

Run Control Parameters at DØ

Marc Paterno
CD/CPD/APS, FNAL

October 9, 2002

Abstract

This document is a short guide to the use of the RCP system for managing run control parameters (RCPs) at DØ.

Please note Appendix C, “Common Tasks”, found at the end of this document, contains instructions for many of the simple tasks one may wish to perform.

1 RCPs and Packages

At DØ, most run control parameter sets (RCPs) are associated with a specific software (CVS) package. Parameter sets that are to be used for official reconstruction must be associated with a CVS package; the release managers use this as the method for controlling what gets into the official RCP database.

In order to allow two packages to have parameter sets with the same name without conflict, parameter sets are named by both the name of their owning package and their own name. Such flexibility may be useful for software development and data analysis tasks

For purposes other than official reconstruction, it is possible to associate a parameter set with a “package” that is not an actual CVS package. Such parameter sets can not at this time be entered into the “official” RCP database (which contains all the RCPs used by the reconstruction program), but they may be entered into a “personal” RCP database. In the future, DØ may establish “physics group” RCP databases, to provide a level of management more strict than the “personal” database, but possibly less strict than the “official” database.

Framework RCPs are handled in a special manner. These RCPs are never read from a database; you must always supply the framework executable with a filename (either absolute or relative from the current working directory) for the RCP file to be read. This is the only case in which an RCP object cannot be extracted from the database (without the need for a local RCP file). It is also the only case in which an RCP script will be picked up from a directory other than the package directories, as described in

Section 5.

The use of *untracked parameters* for some elements in framework RCPs may be useful; this feature was first introduced in release `rcp v00-04-00`. See the document

A Guide for the Use of Untracked Parameters in RCPs at DØ for a description of the official DØ policy on the use of untracked parameters.

2 Use of RCP objects

RCP objects (instances of the class `edm::RCP`) are collections of name-value pairs, where the *name* is a C++ string and the *value* is one of the types listed in Table 1.

<code>bool</code>	
<code>int</code>	<code>std::vector<int></code>
<code>float</code>	<code>std::vector<float></code>
<code>double</code>	<code>std::vector<double></code>
<code>std::string</code>	<code>std::vector<std::string></code>
<code>RCP</code>	<code>std::vector<RCP></code>

Table 1: Data types supported by the RCP system.

Note that `std::vector<bool>` is *not* one of the allowed types; use of this type in any C++ code is discouraged.

2.1 Normal parameters

RCP objects provide access to the parameters they contain by means of various “get” functions. They provide no way to modify these values; an RCP object is essentially a *readonly* object. When asked for a contained parameter, that parameter is returned *by value*, so that the function call can be used as an expression – for example, in the *colon-initialization list* of a constructor for a class. Because of this syntax, an RCP object signals that it could not return a parameter of a given name by throwing an exception; specifically, an instance of the class `XRCPNotFound`. An example of the use of an RCP object is given in

Figure 1. This example shows how to obtain a parameter of type `double` which has the name “ETminimum”, and how to handle the exception which is thrown if no such parameter is found.

RCP objects are obtained by one of two methods: they may be extracted from another RCP object, or they may be obtained from an `RCPManager` (an instance of class `RCPManager`). Each RCP object is labeled with an identifier (an instance of the class `RCPID`). The `RCPID` consists of two parts: a “database id” and a “sequence number”. The “database ID” indicates in what database this parameter set lives; the “sequence number” is an identifier which is unique to this set of parameters, within that database. Since an `RCPID`

```

void someFunction(const edm::RCP& r)
{
    try
    {
        double thresh = r.getString("ETminimum");
        // Code here would use the value of thresh...
    }
    catch (const edm::XRCPNotFound& x)
    {
        // If there is a missing parameter, we catch the
        // appropriate exception here. (Other exceptions are
        // allowed to propagate past this catch block).
        // Handle the error here.
    }
} // void someFunction(const edm::RCP& r)

```

Figure 1: Example of use of the class *RCP*.

is unique only within the database that issues it, it is important that all official reconstruction use the same database information. The class *FileSystemDB* has been provided to meet this need; it is a lightweight “database” designed for the purpose of handling RCP objects, and issuing RCPIDs. The files that make up the body of data managed by the class *FileSystemDB* are ASCII files, so that they are platform independent. It is important not to confuse these files, which are not intended to be directly modified by users, with the RCP scripts discussed in Section 5.

