RPL2 Programmer's Guide

Project title: RPL2

Document title: RPL2 Programmer's Guide

Document identifier: EPCC-BG-PAP-RPL2-PG 0.2

Authorship: Patrick D. Surry

Document history:

Personnel	Date	Summary	Version
PDS	30 November 1993	Initial draft	0.1
PDS	24 January 1994	Draft for alpha-distribution	0.2

Approval List: *ISG Manager, Technical Reviewer, British Gas Project Manager*
Nicholas J Radcliffe, Felicity George, Jim Mallinson

Distribution List: *British Gas Project Team*
Ian Boyd, Mark Green

Contents

List of Figures

List of Tables

Known Problems with this Document

- Genome fields should be read via macros, rather than directly from the structures.
- Should pack, unpack functions be changed to take unsigned int?
- Discuss issues about handling NULL genomes in operators.

1 Overview

This document describes the programmer's interface to RPL2. RPL2 (Reproductive Plan Language) is an extensible interpreted language for experimenting with genetic algorithms and evolutionary computing. User's can make use of existing libraries of genetic operators and specialised representations as discussed in the RPL2 User Guide [6]. Advanced users may wish to code their own evaluation functions, genetic operators, or even an entirely new representation. This is supported in a flexible and modular way.

A step-by-step tutorial explaining how to create a new genetic represenstation with associated operators is presented in *section 2*.

The tutorial is followed by more detailed reference sections. The requirements for representation-independent operator libraries and new representations are given in *section 3.1* and *section 3.2*, followed by an exhaustive listing of all `rpp` macros, in *section 3.3*. The specialised data types used in RPL2 are presented in *section 3.4*, *section 3.5* and *section 3.6*. Finally, framework functions to support I/O, memory allocation, random numbers and error handling are described in *section 3.7*, *section 3.8*, *section 3.9* and *section 3.10* respectively.

Requirements for creating customised versions of RPL2, incorporating any desired combination of libraries, are summarised in *section 4*, and sources of further information are listed in *section 5*.

2 Library Tutorial

2.1 Introduction

This section presents a step-by-step tutorial describing how to build a new library of operators for RPL2. The example presented is of a variable sized set representation in which a genome consists of some set of unique integers in the range 0 to `usize`, where `usize` is the size of the universal set.

Although the example is of a representation library, a representation-independent operator library does not have many fundamental differences and most of what is said here would apply to it as well.

The reference sections presented later in this document may help to clarify details of the tutorial, which is not intended to be entirely rigorous.

2.2 Working from the Template Files

Template files are provided to make the job of building new representations and representation-independent operator libraries easier.

In our case, we want to define a new representation, so we make a new directory and copy the files from `$RPLHOME/repns/template`:

```
mkdir set
cd set
cp $RPLHOME/repns/template/* .
```

The files that are provided are: a header file, which contains macro definitions to be processed by the RPL2 pre-processor, `rpp`; a source file, which outlines the required and optional support routines; and a `Makefile` which is used to build the library with the necessary interface code to export the new operators to the RPL2 language.

2.3 Writing the Interface Header File

The interface header file, named `Repname.h` originally, should be renamed to correspond with the desired name of the representation in RPL2. Typically library names are capitalised in RPL2, so we might call our representation "Set":

```
mv Repname.h Set.h
```

The macro definitions in the header file should be modified as appropriate. In particular the `RPLREPNAME` macro should correspond with the name of the header file, and an appropriate comment should be inserted. Function names should be set to sensible values — typically all external functions in RPL2 libraries include a common three letter prefix ("Set" in this case) to avoid name-space pollution.

The set representation doesn't require any special initialisation or de-initialisation, so we delete the optional `RPLINIT/EXIT` macros. Similarly we'll ignore the checkpointing routines `RPLSAVE/LOADSTATE` for the moment. The representation will take a single parameter, indicating the size of the universal set. We don't want to define any demes, but do want a number of operators, which are defined as `RPLINST`s. The main body of the new header file might then look something like:

```
RPLREPNAME(Set, "A set representation library header");

RPLVER("$Id$");
```

```
RPLPACK  char*  SetPack (void *pGenome, int *pnMsgLen);
RPLUNPACK  void* SetUnpack (char *sMsg, int nMsgLen);
RPLFREE void SetFree (void *pGenome);

RPLSETPARAM BOOL SetSetUniverseSize (BOOL useNotDisamb,
    int  universeSize);

RPLINST (RandomGenome,
 "Create a random genome of the given size")
GENOME *SetRandomGenome (int size);

RPLINST (PrintGenome,
 "Print info about specified genome to file")
void SetPrintGenome (GENOME *g);

RPLINST (Mutate,
 "Mutate a set")
void SetMutate (GENOME *g, double mutrate);

RPLINST (rarw,
 "Random Assorting Recombination, integer weight")
GENOME *Setrarw (GENOME *g1, GENOME *g2, int weight);

RPLINST (rarf,
 "Random Assorting Recombination, real weight")
GENOME *Setrarf (GENOME *g1, GENOME *g2, double weight);

RPLINST (rrr,
 "Random Respectful Recombination")
GENOME *Setrrr (GENOME *g1, GENOME *g2);

RPLINST (Squash,
 "Evaluate by squashing")
void SquashChromosome (GENOME *g);
```

