

JUM ver. 0.9

JANUS User Manual

Contents

Introduction	2
1 JANUS development model	3
2 The JANUS Operating System Daemon (JOSD)	5
3 The JANUS library (JLIB)	7
3.1 Header files, compilation, and shell environment	7
3.2 Starting and closing a JANUS device	7
3.3 <code>spvec</code> JANUS resource array and SP masks	8
3.4 Uploading SP firmware in a selective way	9
3.5 <code>reset_janus_device</code>	10
3.6 <code>janus_comm</code>	10
3.6.1 SP data worms composition	10
3.7 <code>janus_prog</code>	11
3.8 <code>janus_manage_answers</code>	12
3.9 Predefined firmware and library functions	12
3.9.1 The two-replicas Ising 3d firmware and library	12
3.9.2 The Ising and Potts 3D with Parallel Tempering firmware and libraries	16
3.10 Reserved names	21

Introduction

This is *JANUS User Manual* version 0.9. The following chapters cover topics on how, when and what to run on a PC conneted to one or more JANUS board systems, mainly via the `josd` user environment and `jlib` libraries.

Chapter 1

JANUS development model

You may want to develop JANUS applications at different levels. Referring to figure 1, this manual is about the three utmost layers in the chain. A short description of all layers in the picture may be:

- JANUS hardware development: all about architecture, implementation and physical production of a JANUS board system, from PB layout to chip pinouts and assembling everything up. Maybe there will be little or nothing to do at all at this level at the time you are reading. However, this layer includes the IOP firmware development, which may be a very dynamical process: the Input Output Processor firmware may change very often to provide new features and improve communication to/from host PCs.
- Communication interface PC-IOP layer. Everything about instructing the PC in how to speak with one or more JANUS boards. This is mainly an OS kernel (linux) and IOP interaction matter, and includes the definition and implementation of a communication protocol.
- Operating environment development. At this level, development concerns low level interfaces between user and JANUS, masking the two layers below. It is *low level* in the sense that the developer only has to deal with byte streams with no need to give any special meaning to their content. The developer at this layer ignores any detail of the SP firmware, and provides the program and functions needed to send and receive data to/from all devices in the IOP firmware. In the `josd` scheme, it is all about Unix IPC API.
- API layer. Actually it should be joined to the layer immediately below, but in the `josd` architecture, the C language API to JANUS is modularized and can be treated separately. At this level the developer defines what is the programming interface for the user. User programs cannot refer to layers below this one, and routines in this layer must only know how to transfer and receive data to the `josd` batch through the Unix IPC API. There is an obvious advantage in keeping this layer separated from the one below: `josd` only relies on Unix APIs, so the user interface can be written in any language. You may want to build up new libraries and application in your preferred language and dialect (even PERL or Python), and keep on working with `josd` as a JANUS resource manager.
- The User layer. Probably you are planning to develop JANUS applications at this level and above. The layer structure now splits up in two branches.
 - Aware JANUS User: you are writing your own simulation program for a specific SP firmware version. You only need to know what are the JANUS API library functions (and, of course, how the SP firmware works).
 - `jlib` or SP developer: you developed a new SP firmware application, and you need to wrap the API library functions to build a new set of library routines.