Note that the uniqueness refers to the set of parameters (the name-value pairs), not to any name associated with the RCP object – most importantly, not to the name of any script that was read to produce the RCP object. Because the uniqueness of the RCPID refers to the parameters themselves, the EDM makes use of these RCPIDs to help identify the genesis of “chunks” in the Event.

2.2 Untracked Parameters

New to version 0.4 of the RCP system is the concept of *untracked* parameters. Untracked parameters are parameters that can be read from an RCP script, but are never recorded in an RCP database. The only data types allowed for untracked parameters are `bool`, `int`, and `std::string`. Untracked parameters can be used to control those parts of a program (or framework package) that do not affect the results of reconstruction. For example, one might use an `untracked int` to control the level of verbosity of some standard output, or an `untracked string` to provide the name of an output ntuple file. DØ policy is that these parameters may *never* be used to configure parameters of a program (or framework module) that affect the results of reconstruction.

The syntax for inclusion of an untracked parameter in a script is:

```
untracked <type> <name> = <value>
```

where `<type>` is one of `bool`, `int`, or `std::string`, `<name>` is a parameter name, and `<value>` is a string that can be converted to a parameter by the normal rules for a parameter of that type.

Because untracked parameters are never recorded in any RCP database, an RCP will only contain an untracked parameter if the parameter set stored in that RCP was created by reading an RCP script. RCPs that are retrieved by querying and *RCPManager* for the RCP associated with a given *RCPID* will *never* contain any untracked parameters. Since nested *RCPs* are effectively always retrieved by this method, an *RCP* object obtained from within another *RCP* object will also never carry untracked parameters. See the document *A Guide for the Use of Untracked Parameters in RCPs at DØ* for a more detailed explanation of this issue, and for the official DØ policy regarding the use of untracked parameters.

Note that this means the reconstruction run on the production farm will never see untracked parameters; a good synonym for “untracked parameter” might be “parameter for which reconstruction always uses the default”.

Because RCPs retrieved from an RCP database never carry untracked parameters, it is necessary for the syntax for access to untracked parameters to be different from that for normal parameters. To obtain an untracked parameter from an RCP object, use the member function `getUntrackedBool()`, `getUntrackedInt`, or `getUntrackedString`. Each of these functions takes two arguments: the first is a `std::string`, giving the name of the parameter of interest; the second is a default value, to be returned if the RCP does not contain this untracked parameter.¹ One beneficial side effect of having untracked parameters accessed via a different signature is that it allows code inspections to localize the use of untracked parameters more easily. Algorithm group leaders are encouraged to monitor the use of untracked parameters, to assure that those things which need to be recorded in the RCP database are being recorded.

Figure 2 shows an example of code that uses untracked parameters. Note the absence of `try` and `catch` blocks in this example – since untracked parameters can return a default value, there is no need to indicate an missing parameter by throwing an exception.

3 Use of *RCPManager*

The class *RCPManager* is a singleton.² One obtains access to the sole instance of *RCPManager* by calling the static member function `instance()`, which returns a pointer to the *RCPManager*. One can then obtain RCP objects from

¹Note that the returning of a default is one of the reasons that untracked parameters of other types are not provided. Using default values, available only by inspecting the code, would defeat the aim of having configuration parameters available to later reconstruction and analysis programs.

²See Gamma *et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software*.

```

void someFunction(const edm::RCP& r)
{
    int verbosity = r.getUntrackedInt("verbosity", 0);
    bool makeNtuple = r.getUntrackedBool("makeNtuple", false);
    if (verbosity > 0)
    {
        // Yap about our current status here...
    }
    if (makeNtuple)
    {
        // do stuff to make an ntuple here ...
    }
} // void someFunction(const edm::RCP& r)

```

Figure 2: Example of use of untracked parameters.

the manager through the functions `extract(const std::string& pkgName, const std::string& objName)` and `extract(const edm::RCPID& id)`.

