Remote Histogramming Package

February 16, 2001

Ullrich Schwanke

Contents

1 Package Overview

1.1 Function

The remote histogramming package (*rhp*) allows to book and fill ntuples, histograms and monitoring ring buffers. Other processes can subscribe to these objects[1] or interrogate their contents over a network.

The *rhp* package shall be used for monitoring and calibration purposes. Whenever the term 'monitoring data' is used in the following it stands for 'monitoring data and data relevant for calibration'. Routines for conversion of histograms and ntuples between *rhp* and several other packages (HBOOK, ROOT, MIZZI data base) are given.

Send comments, complaints, suggestions for improvements, donations and bug reports to schwanke@ifh.de.

1.2 Types of Client-Server Relations

As for now, there are two distinct types of client-server relations *rhp* allows to build up. We will refer to them as *monitor type* and *gatherer type*, respectively. In the following the notion *server* denotes a source of monitoring data. A program which wants to receive monitoring information (accumulated by a server) is called a *client*.

[1]We use 'object' here as generic term for all *rhp* data types. The *rhp* package is far from being 'object oriented'.

1.2.1 Monitor Type

In this scheme, one server process books for example histograms which are filled with monitoring data. The server process decides at what point in time the histograms are full and publishes them. After sending the data, it should reset the contents of the histograms to accumulate new monitoring data.

Any number of clients may obtain a list of booked objects from the server and subscribe to any histogram on the list. Each client process is informed when the server publishes an object it subscribed to. The clients will receive the data of the histograms to display them or store them in a file. The user may bind a private routine to the arrival of a histogram. Clients have to unsubscribe before exiting. In some cases, the user does not have to unsubscribe explicitly since *rhp* keeps track of subscriptions. (See the discussion of **rhp_subscribe** and **rhp_subscribe_ids**.)

1.2.2 Gatherer Type

In contrast to the server in the monitor type of application, the server process of gatherer type remains passive all the time. There is presumably a large number of server processes (e.g. one on every farm PC). Each process books and fills histograms and ntuples. The server checks occasionally for requests from a client process. It sends histogram and ntuple data only if it is asked to do so explicitly.

In this scheme, there is most probably only one client process (gatherer). At startup, it obtains a list of possible servers from an appropriate name-server (see the discussion on **rhp_set_group**) and a list of booked objects (e.g. from one server). It sends out requests for a specific object to all the server processes, receives the incoming data and merges them into one histogram/ntuple. Furthermore, the client process has the ability to reset objects and two switch on/off the accumulation of data in the server processes (see below).

1.3 Special Data Types

The *rhp* ntuples and histograms mimic HBOOK. *Rhp* implements a subset of HBOOK's functionality. All function names are preceded by an underscore, so instead of `hbook1()` one writes `rhp_hbook1()`. The ordering of arguments is mostly the same as in the corresponding HBOOK functions.

In contrast to HBOOK, *rhp* as such does not encourage the user to unbook or delete objects. This restriction will ensure that both client and server have the correct histogram/ntuple parameters all the time. For the same reason, it's not advisable to alter (rebook) objects to obtain, say, a different number of bins in a histogram. Still, you can do it by setting the flag RHP_FLG_REBOOK when initialising *rhp* (see **rhp_init**). All objects are deleted by a call to **rhp_halt** when terminating the program.

Like HBOOK, *rhp* uses numeric identifiers (object identifiers or IDs for short) to refer to objects. These IDs are 32 bit wide integers which must not be 0. The upper 16 bits should be used to denote the entire set of objects booked for one particular purpose by a specific user. They can be set to 0 when only a single user books objects in a certain process (e.g. in a monitoring program). In case of several users one should use a line like

```
#define MYID(ID) ((COMP_ID_OTR<<16) | (ID))
```

to ensure unique IDs (e.g. in the online ARTE process).

The *rhp* package attaches an active flag to all objects having the same upper 16 bit in the ID. The flag indicates whether this group is active or not. Active means that monitoring or calibration data for these histograms and ntuples should be calculated and accumulated.

A server program can use **rhp_active** to interrogate the state of the active flag and act accordingly. The flag can be set to on and off with **rhp_activate** and **rhp_deactivate** by a client process.

In connection with the active flag, there might be cases where it is appropriate to subdivide the upper 16 bit of the ID even further. Each logical group of histograms and ntuples with the same upper 16 bit in the ID has its own active flag. For two independent monitoring or calibration tasks one should use macros like

```
#define MYID1(ID) ((COMP_ID_OTR<<16+1) | (ID))
#define MYID2(ID) ((COMP_ID_OTR<<16+2) | (ID))
```

to be able to switch them on and off separately.

1.4 Dependencies

The *rhp* package runs in the standard C environment of Unix-like operating systems. It was tested on Linux_intel, AIX_risc, Solaris_sparc and IRIX_mips4.

Rhp requires rpm and pub, but does NOT initialise any of the two. All data types are defined in rhp.h. The *rhp* library `librhp.a` can be found in the usual online library directories.

Each *rhp*-version is labelled with a string of the type `RHP-XX-YY-rZ` where `XX`, `YY` and `Z` are numbers. This version string can be found in `rhp.h` and is printed when initilising *rhp* by calling **rhp_init** while debugging is switched on (see below).

Starting with version `RHP-00-06-r1`, *rhp* supports the multi-threaded capabilities of rpm. It does not provide, however, any mechanisms granting exclusive access to its global data structures and particular objects.

1.5 Examples

There are several examples programs for *rhp*, which we will describe here briefly. Additional information is provided by a README files which comes with each example.

1.5.1 Application of Monitor Type

An example for a client-server application of monitor type resides at

/afs/desy.de/group/hera-b/rhp/examples/monitor.

The server process books histograms and ntuples and publishes them from time to time. The client process subscribes to these publications. A user routine is called whenever data arrive.

1.5.2 Application of Gatherer Type

An example for a client-server application of gatherer type is in

/afs/desy.de/group/hera-b/rhp/examples/gather.

The server process books histograms and ntuples and fills them with data if the active flag is set. The client process sets the active flag in the server and sleeps for some time while the server accumulates data. The client requests the histogram and ntuple data from the server and resets them in the server process.

Working with a group of servers is demonstrated by the programs in

/afs/desy.de/group/hera-b/rhp/examples/groups.

The servers are identical to the ones from the gather-example but register all in a special domain. The client contacts a nameserver to look up all processes in this domain.

1.5.3 Applications using Ring Buffers

The usage of ring buffer objects is demonstrated by the programs in

/afs/desy.de/group/hera-b/rhp/examples/ringbuf.

A number of servers book a ring buffer and fill its segments with some dummy data. The client of gatherer type interrogates the contents of all segments from time to time. A user function is called to import data.

1.5.4 Support of other Histogramming Packages

At present, *rhp* supports the conversion of *rhp* histograms and ntuples to HBOOK and ROOT. The required routines are part of the *rhp* library. Programs calling any of these functions must be linked with the CERN libraries (HBOOK) and the ROOT libraries, respectively. The header files `rhp2root.h` and `rhp2hbook.h` can be found in the *rhp* include directory.

For a closer look see the example programs and README files in

/afs/desy.de/group/hera-b/rhp/examples/rhp2root
/afs/desy.de/group/hera-b/rhp/examples/rhp2hbook.

In both cases a client program requests histograms and/or ntuples from a server and converts them to ROOT or HBOOK for displaying.

1.5.5 Writing *rhp* Histograms and Ntuples to a MIZZI Data Base

There are three routines in `librhp.a` which allow for transfer of data between *rhp* histograms and ntuples and the MIZZI data base. Consult the header file `rhp2mizzi.h` and the example programs in

/afs/desy.de/group/hera-b/rhp/examples/rhp2mizzi

for details.

1.6 Bugs and Features

1.6.1 Histograms

The 1d- and 2d-histograms provided by *rhp* allow to allocate different numbers of bytes to store the content of a bin. One bin is either a char, a short, a float or a double. The bin type to use is selected by passing a symbolic constant (RHP_CHAR, RHP_SHORT, RHP_FLOAT, RHP_DOUBLE) to the booking routines (see **rhp_hbook1t** and **rhp_hbook2t**). The type flags for 1d- and 2d-histograms are RHP_FLG_H1 and RHP_FLG_H2, both defined in `rhp.h`.

1.6.2 Row wise ntuples

Ntuples (type flag RHP_FLG_NT) are stored as a sequence of floats (4 bytes). To fit into a standard rpm message the number of columns times the desired number of rows must be smaller than RHP_MAXCOL (currently 359). Therefore it's not advisable to book larger ntuples in the context of a server. In a client of gatherer type, one wants to work of course with larger ntuples to merge the data of several servers. Since the client inquires about the parameters of an ntuple for booking in the local context a problem arises. It is resolved in the following way: When a server is asked to book an ntuple which is to large to be sent over the network without using **rpm_flood** it will book an ntuple which fits just into one rpm message. The number of requested ntuple rows will be kept. It is sent to the client on request which will then book the larger ntuple.

The disadvantage of this procedure is, of course, that the ntuple size is fixed in the server(s). Another way out is to set the flag RHP_FLG_BIGNT in the server(s) and to use **rhp_ctl** to enlarge the ntuple size in the client.

1.6.3 POTs

The *rhp* package gives the user the possibility to derive the development of a histogram in time. One simply writes a server which books and fills the histogram and a client which requests it occasionally. A special data type, loosely called POT ("Parameter Over Time"), was introduced to narrow this concept to the development of one float value in time. It behaves like a one-bin histogram with a time stamp. See the discussion on **pot_book** and **pot_fill** for the details. They associated type flag is called RHP_FLG_POT.

1.6.4 Ring Buffers

Ring buffers are intended for transmission of a byte stream of arbitrary monitoring data. It is only assumed that all data belonging, say, to one event fit into one memory block (segment) and never exceed a certain maximal size. When booking a ring buffer *rhp* will allocate a user defined number of segments of user defined size to perform buffering. At present, all segments must have the same size. Servers can write data into these segments in a round-robin like way. After writing they have to tell *rhp* how much of the segment is used. The contents of all segments is transferred on a client's request as a byte stream. No swapping is performed. In the client program, one can attach a user function to a ring buffer for reading and analysing all arriving segments.