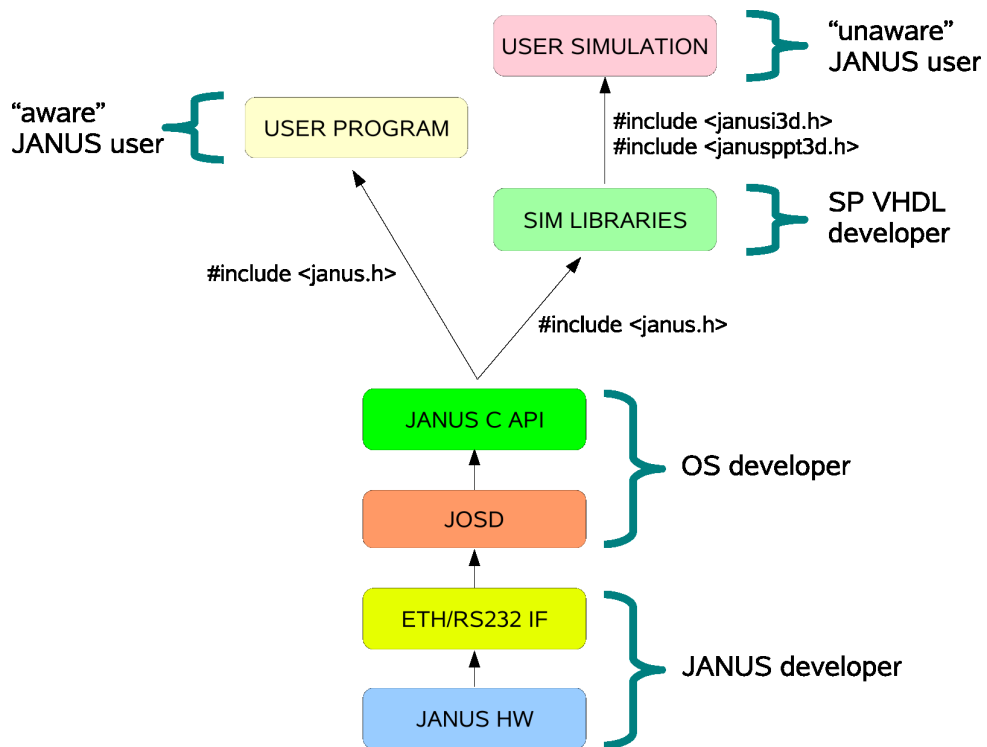


Figure 1.1: A possible schematic of JANUS development layered structure.

The developer at this level is fully aware of the overall JANUS operation, and takes into account for what exactly is doing the SP firmware.

- On top of the `jlib` developer, there is the Unaware JANUS User: you don't care what's happening in that strange black box they sold you as a *JANUS board*, you simply want your simulation program gcc'ed and runned.

Chapter 2

The JANUS Operating System Daemon (JOSD)

`josdis` is a batch process for user-JANUS interaction which provides

- JANUS hardware abstraction
- job scheduling
- distribution of JANUS computing resources
- basic monitoring facilities

Refer to the `josd` manual for details on compiling, installing and running `josd` on a JANUS host PC. `josd` makes the use of JANUS systems transparent enough to let you ignore completely its structure. Figure 2 sketches how `josd` works as a *Multi-User scheduler* for jobs accessing JANUS resources.

A job connects to the `josd` daemon through a well-known socket file (Host channel 0) and `josd` starts a new communication with it through a dedicated channel (say Host channel 1). The job asks the `josd` daemon for an amount of dedicated JANUS resources. `josd` arranges all FPGA nodes (the SPs) of all the JANUS boards it sees connected to the PC in a virtual array. The job asks for resources in terms of *number of nodes*, and may add further “geometric” indications. It may ask, for instance, that `josd` reserved to it 8 nodes with the constraint of all being on the same board. If `josd` sees the request could be fulfilled (there are enough free nodes on a single board), then it labels such nodes as reserved by that specific job, and keeps on listening for communication requests.

In the figure 2, the job by user *Caius* reserved 16 nodes on a single board, while the job by user *Sempronius* asked for the first five nodes available: at the time it asked the nodes were almost all locked by other jobs, so `josd` reserved five scattered SPs. To the jobs point of view, it doesn't matter where actually the reserved nodes are: user *Sempronius* always refers to them as nodes 0, 1, 2, 3, 4. The actual distribution of nodes matters if the user application needs to perform direct inter-node communications, but it's up to the user to ask for specific reserved node patterns. For example, user *Caius* may include every sort of inter-node communication in its program as it is guaranteed all its reserved nodes are mutually connected.

`josd` keeps track of the distribution of reserved nodes, routing communications between the user jobs and the JANUS boards. This task is accomplished in a completely transparent way to the user, who everytime can refer to nodes reserved by the job with an integer index with consecutive values.