The function `extract(const edm::RCPID& id)` is generally useful when one has obtained an *RCPID* from an object in an event. It is the other function that is usually used to obtain a parameter set. The function `extract(const std::string& pkgName, const std::string& objName)` will extract from the (RCP) database the parameter set named “objName” associated with the (CVS) package “pkgName”. This is the same nomenclature as is used in the format of the RCP scripts themselves.

An example of the use of the class *RCPManager* is shown in Figure 3. This example shows the code necessary to access the singleton instance of *RCPManager*, and that necessary to get an RCP object from that manager.

4 RCP Databases

The *RCPManager* makes use of one or more RCP databases, from which it retrieves parameter sets and to which it may add parameter sets. Each *RCPManager* can make use of any number of *readonly* databases, and zero or one *writable* database. The list of databases to be used are specified by the environment variable `RCP_DATABASE_PATH`.

The environment variable `RCP_DATABASE_PATH` is to be defined to be a colon-separated list of database specifications. Each database specification comes in two parts: the first is the name of the database, and the second is the name of the class to be used to communicate with the database. The databases named in this list are those that will be searched, in the given order, in order to find a parameter set. Under normal conditions, the last database named in this list is used as the *writable* database, and all others (there may be zero others) are used as *readonly* databases. No more than one database may be used in

```

void someFunction()
{
    edm::RCPManager* pman = edm::RCPManager::instance();
    // No need to check the return value; if the RCPManager
    // instance can't be created, an exception will be thrown.
    // The try block below is to handle exceptions thrown if we
    // can't get the parameter sets we request.
    try
    {
        // Get the parameter set named 'golden', from the package
        // 'EMID'
        edm::RCP r1 = pman->extract("EMID", "golden");
        // ... now do something useful with the parameters ...
    }
    catch (const edm::XRCP& x)
    {
        // Handle the error condition here.
        // ...
    }
}

```

Figure 3: Use of the class *RCPManager*.

writable mode in a single program. To indicate that *no* writable databases are to be used (all databases are to be used in readonly mode), one must define `RCP_DATABASE_PATH` with a final forward slash (/). For example, with the definition `RCP_DATABASE_PATH="official/FileSystemDB:personal/FileSystemDB"` two databases are used. The first, named "official", will be used in readonly mode. The second, named "personal", will be used in writable mode. If, instead, one had `RCP_DATABASE_PATH="official/FileSystemDB:personal/FileSystemDB/"`

(note the trailing forward slash) then both databases would be used in readonly mode.

As of release t00.91.00 of the DØ offline software, the allowable database names are "official" (wherein all the parameter sets associated with released code will reside), and "personal" (which indicates a database that will be created for the personal use of the user). In release v00-10-00 of the RCP system, the available database classes are *FileSystemDB* and *RCPDatabaseInMemory*.

4.1 RCPDatabaseInMemory

Use of *RCPDatabaseInMemory* is appropriate for testing purposes only. This class provides no permanence for RCPIDs past the end of the running program – as the class name says, the "database" kept is only in memory, and is thus deleted at the end of the execution of the program.

4.2 FileSystemDB

The class *FileSystemDB* is appropriate for general use. In order to use this class, one must define two environment variables:

`RCP_FILESYSTEMDB_READONLY_DIR` and `RCP_FILESYSTEMDB_WRITABLE_DIR`.

`RCP_FILESYSTEMDB_READONLY_DIR` is the name of the directory in which the database files for the *readonly* databases used by the RCPManager are to be found. This environment variable is generally set to an appropriate value by the command *setup DORunII*.

`RCP_FILESYSTEMDB_WRITABLE_DIR` is the name of the directory in which the database files for the *writable* database used by the RCPManager is to be found. This environment variable is not set by the command *setup DORunII*; it should be set by the user to an appropriate value. A reasonable choice is to define this as the default login directory of the user.

