Single Library Version of SPRNG

Mike Zhou and Michael Mascagni

June 6, 1999

1 SPRNG Software Structure

1.1 Application Programming Interface (API)

The API has several interfaces in order to meet the various needs of users. For both languages `C` and `Fortran`, SPRNG has **simple**, **standard**, and **pointer checking** interfaces. It also needs to support `MPI`. These are accomplished by header files and wrapper files. A user needs not to know the details of the header file and wrapper files. However, he must have a clear picture of the front interface in order to define the right macros, include the right header files and call the right functions.

First, if the user is programming in `MPI`, he needs to define the macro `USE_MPI` before including a SPRNG header file. Message passing of `MPI` is used by SPRNG only in two cases:

1. when the same seed is required on all the processes in a call to `make_sprng_seed`, this function broadcasts a single seed to all the processes and
2. during initialization with the simple interface, SPRNG needs to determine the total number of processes and the rank of the local process.

Then the user has three sub-interfaces to choose: **simple**, **standard**, and **pointer checking**. The **simple** interface is invoked by defining the macro `SIMPLE_SPRNG`. We know that SPRNG can provide a lot of random number streams. In fact, in **standard** interface (non-simple), user can initialize different streams and get random numbers from them by providing stream IDs. However, if you choose the **simple** interface, you have only one stream. The advantage of this interface is simplicity. You don't need to call the initialization routine and you don't need to provide the stream ID to get the next random number. The third interface is **pointer checking** which is invoked by defining the macro `CHECK_POINTERS`. It is similar to the default **standard** interface, except that it checks for the validity of the stream ID each time a SPRNG function is called. This facility slows down the SPRNG routines, and so would normally be used only while debugging the program.

Each sub-interface has a set of functions. Before using the functions, a user needs to include a header file which contains the function declarations. `C` programmers must include the file `sprng.h`, which in turn includes `interface.h`. A Fortran user needs `sprng_f.h`. A macro called `SPRNG_POINTER` is defined in `sprng_f.h` for the stream ID. The SPRNG initialization routine returns a unique stream ID for each stream, based on which the different streams can be distinguished. In the implementation, an ID is actually a pointer to the memory location where the state of the stream is stored. Standard `FORTRAN 77` does not have a pointer type. The job is done by `SPRNG_POINTER`. The FORTRAN programmer can use the type `SPRNG_POINTER` just as if it were a FORTRAN data type. All interfaces have the same set of function names:

- `init_sprng`
- `sprng`
- `isprng`
- `print_sprng`
- `make_sprng_seed`
- `pack_sprng`
- `unpack_sprng`
- `free_sprng`
- `spawn_sprng`

The differences between `Fortran` and `C` interfaces functions are data type names, like `integer` vs `int`, `SPRNG_POINTER` vs `int *`. The difference between the the function set of **simple** interface and that of the **standard** and **pointer checking** interfaces is that the **simple** interface functions don't have the stream ID related arguments.

Typically a user calls (we use **standard** interface as an example and provide both `C` and `Fortran` versions)

```
int       *init_sprng(int streamnum, int nstreams, int seed, int param)
```

```
SPRNG_POINTER  init_sprng(integer streamnum, integer nstreams, integer seed, integer param)
```

to initialize a random number stream. This call returns the ID of the stream. The user can then use

```
int       isprng(int *stream)
```

```
real*8    sprng(SPRNG_POINTER stream)
```

to get the next random integer in $[0, 2^{31})$, or

```
double    sprng(int *stream)
```

```
real*8    sprng(SPRNG_POINTER stream)
```

to get the next random number of double precision in $[0, 1)$. When the user doesn't need the stream anymore, the stream can be deleted via

```
int       free_sprng(int *stream)
```

```
integer   free_sprng(SPRNG_POINTER stream)
```

The other functions are auxiliary. `print_sprng` prints information about streams after initialization or spawning. `make_sprng_seed` produces a new seed using system date and time information. `pack_sprng` packs the state of the stream with ID `stream` into an array. `unpack_sprng` does the opposite. They can be used to pass a stream between processes. `spawn_sprng` creates new random number streams when given a stream ID `stream`.