2.4 Writing the Operator and Support Code

Once we've defined the interface to our new library, we have to actually write some code for the required operators and support functions that we specified in the header file. We can either start from scratch or use the template source file (`Repname.c`) as a model. There are no naming or other requirements on source files so they may be broken down in any desired manner. Other header files can also be used without restriction, provided that all `rpp` macros are specified in the single `Set.h` header.

For the set representation, we'll use a number of global variables to track the current state (dependant on the size of the universal set, etc.), and we define the `CHROMOSOME` structure which will be our user-defined portion of a `GENOME`. Whatever method is most suitable to the particular representation and programming style should be adopted.

```
static int usize; /* Size of Universal Set
 * Set by the parameter
 * passed to the representation */

static int *universe; /* Contains the entire universal set */
```

```
static int *excluded; /* Used by Mutate */

/* Used by RAR operators: */
/* ---------------------- */
static int *common; /* common alleles */
static int *unique; /* unique alleles */
static int *childbgene; /* binary child   */
static int *foobgene; /* binary foo   */
static int *barbgene; /* binary bar   */


typedef struct
{
int size; /* number of elements in the set */
int *member; /* the members of the set */
/*  --- all in the range [0, usize-1] */
} CHROMOSOME;
```

We now proceed to write code to implement both the support functions and the operators we defined. Some examples are given below, and complete details are presented in the later reference sections (e.g. *section 3*).

```
char *SetPack (void *vchr, int *length)
{
int i;
int *buf, *gene;
CHROMOSOME *chr = vchr;

if (NULL == vchr)
{
*length = 0;
return NULL;
}

*length = chr->size * sizeof (int);
gene = chr->member;

buf = (int *) MemAlloc ((unsigned) *length);

for (i = 0; i < chr->size; ++i)
{
buf[i] = gene[i];
}

return (char *) buf;
}
void *SetUnpack (char *buf, int length)
{
CHROMOSOME *chr;
int i;
int *gp;
int *ibuf = (int *) buf;


ErrAssert (length % sizeof (int) == 0);
```

```
if (NULL == buf)
{
return NULL;
}

chr = (CHROMOSOME *) MemAlloc ((unsigned) sizeof (CHROMOSOME));

chr->size = length / sizeof (int);
chr->member = gp = (int *) MemAlloc ((unsigned) length);

for (i = 0; i < chr->size; ++i)
{
gp[i] = ibuf[i];
}

return chr;
}
void SetFree (void *vchr)
{
CHROMOSOME *chr = (CHROMOSOME *) vchr;

MemFree ((char *) chr->member);
MemFree ((char *) vchr);
}
```

2.5 Modifying the Template Makefile and Building the Library

We need to make a few minor changes to the template `Makefile` in order to build our new library:

- Change `RPLLIBNAME` to the name of the representation (exactly the same as both the root name of the representation header file, and the `RPLREPNAME` macro in the header).
- List all of the C source files with the library's code in the `CSRC` line.
- Possibly change `RPLLIBDIR` if you want to produce the library in a different directory.
- Possibly add any extra source dependencies or special compilation rules at the end of the file.

We're now ready to compile the library. This should be as simple as typing `make`. After fixing any compilation errors, we should be left with a library called `libSet.a`, which can then be linked with other libraries and the framework code as discussed below.

2.6 Building a Customised Version of RPL2

By this point, we've produced a new representation library which exports a number of operators along with a new type of genome to be used in RPL2 plans. All we need to do now is create a customised version of the RPL2 executables that incorporates this library, perhaps along with some others we are interested in using.