Under no circumstances should non-experts alter any of the files that make up a *FileSystemDB* database. These files are automatically managed by the RCP system itself, and alteration of any one of the files may lead to a corruption of the database.

5 Reading an RCP Script

The RCP system has been designed to provide automatic storage of parameter sets. In a controlled environment such as the reconstruction farm, parameter sets will only be obtained from a downloaded RCP database. In the development and analysis environments, however, such control is too strict. Thus the RCP system has the ability to read parameter sets from a secondary source: ASCII files called RCP scripts. The formal grammar of the RCP script is given (as a YACC grammar) in the file *rcp_parse.y*, which is part of the **RCP** source code. A more concise (though less precise) summary of the syntax is given in Appendix A.

In order for the RCP system to find a script, the user's environment must be setup up correctly. The RCP system uses an environment variable `RCP_SCRIPT_BASE` to find scripts. `RCP_SCRIPT_BASE` is a colon-separated list of directories³ used as the base directories for starting a search. If this environment variable is not defined, then the search begins in the current working directory. Under a base directory, the system will look for a directory structure of the same form as that used by the SRT build environment. Specifically, the required directory structure under the base directory is: (1) a subdirectory with the name of the package with which the RCP is associated, under which is (2) a subdirectory named "rcp", in which is to be found the RCP script itself. Thus, if `RCP_SCRIPT_BASE` is defined to be `"/home/me/test:/home/you/test"`, then the search for a script for an RCP named `<io_packages ReadEvents>` will look for a file `"/home/me/test/io_packages/rcp/ReadEvents.rcp"`, and if that is not found, it will search for `"/home/you/test/io_packages/rcp/ReadEvents.rcp"`.

³In versions of **RCP** prior to v00-09-00, `RCP_SCRIPT_BASE` was a single directory, not a list.

If a script is found by this search, it will be read, and the parameter set it defines will be searched for in the readonly databases. If a matching parameter set is found, it will be returned to the user. If no matching parameter set is found, then the system will look in the writable database for a matching parameter set. If none is found in the writable database, then the system will attempt to add this parameter set to the writable database. If this fails, then an exception will be thrown by the system.

If no script by this name is found, then the databases are searched (in the order given above) for a parameter set known by this name. The first one found is returned. If none is found, then an exception will be thrown by the system.

A RCP Script Grammar

The exact grammar of RCP scripts is given in the file *rcp_parse.y*, which is part of the **RCP** source code. This appendix gives a more concise description.

Comments are denoted by two forward slashes, following the style of C++ comments. Comments can come at the beginning of a line, or at the end of a line, as in C++ code.

Each entry in an RCP script defines a single parameter. The types of these parameters are the following: `bool`, `int`, `float`, `double`, `std::string`, `RCP`, and the vectors: `std::vector<int>`, `std::vector<float>`, `std::vector<double>`, `std::vector<std::string>`, and `std::vector<RCP>`⁴. Each entry specifies the name to be assigned to that parameter, and the value to be assigned to that parameter.

The form of the line is:

type-tag name = assignment-statement

The *type-tag* must be one of the keywords in Table 2. The type of the parameter created is determined by the combination of the keyword and by whether the assignment statement is a *scalar-assignment* or an *aggregate-assignment*. An aggregate-assignment is indicated by containment within parentheses; any other assignment is a scalar assignment.

<i>type-tag</i>	parameter types
bool	<code>bool</code>
int	<code>int</code> and <code>std::vector<int></code>
float	<code>float</code> and <code>std::vector<float></code>
double	<code>double</code> and <code>std::vector<double></code>
string	<code>std::string</code> and <code>std::vector<std::string></code>
RCP	<code>RCP</code> and <code>std::vector<RCP></code>
untracked bool	<code>bool</code>
untracked int	<code>int</code>
untracked string	<code>std::string</code>

Table 2: The keywords recognized in an RCP script, and the associated parameter types.

Several of these types require special syntax. These types are listed below. The numeric types `int`, `float`, and `double` all take assignment by a number of the correct type.