In an application of gatherer type, one is more or less forced to set a user function (using **rhp_set_routine**) to see all arriving segments. Consider the case of a gatherer wanting to request data from M sources with N segments per ring buffer each. A solution would be to allocate $N\dot{M}$ segments in the gatherer, collect all data and then perform the transformation from ring buffer format to user format. This approach requires a lot of memory and is rather inefficient, since the ring buffer segments serve for intermediate storage only. For this reason *rhp* employs another strategy which allows for the same number of segments in gatherer and sources. When collecting data from all sources using **rhp_req2** *rhp* will call the user function every time when N segments arrived. See the client in the `ringbuf` example for an appropriate user function.

The type flag for ring buffers is RHP_FLG_RBF.

1.6.5 Title Strings for Objects

The title strings for *rhp* objects are derived from the user supplied title and the ID of the object as follows: The first two characters in the string are `NT`, `PO`, `H1`, `H2` and `RB` for ntuples, POTs, 1d-histograms, 2d-histograms and ring buffers, respectively. Then follows the object ID in hexadecimal representation and a colon plus the title string passed when booking the object. For example, a call like (macro MYID defined above)

```
rhp_hbookn( MYID(1),"my_title",2,-1);
```

will result in a title

```
"NT0x04000001:my_title",
```

since COMP_ID_OTR equals 0x0400. A list of titles of booked objects can be obtained from a server with **rhp_list**.

1.6.6 Publication Names for Objects

As elaborated above, a client process may subscribe to an *rhp* object. The publication service (pub) uses strings with a length of up to PUB_LNAME (see pub.h) to identify a publication. *Rhp* uses the function **pub_id2nam**[2] to turn the object ID into a unique publication name. The publication name is simply the hexadecimal representation of the ID, preceeded by "0x". (See the documentation of the subscription service.)

[2]This feature exists only in newer versions of the subscription service. It is being used since RHP-00-07-r1. In all older *rhp* versions, the publication name was identical to the object title string.

1.6.7 Debugging

The *rhp* package is capable of producing ample debugging information at runtime. It simply looks at the debug flag of the rpm package. The debug level can be set when starting a program like that:

```
a.out -rpmdebug:<n>
```

When `n` is set to 0, no warnings and error messages are printed. Any value greater than 0 will make *rhp* report all errors returned from rpm and pub calls. Also warnings are printed, e.g. when one uses an ID not known to *rhp* or passes invalid arguments to *rhp* routines.

Another handy routine for debugging is **rhp_dump** which prints out object data to the screen. See the discussion there.

1.7 Dictionary of Terms

Server: A process which books and fills objects. It acts as a **source** of monitoring information. It either publishes the booked objects or answers requests for histogram/ntuple data.

Client: A process which either subscribes to objects, booked by a server, or interrogates their contents. It acts as a **receiver** of monitoring data.

2 User Interface

The following routines can be used regardless of the application type.

- **rhp_init:** initialise the *rhp* package
- **rhp_halt:** stop the *rhp* package
- **rhp_ctl:** set *rhp* parameters
- **rhp_reset:** reset a local object to its initial state
- **rhp_hexist:** check existence of an object
- **rhp_hid:** retrieve IDs of local objects
- **rhp_localIDs:** retrieve IDs of local objects
- **rhp_hnoent:** query number of entries in an object
- **rhp_hi:** query number of entries in a 1d-histogram bin by bin number

- **rhp_hx:** query number of entries in a 1d-histogram bin by bin content
- **rhp_hij:** query number of entries in a 2d-histogram bin by bin number
- **rhp_hxy:** query number of entries in a 2d-histogram bin by content
- **rhp_mult:** multiply histogram by a factor
- **rhp_hgn:** read data from ntuple
- **rhp_start_nt:** prepare reading data from ntuple
- **rhp_read_nt:** read data from ntuple
- **rhp_type:** return type of object
- **rhp_htype:** return type of histogram
- **rhp_hgive1:** query parameters of 1d-histogram
- **rhp_hgive2:** query parameters of 2d-histogram
- **rhp_hgiven:** query parameters of ntuple
- **rhp_title:** query object title
- **rbf_give:** query parameters of a ring buffer
- **rbf_obtain:** get next ring buffer segment
- **rhp_bins:** query number of histogram bins
- **rhp_hunpak:** transfer histogram data to user buffer
- **rhp_hpak:** transfer array to histogram
- **rhp_hcopy:** copy one histogram into another
- **rhp_dump:** dump object to screen
- **rhp_set_userdata:** attach a pointer to user data to an *rhp* object
- **rhp_get_userdata:** retrieve pointer to user data
- **rhp_sleep_and_dispatch:** wait some time for messages
- **rhp_wait_for_one_msg_and_dispatch:** receive one message

Initialise the *rhp* package.

Synopsis

```
int rhp_init(int flags);
```

Parameters

flags Flags altering the behaviour of *rhp*.

Return values

GRR_OK Initialisation completed.

Discussion

This function is documented here only to discuss the meaning of flags altering the behaviour of *rhp*. Users should not call it directly but rather use **rhp_mcinit**, **rhp_msinit**, **rhp_gcinit** and **rhp_gsinit** (see there).

The **flags** argument can take any ored combination of the the following flags:

- RHP_FLG_PUBLISH: if set, every booked object is made public for subscription. Should be used when using *rhp* in a server process of monitor type.
- RHP_FLG_RRESET: if set, booked objects can be reseted remotely by a client process. Should be used for servers in a gatherer type of application.
- RHP_FLG_RESET: if set, the process is allowed to reset its local objects.
- RHP_FLG_ON: if set, the active flag of all objects is set to ON at booking time. Otherwise is defaults to OFF.
- RHP_FLG_BIGNT: if set, the process is allowed to book ntuples which are larger then one standard rpm message. This is useful for client programs which merge a lot of small ntuples.

- RHP_FLG_ACCUMULATE: if set, a client will add arriving histogram data to the local instance instead of overwriting the local histogram with the new data. This flag is meaningless for other types of objects.
- RHP_FLG_REBOOK: if set, object IDs can be reused. Without this flag it is impossible to rebook a histogram or ntuple.

Stop the *rhp* package.

Synopsis

```
int rhp_halt(void);
```

Return values

GRR_OK	Normal successful return.

Discussion

Stops the *rhp* package. Any subscriptions made for objects using rhp_subscribe are automatically deleted. All objects booked are released. Should be the last function called.

Set parameters of the *rhp* package.

Synopsis

```
int rhp_ctl(int flag,int ID,int value);
```

Parameters

flag	Parameter to modify.
ID	Identifier of the object(s).
value	New value for parameter **flag**.

Return values

GRR_OK	Normal successful return.
GRR_INVALID	Invalid input parameter.
GRR_FISHY	Not enough memory.

Discussion

This function can be used to modify the properties of objects. If the lower 16 bit of **ID** are nonzero the function will operate on the single object denoted by **ID**. Otherwise all objects which have the same upper 16 bit in their IDs as **ID** are affected.
GRR_INVALID is returned whenever

- **flag** is not any of the flags described below.
- **ID** matches none of the booked objects.

These are the supported flags:

RHP_FLG_RESET_ON_QUERY: If set, *rhp* clients will will try to reset the object after querying the object data with **rhp_req** or **rhp_req2**. This will only work when RHP_FLG_RRESET was set in the server process. See the discussion on **rhp_init** for details. This flag defaults to off. The flag will be set (cleared) in case **value** is nonzero (zero). Ring buffer do not respect this flag: each segment is reseted automatically after its contents was transferred.

RHP_FLG_ACCUMULATE: With this flag a client can modify the handling of arriving histogram data. If this flag is enabled the arriving histogram will be added to the local instance instead to overwrite it. The initial setting of this flag is off, unless you called **rhp_gcinit** for initialising *rhp*. The flag will be set (cleared) in case **value** is nonzero (zero).

RHP_FLG_NTSIZE: This will set the number of rows in the ntuple(s) denoted by **ID** to **value**. GRR_INVALID is returned if **ID** does not refer to an existing ntuple or **value** is lower than 1. This call is nothing more than a `realloc()` for the memory reserved for the ntuple. If it fails due to lack of memory the ntuple is deleted completely and the return code is GRR_FISHY. If successful, the ntuple is empty. Remember that memory blocks are possibly moved in this call, so mixing **rhp_ctl** with **rhp_start_nt** or **rhp_read_nt** will give unpredictable results.

RHP_FLG_HBOOKED: This flag is set by *rhp* for objects after booking the corresponding HBOOK histogram or ntuple to avoid double booking when calling **rhp_2hbook** several times. You can clear (set) it by passing zero (nonzero) in **value**.

Reset a local object to its initial state.

Synopsis

```
int rhp_reset(int ID);
```

Parameters

ID	Object identifier.

Return values

GRR_OK	Normal successful return.
GRR_INVALID	No object with this ID booked.
GRR_ACCESS	Permission to reset denied.

Discussion

For histograms, this call will clear the content of all bins. Ntuples will contain no entries anymore. As for ring buffers, all segments are marked as empty. This routine operates locally in server and client processes. (See **rhp_rreset** and **rhp_rreseta** to find out how a client can remotely reset an object in a server.) It will fail with GRR_ACCESS if RHP_FLG_RESET was not passed to **rhp_init** at initialisation.

Check existence of an object.

Synopsis

```
int rhp_hexist(int ID);
```

Parameters

ID Object identifier.

Return values

1 Object(s) exist(s).

0 No object(s) matching ID known.

Discussion

If the lower 16 bit of **ID** are not zero this function will check the existence of the particular object denoted by ID. Otherwise it is checked whether there are any objects with IDs returning true when a bit-wise AND is performed with **ID**.

Retrieve object IDs.

Synopsis

```
int rhp_hid(int ID,int *IDvec,int veclen,int *found);
```

Parameters

ID Identifier mask.

IDvec User vector for storage of IDs.

veclen Size of **IDvec**.

found Returns number of objects found.

Return values

GRR_OK Successful return.

Discussion

This function can be used to count the number of objects booked or/and to retrieve their IDs. The first argument **ID** is a mask indicating the objects of interest. Only objects the IDs of which have the same upper 16 bit as in **ID** are affected. A value of 0 for the mask matches all possible IDs. The lower 16 bits can take any ored combination of RHP_FLG_H1, RHP_FLG_H2, RHP_FLG_NT, RHP_FLG_POT and RHP_FLG_RBF to tell the function on what kind of objects to operate.