In a new directory, make a copy of `$(RPLHOME)/local/Makefile`:

```
mkdir myrpl
cd myrpl
cp $(RPLHOME)/local/Makefile .
```

Make the following modifications to the `Makefile`:

- add the full pathname of the directory containing your new representation to the `RPLLIBDIRS` list
- change the `RPLLIBS` list to include your representation and possibly remove others
- change the `BINDIR` variable to point at where your customised executable should go

These variables might look something like this for our set representation:

```
RPLLIBDIRS=     $(LIBDIR) \
                /my-home-dir/set

RPLLIBS=        StdInst \
                Binary \
                Set \
                Debug

BINDIR=         .
```

Now build the customised executables simply by typing `make`. This will generate a linkage stub C file (`rpl_lib_init.c`), and will link against all of the specified libraries and the rpl2 framework to produce the desired executables. These executables can now be used to run plans that make use of the new representation and operators, as discussed in the RPL2 User Guide [6].

3 Library Reference

This section describes in detail the interface to RPL2 presented to the designer of new operators and representations. The entire interface is made visible to user C code by including the file `$RPLHOME/include/rpl2.h`. RPL2 Pre-processor macros are defined in the file `$RPLHOME/include/rpp.h`.

At present, the only type of structured population instructions that may be defined by the user are new `demes`. This is primarily due to the difficulty of presenting a clean interface to the low-level message passing that must occur in most instructions which contain "explicit" parallelism.

An important point to keep in mind when modifying or adding operators to RPL2 is the naming and calling scheme adopted. It is important from the user's perspective that there should not be a proliferation of strangely named functions that take strange arguments and do not quite what you might think... If a similar function exists in a different representation, consider using function overloading. Use arguments in the same order as existing functions, and so on.

3.1 Representation-Independent Operator Libraries

A representation-independent operator library is a collection of RPL2 instructions which can act on arbitrary genomes (along with standard RPL2 data types). They can make use only of the generic fields in the genome structure (see *section 3.5*), and may not assume anything about the user-defined part of the genome.

Such a library of operators is constructed by writing ANSI-C functions and declaring an interface to RPL2 by means of a header-file containing function prototypes which contain `rpp` macros (see *section 3.3*). This allows the library not only to export the required operator names, but also to provide initialisation and termination routines, as well as checkpointing support.

The template files `Rioname.h`, `Rioname.c` and `Makefile` may be found in `$RPLHOME/riops/template`. The C source file is intended mainly as a guide, while the header and `Makefile` may most often be directly modified for a particular library.

The requirements for the library header file (`rpp` macros) are summarised below:

RPLRIONAME Must appear exactly once before any other rpp macro.

RPLVER May optionally appear once.

RPLINIT May optionally appear once.

RPLEXIT May optionally appear once.

RPLSETPARAM May optionally appear once.

RPLSAVESTATE May optionally appear once.

RPLLOADSTATE Must appear once if `RPLSAVESTATE` is used. Otherwise it cannot appear.

RPLDEME Zero or more such definitions may appear in a representation-independent library.

RPLINST Zero or more such definitions may appear in a representation-independent library.

The template `Makefile` which should be used to build the library requires several modifications to the variables, as listed:

RPLLIBNAME The name of the library, exactly as in the RPLRIONAME macro.

RPLLIBDIR The output directory for the resulting library.

CSRC The list of C files the library depends on.

Additional make dependencies may be added at the end of the file, if desired.

3.2 Representation-Dependent Operator Libraries

A representation-dependent library specifies a user-defined genome type, along with a number of operators that act upon it (and the standard RPL2 data types). It is a superset of a representation-independent library (see *section 3.1*), and requires a number of additional functions in order to allow the run-time system to handle the abstract data types which define the new representation. These functions provide for the packing, unpacking and freeing of the user-defined data in order to perform message passing and garbage collection respectively.

The library is built by writing ANSI-C functions along with an interface to RPL2, specified via a header file containing `rpp macros` (see *section 3.3*). An illustrative source file can be found in `$RPLHOME/repns/template/Repname.c`. A template header can be found in `$RPLHOME/repns/template/Repname.h`. The requirements for this header file are summarised below:

RPLREPNAME Must appear exactly once before any other rpp macro.

RPLVER May optionally appear once.

RPLINIT May optionally appear once.

RPLEXIT May optionally appear once.

RPLSETPARAM May optionally appear once.

RPLSAVESTATE May optionally appear once.

RPLLOADSTATE Must appear once if `RPLSAVESTATE` is used. Otherwise it cannot appear.

RPLPACK Must appear once.

RPLUNPACK Must appear once.

RPLFREE Must appear once.