In addition, starting with **RCP** version v00-10-00, and for `doubles` only, several special values are recognized:

- the character strings “inf” and “+inf” (any capitalization, no space between the “+” and the “inf”) is translated to IEEE 754 positive infinity;

⁴Note the `std::vector<bool>` is missing from the list. This is because the class `std::vector<bool>` is not handled correctly by the KAI C++ compiler (v3.3f1), and also because this class has come into question by the C++ Standards committee.

- the character string “-inf” (any capitalization, no space between the “-” and the “inf”) is translated to IEEE 754 negative infinity;
- the character string “nan” (any capitalization) is translated to IEE 754 quiet NaN (“not a number”).

Please note while RCP will read these values, it can not guarantee that the C++ implementation correctly handles all the behavior required by IEEE 754 for infinities and NaNs. Care must be taken to test code relying on infinities and NaNs even more thoroughly than other code is tested.

The use of “inf” and “-inf” may be interesting for parameters such as floating-point thresholds, which can effectively be “turned off” by setting them to the appropriate infinity. The use of “nan” is more tricky, and probably should be avoided by all except for numeric programming experts.

A.1 bool

The value of a bool can be set with any of the following values: for false, use “false”, “FALSE”, “False”, “F”, “f”, “off”, “OFF”, “no”, “No” or “NO”. For true, use “true”, “TRUE”, “True”, “T”, “t”, “on”, “ON”, “yes”, “Yes”, or “YES”.

A.2 string

A string must be delimited by double quotes. Strings may include printable characters and white space, including newlines. To include a double quote mark itself, it must be “escaped” by a backslash: `"This is a \"quoted\" string"`.

A.3 RCP

An RCP is included by giving the name of the package with which the RCP is associated and the name of the RCP object itself, within angle brackets: `<PackageName ObjectName>`.

A.4 Untracked Parameters

Only the types `bool`, `int`, and `std::string` can be preceded by the modifier `untracked`. Parameters marked as untracked (by use of the script keyword `untracked`) will not be entered into any RCP database, and are not considered as part of the parameter set for assignment of *RCPIDs*, nor for equality comparisons. See section 2.2 for more details.

A.5 Examples

This section contains several examples of legal syntax for RCP scripts. Note that the length of a vector created by defining a vector parameter is determined by the number of entries one places in the parameter definition. There is no facility for defining a larger vector, and filling only part of it.

```
// This a comment at the beginning of a line
# This is also a comment at the beginning of a line
bool myBool = true // or false, if you prefer
int myInt = 21 // Trailing comments are also legal
float myFloat = 8.9 # Trailing comments can also follow a sharp
double myDouble = 0.5 // doubles are distinguished from floats
string myString = "This is a \"string\" with a quote embedded"
string string2 = "This string has
a newline in the middle of it"
RCP myRCP = <SomePackage SomeObject>
// The following are examples of vectors
// vectors of bools are not supported
int myIntVector = (1 2 3) // This makes a vector with 3 entries
float myFloatVector = (1.2) // A vector may have one entry
double myDoubleVector = () // Or even no entries
string myStringVector = ("Howdy" "Hi" "Good to see you")
RCP myRCPVector= (<P1 Obj1> <P2 Obj2>)
```

Figure 4: An example of legal RCP script syntax

A.6 “Interesting Features” of the New Parser

One of the changes between the old RCP system (used up to t00.64.00) and the new RCP system (t00.65.00 and beyond) is the introduction of a new parser, which understands a richer set of constructs (such as vectors). Here is a short list of quirks in the parser used in t00.65.00, which I encountered while trying to parse the set of RCPs present in t00.62.00 (which was the most recently available test set, using during testing of the parser).

1. Trailing comments must contain whitespace between the end of the preceding value and the double-slash (or sharp) that introduces the comment.

Bad:
float x = 1.23// This will fail

Good:
float x = 1.23 // This is OK

This “feature” is caused by some difficulties with the regular expressions recognized by flex; it may be relaxed in a future release.

2. Comments can be introduced only with two forward slashes or a sharp sign (// or #). Backward slashes are not legal. This is by design; only one file in t00.62.00 tried to use backward slashes.