If **IDvec** is NULL only the number of found objects is returned through **found**. The **veclen** parameter is meaningless then. If **IDvec** is not NULL up to **veclen** found IDs are placed in the user vector **IDvec**. One can use the following calling sequence to obtain, say, all OTR histograms IDs:

```
int found,*vec=NULL;
/* first, count histograms ... */
int mask = (COMP_ID_OTR<<16) | RHP_FLG_H1 | RHP_FLG_H2;
rhp_hid(mask,NULL,0,&found);
if(found>0)
```

```
{
  vec=(int*)malloc(found*sizeof(int));
  /* retrieve the IDs into the allocated vector */
  rhp_hid(mask,vec,found,&found);
}
```

Retrieve object IDs.

Synopsis

```
int* rhp_localIDs(int ID,int *found);
```

Parameters

ID Identifier mask.

found Returns number of objects found.

Return values

NULL No objects matching **ID** found.

≠NULL Allocated vector with matching IDs.

Discussion

This function implements the example shown in the description of **rhp_hid** (see there). The **ID** and **found** arguments are the same as the first and last of **rhp_hid**. The returned vector contains ***found** elements and was allocated using `malloc()`. It is the user responsibility to free the memory again. A return value of NULL indicates that no matching IDs were found.

Return the number of entries in an object.

Synopsis

```
int rhp_hnoent(int ID);
```

Parameters

ID Object identifier.

Return values

$\geq$0 Number of entries.

-1 No object with this ID booked.

Discussion

If **ID** denotes a histogram the number of entries is returned. For ntuples the return value is the number of filled rows. Note that this value will be a constant when the ntuple is full since *rhp* will begin to overwrite the oldest rows. For ring buffers the number of filled segments is returned.

Returns the number of entries in a specific bin of a 1d-histogram.

Synopsis

```
float rhp_hi(int ID,int bin);
double rhp_hidbl(int ID,int bin);
```

Parameters

ID Identifier of the histogram.

bin Index of the bin the content of which is requested.

Return values

≥0 Number of entries.

-1 1d-histogram with this ID not booked.

Discussion

Returns the number of entries in a specific bin or the number of overflows or underflows according to the following table. **nbins** is the number of bins passed to **rhp_hbook1** when booking the histogram.

	return value
bin<1	number of underflows
1≤**bin**≤**nbins**	bin content
bin>**nbins**	number of overflows

Return the number of entries in a specific bin of a 1d-histogram.

Synopsis

```
float rhp_hx(int ID,float x);
double rhp_hxd(int ID,float x);
```

Parameters

ID Identifier of the histogram.

x Value which denotes the bin requested.

Return values

≥0 Number of entries.

-1 1d-histogram with this ID not booked.

Discussion

Returns the number of entries in a specific bin or the number of overflows or underflows according to the following table. **xmin** and **xmax** are the edges of the histogram as supplied to **rhp_hbook1**.

	return value
x<xmin	number of underflows
xmin≤x<xmax	content of bin where **x** would go in.
x≥xmax	number of overflows

Return the number of entries in a specific bin of a 2d-histogram.

Synopsis

```
float rhp_hxy(int ID,float x,float y);
double rhp_hxyd(int ID,float x,float y);
```

Parameters

ID Identifier of the histogram.

x Value which denotes the bin in x requested.

y Value which denotes the bin in y requested.

Return values

$\geq$0 Number of entries.

-1 2d-histogram with this ID not booked.

Discussion

Returns the number of entries in a specific bin or the number of overflows or underflows according to the following table. **xmin**, **xmax**, **ymin** and **ymax** are the edges of the histogram as supplied to **rhp_hbook2**.

	return value
xmin$\leq$**x**$<$**xmax** and **ymin**$\leq$**y**$<$**ymax**	content of bin where (**x**,**y**) would go in.
xmin$\leq$**x**$<$**xmax** amd **y**$<$**ymin**	underflow in y
xmin$\leq$**x**$<$**xmax** amd **y**$\geq$**ymax**	overflow in y
x$<$**xmin** and **y**$<$**ymin**	underflow in x and y
x$<$**xmin** and **ymin**$\leq$**y**$<$**ymax**	underflow in x
x$<$**xmin** and **y**$\geq$**ymax**	underflow in x, overflow in y
x$\geq$**xmax** and **y**$<$**ymin**	overflow in x, underflow in y
x$\geq$**xmax** and **ymin**$\leq$**y**$<$**ymax**	overflow in x
x$\geq$**xmax** and **y**$\geq$**ymax**	overflow in x and y

Return bin contents of a 2d-histogram.

Synopsis

```
float rhp_hij(int ID,int x,int y);
double rhp_hijd(int ID,int x,int y);
```

Parameters

ID Identifier of the histogram.

x Bin number in x.

y Bin number in y.

Return values

≥0 Number of entries in bin.

-1 **ID** denotes no 2d-histogram.

Discussion

Returns bin contents of a 2d-histogram according to the following table where **nbinsx** and **nbinsy** denote the number of bins in x and y.

	return value
1≤**x**≤**nbinsx** and 1≤**y**≤**nbinsy**	number of bin entries
1≤**x**≤**nbinsx** and **y**<1	underflow in y
1≤**x**≤**nbinsx** and **y**>**nbinsy**	overflow in y
x<1 and **y**<1	underflow in x and y
x<1 and 1≤**y**≤**nbinsy**	underflow in x
x<1 and **y**>**nbinsy**	underflow in x and overflow in y
x>**nbinsx** and **y**<1	overflow in x and underflow in y
x>**nbinsx** and 1≤**y**≤**nbinsy**	overflow in x
x>**nbinsx** and **y**>**nbinsy**	overflow in x and y

Multiply a histogram by a factor.

Synopsis

```
int rhp_mult(int ID,float factor);
```

Parameters

ID	Identifier of the histogram.
factor	Scaling factor.

Return values

GRR_OK	Successful return.
GRR_INVALID	Invalid parameter.

Discussion

This function scales the contents of all histogram bins by **factor**. GRR_INVALID is returned if

- **ID** denotes no known histogram.
- **factor** is ≤ 0.

Read data from one row of an ntuple.

Synopsis

```
int rhp_hgn(int ID,int row,float *values);
```

Parameters

ID Identifier of the ntuple.

row Number of row to read data from.

values Buffer for data.

Return values

GRR_OK Successful return.

GRR_INVALID Invalid parameter.

GRR_EMPTY Ntuple contains no data.

Discussion

This function copies the data from one row denoted by **row** to the user buffer **values**. GRR_INVALID is returned in case

- **ID** denotes no ntuple.
- **row** is lower than 1 or too large.

Suppose an ntuple with ID 42 and three columns was booked. One can access its data like this:

```
float data[3];
int i,rows = rhp_hnoent(42);
for(i=1;i<=rows;i++)
{
  rhp_hgn(ID,i,data);
  /* do something with data */
}
```

See **rhp_start_nt** and **rhp_read_nt** for a more efficient way of accessing ntuple data.

Prepare reading of data from an ntuple.

Synopsis

```
void* rhp_start_nt(int ID);
```

Parameters

ID Identifier of the ntuple.

Discussion

This function looks up the address of an ntuple for a more rapid access to ntuple data. NULL is returned when **ID** denotes no ntuple. An internal pointer is set to the first row. See **rhp_read_nt** for an example.

Read all rows of an ntuple consecutively.

Synopsis

```
float* rhp_read_nt(void *addr,float *buffer);
```

Parameters

addr Address of ntuple obtained with **rhp_start_nt**.

buffer Buffer to copy data to.

Discussion

The fastest way of accessing ntuple data is a call to **rhp_start_nt** followed by several calls to **rhp_read_nt**.

This function will return a pointer to the first element of every row in subsequent calls. NULL is returned when all rows were read. If **buffer** is not NULL the data of the present row will be copied there. When ntuple 42 was booked with three columns one can traverse all rows like this:

```
int i;
float *p;
void *addr = rhp_start_nt(42);
for( i=1; p=rhp_read_nt(addr,NULL) ; i++ )
{
  printf("row %i: x0=%f x1=%f x2=%f\n",i,p[0],p[1],p[2]);
}
```

Return the type of an object.

Synopsis

```
int rhp_type(int ID);
```

Parameters

ID Identifier of a histogram or ntuple.

Return values

RHP_FLG_H1	**ID** denotes a 1d-histogram.
RHP_FLG_H2	**ID** denotes a 2d-histogram.
RHP_FLG_NT	**ID** denotes an ntuple.
RHP_FLG_POT	**ID** denotes an pot object.
RHP_FLG_RBF	**ID** denotes a ring buffer.
RHP_FLG_NOTYPE	**ID** is unknown.

Discussion

Returns the type of an object. The symbolic constants are defined in rhp.h.

Return the type of a histogram.

Synopsis

```
int rhp_htype(int ID);
```

Parameters

ID Identifier of a histogram.

Return values

RHP_CHAR	Histogram with sizeof(char) bytes/bin.
RHP_SHORT	Histogram with sizeof(short) bytes/bin.
RHP_FLOAT	Histogram with sizeof(float) bytes/bin.
RHP_DOUBLE	Histogram with sizeof(double) bytes/bin.
-1	**ID** denotes no histogram.

Discussion

Returns the type of a histogram. The symbolic constants are defined in rhp.h.

Query parameters of a 1d-histogram.

Synopsis

```
int rhp_hgive1(int ID,char *title,int *nbins,float *xmin,float *xmax);
```

Parameters

ID Identifier of a 1d-histogram.

Return values

GRR_INVALID	**ID** denotes no 1d-histogram.
GRR_OK	Successful return.

Discussion

This function returns the parameters of a 1d-histogram. The title of the histogram is copied to the user buffer **title**. Make sure its size is at least RHP_TITLE_LEN. The number of bins is returned through **nbins**. The edges of the histogram are copied to **xmin** and **xmax**. Copying is only performed if the pointer in question is not NULL. So it's legal to obtain the number of bins in a histogram with ID 42 like this:

```
int nbins;
rhp_hgive1(42,NULL,&nbins,NULL,NULL);
```

Query parameters of a 2d-histogram.

Synopsis

```
int rhp_hgive2(int ID,char *title,int *nbinsx,float *xmin,float *xmax,
int *nbinsy,float *ymin,float *ymax);
```

Parameters

ID Identifier of a 2d-histogram.

Return values

GRR_INVALID	**ID** denotes no 2d-histogram.
GRR_OK	Successful return.