1.2 Back-End Implementation

The API is the front end of SPRNG. In the previous back end implementation, a library is created for each generator. Currently there are six generators implemented: **Modified Additive Lagged Fibonacci**, **Multiplicative Lagged Fibonacci**, **Combined Multiple Recursive generator**, and three types of **Linear Congruential**. Users need to link their programs with one of the generator libraries at compile time. Different libraries implement the same set of function calls:

- `init_rng`
- `get_rn_int`
- `get_rn_flt`
- `get_rn_dbl`
- `spawn_rng`
- `get_seed_rng`
- `free_rng`
- `pack_rng`
- `unpack_rng`
- `print_rng`

Most of the front end function calls for different interfaces, `C`, `Fortran`, **simple**, **standard**, and **pointer checking**, all end up to be calls to this set of functions through the header files and wrapper files. These files can be grouped according to the interfaces they serve for. One can see the modularity of the SPRNG software. The interface files are in a module and files of each generator are in separate modules. When you add or change a generator, you don't need to worry about the interfaces files and other generators. You only need to make sure that you correctly implemented the required set of functions.

The following are for the MPI interface:

`communicate.c`

`simple_mpi.c`

`fwrap_mpi.c`

These files have functions using `MPI` libraries for communication among processors. The functions will be called when the MPI interface is chosen.

The header file `sprng.h` is for `C` interface. For `Fortran` interface, we have the following header and wrapper files:

`sprng_f.h`

`fwrap.h`

`fwrap_.h`

`fwrap_mpi.c`

We know that the generator functions are implemented in `C` and these `C` functions can not be called directly from a `Fortran` program since `Fortran` function calls **use call-by-reference** convention and `C` function calls use **call-by-value** convention. The wrapper functions in these files are the `C` equivalents of the Fortran functions so they can be called from `Fortran` programs. The wrapper functions then call the generator `C` functions.

The files `simple_.h` and `simple_mpi.c` are for the simple interface. The file `checkid.c` is for the **pointer checking** interface. Each generator has a separate directory which contains files implementing the same set of functions. The functions are declared in the header file `interface.h`.

2 The New SPRNG

Using the previous version of SPRNG, a user can use only one of the SPRNG generators in one run of the program. The objective of the new version is to combine the current random number generators (RNGs) into a single library so that a user can use all of them in a single program at the same time.

2.1 Changes to the User Interface (API)

The user now is able to and needs to specify the type of RNG when a random number stream is initialized. We add one integer argument `rng_type` to the front of the argument list of the function `init_sprng`,

```
int       *init_sprng(int rng_type, int stream_number, int nstreams, int seed, int rng_parame
```

```
SPRNG_POINTER  init_sprng(integer rng_type, integer streamnum, integer nstreams, integer see
```

User can specify one of the follows for `rng_type`:

- SPRNG_LFG
- SPRNG_LCG
- SPRNG_LCG64
- SPRNG_CMRG
- SPRNG_MLFG
- SPRNG_PMLCG

The following macros are added to `sprng.h` and `sprng_f.h`:

```
#define   SPRNG_LFG 0

#define   SPRNG_LCG 1

#define   SPRNG_LCG64 2

#define   SPRNG_CMRG 3

#define   SPRNG_MLFG 4

#define   SPRNG_PMLCG 5
```

For **simple** interface, a user can only have one random number stream at one time since he doesn't specify stream ID. The default generator is `"SPRNG_LFG"`. User still can change random number type at runtime by calling

```
init_sprng:  int *init_sprng(int rng_type, int seed, int rng_parameter)

SPRNG_POINTER  init_sprng(integer rng_type, integer seed, integer param)
```

The above are all the changes a user needs to know. Behind the scenes, a lot of changes are incurred to interface related and generator related files.

2.2 Changes to Interface Implementation Files

In the `C` interface header file, `sprng.h` and `interface.h`, the following modifications are needed because of the addition of the `rng_type argument,`

```
init_sprng(A,B,C,D)  --> init_sprng(A,B,C,D,E)

init_rng(A,B,C,D)  --> init_rng(A,B,C,D,E)

init_rng  ANSI_ARGS(( int gennum, ... -->

    init_rng  ANSI_ARGS(( int rng_type, int gennum, ...
```

In the Fortran wrapper files, `fwrap_.h` and `fwrap_mpi.c`, do initialization related modifications like,

```
FNAMEOF_finit_rng(  int *gennum, ... -->

    FNAMEOF_finit_rng(  int *rng_type, int *gennum,

init_rng(*gennum,  ... --> init_rng(*rng_type, *gennum, ...
```

Note that this is not a complete list. Anywhere containing the key string "`init`" should be checked.