RPLDEME Zero or more such definitions may appear in a representation-dependent library.

RPLINST Zero or more such definitions may appear in a representation-dependent library.

A `Makefile` which should be used to build the library is found in `$RPLHOME/repns/template/Makefile`. Several of the make variables should be set, as listed:

RPLLIBNAME The name of the library, exactly as in the RPLREPNAME macro.

RPLLIBDIR The output directory for the resulting library.

CSRC The list of C files the library depends on.

Additional make dependencies may be added at the end of the file, if desired.

3.3 RPP Macros

In order to allow a transparent interface between RPL2 operators and normal ANSI C functions, `rpp` (`rpl2` pre-processor) is provided. The prototypes for the interface functions between C and RPL2 are flagged with macros are flagged with specially defined macros which are listed below. The `Makefile` for the operator library will invoke `rpp` on the library's header file and will generate a corresponding C interface file (e.g. `rpp Binary.h` would generate the file `Binary_rpp.c`). This C file contains all of the necessary wrapper functions to allow your C functions to be used transparently by the RPL2 run-time environment.

3.3.1 RPLREPNAME

```
RPLREPNAME(repname,"Description");
```

This macro must be defined for a representation-dependent library of operators, and specifies the name which will be exported to the RPL2 language. It must appear before any other rpp macros in the header file. The library can then be `used` by specifying the *repname* string and any parameters (see the `RPLSETPARAM` macro) in a plan. The textual description is used in on-line help.

The macro also defines *repname* as a `LIB` pointer, to be used in the `GNew()` call (see *section 3.5*).

3.3.2 RPLRIONAME

```
RPLRIONAME(rioname,"Description");
```

This macro must be defined for a representation-independent library of operators, and specifies the name which will be exported to the RPL2 language. It must appear before any other rpp macros in the header file. The library can then be `used` by specifying the *rioname* string and any parameters (see the `RPLSETPARAM` macro) in a plan. The textual description is used in on-line help.

The macro defines *rioname* as a `LIB` pointer (as does the `RPLREPNAME` macro), although this is currently not required for representation-independent libraries.

3.3.3 RPLVER

```
RPLVER("Version info");
```

This optional macro can be specified for both representation-independent and representation-dependent libraries. It simply associates an arbitrary version identifier string with the library in order to later distinguish different executable versions of RPL2, possibly linked with different combinations of libraries.

3.3.4 RPLINIT

```
RPLINIT BOOL fname(void);
```

This macro may optionally be defined for either type of library. The macro specifies an initialisation routine which takes no arguments and returns `TRUE` on success or `FALSE` on failure. If it is defined, the function will be called once as the first call to the user's code.

3.3.5 RPLEXIT

```
RPLEXIT BOOL fname(void);
```

This macro may optionally be defined for either type of library. The macro specifies an exit or clean-up routine which takes no arguments and returns `TRUE` on success or `FALSE` on failure. If it is defined, the function will be called once as the last call to the user's code.

3.3.6 RPLSAVESTATE

```
RPLSAVESTATE BOOL fname(FILE *fpWriteBinary);
```

This macro may optionally be defined for either type of library. If `RPLSAVESTATE` is defined, `RPLLOADSTATE` must also be defined, and vice versa. The macro specifies a function which takes a file handle (pointing at a file which is opened for writing in binary mode) and returns a boolean which is `TRUE` on success and `FALSE` on failure.

The function is called during a checkpointing operation in order to save any internal state of the library. It should write sufficient information to the file to allow its counterpart to duplicate the current state of the library's global variables and so on. The function should not do any manipulation (`rewind, fseek`) of the file pointer.

3.3.7 RPLLOADSTATE

```
RPLLOADSTATE BOOL fname(FILE *fpReadBinary);
```

This macro may optionally be defined for either type of library. If `RPLLOADSTATE` is defined, `RPLSAVESTATE` must also be defined, and vice versa. The macro specifies a function which takes a file handle (pointing at a file which is opened for reading in binary mode) and returns a boolean which is `TRUE` on success and `FALSE` on failure.

The function is called during a resume-from-checkpoint operation, should read the identical data that its counterpart wrote during a previous checkpointing operation, and should replicate the internal state of the library at that time. The function should not manipulate the file pointer.