Discussion

This function returns the parameters of a 2d-histogram. See the discussion on **rhp_hgive1**. Again, any pointer can be NULL if this information is not needed.

Query parameters of an ntuple.

Synopsis

```
int rhp_hgiven(int ID,char *title,int *ncol,char *varnames);
```

Parameters

ID Identifier of an ntuple.

Return values

GRR_INVALID	**ID** denotes no ntuple.
GRR_OK	Successful return.

Discussion

This function returns the parameters of an ntuple. The title is copied to the char-array **title** which should have a length of at least RHP_TITLE_LEN bytes. The number of columns is returned through **ncol**. A comma-separated list of variable names is copied to **varnames** (length RHP_VAR_LEN). Any of the pointers passed can be NULL if this information is not needed.

Obtain a copy of an object's title.

Synopsis

```
int rhp_title(int ID,char *userbuf,int size);
```

Parameters

ID	Object identifier.
userbuf	Pointer to user buffer.
size	Size of buffer in bytes.

Return values

GRR_INVALID	**ID** denotes no known object.
GRR_OK	Successful return.

Discussion

This function can be used to obtain a copy of an object's title. Up to **size** bytes (including a trailing 0) are copied to the user buffer **userbuf**. It's a good idea to provide a buffer with RHP_TITLE_LEN (see rhp.h) bytes to prevent any truncation of the title string.

Query parameters of a ring buffer.

Synopsis

```
int rbf_give(int ID,char *title,int* nseg,int* segsize);
```

Parameters

ID Identifier of a ring buffer.

Return values

GRR_INVALID	**ID** denotes ring buffer.
GRR_OK	Successful return.

Discussion

This function returns the parameters of a ring buffer. The title is copied to the char-array **title** which should have a length of at least RHP_TITLE_LEN bytes. The number of segments is returned through **nseg**, their size through **segsize**. Any of the pointers passed can be NULL if this information is not needed.

rbf_obtain rbf_obtain

Get address and size of next segment in ring buffer.

Synopsis

```
char* rbf_obtain(int ID,int mode,int* segsize,int *usedsize);
```

Parameters

ID Identifier of a ring buffer.

mode Access mode.

segsize Returns segment size.

usedsize Returns number of used bytes in segment.

Discussion

This function returns the address of the next segment in the ring buffer. The value of **mode**, which can be either RBF_READ or RBF_WRITE, specifies whether the segment is accessed for reading or writing. In the latter case the buffer is declared as 'empty' before returning its address. The nominal size of the segment is returned through **segsize**, the number of used bytes through **usedsize**. Both pointers can be NULL if the information is not needed. NULL is returned if **ID** denotes no ring buffer or **mode** is invalid. Remember that the returned pointer refers to a memory block allocated by *rhp*. Don't keep it longer than needed and don't free it.

When obtaining the buffer address for writing you should call **rbf_release** after copying in the data to tell *rhp* how many bytes were written. Make sure you don't write more than ***segsize** bytes.

Query number of histogram bins.

Synopsis

```
int rhp_bins(int ID,int *nbinsx,int *nbinsy);
```

Parameters

ID Identifier of a histogram.

Return values

GRR_INVALID	**ID** denotes no histogram.
GRR_OK	Successful return.

Discussion

This function returns the number of bins for 1d- and 2d-histograms. In case of a 1d-histogram **nbinsy** is not used and can be set to NULL.

Transfer data of a histogram to a user buffer.

Synopsis

```
int rhp_hunpak(int ID,float *dest);
int rhp_hunpakd(int ID,double *dest);
```

Parameters

ID Identifier of a histogram.

dest User buffer for histogram data.

Return values

GRR_INVALID **ID** denotes no histogram.

GRR_OK Successful return.

Discussion

This function copies the histogram data to a user buffer. The length of the buffer should equal the number of bins (the product of bins in x and y) for 1d-histograms (2d-histograms).

Note that *rhp* uses FORTRAN's array convention when storing the data of a 2d-histogram to speed up the data transfer between *rhp* and HBOOK histograms. One must use ($0 \leq \mathbf{x} < \mathbf{nbinsx}$, $0 \leq \mathbf{y} < \mathbf{nbinsy}$)

```
offset = y*nbinsx + x;
```

instead of

```
offset = x*nbinsy + y;
```

to calculate the offset at which the contents of bin (**x**,**y**) can be found.

Transfer array to histogram.

Synopsis

```
int rhp_hpak(int ID,float *vec);
int rhp_hpakd(int ID,double *vec);
```

Parameters

ID Identifier of a histogram.

vec User buffer with histogram data.

Return values

GRR_INVALID **ID** denotes no histogram.

GRR_OK Successful return.

Discussion

This function will set the **i**-th bin of the histogram to the value stored in **vec**[**i**]. It is assumed that the length of the passed vector equals the number of histogram bins. The histogram is reseted before filling. See the discussion for **rhp_hunpak** on how to store the data for 2d-histograms.

Copy one histogram onto another.

Synopsis

```
int rhp_hcopy(int ID1,int ID2);
```

Parameters

ID1	Identifier of the source histogram.
ID2	Identifier of the destination histogram.

Return values

GRR_INVALID	Error related to passed IDs.
GRR_OK	Successful return.
GRR_FISHY	Error in histogram parameters.

Discussion

The above call will make the histogram **ID2** an exact copy of histogram **ID1**. If **ID2** is not known yet an appropriate histogram will be booked. GRR_INVALID is returned in case

- **ID1** is unknown or denotes no histogram.
- **ID2** is identical with **ID1** or denotes another type of object.

Calling **rhp_hcopy** will fail and return GRR_FISHY if the histogram **ID2** was booked with other parameters (number of bins, lower and upper edges) as histogram **ID1**. This restriction does *not* exist in HBOOK. It is enforced by the usage of *rhp* in client-server relations.

Dump object to screen.

Synopsis

```
int rhp_dump(int ID);
```

Parameters

ID Object identifier.

Return values

GRR_INVALID	**ID** is unknown.
GRR_OK	Successful return.

Discussion

This function will print information about the object with identifier **ID** on the screen. All booking parameters and data are included. If 0 is passed for **ID** a list of all objects booked will be shown.

rhp_set_userdata

Associate pointer to user data with an object.

Synopsis

```
int rhp_set_userdata(int ID,void* yourdata);
```

Parameters

ID	Object identifier.
yourdata	Any pointer.

Return values

GRR_INVALID	**ID** is unknown.
GRR_OK	Successful return.

Discussion

This function might be handy if one wants to associate additional information with an *rhp* object. The passed pointer is stored in the object structure and can be retrieved using **rhp_get_userdata**.

rhp_get_userdata rhp_get_userdata

Retrieve pointer to user data.

Synopsis

```
void* rhp_get_userdata(int ID);
```

Parameters

ID Object identifier.

Return values

NULL **ID** is unknown or stored pointer is NULL.

!=NULL Successful return.

Discussion

This function returns the pointer which was set by calling **rhp_set_userdata**. NULL is returned when no such pointer was set before.

rhp_sleep_and_dispatch rhp_sleep_and_dispatch

Wait for messages and handle them.

Synopsis

```
int rhp_sleep_and_dispatch(int secs);
```

Parameters

secs Time to wait in seconds.

Return values

GRR_OK Time is up.

Discussion

This function will return after **secs** seconds. In the meantime it receives messages of arbitrary type with **rpm_receive** and delivers them using **rpm_dispatch**. If **secs** is lower than 1 the function will return immediately.

The function handles messages of any size and **rpm_flood** is called when necessary. Starting with `RHP-00-06-r1` it is also aware of the rpm features connected with the usage of POSIX threads.

Wait for one message and handle it.

Synopsis

```
int rhp_wait_for_one_msg_and_dispatch(int block);
```

Parameters

block Indicates blocking mode.

Return values

GRR_OK	A message was received and handled successfully.
GRR_TIME	No message was received.
rpm error code	Error reported by **rpm_receive**.

Discussion

The name of this function is so long to explain all of its functionality. It checks for messages in the mode specified by **block** which should be either NO_BLOCK, BLOCK_TIL_TIMEOUT or BLOCK_FOREVER. If a message is received it is passed to **rpm_dispatch**. See the rpm documentation for more information on the different blocking modes and the dispatch mechanism.

The function handles messages of any size and **rpm_flood** is called when necessary. Starting with `RHP-00-06-r1` it is also aware of the rpm features connected with the usage of POSIX threads.

3 Server Functions

The following routines are called in server processes for booking and filling of objects.

- **rhp_hbook1t:** book a 1d-histogram
- **rhp_hbook1:** book a 1d-histogram (1 float/bin)
- **rhp_hf1:** fill a 1d-histogram
- **rhp_hbook2t:** book a 2d-histogram
- **rhp_hbook2:** book a 2d-histogram (1 float/bin)
- **rhp_hfill:** fill a histogram
- **rhp_hfpak1:** fill a vector into a 1d-histogram
- **rhp_hbookn:** book a row wise ntuple
- **rhp_hbookn2:** book a row wise ntuple
- **rhp_hfn:** fill a row in an ntuple
- **pot_book:** book a POT
- **pot_fill:** fill a POT
- **rbf_book:** book a ring buffer
- **rbf_release:** signal that writing to the ring buffer is done

3.1 Server of Monitor Type

These functions help to implement a server in a monitor type of application.

- **rhp_msinit:** initialise *rhp* for use in a server process
- **rhp_msbind:** bind routines for a server of monitor type
- **rhp_publish:** publish a booked object to all subscribers
- **rhp_test:** test whether an object has subscribers

3.2 Server of Gatherer Type

These routines can be used in a server in a gatherer type of application.

- **rhp_gsinit:** initialise *rhp* for use in a server process
- **rhp_gsbind:** bind routines for a server of gatherer type
- **rhp_active:** interrogate active flag of a group of histograms/ntuples

Book a 1d-histogram.

Synopsis

```
int rhp_hbook1t(int ID,char* title,int nbins,float xmin,float xmax,int type);
```

Parameters

ID	Identifier of the 1d-histogram.
title	Histogram title.
nbins	Number of desired bins.
xmin	Lower edge of the histogram.
xmax	Upper edge of the histogram.
type	Desired histogram type.

Return values

GRR_OK	Normal successful return.
GRR_DEFINED	**ID** is already in use.
GRR_INVALID	Invalid input parameter.
GRR_FISHY	Not enough memory for histogram.