For the **simple** interface to work, we define a macro called "`DEFAULT_RNG_TYPE`" in interface.h,

```
#define   DEFAULT_RNG_TYPE SPRNG_LFG
```

Then we add an argument `DEFAULT_RNG_TYPE` for function "`init_rng`" in files `simple_.h` and `simple_mpi.c`,

```
init_rng(DEFAULT_RNG_TYPE,  ...
```

2.3 The Back-End

Now we need to combine the separate generator libraries into a single library. To keep the modularity, we didn't combine the different generator functions into single functions. In stead, we keep them in original files and directories. The RNGs define the same set of function prototypes with which the various interface function calls end up. Including them in a same library will cause name conflicts. To distinguish functions for different RNG, we expand the name space by prefixing each function with the name of the generator, for example, we change `init_rng` to `lcg_init_rng` for generator `LCG`. This way, they can co-exist in one library.

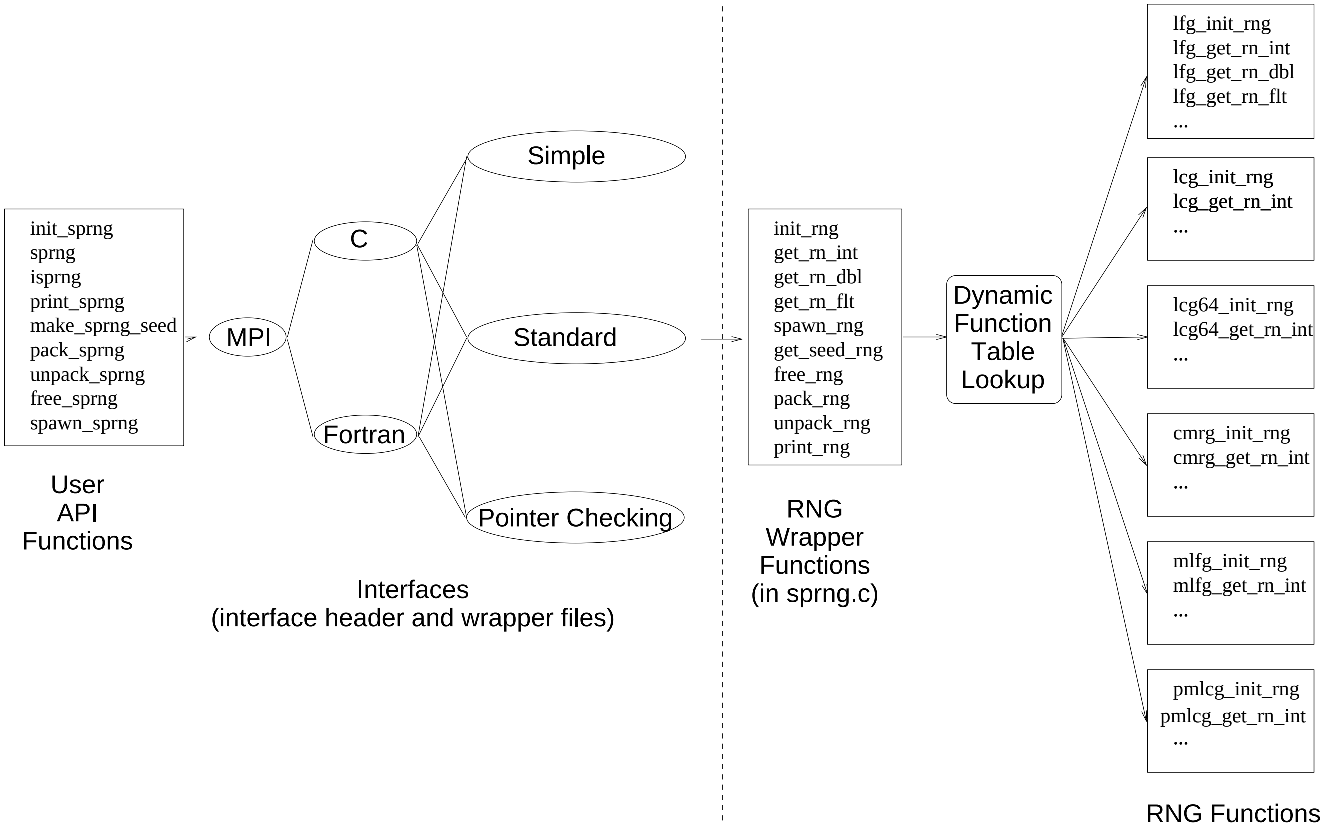

Figure 1: Single library SPRNG structure

The Wrapper File and the Dynamic Function Table Lookup

We then write wrapper functions with the original function names. The user interface functions call the wrapper functions while these wrapper functions call the right RNG functions according to the random number type. The whole picture is shown in Fig.(1).