3.3.8 RPLPACK

```
RPLPACK char* fname(void *pGenome, int *pnMsgLen);
```

This macro must be defined for each new representation library. It will be used on parallel hardware to convert the user-defined part of a genome to a flat sequence of bytes so that it can be passed from one processor to another by the framework. If the `pGenome` field of the `GENOME` structure (see *section 3.5*) is non-`NULL`, this function will be called with the value of the pointer. The function must then return a pointer to a block of memory (which it allocates using `MemAlloc()` — see *section 3.8*) that contains sufficient information to reconstruct the representation dependent information on a different processor. The message length (size of the memory block) should be returned in `*pnMsgLen`.

3.3.9 RPLUNPACK

```
RPLUNPACK void* fname(char *sMsg, int nMsgLen);
```

This macro must be defined for each new representation library. It acts in reverse to the the `RPLPACK` macro, and returns a pointer to a decoded version of the representation-dependent part of a genome which is encoded in the buffer `sMsg`. The buffer is of size `nMsgLen`. `NULL` should be returned on error. The buffer space will be freed by the framework after the call has completed.

3.3.10 RPLFREE

```
RPLFREE void fname(void *pGenome);
```

This macro must be defined for each new representation library. The provided function should free all of the representation dependent information associated with the given pointer (the `pGenome` field of a `GENOME` structure — sec *section 3.5*). This function will be used by the RPL2 framework to provide automated garbage collection.

3.3.11 RPLSETPARAM

```
RPLSETPARAM BOOL fname(BOOL bUseNotDisamb, argtype1 argname1, ... );
```

This macro may optionally be defined for either type of library, to allow the library to be parameterised. If it is present, it specifies a function which returns `TRUE` on success and `FALSE` on failure, and which takes a boolean first argument. The remaining arguments (there must be at least one) of the function are arbitrary within the constraints on *section 3.4*.

The function will be called in two separate ways. The first call(s) to the routine will be when the representation appears with a proper parameter list in the `use` line of an RPL2 plan. During these calls, the first argument flag `bUseNotDisamb` will be `TRUE`. Subsequent calls to the routine will occur when an instruction from this library is preceded by a parameterised disambiguator (e.g. `::Binary(64)::Mutate(g)`). In these cases the flag `bUseNotDisamb` will be `FALSE`. Such calls will typically occur when the user has written a plan that uses more than one version of the same library (e.g. `use Binary(64), Binary(128)`), so that the routine should set some global flag (if necessary) so that operators know what parameters are in force when called later (e.g. an instruction to create a random new genomewould have to know what parameters the new genome should have).

3.3.12 RPLINST

```
RPLINST(rplfname,"Description")
rettype fname(argtype1 argname1, ...);
```

An RPL2 instruction (or operator), whether it be representation-independent or part of a new representation, is simply a normal ANSI-C function with its return and argument types corresponding to the RPL2 to C data type mapping discussed in *section 3.4*.

The macro specifies the exported name of the function in RPL2, a description of the operator (used in on-line help) and is followed by an ANSI function prototype. The C function may have arbitrary name and arbitrary argument list (within the constraints of *section 3.4*). Every argument must be given an explicit name as well as a type (which is not required by ANSI).

This macro may be used in both representation-dependenent and representation-independent operator libraries.

3.3.13 RPLDEME

```
RPLDEME(rpldemename,"Usage")
BOOL* fname(int nDim, int *pnSizeList, argtype1 argname1, ...);
```

This macro may be used to define new deme types in both representation-independent and representation-dependent libraries. The macro specifies the name of the name exported to RPL2 along with a one line description of the deme. The deme function takes a number of dimensions, an empty vector of integers, and an optional list of additional parameters corresponding to RPL2 data types as specified in *section 3.4* (which will be supplied in the `structure/deme` construct of an RPL2 plan). The deme itself is returned as an n-dimensional array of booleans.

The `nDim` argument is set by the caller to the number of fine-grained axes in the currently defined population structure. The `pnSizeList[]` argument is provided as an empty vector of `nDim` elements, to be filled by the routine. The routine should set each element in this array to the half-extent of the stencil in the given axis. The routine should return a list of `BOOL` values (in a block of memory that it allocates) which corresponds to an n-dimensional array of size equal to the product of the values `2*pnSizeList[i]+1`, as explained in [2].

3.4 Mapping RPL2 to C data types

All RPL2 instructions are implemented as C functions. Each of the RPL2 data types corresponds to a particular C data type, as indicated in *table 1*. This determines the proper argument and return types for the C code of an RPL2 instruction.