Discussion

Books a 1d-histogram. GRR_INVALID is returned when

- the lower 16 bit of the ID are zero.
- **xmin**$\geq$**xmax**.
- **nbins**$<$1.
- **type** is none of RHP_CHAR, RHP_SHORT, RHP_FLOAT or RHP_DOUBLE.

The **type** parameter specifies how many bytes are allocated per histogram bin. The title string is copied.

Book a 1d-histogram with one float per bin.

Synopsis

```
int rhp_hbook1(int ID,char* title,int nbins,float xmin,float xmax);
```

Discussion

Same functionality as `rhp_hbook1t(ID,title,nbins,xmin,xmax,RHP_FLOAT)`. See the discussion there.

Fill a 1d-histogram.

Synopsis

```
int rhp_hf1(int ID,float x,float weight);
int rhp_hf1d(int ID,float x,double weight);
```

Parameters

ID	Identifier of a 1d-histogram.
x	Value to fill in.
weight	Weight for the bin.

Return values

GRR_OK	Normal successful return.
GRR_INVALID	Invalid parameter.

Discussion

Accumulates an entry in a 1d-histogram. GRR_INVALID is returned if **ID** denotes no 1d-histogram. The overflow bin (underflow bin) is increased if **x** is greater or equal than the upper edge (smaller than the lower edge) of the histogram.

rhp_hbook2t rhp_hbook2t

Book a 2d-histogram.

Synopsis

```
int rhp_hbook2t(int ID,char* title,int nbinsx,float xmin,float xmax,
                               int nbinsy,float ymin,float ymax,int type);
```

Parameters

ID Identifier of the 1d-histogram.

title Histogram title.

nbinsx Number of desired bins in x.

xmin Lower edge of the histogram in x.

xmax Upper edge of the histogram in x.

nbinsy Number of desired bins in y.

ymin Lower edge of the histogram in y.

ymax Upper edge of the histogram in y.

type Desired histogram type.

Return values

GRR_OK Normal successful return.

GRR_DEFINED **ID** is already in use.

GRR_INVALID Invalid input parameter.

GRR_FISHY Not enough memory for histogram.

Discussion

Books a 2d-histogram. GRR_INVALID is returned when

- the lower 16 bit of the ID are zero.
- **xmin**$\geq$**xmax**.
- **ymin**$\geq$**ymax**.
- **nbinsx**<2 or **nbinsy**<2.
- **type** is none of RHP_CHAR, RHP_SHORT, RHP_FLOAT or RHP_DOUBLE.

The **type** parameter specifies how many bytes are allocated per histogram bin. The title string is copied.

Book a 2d-histogram with one float per bin.

Synopsis

```
int rhp_hbook2(int ID,char* title,int nbinsx,float xmin,float xmax,
                              int nbinsy,float ymin,float ymax);
```

Discussion

Same as `rhp_hbook2t(ID,title,nbinsx,xmin,xmax,nbinsy,ymin,ymax,RHP_FLOAT)`. See the discussion there.

rhp_hfill rhp_hfill

Fill a histogram.

Synopsis

```
int rhp_hfill(int ID,float x,float y,float weight);
int rhp_hfilld(int ID,float x,float y,double weight);
```

Parameters

ID	Identifier of the histogram.
x,y	Value to fill in.
weight	Weight for the bin.

Return values

GRR_OK	Normal successful return.
GRR_INVALID	Invalid parameter.

Discussion

Accumulates an entry in a histogram. GRR_INVALID is returned if **ID** denotes no histogram. For 1d-histograms the third argument **y** is ignored. The function has then the same functionality as `rhp_hf1(ID,x,weight)` or `rhp_hf1d(ID,x,weight)`, respectively.

Fill a vector into a 1d-histogram.

Synopsis

```
int rhp_hfpak1(int ID,float *vec,int vec_len);
```

Parameters

ID	Identifier of the histogram.
vec	Vector to fill in.
vec_len	Number of elements in **vec**.

Return values

GRR_OK	Normal successful return.
GRR_INVALID	1d-histogram with this ID not booked.

Discussion

Fills a vector into the 1d-histogram denoted by **ID**. This function is equivalent to calling `rhp_hf1(ID,vec[i],1)` for all elements `i` in the vector, but faster.

Book a row wise ntuple (RWN).

Synopsis

```
int rhp_hbookn(int ID,char* title,int ncol,int nrow);
```

Parameters

ID	Identifier of the RWN.
title	Title string for the ntuple.
ncol	Number of columns (i.e. number of variables).
nrow	Maximal number of rows.

Return values

GRR_OK	Ntuple booked successfully.
GRR_DEFINED	**ID** is already in use.
GRR_INVALID	Invalid input parameter.
GRR_FISHY	Not enough memory for ntuple.

Discussion

Books a RWN. The number of desired columns is given by **ncol**. GRR_INVALID is returned under the following circumstances:

- The lower 16 bit of the ID are zero.
- **ncol**<1.
- **ncol**>RHP_MAXCOL.

If **nrow** is lower than 1 *rhp* will allocate so many rows that one standard rpm message can contain the entire ntuple data.

If RHP_FLG_BIGNT is not set and **nrow** is larger than the maximal possible value compatible with **ncol**, it is set to RHP_MAXCOL/**ncol**. The title string is copied.

When writing an ntuple booked with **rhp_hbookn** to an HBOOK file the names of the variables will be `x0,x1,x2,...`.

Book a row wise ntuple (RWN).

Synopsis

```
int rhp_hbookn2(int ID,char* title,char *varstring,int nrow);
```

Parameters

ID Identifier of the RWN.

title Title string for the ntuple.

varstring A string with names for variables (i.e. columns).

nrow Maximal number of rows.

Return values

GRR_OK Ntuple booked successfully.

GRR_DEFINED **ID** is already in use.

GRR_INVALID Invalid input parameter.

GRR_FISHY Not enough memory for ntuple.

Discussion

Books a RWN. The number of desired columns and the associated names are derived from **varstring**. The string **varstring** is a list of identifiers which are separated by spaces, commas, colons or semicolons. A string like `"px:py:pz"` would result in an ntuple with three columns. The identifiers have to start with a letter and may continue with letters and digits. Like in HBOOK, underscores are forbidden. Only the first 8 characters are used. A string like `"x12345678,a,b"` is converted to `"x1234567,a,b"`. GRR_INVALID is returned under the following circumstances:

- The lower 16 bit of the ID are zero.

- **ncol**<1.
- **ncol**>RHP_MAXCOL.
- Error when parsing **varstring**.
- **varstring** is too long.

A detailed report on parsing errors is given if debugging is switched on.

If **nrow** is lower than 1 *rhp* will allocate so many rows that one standard rpm message can contain the entire ntuple data.

If RHP_FLG_BIGNT is not set and **nrow** is larger than the maximal possible value compatible with **ncol**, it is set to RHP_MAXCOL/**ncol**. The title string is copied.

When writing an ntuple booked with **rhp_hbookn2** to an HBOOK file the names of the variables will be like in **varstring**.

Fill a row in a RWN.

Synopsis

```
int rhp_hfn(int ID,float* values);
```

Parameters

ID Identifier of the RWN.

values Pointer to float-array with (at least) as many entries as columns in the RWN.

Return values

GRR_OK Ntuple filled correctly.

GRR_INVALID The ID denotes no ntuple.

Discussion

Fills a row in a RWN with values from a float-array. If the memory for storing the data is exhausted, the oldest rows are overwritten.

pot_book pot_book

Book a POT.

Synopsis

```
int pot_book(int ID,char *title);
```

Parameters

ID Identifier for the POT.

title Title or name.

Return values

GRR_OK	POT booked successfully.
GRR_INVALID	The ID is zero.
GRR_DEFINED	The ID is already used.
GRR_FISHY	Not enough memory.

Discussion

This function books a POT. The title string is copied. Both contents and time stamp are initialised to zero.

pot_fill pot_fill

Fill a POT.

Synopsis

```
int pot_fill(int ID,float* value);
```

Parameters

ID Identifier of the POT.

value Value to fill.

Return values

GRR_OK Successful return.

GRR_INVALID The ID denotes no POT.

Discussion

This routines will store **value** in the POT denoted by **ID**. The time stamp is set by calling `time(0)`, which returns the number of seconds since 00:00:00 UTC, January 1, 1970. Note that only one single value is stored in a POT at a time. So you have to call **rhp_publish** in a server of monitor type immediately after each fill command. If no client sends a request for the POT to a server of gatherer type between to fill calls the stored value will be overwritten and is lost.

rbf_book rbf_book

Create a ring buffer.

Synopsis

```
int rbf_book(int ID,char *title,int nseg,int segsize);
```

Parameters

ID	Identifier of the ring buffer.
title	A title string.
nseg	Number of desired segments in ring buffer.
segsize	Size of one segment in bytes.

Return values

GRR_OK	Successful return.
GRR_DEFINED	**ID** is already in use.
GRR_INVALID	Invalid parameter.
GRR_FISHY	Out of memory.

Discussion

This function will book a ring buffer with **nseg** segments, each with a size of **segsize** bytes. GRR_INVALID is returned

- the lower 16 bit of the ID are zero.
- **nseg**<1.

If **segsize** is lower than 1 it is set to RBF_MXTXT (1416). This results in a canonical rpm message size of RPM_MXTXT bytes, which still can be transferred using **rpm_send**. The title string is copied.

rbf_release rbf_release

Signal that a ring buffer segment was filled.

Synopsis

```
int rbf_release(int ID,char *buffer,int nbytes);
```

Parameters

ID Identifier of the ring buffer.

buffer Segment address returned from **rbf_obtain**.

Return values

GRR_OK Successful return.

GRR_INVALID The ID denotes no ring buffer.

Discussion

To write data into a ring buffer segment a process should call **rbf_obtain** to get the buffer address. The above function tells *rhp* that **nbytes** bytes were written into the buffer.

Initialise the *rhp* package for use in server of monitor type.

Synopsis

```
int rhp_msinit(void);
```

Return values

GRR_OK	Initialisation completed.

Discussion

This function should be called in a server of monitor type before any other function described in this manual. It assumes that rpm and pub have been initialised before. It is equivalent to calling

```
rhp_init(RHP_FLG_PUBLISH | RHP_FLG_RESET | RHP_FLG_ON).
```

Set up handler for incoming *rhp* mmessages in a server of monitor type.