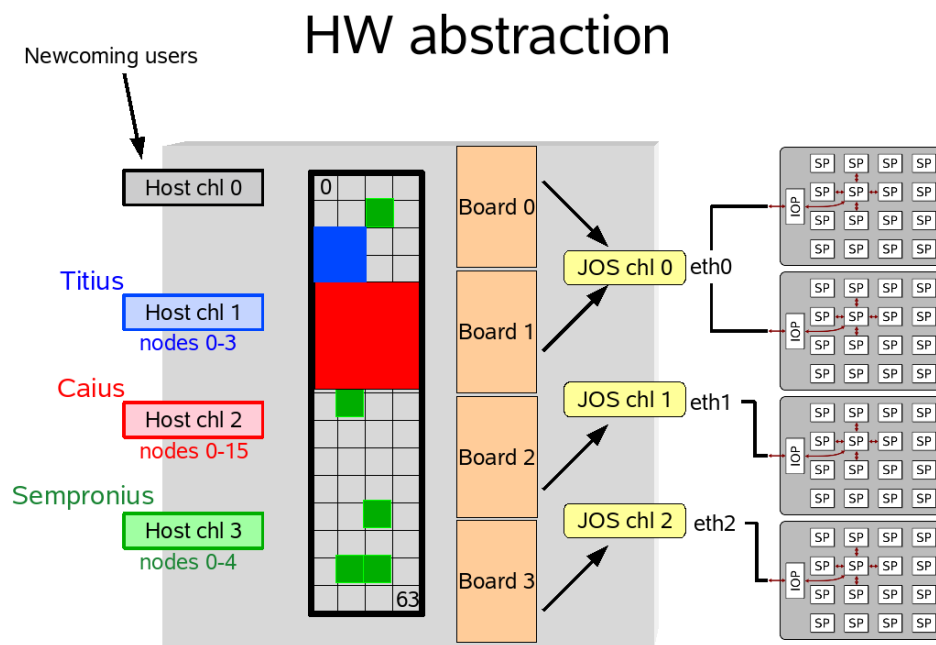


Figure 2.1: josd batch turning JANUS in a FPGA cluster.

Chapter 3

The JANUS library (JLIB)

3.1 Header files, compilation, and shell environment

A C program must include the header file `janus.h` in order to connect to the `josed` daemon. Such header file can be found in the `josed/usr/include` directory of the `josed` distribution (e. g. `/usr/local/josed/usr/include`, where `josed` is not a misspelling). Shared libraries (`libjanus.so`) for linking are in the `josed/usr/lib` directory. Remember to add the complete path to `josed/usr/lib` in your `LD_LIBRARY_PATH` environment variable. Example (bash shell):

```
prompt> export LD_LIBRARY_PATH="/usr/local/josed/usr/lib"
```

or alternatively run your simulation program `run.exe` as

```
prompt> LD_LIBRARY_PATH="/usr/local/josed/usr/lib" ./run.exe
```

3.2 Starting and closing a JANUS device

Remember to include the header file

```
#include <janus.h>
```

In a C program a JANUS set of resources is a *JANUS device*, and is represented by a structure, defined in the `janus.h` header.

```
struct janus_dev {  
  
    struct sockaddr_un un;  
    struct msghdr      msg;  
    struct jos_ctrl    ctrl;  
    union jos_rqst     rqst;  
    int                fd;  
    size_t             un_len;  
    pid_t              pid;  
    uid_t              uid;  
    gid_t              gid;  
    int*               svec;  
    int                splen;  
    char*              sockf;  
};
```

```
};
```

The structure stores information about the user and the current job needed by `josd` to manage communication. The only field you have to change at any rate is the `spvec` one, as described below. An instance of this `struct janus_dev` is your *phone jack* to JANUS. So your program must contain a declaration

```
struct janus_dev JDEV
```

or equivalently

```
struct janus_dev JDEV[1]
```

so we can use the name `JDEV` directly as a pointer to a `struct janus_dev`.

Once the *phone jack* has been declared, plug it with the function

```
int init_janus_device (struct janus_dev* idev, int NSP , int HOW, char* FWSYS)
```

where the first argument is the pointer to the `struct janus_dev` we are trying to connect and `NSP` is the number of SP nodes you want to reserve.

You also have to provide a geometric argument as the `HOW` parameter: allowed values are the macros

1. `IG_ANYWHERE` : SP nodes are reserved anywhere in the SP node grid
2. `IG_ONEBOARD` : SP nodes must be in the same JANUS board
3. `IG_GROUP` : SP nodes must be in the same JANUS board and must be consecutive indexed nodes (index= $4 * y + x$, remember a JANUS board is a 4×4 array of SP nodes);

More geometry macros will be added in the future.

Last, you may provide `FWSYS` the full pathname to a specific SP firmware: the corresponding binfile will be uploaded to all SPs that will be reserved by `josd`. Most users need not provide any special firmware, so a `NULL` may be passed to `FWSYS`.