3. It isn't a good idea to have too many significant digits in a `float` constant. `std::numeric_limits<float>`, under KCC on both Linux and IRIX6, and also under MSVC++ 6.0, all report that the number of significant digits in a float is 6. If you need more, use a `double`.

This is a feature of the language; the (new) ability to use doubles in RCP scripts should help solve the problem.

4. The parser will choke if you try to include another RCP which neither exists in the database(s) you're reading nor can be created from a script that can be found under `RCP_SCRIPT_BASE`.

This is by design; such an RCP would be malformed, and is not allowed. Note that you should receive an error message via email from the DØ Release Managers if you have a malformed RCP in your package's `rcp` subdirectory; this is because such a malformed RCP can not be entered into the RCP database. Sometimes you may receive a complaint that is a result of a "cascade", when you have included an RCP that was malformed because an RCP that *it* included was malformed.

5. The parser sometimes chokes (well, only once really, and since I fixed some other things in the parser this hasn't happened again) on lines that contain nothing but a comment, if that comment doesn't start at the beginning of the line.

I think this "feature" doesn't exist any more, but if you have an example of an RCP file with such a comment that does kill the parser, please send me e-mail (paterno@fnal.gov) with the killer RCP file, so that I can track down the bug.

B Environment Variable Summary

This appendix contains a summary of the environment variables used by the RCP system.

B.1 The database path

The environment variable `RCP_DATABASE_PATH` is used to define which databases are to be searched for parameter sets. Note that this is a list of databases and class names, and *not* a list of directories, and this environment variable has nothing to do with the searching for RCP scripts (files). See Section 4 for a more complete description.

B.2 File System database

The environment variable `RCP_FILESYSTEMDB_READONLY_DIR` defines a search list of directories in which the class *FileSystemDB* will search for the database files to be read for the *readonly* databases. This variable is generally set by the command `d0setwa`. See Section 4.2 for a more complete description.

The environment variable `RCP_FILESYSTEMDB_WRITABLE_DIR` defines the directory in which the class *FileSystemDB* will search for the database files to be read and written to for the *writable* database. This variable is generally set by the command `d0setwa`. See Section 4.2 for a more complete description.

B.3 RCP Scripts

The environment variable `RCP_SCRIPT_BASE` defines a colon-separated list of directories, each of which is used as a base directory for the searching for RCP scripts (files). If this environment variable is not defined, then the search begins from the current working directory. See Section 5 for a more complete description.

B.4 Mapping Database names to Database IDs

The environment variable `RCP_DB_NAMES_FILE` is used to find a file that provides the mapping from database names to database IDs. This environment variable is defined by `d0setwa`, and should not be modified by the user under normal circumstances.

C Common Tasks

This section contains instructions (or suggestions) for how to perform some common tasks related to the management or use of RCPs.

C.1 How do I run component and integrated tests?

There are several solutions possible, each of which may be achieved by defining `RCP_SCRIPT_BASE` appropriately, and by copying RCP files if necessary.

Perhaps the simplest method is to define `RCP_SCRIPT_BASE` to be the base directory of your test release (known in SRT as `SRT_PRIVATE_CONTEXT`). If you put the scripts for all the RCPs you use in your tests in the “rcp” subdirectory of your package, they will then be found by the system when you run your tests. This allows you to modify the RCP scripts in place, and to have those modifications seen by your tests.

It is also recommended that you give names to these scripts that will make it clear to others that they are used for tests, and not for normal data processing. These RCPs will be entered into the “official” RCP database, and will thus be available for use by tests in other packages.

You should note that running such tests will populate your personal database with the modified RCPs. If you want to get rid of these after your testing session, and do not need to have a permanent record of the RCPIDs generated for these RCPs, it is safe to delete the entire database directory, and to then recreate it with the shell script `rcp_setup.db`.

C.2 How do I modify an RCP in a package I have checked out?

