_header_f 18
Consume_IOType_defs 28
Consume_log 30
Consume_param_defs 28
Consume_start 17
Consume_start_blocking 17
consume_start_blocking_f 17
consume_start_f 17
Consume_status 30
Consume_stop 19
consume_stop_f 19
Delete_sim_table_entry 27
Disable_IOType 13
Disable_IOType_acks 14
disable_iotype_acks_f 14
disable_iotype_f 13
Disable_param_logging 10
disable_param_logging_f 10
Dump_sim_table 36
Emit_control 28
Emit_data_slice 16
emit_data_slice_f 16
Emit_pause_cmd 29
Emit_restart_cmd 29
Emit_resume_cmd 29
Emit_retrieve_param_log_cmd 29
Emit_start 15
emit_start_f 15
Emit_stop 16
emit_stop_f 16
Enable_all_param_logging 11
enable_all_param_logging_f 11
Enable_IOType 13
Enable_IOType_acks 14
enable_iotype_acks_f 14
enable_iotype_f 13
Enable_IOTypes_on_registration 13
enable_iotypes_on_registrn_f 13
Enable_param_logging 10
enable_param_logging_f 10
Get_chk_log_entries 34
Get_chk_log_entries_reverse 34
Get_chk_log_number 33
Get_chktype_number 32
Get_chktypes 32
Get_iotype_number 32
Get_iotypes 32
Get_next_message 28

Get_param_log 31
Get_param_number 30
Get_param_values 31
Get_sim_list 26
Get_supp_cmd_number 33
Get_supp_cmds 33
Record_checkpoint_set 22
record_checkpoint_set_f 22
Register_ChkType 21
register_chktype_f 21
Register_ChkTypes 20
register_chktypes_f 20
Register_IOType 12
register_iotype_f 12
Register_IOTypes 11
register_iotypes_f 11
Register_param 9
register_param_f 9
Register_params 7
register_string_param_f 9
Set_chktype_freq 35
Set_iotype_freq 35
Set_param_values 35
Sim_attach 27
Sim_detach 27
Steerer_finalize 26
Steerer_initialize 26
Steering_control 23
steering_control_f 23
Steering_enable 6
steering_enable_f 6
Steering_finalize 7
steering_finalize_f 7
Steering_initialize 6
steering_initialize_f 7
Steering_pause 24
steering_pause_f 24
Unregister_params 10