Synopsis

```
int rhp_msbind(void);
```

Return values

GRR_OK Binding performed.

Discussion

This function binds routines which are called to handle incoming requests for object data.

Publish one or several objects to possible subscribers.

Synopsis

```
int rhp_publish(int ID);
```

Parameters

ID Object identifier.

Return values

GRR_OK	Successful return.
GRR_INVALID	**ID** is unknown.
pub error code	Error using **pub_publish**.

Discussion

This function will publish the object denoted by **ID**. All possible subscribers will receive a message with the object data. For ring buffers one message per filled segment is sent. If **ID** is zero all local objects are published.

Test whether an object has subscribers.

Synopsis

```
int rhp_test(int ID);
```

Parameters

ID Object identifier.

Return values

GRR_INVALID	**ID** is unknown.
GRR_OK	**ID** exists and has subscribers.
GRR_EMPTY	**ID** exists but has no subscribers.
GRR_UNINITIALIZED	pub was not initialised.
GRR_UNKNOWN	Object not published.

Discussion

This function tests whether the object denoted by **ID** has subscribers. A return value of GRR_UNKNOWN indicates that the flag RHP_FLG_PUBLISH was not set when initialising *rhp*.

Initialise the *rhp* package for use in server of gatherer type.

Synopsis

```
int rhp_gsinit(void);
```

Return values

GRR_OK Initialisation completed.

Discussion

This function should be called in a server of gatherer type before any other function described in this manual. It assumes that rpm has been initialised before. It is equivalent to

`rhp_init(RHP_FLG_RRESET).`

rhp_gsbind rhp_gsbind

Set up handler for incoming *rhp* messages in a server of gatherer type.

Synopsis

```
int rhp_gsbind(void);
```

Return values

GRR_OK	Binding performed.

Discussion

This function binds routines which are called to handle incoming requests for object data.

Query active flag of a group of objects.

Synopsis

```
int rhp_active(int ID);
```

Parameters

ID ID of a group of objects.

Return values

1 Group with **ID** is active.

0 Group with **ID** is inactive.

Discussion

This function returns the state of the active flag for a group of objects which have the same upper 16 bit as in **ID**. Unknown IDs are regarded as inactive, 0 is returned.

4 Client Functions

These routines help to obtain and parse lists of booked objects from a server.

- **rhp_list:** obtain a list of booked objects
- **rhp_book_list:** book all members of a list in local context
- **rhp_parse_list:** dissect the list obtain by **rhp_list**
- **rhp_book:** book a list entry in local context
- **rhp_rhid:** retrieve IDs of objects remotely
- **rhp_readIDs:** retrieve vector with object IDs from a server
- **rhp_book_nr:** book an object in local context
- **rhp_obj_info:** read object booking parameters from a server
- **rhp_read_object_info:** read object booking parameters from a server

This call binds the execution of a user routine to the arrival of an object.

- **rhp_set_routine:** bind a user routine to the arrival of data

4.1 Client of Monitor Type

The listed routines help to implement a client in a monitor type of application.

- **rhp_mcinit:** initialise the *rhp* package for a client process
- **rhp_mcbind:** bind routines for the client process
- **rhp_subscribe:** subscribe to an object
- **rhp_subscribe_ids:** subscribe to several objects
- **rhp_unsubscribe:** end subscription to an object
- **rhp_unsubscribe_ids:** end subscription to several objects

4.2 Client of Gatherer Type

These functions can be used in a client of gatherer type.

- **rhp_gcinit:** initialise the *rhp* package for a client process
- **rhp_gcbind:** bind routines for the client process
- **rhp_set_group:** look up server processes in nameserver
- **rhp_rem_group:** delete list of server processes
- **rhp_activate:** set active flag in server to on
- **rhp_deactivate:** set active flag in server to off
- **rhp_req:** request object data from a server
- **rhp_req2:** more clever version of **rhp_req**
- **rhp_rreset:** reset object in a remote process
- **rhp_rreseta:** reset objects in remote process

Obtain a list of objects booked by a server process.

Synopsis

```
int rhp_list(char *srv,int pattern,char *buf,int sizeofbuf);
```

Parameters

srv	Name of server or group.
pattern	A bit pattern denoting which objects are of interest.
buf	Buffer to store the obtained list.
sizeofbuf	Size of buffer.

Return values

GRR_OK	Obtained list.
rpm error code	Could not locate server or error using **rpm_request**.

Discussion

This function tries to obtain a list of objects from a server. The server will only send information about objects with IDs having the same upper 16 bit as **pattern**. The lower 16 bits of **pattern** can take any ored combination of RHP_FLG_H1, RHP_FLG_H2, RHP_FLG_NT, RHP_FLG_POT and RHP_FLG_RBF to select the desired objects type(s). A **pattern** of 0 matches all possible IDs.

The buffer must be large enough to store all received description strings. If N objects are present in the server process a buffer size of $N \cdot$ RHP_TITLE_LEN should do. (One can use **rhp_rhid** to obtain N.)

On successful return, **buf** contains a sequence of 0 terminated object description strings. The last string is terminated by two zeros. There are basically two things one can do with a list:

a) Pass it on to **rhp_book_list** to book all list entries in the local context.

b) Use **rhp_parse_list** to dissect the list. One can then select the description strings of interest and book them in the local context with **rhp_book**.

rhp_book_list rhp_book_list

Book all objects on the list in local context.

Synopsis

```
int rhp_book_list(char *srv,char *buf,int *IDvec,int max,int* found);
```

Parameters

srv	Name of server or group.
buf	Buffer passed to **rhp_list**.
IDvec	Array of int to store IDs.
max	Number of elements in **IDvec**.
found	Returns number of valid entries in **IDvec**.

Return values

GRR_OK	Successful return.
GRR_INVALID	List contains corrupted description strings.
rpm error code	Error using **rpm_locate** or **rpm_request**.

Discussion

Parses a list obtained with **rhp_list** and books all objects in the local context. The server is contacted to find out about the various object parameters. The IDs are returned in **IDvec**, which can hold **max** elements. The IDs in **IDvec[0],...,IDvec[*found-1]** are valid. All remaining elements are set to zero.

rhp_parse_list rhp_parse_list

Dissect a list obtained with **rhp_list**.

Synopsis

```
char* rhp_parse_list(char *buf);
```

Parameters

buf Buffer passed to **rhp_list**.

Return values

GRR_OK List parsed.

Discussion

Dissects a list obtained with **rhp_list**. The function is similar to `strtok()` from the ANSI-C library. When calling it for the first time one should pass the buffer from the call to **rhp_list**. In subsequent calls the argument should be NULL. The function will return pointers to the object description strings. A NULL pointer is returned when the list was traversed completely. Example (`list_buf` is the buffer passed to **rhp_list**):

```
char *s;
int i=0;
for(s=rhp_parse_list(list_buf); s ;s=rhp_parse_list(NULL))
   printf("%i-th description string is %s\n",++i,s);
```

This function is not reentrant.

Book an entry of a list in the local context.

Synopsis

```
int rhp_book(char *server,char *desc_str,int *ID);
```

Parameters

server	Name of server or group.
desc_str	Object description string.
ID	Returns the ID of the booked object.

Return values

GRR_OK	Successful return.
GRR_INVALID	**desc_str** is corrupted.
rpm error code	Error using **rpm_locate** or **rpm_request**.

Discussion

Books an entry of a list obtained with **rhp_list** and **rhp_parse_list** in the local context. The server is contacted to find out about the object parameters. The ID is returned through **ID** and will be zero in case of an error.

Retrieve object IDs remotely.

Synopsis

```
int rhp_rhid(char *server,int ID,int *IDvec,int veclen,int *found);
```

Parameters

server	Name of server or group.
ID	Identifier mask.
IDvec	User vector for storage of IDs.
veclen	Size of **IDvec**.
found	Returns number of objects found.

Return values

GRR_OK	Successful return.
rpm error code	Error using **rpm_locate** or **rpm_request**.

Discussion

This is the remote version of **rhp_hid**. Apart from the name of the server(s) all arguments are the same as for **rhp_hid** (see discussion there). The function gives information about objects booked in a server process, while **rhp_hid** operates locally. Again, **IDvec** may be set to NULL to count the number of matching objects first. The obtained IDs can be passed to **rhp_book_nr** for booking in the local context.

Retrieve a vector with object IDs from a server.

Synopsis

```
int* rhp_readIDs(char *server,int mask,int *found);
```

Parameters

server Name of server or group.

mask Mask denoting type of desired objects.

found Returns number of IDs in returned vector.

Return values

!=NULL Successful return.

NULL An error occurred.

Discussion

This routine calls **rhp_rhid** twice. First to count the number of objects matching **mask** in the server process. It then allocates a buffer using `malloc()` which can hold the IDs and retrieves them with a second call to **rhp_rhid**. The buffer is returned to the user who is responsible to `free()` it after using. The number of elements in the returned vector is copied to ***found**. NULL is returned in case of error. Setting the debug flag should reveal the actual source of the failure.

Book an object in the local context.

Synopsis

```
int rhp_book_nr(char *server,int ID);
```

Parameters

server	Name of server or group.
ID	ID of object to book.

Return values

GRR_OK	Successful return.
GRR_DEFINED	**ID** already booked locally.
GRR_INVALID	No object with **ID** in server process.
rpm error code	Error using **rpm_locate** or **rpm_request**.

Discussion

The function **rhp_book_nr** takes an object identifier (possibly obtained using **rhp_rhid**) and contacts the server to inquire about the object parameters. The object is booked in the local context.

Retrieve object booking parameters from a server.

Synopsis

```
int rhp_obj_info(char *server,int mask,char *buf,int size,int *found);
```

Parameters

server	Name of server or group.
mask	An identifier mask.
buf	User buffer.
size	Size of user buffer in bytes.
found	Returns number of objects found.

Return values

GRR_OK	Successful return.
GRR_INVALID	The **mask** is invalid.
rpm error code	Error using **rpm_locate** or **rpm_request**.