RPL2 Data Type	C Type
int	int
real	double
bool	BOOL
string	const char *
genome	GENOME *
gstack	GSTACK *

Table 1: Mapping from RPL2 to C data types

RPL2 functions may take any of the six listed types as arguments. The four base types are passed by "value" (the string type is a C pointer, but to constant data). The two special types `genome` and `gstack` correspond to the listed pointer types in C, and hence can have their value changed by the function. Any type except `gstack` may be returned by an RPL2 function.

The C type `BOOL` in *table 1* is a `typedef`'d `enum` which may take on the symbolic values `TRUE` and `FALSE` which are sensibly defined for direct use in conditionals.

The `GENOME` and `GSTACK` types are explained in *section 3.5* and *section 3.6*.

3.5 The `GENOME` type

The type `GENOME` is a `typedef`'d `struct` which has the following fields that can be accessed by the programmer:

```
void    *pGenome;
double  rRawFitness;
BOOL    bEvaluated;
```

`GENOME` structures are subject to automatic garbage collection as discussed in [5]. Only the `pGenome` pointer should be accessed directly by the programmer. The other fields may be read directly, but should only be changed (written) using the accessor functions described below. The `pGenome` pointer is provided so that the programmer can attach representation dependent information to the structure. If the pointer is non-`NULL` when the garbage collection system deletes the structure, the representation dependent `RPLFREE` function will be called (see *section 3.3*). New structures should only be allocated by using the following function:

Function `GNew`

`GENOME*` GNew (

`LIB *plibOwner,`	*The name of the representation in `RPLREPNAME()`*
`void *pGenome`	*Value for the `pGenome` field in the new struct*

)

This function allocates space for a new `GENOME` structure, to which it returns a pointer. The `pGenome` field in the structure is set to the provided value, which may be `NULL`. The first argument should be the name of the representation as specified in the `RPLREPNAME` macro — this is initialised by the `rpp` startup code as a pointer to the representation library structure, and is required so that the framework knows which representation owns each given genome.

Functions are also provided to set and unset the raw fitness values of the structure. Unset in this context simply means to mark the `GENOME` as unevaluated. This should be called if some of the representation dependant information is changed without re-evaluating the genome — for instance after a mutation operation.

Function `GSetRawFitness`

`void` GSetRawFitness (

`GENOME`	`*pg,`	*Pointer to the genome to be changed*
`double`	`rRawFitness`	*The new value for the raw fitness*

)

This function is used to set the raw fitness of the specified genome. This will modify the field `pg->rRawFitness`, and will set the value of `pg->bEvaluated` to `TRUE`.

Function `GUnsetRawFitness`

`void` GUnsetRawFitness (

`GENOME`	`*pg`	*Pointer to the genome to be changed*

)

This function is used to mark the specified genome as unevaluated. This will set the value of `pg->bEvaluated` to `FALSE`.

3.6 The `GSTACK` and `GWRAP` types

The type `GSTACK` is a `typedef`'d struct with the following visible fields. None of these fields should be changed (written) directly, but only through the provided functions, described below.

```
GWRAP    *pgwHead;
int      nElem;
```

A `GSTACK` consists of a linked list of `GENOME` structures, each of which is wrapped by a `GWRAP` structure. This is necessary to allow `GENOME`s to appear in more than one `GSTACK` at a time. The top element of the stack (first to be popped) is pointed to by `pgwHead`, and subsequent elements are linked to this. The `nElem` field indicates the number of genomes currently in the stack.

The type `GWRAP` is a typedef'd struct used to build the linked lists of the `GENOME` type in a `GSTACK`. This wrapper structure contains information about the genome which can change depending on its context, such as the scaled fitness value. The visible fields are shown below, and are to be treated as read-only:

```
GWRAP    *pgwNext;
double   rScaledFitness;
BOOL     bScaled;
GENOME   *pg;
```

The `pgwNext` field points to the next lower member on the stack, with the bottom-most element (last to be popped) having this pointer equal to `(GWRAP*)NULL`. The `pg` field points to a GENOME structure as described above.

Several functions are provided to manipulate the `GSTACK` and associated `GWRAP` types. These are discussed below.

Function `GSInsert`

`void` GSInsert (

`GSTACK`	`*pgs,`	*Stack to insert a genome in*
`GWRAP`	`*pgw,`	*The genome wrapper to insert after*
`GENOME`	`*pg`	*The genome to insert*

)

This function inserts a `GENOME` structure in the specified `GSTACK` structure. The insertion is made between the wrappers `*pgw` and `*(pgw->pgwNext)`. To insert at the top of the stack (push), `(GWRAP*)NULL` should be provided as the second argument. If `pgw` is not `NULL`, then it must appear in the specified `GSTACK`. On successful completion, a new wrapper pointing to the specified `GENOME` will be inserted in the stack, pointed to by `pgw->pgwNext`, and the value of `pgs->nElem` will be incremented. The reference count of the newly inserted genome will also be incremented. In the case of an error, no change to the stack will occur.