Here are the details of the change. First we create a subdirectory `sprng` which is in parallel to the generator directory like `lcg`. In this directory we create the wrapper file "`sprng.c`" (see Appendix A).

In the code we first create global arrays of function pointers. The elements of the arrays are pointers pointing to corresponding real RNG functions. The indices to the array are generator types. This dynamic table lookup approach is more efficient than using "`switch`" or "`if ... else...`" statements. The latter will involve some comparisons. The maximum number of comparisons is equal to the number of generators. In the function table approach, there is no comparison at all. Since the functions will be called over and over, this will make a difference.

Retrieve of RNG type in the Wrapper Functions

A random number stream's state information is stored in a structure called "`struct rngen`". It is natural for us to add the random number type `rng_type` to this structure. When the initialization function "`init_rng`" is called, the random number type is stored in "`struct rngen`".

Subsequent calls for next random number will only provide the stream ID, which is the address of "`struct rngen`". The wrapper function needs to get the RNG type from the stream structure. This could be a problem since RNGs have different definitions for "`struct rngen`". We solved the problem by forcing the first field of each "`struct rngen`" be the integer `rng_type`. In the wrapper file we cast the stream ID to a structure with one and only one field `rng_type`. We then access the first field through structure member dereference. This way we get the random number type without knowing all the details of the structures.

The `unpack_rng` function is different from the others in that here we are given a packed RNG state package instead of the `rngen struct` itself. To get the rng_type we need to unpack the package but to unpack the package we need to know the rng_type to apply the right routine. To solve this chicken-egg problem we again force each RNG to pack `rng_type` first and in the same way so that we can partially unpack the package and get the `rng_type` in an RNG independent manner.

Changes to Generator Implementation Directories

Each random number generator directory, `lfg`, `lcg, mlfg, lcg64, pmlcg` and `cmrg` undergoes some changes. As mentioned above, we changed the function names by prefixing them with the generator name. We also need to store the random number type in the state structure. In the following we use "`lcg`" as an example.

We need a new header file "lcg.h" for the prototype declarations of the functions with new names. "lcg.h" should be included in sprng.c.

For "lcg.c", we prefix each function name with "lcg_" by using the following preprocessor directives:

```
#define   init_rng lcg_init_rng

#define   get_rn_int lcg_get_rn_int

#define   get_rn_flt lcg_get_rn_flt

#define   get_rn_dbl lcg_get_rn_dbl

#define   spawn_rng lcg_spawn_rng

#define   get_seed_rng lcg_get_seed_rng

#define   free_rng lcg_free_rng

#define   pack_rng lcg_pack_rng

#define   unpack_rng lcg_unpack_rng

#define   print_rng lcg_print_rng
```

Global variable names also need to be prefixed to avoid conflicts. For all generators, we apply

```
#define   MAX_STREAMS lcg_MAX_STREAMS

#define   NGENS lcg_NGENS
```

For lfg and mlfg, the name valid needs to be taken care while for lcg64 and cmrg we have PARAMLIST.

Another thing to take care is to store random number type information. First we modify struct rngen to include

```
int       rng_type;
```

as the first field. Then we modify the function init_rng to initialize the field rng_type using the rng_type passed in as argument:

```
genptr->rng_type  = rng_type;
```

The functions initialize, spawn_rng, pack_rng and unpack_rng all need be modified to deal with the new field rng_type in the structure. This is a tedious work since different generators were implemented in different ways. You must figure it out one by one.

There are three generators that need prime number routines. Unfortunately, there are two versions of them. To avoid name conflicts, we postfix the file name and function names with _32 for 32bit version and _64 for 64bit version. We also move them to the parent "SRC" directory for consistency since the common dependent files live there. Corresponding changes are needed in the generator source files. These include the include file name changes and getprime function name changes.

3 Incorporate SPRNG to Condor

Condor is a software system collecting workstation CPU cycles for computation intensive programs like Monte Carlo applications. Monte Carlo applications is not only a big CPU time consumer but also a big random number consumer. The Condor people want to include SPRNG as an integrate part of Condor so that a Condor job can be linked with SPRNG random number library easily. We describe here the technical details of this “marriage”.

In the following we assume the Condor release directory name is “`condor`” and the SPRNG root directory name is “`sprng`”.

1. Create directory, `condor/include`, and copy `"interface.h"`, `"sprng.h"` and `"sprng_f.h"` from `sprng/include` to it.