Discussion

This function allows to obtain an array with booking parameters of objects in a server. The lower 16 bit of **mask** must be one (and only one) of *rhp*'s object type flags. The upper 16 bit exclude all objects the IDs of which do not match this pattern. The information, which is an array of structures, is copied to the user buffer **buf** of size **size**. The number of array elements is returned through **found** and is zero in case of an error. The following table shows the relation between *rhp*'s type flags and the structure of the array.

lower 16 bit of **mask**	structure
RHP_FLG_H1	`struct rhp_h1_info`
RHP_FLG_H2	`struct rhp_h2_info`
RHP_FLG_NT	`struct rhp_nt_info`
RHP_FLG_POT	`struct rhp_pot_info`
RHP_FLG_RBF	`struct rhp_rbf_info`

The structure definitions can be found in `rhp.h`. If you consider using this function have a look at **rhp_read_obj_info**.

Retrieve object booking parameters from a server.

Synopsis

```
char* rhp_read_obj_info(char *server,int mask,int *found);
```

Parameters

server	Name of server or group.
mask	An identifier mask.
found	Returns number of objects found or error code.

Return values

NULL	Error or no objects found.
!=NULL	Pointer to structure array.

Discussion

A disadvantage of **rhp_obj_info** (see there) is that the user must supply a buffer the size of which depends on the (unknown) number of objects in the server. This function uses **rhp_rhid** to count the number of objects matching **mask** in the server. It then allocates a buffer using `malloc()` and calls **rhp_obj_info**. to retrieve the data. On success the pointer is returned and the number of elements in the array is copied to ***found**. The user is responsible for freeing the memory again. In case of a failure, NULL is returned and ***found** contains the return code of **rhp_obj_info**. A return value of NULL and ***found**==GRR_OK indicate that no data matching **mask** was found.

rhp_set_routine rhp_set_routine

Attach user routine to an *rhp* object.

Synopsis

```
int rhp_set_routine(int ID,int (*user)(int ID,void *object));
```

Parameters

ID Object identifier.

user User routine.

Return values

GRR_OK	Successful return.
GRR_INVALID	**ID** is unknown.

Discussion

This call binds a user routine to an object. It will be called whenever data for this object arrive. The arguments passed to the user routine are the object identifier and its address which is of type `Rhp_nt*`, `Rhp_h1*`, `Rhp_h2*`, `Rhp_pot*` or `Rhp_rbf*`. It is legal to supply NULL for **user** to remove the user routine from a former call to **rhp_set_routine**.

rhp_mcinit rhp_mcinit

Initialise the *rhp* package for use in a client of monitor type.

Synopsis

```
int rhp_mcinit(void);
```

Return values

GRR_OK Initialisation completed.

Discussion

This function should be called in a client of monitor type before any other function described in this manual. It assumes that rpm and pub have been initialised before. It has the same effect as

```
rhp_init(RHP_FLG_RESET | RHP_FLG_BIGNT).
```

rhp_mcbind rhp_mcbind

Set up handler for incoming *rhp* messages in a client of monitor type.

Synopsis

```
int rhp_mcbind(void);
```

Return values

GRR_OK	Binding performed.

Discussion

This function call binds routines in a client process of monitor type which are called for handling of incoming object data.

Subscribe to an object.

Synopsis

```
int rhp_subscribe(char *server,int ID);
```

Parameters

server Name of server or group.

ID Object identifier.

Return values

GRR_OK Successful return.

GRR_INVALID **ID** is unknown.

rpm error code Unable to locate server or error when using **rpm_request**.

pub error code Error using **pub_subscribe**.

Discussion

With this function a client process subscribes to an object. The histogram or ntuple must exist in the local context. On success, all publications will be received by the client process. The data will be added to the local instance of the object. A possible user routine set with **rhp_set_routine** is called afterwards.

The functions adds an entry to an internal list, which helps *rhp* to keep track of subscriptions. The list is used for terminating possible subscriptions when either **rhp_halt** or **rhp_unsubscribe** is called (see there).

Subscribe to several objects with a single message.

Synopsis

```
int rhp_subscribe_ids(char *server,int* idvec,int veclen);
```

Parameters

server Name of server or group.

idvec Vector with object identifiers.

idlen Number of elements in **idvec**.

Return values

GRR_OK Successful return.

rpm error code Unable to locate server or error when using **rpm_request**.

pub error code Error using **pub_subscribe_ids**.

Discussion

With this function a client process tries to subscribe to several objects with a single message. The function returns GRR_OK if at least one subscription could me made. The publications will then be received by the client process. The data will be added to the local instance of the objects. A possible user routine set with **rhp_set_routine** is called afterwards.

In contrast to **rhp_subscribe**, the function does not keep track of subscriptions. A call to **rhp_unsubscribe_ids** with the same arguments should be made before terminating client programs.

Stop subscription to an object.

Synopsis

```
int rhp_unsubscribe(char *server,int ID);
```

Parameters

server	Name of server or group.
ID	Object identifier.

Return values

GRR_OK	Successful return.
GRR_INVALID	**ID** is not known.
rpm error code	Could not locate server.
pub error code	Error using **pub_unsubscribe**.

Discussion

End the subscription to an object. The same arguments as in the former call to **rhp_subscribe** should be passed.

The passed values for **server** and **ID** can have the special values NULL and 0, respectively. NULL refers to all possible servers, 0 to all objects. A call `rhp_unsubscribe(NULL,42)` will end the subscription to the object with ID 42 in all possible servers. After `rhp_unsubscribe(NULL,0)` all subscriptions made by the process using **rhp_subscribe** are deleted. This is done explicitly in **rhp_halt**.

rhp_unsubscribe_ids rhp_unsubscribe_ids

Stop subscription to several objects.

Synopsis

```
int rhp_unsubscribe_ids(char *server,int* idvec,int veclen);
```

Parameters

server	Name of server or group.
idvec	Vector with object identifiers.
veclen	Number of elements in **idvec**.

Return values

GRR_OK	Successful return.
rpm error code	Could not locate server.
pub error code	Error using **pub_unsubscribe_ids**.

Discussion

End the subscription to the objects in the vector **idvec** with **veclen** elements. The same arguments as in a former call to **rhp_subscribe_ids** should be passed. The function returns GRR_OK if the subscription to at least on identifier in the vector was successfully terminated.

In contrast to **rhp_unsubscribe**, neither the passed value of **server** or **ID** may be NULL and 0, respectively. See the discussion there. The function has to be called to complement **rhp_subscribe_ids**.

Initialise the *rhp* package for use in a client of gatherer type.

Synopsis

```
int rhp_gcinit(void);
```

Return values

GRR_OK Initialisation completed.

Discussion

This function should be called in a client of gatherer type before any other function described in this manual. It assumes that rpm has been initialised before. It is the equivalent to

```
rhp_init(RHP_FLG_RESET | RHP_FLG_BIGNT | RHP_FLG_ACCUMULATE).
```

rhp_gcbind rhp_gcbind

Set up handler for incoming *rhp* messages in a client of gatherer type.

Synopsis

```
int rhp_gcbind(void);
```

Return values

GRR_OK	Binding performed.

Discussion

This function call binds routines in a client process of gatherer type which will handle incoming object data.

rhp_set_group rhp_set_group

Look up server processes in a name server.

Synopsis

```
char* rhp_set_group(char *domain,char *wildcard,int *nServers);
```

Discussion

This function contacts the nameserver denoted by **domain** and tries to obtain a list of all processes matching the pattern **wildcard**. On success, a non zero pointer is returned which can be passed to every *rhp* function which takes the name of a server as the first argument (e.g. **rhp_list**, **rhp_req**, **rhp_req2**). The number of server is returned through **nServers**. In case of failure NULL is returned and ***nServers** equals zero.

At present, *rhp* supports working with three groups of servers at a time. So, a fourth call to **rhp_set_group** with distinct parameters will fail. One can use **rhp_rem_group** to remove a server group (see discussion there). Passing the same arguments as in a previous (successful) call will provide you with an up-to-date process list replacing the old list. There is no need to call rhp_rem_group before.

rhp_rem_group rhp_rem_group

Delete a process list.

Synopsis

```
int rhp_rem_group(server *domain);
```

Parameters

server	Name of server or group.

Return values

GRR_OK	Successful return.
GRR_INVALID	**server** denotes no valid group.

Discussion

This function will delete the server group denoted by **server**. The argument passed should be the return value of **rhp_set_group**. If **server** is NULL all existing process lists are deleted. This is done automatically when calling **rhp_halt**.

rhp_activate rhp_activate

Activate a group of objects.

Synopsis

```
int rhp_activate(char *server,int ID);
```

Parameters

server	Name of server or group.
ID	Upper 16 bit of a group of objects.

Return values

GRR_OK	Successful return.
rpm error code	Could not locate server or error using **rpm_send**.

Discussion

This function will set the active flag of objects in a server process which have the same upper 16 bit as in **ID** to on. On success calls to **rhp_active** will return 1 in the server process.

Deactivate a group of objects.

Synopsis

```
int rhp_deactivate(char *server,int ID);
```

Parameters

server	Name of server or group.
ID	Upper 16 bit of a group of objects.

Return values

GRR_OK	Successful return.
rpm error code	Could not locate server or error using **rpm_send**.

Discussion

This function will set the active flag of objects in a server process which have the same upper 16 bit as in **ID** to off. On success calls to **rhp_active** will return 0 in the server process.

rhp_req rhp_req

Request an object's data from a server.

Synopsis

```
int rhp_req(char *server,int ID);
```

Parameters

server Name of server or group.

ID Object identifier.

Discussion

This function has the same effect as `rhp_req(server,ID,0,0,NULL)`.
Please turn over.

Request an object from a server or a group of servers.

Synopsis

```
int rhp_req2(char *server,int ID,int ask,int retry,int *nreceived);
```

Parameters

server	Name of server or group.
ID	Object identifier.
ask	Number of data sources to ask at a time.
retry	Number of retries for sources which did not sent data.
nreceived	Number of sources which sent data.

Return values

GRR_OK	Successful return.
GRR_INVALID	**ID** not known.
GRR_CONTINUE	Function was interrupted.
rpm error code	Could not locate server or error using **rpm_send/flood**.

Discussion

This function is a more clever version of **rhp_req**, to which it is identical in case one requests data from a single server. For groups of servers, set with rhp_set_group, there are two more parameters to steer the data transmission. While **rhp_req** prompts all servers to send data at the same time and then enters a receive loop, this function will only ask **ask** sources at a time. If **ask** is lower than 1 or larger than the number of servers in the group **rhp_req2** will fall back to ask all data sources at the same time. If the **retry**-parameter is greater than 0 **rhp_req2** will find out which servers failed to deliver data and will re-initiate the sending **retry** times. The number of sources from which data were received is returned through **found**. The **found** argument can be set to NULL if the returned information is not required.