Function `GSRemove`

`void` GSRemove (

`GSTACK`	`*pgs,`	*The stack to be modified*
`GWRAP`	`*pgw`	*The wrapper to be removed*

)

This function removes an element from the specified `GSTACK`. The `pgs` argument must be a valid, non-`NULL` pointer to a `GSTACK` structure, and must contain the wrapper pointed to by `pgw`. If these conditions are met, the specified wrapper will be removed from the stack, and the value of `pgs->nElem` will be decremented. The reference count of the wrapped `GENOME`, `pgw->pg` will also be reduced by one. The call `GSRemove(pgs, pgs->pgwFirst)` is equivalent to popping and discarding the top element of the stack.

Function `GWSetScaledFitness`

`void` GWSetScaledFitness (

`GWRAP`	`*pgw,`	*The genome wrapper to modify*
`double`	`rScaledFitness`	*The new scaled fitness value*

)

This function is used to set the scaled fitness value of a genome in a particular stack (the wrapper structure). The `pgw->rScaledFitness` value will be changed to that specified, and the `pgw->bScaled` field will be set to `TRUE`.

Function `GWUnsetScaledFitness`

`void` GWUnsetScaledFitness (

`GWRAP`	`*pgw`	*The genome wrapper to modify*

)

This function is used to mark a genome wrapper as having no valid scaled fitness value. The `pgw->bScaled` flag will be set to `FALSE`.

3.7 Input/Output and Communications Facilities

Function `RPLBroadcast`

`void` RPLBroadcast (

`int`	`nSize,`	*The size in `chars` of the data to exchange*
`char`	`*pBuf`	*A buffer to hold the data*

)

In order to allow a library to avoid multiple reading of data (in the `RPLINIT` function), the `RPLBroadcast` function and `bRPLIONODE` global variable are provided.

The `bRPLIONODE` variable will be `TRUE` on a single processor and `FALSE` elsewhere. The `RPLINIT` function can make use of this by only reading/calculating initial data there.

To share data between processors, the `RPLBroadcast` function can be called. The function will copy the contents of the buffer on the processor where `bRPLIONODE` is `TRUE` into the equivalent buffers on the other processors. The size of the buffer must be identical on all processors. The function will only return on success, terminating with an error otherwise.

3.8 Memory Allocation in Operators

An interface to the standard C memory allocation and deallocation routines is provided by the framework. The framework assumes in some cases that memory space provided by the programmer has been allocated using these routines, and frees it accordingly. These cases include:

- The buffer space returned by an RPLPACK function
- The stencil returned by an RPLDEME instruction

It is important in these cases that the provided memory allocation/deallocation routines are used, and it is recommended that they be used in all cases in preference to the standard `malloc()` and `free()`, since they can provide instrumentation to trap memory leaks and bad frees.

The provided functions are:

Function `MemAlloc`

`char*` MemAlloc (

`unsigned int`	`nBytes`	*The number of bytes to allocate*

)

This function is equivalent to the standard `malloc()` routine.

Function `MemStrDup`

`char*` MemStrDup (

`char*`	`s`	*A zero-terminated string to duplicate*

)

The function is equivalent to the `strdup()` routine.

Function `MemFree`

`void` MemFree (

`char*` `pBlock` *A block of memory allocated with MemAlloc*

)

This function is equivalent to the standard `free()` routine.

3.9 Random Numbers in Operators

The `RPLUniformDeviate` function is provided for the purpose of generating "good" random numbers, based on the "universal" generator of described in [3]. The generator is initialised with default seed values when the framework starts up, and may be explicitly initialised from an RPL program using the `Randomize` instruction. Library routines should not attempt to re-seed the generator.

Function `RPLUniformDeviate`

`double` RPLUniformDeviate (

`void`

)

This function returns a double precision random uniform deviate in the interval $[0, 1)$. The actual generator produces 24 bit binary fractions, which may not be the full precision of a double on some hardware.

3.10 Error Handling in Operators

debugging?

sERR

Function `ErrRun`

`void` ErrRun (

`char*` `s` *Explanatory string for error condition*

)

This routine signals a run-time error and prints the given error message. Control *is* returned to the caller, so that typical usage might be:

```
ErrRun("Can't handle some condition");
return;
```

Once the operator completes, the framework will halt plan execution before the next instruction, and will checkpoint if possible.