2. Copy `"libsprng.a"` from `sprng/lib` to `condor/lib`.

3. Modify `"condor/bin/condor_compile"`. Under `"CONDOR_LIBDIR= .."`, add `SPRNG_INCLUDEDIR=$CONDOR_LIBDIR/../include .`

4. Change all `"$*"` to `"$* -I$SPRNG_INCLUDEDIR -L$CONDOR_LIBDIR -lsprng -lgmp"`

User doesn’t need to specify SPRNG library to link on command line, e.g.

```
condor_compile  cc foo.c
```

Appendix A: RNG wrapper file sprng.c

```
/************************************************************************/
/*               SPRNG single library version                           */
/*               sprng.c, Wrapper file for rngs                         */
/*                                                                      */
/* Author: Mike H. Zhou,                                                */
/*             University of Southern Mississippi                       */
/* E-Mail: Mike.Zhou@usm.edu                                            */
/* Date: April, 1999                                                    */
/*                                                                      */
/* Disclaimer: We expressly disclaim any and all warranties, expressed  */
/* or implied, concerning the enclosed software.  The intent in sharing */
/* this software is to promote the productive interchange of ideas      */
/* throughout the research community. All software is furnished on an   */
/* "as is" basis. No further updates to this software should be         */
/* expected. Although this may occur, no commitment exists. The authors */
/* certainly invite your comments as well as the reporting of any bugs. */
/* We cannot commit that any or all bugs will be fixed.                 */
/************************************************************************/


#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "memory.h"
#include "sprng.h"
#include "interface.h"
#include "lfg/lfg.h"
#include "lcg/lcg.h"
#include "lcg64/lcg64.h"
#include "cmrg/cmrg.h"
#include "mlfg/mlfg.h"
#include "pmlcg/pmlcg.h"
#define NDEBUG
#include <assert.h>

#define VERSION "00"
#define GENTYPE  VERSION "SPRNG Wrapper"

/*
 *  This struct is used to retrieve "rng_type" from the rng specific
 *  "struct rngen". RNGs have different definations for "struct rngen",
 *  however, its first field must be the integer "rng_type"
 */
struct rngen
{
    int rng_type;
};

/*
 *  The function tables, the order of the RNG functions in each table
 *  must conform to that of the macro definations in sprng.h and
 *  sprng_f.h,
 *  #define SPRNG_LFG   0
 *  #define SPRNG_LCG   1
 *  #define SPRNG_LCG64 2
 *  #define SPRNG_CMRG  3
 *  #define SPRNG_MLFG  4
 *  #define SPRNG_PMLCG 5
 */

int *(*init_rng_tbl[])(int rng_type,int gennum,int total_gen,int seed,int mult)\
    = { lfg_init_rng, \
        lcg_init_rng, \
        lcg64_init_rng, \
        cmrg_init_rng,\
        mlfg_init_rng, \
        pmlcg_init_rng};

double (*get_rn_dbl_tbl[])(int *igenptr)\
    ={  lfg_get_rn_dbl, \
        lcg_get_rn_dbl, \
        lcg64_get_rn_dbl, \
        cmrg_get_rn_dbl,\
        mlfg_get_rn_dbl, \
        pmlcg_get_rn_dbl};

int (*get_rn_int_tbl[])(int *igenptr)\
    ={  lfg_get_rn_int, \
        lcg_get_rn_int, \
        lcg64_get_rn_int, \
        cmrg_get_rn_int,\
        mlfg_get_rn_int, \
        pmlcg_get_rn_int};

float (*get_rn_flt_tbl[])(int *igenptr)\
    ={  lfg_get_rn_flt, \
        lcg_get_rn_flt, \
        lcg64_get_rn_flt, \
        cmrg_get_rn_flt,\
        mlfg_get_rn_flt, \
        pmlcg_get_rn_flt};

int (*spawn_rng_tbl[])(int *igenptr, int nspawned, int ***newgens, int checkid)\
    ={  lfg_spawn_rng, \
        lcg_spawn_rng, \
        lcg64_spawn_rng, \