What values to choose for **ask** and **retry** depends on your application. When a long transmission time poses no problem and you want to keep the peak network bandwidth consumed as low as possible you should set **ask** to 1. Then only one source will use **rpm_send/flood** at a time to deliver data to your client.

A return type of GRR_CONTINUE indicates that **rhp_req2** was interrupted by calling **rhp_break**. Please, see the discussion there.

rhp_break

Signal **rhp_req2** to stop immediately.

Synopsis

```
int rhp_break(void);
```

Return values

GRR_OK	Successful return.

Discussion

There might be situations where you want to exit **rhp_req2** immediately when you receive a certain message (for instance from the state machine). The function **rhp_req2** works such that it (a) initiates the sending of a certain object and (b) enters a loop where it handles all arriving messages using **rpm_dispatch**. This loop is left when either the right number of messages with object data arrived or **rhp_break** was called in any of the handler functions.

Reset an object in a server process remotely.

Synopsis

```
int rhp_rreset(char *server,int ID);
```

Parameters

server	Name of server or group.
ID	Object identifier.

Return values

GRR_OK	Successful return.
GRR_INVALID	No object with this ID in remote process.
GRR_ACCESS	Permission to reset denied.
rpm error code	Could not locate server or error using **rpm_request**.

Discussion

A client can use this routine to reset an object in a server process remotely. The function uses **rpm_request.** The permission to reset is denied if RHP_FLG_RRESET was not passed to **rhp_init** in the server process.

Reset objects in a server process remotely.

Synopsis

```
int rhp_rreseta(char *server,int *IDarray,int nID);
```

Parameters

server	Name of server or group.
IDarray	Array with object identifiers.
nID	Number of array elements.

Return values

GRR_OK	Successful return.
GRR_INVALID	None of the passed IDs is known in remote process.
GRR_ACCESS	Permission to reset denied.
rpm error code	Could not locate server or error using **rpm_request**.

Discussion

This function has the same functionality like calling **rhp_rreset** for each single ID in the array but is much faster. Only one message containing all the IDs is sent. Since only 360 IDs fit into one standard rpm message more than one message is sent if **nID** exceeds 360.

5 Miscellaneous Functions

These routines will convert histograms and ntuples booked by *rhp* to ROOT. They have C^{++} linkage. The ROOT libraries have to be linked in addition. (**r2r_** is short for **rhp2root_**.)

- **r2r_book_TH1F:** book 1d-ROOT-histogram, 1 float/bin
- **r2r_book_TH1D:** book 1d-ROOT-histogram, 1 double/bin
- **r2r_book_TH2F:** book 2d-ROOT-histogram, 1 float/bin
- **r2r_book_TH2D:** book 2d-ROOT-histogram, 1 double/bin
- **r2r_book:** book ROOT ntuple
- **r2r_fill:** fill ROOT ntuple or histogram

These routines provide a way of converting *rhp* histograms to HBOOK histograms. You have to add the HBOOK libraries to your linker call.

- **rhp_2hbook:** convert a *rhp* histogram or ntuple to HBOOK
- **rhp_2file:** write all HBOOK ntuples/histograms to a file

The following functions can be used to store/retrieve ntuples and histograms in/from a MIZZI data base. The MIZZI library must be linked to your program.

- **rhp_BookTable:** book a table for an *rhp* histogram or ntuple.
- **rhp_PutTableFields:** prepare data storage
- **rhp_GetTableFields:** read histogram/ntuple data from data base

r2r_book_TH1X

Convert a 1d-histogram to ROOT.

Synopsis

```
TH1F* r2r_book_TH1F(int ID);
TH1D* r2r_book_TH1D(int ID);
```

Parameters

ID 1d-histogram ID.

Discussion

The two functions above take the ID of an 1d-*rhp* histogram and book a ROOT histogram with the same parameters. In case of an invalid **ID** or a booking error NULL is returned.

This function performs only the booking. No data is transferred between *rhp* and ROOT. The user is responsible for deleting the returned pointer.

r2r_book_TH2X r2r_book_TH2X

Convert a 2d-histogram to ROOT.

Synopsis

```
TH2F* r2r_book_TH2F(int ID);
TH2D* r2r_book_TH2D(int ID);
```

Parameters

ID 2d-histogram ID.

Discussion

The two functions above take the ID of a 2d-*rhp* histogram and book a ROOT histogram with the same parameters. In case of an invalid **ID** or a booking error NULL is returned.

This function performs only the booking. No data is transferred between *rhp* and ROOT. The user is responsible for deleting the returned pointer.

Convert an ntuple to ROOT.

Synopsis

```
TNtuple* r2r_book(int ID);
```

Parameters

ID Ntuple ID.

Discussion

The function above takes the ID of an *rhp* ntuple and books a ROOT ntuple with the same parameters. In case of an invalid **ID** or a booking error NULL is returned. On success, one can then use **r2r_fill** to transfer data between *rhp* and ROOT. The user is responsible for deleting the returned pointer.

r2r_fill r2r_fill

Copy data from an *rhp* histogram to a ROOT histogram.

Synopsis

```
int r2r_fill(TH1 *histo,int ID);
int r2r_fill(TH2F *histo,int ID);
int r2r_fill(TH2D *histo,int ID);
```

Parameters

histo Pointer to ROOT-histogram.

ID ID of histogram.

Return values

GRR_OK Successful return.

GRR_INVALID No matching histogram with this ID known.

Discussion

This call looks up the *rhp* histogram denoted by **ID** and fills its data into the ROOT histogram. Note that TH1 is a super class of TH1F and TH1D.

r2r_fill r2r_fill

Copy data from a *rhp* ntuple to a ROOT ntuple.

Synopsis

```
int r2r_fill(TNtuple *nt,int ID);
```

Parameters

nt Pointer to ntuple.

ID Ntuple ID.

Return values

GRR_OK Successful return.

GRR_INVALID No ntuple with this ID known.

Discussion

This call looks up the *rhp* ntuple denoted by **ID** and fills its data into the passed ROOT ntuple.

Convert a histogram or ntuple to HBOOK.

Synopsis

```
int rhp_2hbook(int ID,int unused,char *topdir);
```

Parameters

ID	ID of histograms/ntuples.
unused	An obsolete parameter.
topdir	Pathname for ntuples.

Return values

GRR_OK	Successful return.
GRR_INVALID	No object with this ID known.

Discussion

This function will convert a single *rhp* histogram or ntuple to HBOOK unless **ID** is 0. Then all booked objects are converted.

Converting means two things, namely booking and copying over the entries. *Rhp* will remember whether a histogram/ntuple was booked in HBOOK before. So there will be no double-booking when calling **rhp_2hbook** twice. All entries present in the *rhp* histogram or ntuple will be filled into the matching HBOOK histogram or ntuple. If one does not reset the objects with **rhp_reset** all data will be filled in again at a second call to **rhp_2hbook**.

The second parameter **unused** was used to initialise HBOOK by calling `hlimit()`. This led to some confusion. From now on, the user is responsible for initialising HBOOK properly and **unused** is ignored and should be set to 0.

The last parameter **topdir** has a meaning for ntuples exclusively. It's safe to set it to NULL or "/" if it is not needed. If set it specifies the booking path in a file that was opened before with **hropen** (See HBOOK documentation).

One additional complication exists. HBOOK seems to use 32 bit wide IDs for histograms, too. As for ntuples, only IDs lower than 10000 are allowed. Therefore **rhp_2hbook** will use only the lower 16 bit of the ID when converting histograms and ntuples.

Write all HBOOK histograms/ntuples to a file.

Synopsis

```
int rhp_2file(char* filename);
```

Parameters

filename Desired filename.

Return values

GRR_OK Successful return.

Discussion

This call will put all HBOOK histograms and ntuples into a file. It should not be used in an ARTE environment, since ARTE opens its own HBOOK file.

Book a MIZZI table for a histogram or ntuple.

Synopsis

```
int rhp_BookTable(int ID,Tables x,char* tablename);
```

Parameters

ID	Histogram or ntuple ID.
Tables	Data base id.
name	Table name.

Return values

0	An error occurred.
>**0**	MIZZI table entry number.

Discussion

When **ID** is a valid ID and denotes an existing histogram or ntuple this function will book an appropriate MIZZI table. The name of the title will be **name**. There is a convention that **name** has to end with `".h1"`, `".h2"` or `".nt"` for 1d/2d-histograms and ntuples, respectively. *Rhp* will append the right extension automatically to ensure that this convention is met.

If **ID** is not a valid ID (e.g. because you just know the table name and want to read from the data base) the function will look at the extension of **name** to book the appropriate type of table. 0 is returned when the table name is invalid for *rhp* or a booking error occurs.

rhp_PutTableFields rhp_PutTableFields

Put fields in a data base table.

Synopsis

```
int rhp_PutTableFields(int ID,Tables x);
```

Parameters

ID Histogram or ntuple ID.

Tables Data base id.

Return values

0 **ID** denotes no histogram or ntuple.

1 Successful return.

Discussion

This function will call `SkipTableField(x,1)` followed by several calls to `PutTableField()` to put the histogram or ntuple data into a data base table. `StoreTable()` should be executed afterwards. When writing histogram data, zeros at the end of the histogram vector are *not* stored in the data base. This simple zero-suppression should reduce the size of the data base files.

Read histogram/ntuple data.

Synopsis

```
int rhp_GetTableFields(int ID,Tables x,int xe);
```

Parameters

ID	Histogram or ntuple ID.
Tables	Data base id.
xe	Entry number of table.

Return values

0	An error occurred.
>**0**	ID of the histogram or ntuple.

Discussion

This function transfers the histogram or ntuple data from the data base to a *rhp* histogram or ntuple. The object is created or overwritten when necessary. **xe** is the entry number of the MIZZI-table returned by **rhp_BookTable**.

Once a histogram or ntuple is stored in the data base you can decide whether you want to access it with a newly invented ID or with the ID which was used to store it in the data base.

Pass 0 in **ID** to stay with the original ID, which is then the return value of the function. If you pass any non-zero number you can use it later to refer to this histogram or ntuple (when the function returns non-zero). This provides a way of having different versions of the same histogram or ntuple in memory simultaneously.

0 is returned in case **xe** denotes no valid histogram or ntuple table.