Function `ErrAssert`

```
void ErrAssert (
    int bFlag        Boolean flag or expression which is asserted to be true
)
```

This function is similar to the `assert()` macro in `assert.h`. The function is called with an expression that evaluates to true or false. If the expression is false, an immediate fatal error is triggered, with checkpointing occurring if possible. Use of this routine is discouraged in preference to `ErrRun()` but it is sometimes convenient for cases which "never" happen...

4 Building RPL2 Executables

4.1 System Dependencies

All of the system dependent definitions have been defined in the make include file `$RPLHOME/make.defs`.

4.2 Creating a local version of RPL2

RPL2 is designed to be used in a modular way by a number of users, who may each create and maintain private libraries of representations and operators, while still making use of common facilities in the main distribution. To take advantage of this feature, it is assumed that users will maintain private copies of the `rpl2` executable while they design and test their own features.

A local copy of `rpl2` can be created by copying and modifying `$(RPLHOME)/local/Makefile`. The list of paths specifying the locations of representations and representation-independent operator (rio) libraries should be modified according to the user's preference. In the example below, the directory `./MySet` has been added to the list.

```
RPLLIBDIRS=     $(RPLHOME)/lib/$(ARCH) \
                ./MySet
```

The list of representations and other operators should be modified as appropriate. This list will be used to locate the actual library code (e.g. `libMySet.a`) as well as to generate a wrapper code which will cause them to be linked and register all of the functions defined within them. The representation `MySet` is indicated in the list below:

```
RPLLIBS=        StdInst \
                Binary \
                MySet
```

The `BINDIR` variable can also be changed to specify the destination path for the customised executables.

Once these modifications are complete, the `make` command will build an executable version of RPL2 incorporating the desired combination of command and private operator libraries.

4.3 Building a Source Distribution

This should be as simple as typing `make` in the `$RPLHOME` directory. Modifications to the make definitions in `$RPLHOME/include/make.defs` may need to be made, as described in *section 4.1*.

5 Further Information

Further information about RPL2 and related issues is available as follows:

- RPL2 Functional Specification, [5]
- RPL2 Design Document, [4]
- RPL2 User's Guide, [6]
- PUL-RD, [2]
- PUL-TF, [7]
- PUL-EM, [8]
- CHIMP V2, [1]

The authors of RPL2 can be reached at the following electronic mail addresses:

rpl2-support@epcc.ed.ac.uk Usage queries, design questions and suggestions for improvements should be reported here, and may be answered on a time-permits basis after the alpha and beta test phases.

rpl2-faults@epcc.ed.ac.uk Bug reports received at this address will be acknowledged, and may be corrected in future releases.

rpl2-contributions@epcc.ed.ac.uk Contributions of new libraries of operators and representations may be sent to this address, and will be considered for inclusion with future general releases of RPL2.

References

[1] R. A. A. Bruce and A. G. Smith. CHIMP version 2.0 interface. Technical Report EPCC-KTP-CHIMP-V2-IFACE, Edinburgh Parallel Computing Centre, 1993.

[2] S. Chapple. *PUL-RD Prototype User Guide*. Technical Report EPCC-KTP-PUL-RD-PROT-UG 1.5, Edinburgh Parallel Computing Centre, 1993.

[3] G. Marsaglia, A. Zaman, and W. W. Tsang. Toward a universal random number generator. *Statistics and Probability Letters*, 9(1):35–39, 1990.

[4] P. D. Surry. RPL2 Design Document. Technical Report EPCC-BG-PAP-RPL2-DD, Edinburgh Parallel Computing Centre, November 1993.

[5] P. D. Surry. RPL2 Functional Specification. Technical Report EPCC-BG-PAP-RPL2-FS, Edinburgh Parallel Computing Centre, July 1993.

[6] P. D. Surry. RPL2 User Guide. Technical Report EPCC-BG-PAP-RPL2-UG, Edinburgh Parallel Computing Centre, November 1993.

[7] S. Trewin. *PUL-TF Prototype User Guide*. Technical Report EPCC-KTP-PUL-TF-PROT-UG 1.7, Edinburgh Parallel Computing Centre, 1993.

[8] S. M. Trewin and S. R. Chapple. *PUL-EM Prototype User Guide*. Technical Report EPCC-KTP-PUL-EM-PROT-UG, Edinburgh Parallel Computing Centre, 1993.