If you want to modify an RCP in a package that you have checked out, all you need to do is to modify the RCP script found in that package’s “rcp” directory. If you have `RCP_SCRIPT_BASE` defined to be the top-level directory of your test release (the same directory as the SRT environment variable `SRT_PRIVATE_CONTEXT`), the modified RCP script will be read, and the resulting RCP will be used in your program. Note that this new parameter set will be entered into your personal RCP database.

C.3 How do I modify an RCP in a package that I do not have checked out?

This is almost identical to the scenario in Section C.2. All you need to do is to create the appropriate directory structure (a subdirectory named for the package from which you want to modify the RCP, and a subdirectory below that named “rcp”). Into this directory structure, you copy the RCP script you want to modify, and modify as needed. If you have `RCP_SCRIPT_BASE` defined to be the top-level directory of your test release, the modified script will be read, and the resulting RCP will be used in your program.

Note that this new parameter set will be entered into your personal RCP database.

C.4 How do I run with only the RCPs in the official release?

The easiest method is to have `RCP_SCRIPT_BASE` point to a non-existent directory. Note that this is *not* the same as having `RCP_SCRIPT_BASE` undefined (see Section 5). If `RCP_SCRIPT_BASE` points to a non-existent directory, then the search for a script will never find one, and all the RCPs your program uses will come from a database (either the official database, or, if not found there, from your personal database).

C.5 How do I introduce a new RCP into my local test release?

If you need to invent a new RCP, which hasn't been used before, for a package you are working on, all you need to do is to put the correctly named script into the appropriately named directory. See Section C.2 for how to do so. This will result in the new parameter set being added to your personal database. If you need to introduce a new RCP to a package you are not working on, you merely need to make the directory structure (described in Section C.2) and put the new script there.

C.6 How do I introduce a new RCP into a release?

All RCPs in the “rcp” directory of your package will be added to the official RCP database during the release procedure. If you have put the RCP script into the “rcp” directory, and have requested a release of the package, the addition (or updating) of the RCP database is automatic.

C.7 How do I add a new RCP to my personal database?

Any time you modify (or add) an RCP script to your local test release, and that RCP script is read by the RCP system, the corresponding RCP is automatically entered into your personal RCP database. No special action is required.

C.8 What happens if I delete my personal RCP database?

If you delete your personal RCP database, you will need to run the script `rep_setup_db` to create an empty personal database before you can begin to put new RCPs into your now-empty personal database. Any RCPIDs issued by the old (deleted) database will not be meaningless; the new database will begin issuing IDs with number 1. Any data files you have written with the old RCPIDs will now contain misleading information, since the RCPIDs issued by your (deleted) personal database can no longer be resolved by that database –

or worse, they will be resolved incorrectly after you begin to populate your new database.

It is recommended to delete your personal database only when you are sure that none of the RCPIDs issued by that database need to be meaningful – for example, after the end of a long testing session.

If you want to retain meaning for the RCPIDs you use in your analysis, you should not delete your personal RCP database.

C.9 How do I set up the correct environment variables?

As of release t00.65.00 build 4, the command *setup D0RunII [version]* will set up reasonable defaults for most of required environment variables.⁵ This does not define the environment variable `RCP_SCRIPT_BASE` which may need to be reset whenever you switch the test release base directory in which you are working. To set this variable, use the command *d0setwa* (DØ set working area).

C.10 How are framework RCPs handled?

(This answer is a repeat of the information in Section 1.) In the first release of the system, framework RCPs are handled in a special manner. These RCPs are never read from a database; you must always supply the framework executable with a filename (either absolute or relative from the current working directory) for the RCP file to be read.

Note that this behavior is peculiar to framework RCPs. In all other cases, an RCP

C.11 Why doesn't my program pick up my modified RCP?

If one is modifying one or more parameters for an algorithm that come from an *RCP* object deeply nested under the framework *RCP* (either tracked and untracked parameters), one should watch out for them coming from the database rather than the local file. In order for parameters in a nested *RCP* to be controlled by the contents of a local script, *all* of the *RCPs* from the top-level framework *RCP* down to the *RCP* one is modifying must be present as local scripts.

⁵Of course, the value of *version* must be at least t00.65.00.