        cmrg_spawn_rng,\
        mlfg_spawn_rng, \
        pmlcg_spawn_rng};

int (*free_rng_tbl[])(int *genptr)\
    ={  lfg_free_rng, \
        lcg_free_rng, \
        lcg64_free_rng, \
        cmrg_free_rng,\
        mlfg_free_rng, \
        pmlcg_free_rng};

int (*pack_rng_tbl[])( int *genptr, char **buffer)\
    ={  lfg_pack_rng, \
        lcg_pack_rng, \
        lcg64_pack_rng, \
        cmrg_pack_rng,\
        mlfg_pack_rng, \
        pmlcg_pack_rng};

int *(*unpack_rng_tbl[])( char *packed)\
    ={  lfg_unpack_rng, \
        lcg_unpack_rng, \
        lcg64_unpack_rng, \
        cmrg_unpack_rng,\
        mlfg_unpack_rng, \
        pmlcg_unpack_rng};

int (*get_seed_rng_tbl[])(int *gen)\
    ={  lfg_get_seed_rng, \
        lcg_get_seed_rng, \
        lcg64_get_seed_rng, \
        cmrg_get_seed_rng,\
        mlfg_get_seed_rng, \
        pmlcg_get_seed_rng};

int (*print_rng_tbl[])( int *igen)\
    ={  lfg_print_rng, \
        lcg_print_rng, \
        lcg64_print_rng, \
        cmrg_print_rng,\
        mlfg_print_rng, \
        pmlcg_print_rng};


#ifdef __STDC__
int *init_rng(int rng_type, int gennum,  int total_gen,  int seed, int mult)
#else
int *init_rng(rng_type,gennum,total_gen,seed,mult)
int rng_type,gennum,mult,seed,total_gen;
#endif
{
    if (rng_type==SPRNG_LFG     || \
        rng_type==SPRNG_LCG     || \
        rng_type==SPRNG_LCG64   ||\
        rng_type==SPRNG_CMRG    || \
        rng_type==SPRNG_MLFG    || \
        rng_type==SPRNG_PMLCG)
    {
        return init_rng_tbl[rng_type](rng_type,gennum,total_gen,seed,mult);
    }else{
        fprintf(stderr, "Error: in init_rng, invalid generator type.\n");
        return NULL;
    }
}



#ifdef __STDC__
int get_rn_int(int *igenptr)
#else
int get_rn_int(igenptr)
int *igenptr;
```

```
#endif
{
    struct rngen * tmpgen = (struct rngen *)igenptr;
 return get_rn_int_tbl[tmpgen->rng_type](igenptr);
}

#ifdef __STDC__
float get_rn_flt(int *igenptr)
#else
float get_rn_flt(igenptr)
int *igenptr;
#endif
{
    return get_rn_flt_tbl[((struct rngen *)igenptr)->rng_type](igenptr);
} /* get_rn_float */


#ifdef __STDC__
double get_rn_dbl(int *igenptr)
#else
double get_rn_dbl(igenptr)
int *igenptr;
#endif
{
     return get_rn_dbl_tbl[((struct rngen *)igenptr)->rng_type](igenptr);
} /* get_rn_dbl */



#ifdef __STDC__
int spawn_rng(int *igenptr,  int nspawned, int ***newgens, int checkid)
#else
int spawn_rng(igenptr,nspawned, newgens, checkid)
int *igenptr,nspawned, ***newgens, checkid;
#endif
{
     return spawn_rng_tbl[((struct rngen *)igenptr)->rng_type]\
         (igenptr,nspawned,newgens,checkid);
}



#ifdef __STDC__
int free_rng(int *genptr)
#else
int free_rng(genptr)
int *genptr;
#endif
{
    return free_rng_tbl[((struct rngen *)genptr)->rng_type](genptr);
}


#ifdef __STDC__
int pack_rng( int *genptr, char **buffer)
#else
int pack_rng(genptr,buffer)
int *genptr;
char **buffer;
#endif
{
    return pack_rng_tbl[((struct rngen *)genptr)->rng_type](genptr,buffer);
}



#ifdef __STDC__
int get_seed_rng(int *gen)
#else
int get_seed_rng(gen)
int *gen;
#endif
{
    return get_seed_rng_tbl[((struct rngen *)gen)->rng_type](gen);
}


#ifdef __STDC__
int *unpack_rng( char *packed)
#else
int *unpack_rng(packed)
char *packed;
#endif
{
    int rng_type;

    load_int(packed,4,(unsigned int *)&rng_type);
    /*return unpack_rng_tbl[((struct rngen *)packed)->rng_type](packed);
      */
    return unpack_rng_tbl[rng_type](packed);
}



#ifdef __STDC__
int print_rng( int *igen)
#else
int print_rng(igen)
int *igen;
#endif
{
    return print_rng_tbl[((struct rngen *)igen)->rng_type](igen);
}


#include "../simple_.h"
#include "../fwrap_.h"
```