The `init_janus_device` call **MUST BE** the first JANUS library call function before any other operation on a `struct janus_dev`.

The JANUS device must be closed on exit. This can be achieved with a call to

```
int release_janus_device (struct janus_dev* idev);
```

It is advisable that you always call the release function on exit, but `josd` should be able to detect broken connection and to wipe out dead devices by itself. You may also release a janus device at any point in your program, and then re-initialize it by the `init_janus_device`. this way, however, all reserved resources are lost, and a brand new connection established.

The `init_janus_device` function returns a negative value on failure. Please check the return value before going on polling the device for operation.

3.3 spvec JANUS resource array and SP masks

```
int janus_set_sp_order (struct janus_dev* idev, int* neworder);
```

After the `init_janus_device(...)` call, the corresponding `struct janus_dev` will have its `splen` field set to `NSP` and the `spvec` array alloc'ed to `splen` elements, whose values are initialized to integers between 0 and `NSP - 1`. This would be of little or no use to the user if there wouldn't be a library function for swapping the SPs order. Whenever you need to swap SPs order, declare an array of `NSP` `int` integers, and fill it with every value in the range $0 \dots NSP - 1$ in the order you want, then call the `janus_set_sp_order` function:

```

struct janus_dev JDEV[0];
int swaporder[] = {3,2,4,5,7,1,6,0};
int oldorder [] = {0,1,2,3,4,5,6,7};
...
init_ianus_device(JDEV,8,IG_ANYWHERE,NULL);
...
...
...

janus_set_sp_order(JDEV,swaporder);
...
...
janus_set_sp_order(JDEV,oldorder);
...
...
release_janus_device(JDEV);

```

The `janus_set_sp_order` function always return 0 as it performs no control over bad sized arrays passed as argument, so watch out. It is possible to pass as a new order with two or more or all elements containing the same value.

Several functions in the library may need a SP mask to prevent operation on a subset of the reserved nodes. An SP mask for a JANUS device is an array of NSP (or equivalently `JDEV->spLen`) chars. An array value of 1 sets the corresponding SP in the `spvec` list as *activated*, while a 0 (zero) value sets it as *skipped*. You may define several SP masks as simple arrays of chars, set to 0s and 1s and pass them as argument to functions that will require them. An example of the use of a SP mask is in the next section.

3.4 Uploading SP firmware in a selective way

You can select a binfile for upload to all SPs you reserve during the `init_ianus_device` as described above, or call an upload function directly to maintain more control over the process. The prototype for the upload function is

```
int janus_load (struct janus_dev* jdev, char* filename, int Fsp, int Nsp, char* Spmask)
```

where the first argument is the JANUS device pointer and the second one is the full path of a binfile firmware. The following arguments select which SPs have to be programmed. If you specify `NULL` as `char* Spmask`, then you have to provide `FSP` and `NSP`. `NSP` is the number of SPs to be programmed starting from element numbered `FSP` in the `jdev->spvec` array. In the event that the provided values for `FSP` and `NSP` will generate a bad SP index, `jospd` could raise a `BAD_RQST` condition. In this case `jospd` will command a shutdown to the program.

If `Spmask` is not a `NULL` pointer, then the function ignores `FSP` and `NSP`. In this case, for every value of 1 in the SP mask, the programmer will upload the firmware on the corresponding SP in the `spvec` array. Old firmware persists on all SPs `spvec` array whose corresponding values in the SP mask are 0 (zero). An example:

```

struct janus_dev JDEV[0];
int swaporder[] = {3,2,4,5,7,1,6,0};
char loadmask[] = {1,1,1,0,0,0,0,1}
init_ianus_device(JDEV,8,IG_ANYWHERE,NULL);
janus_set_sp_order(JDEV,swaporder);
janus_load(JDEV,/my/path/fwA.bin,3,4,NULL);
janus_load(JDEV,/my/path/fwB.bin,0,0,loadmask);
...

```

given the `swaporder` values, after the two `janus_load` calls, SPs numbered 1, 5, 6, 7 will be programmed with `/my/path/fwA.bin`, while SPs numbered 0, 2, 3, 4 will be programmed with `/my/path/fwB.bin`. Passing an SP mask with all zeroes will generally give unpredictable results, as well as passing an undersized array.

`janus_load` return zero in successful completion, a negative value otherwise

3.5 reset_janus_device

This function is used by the `init_janus_device` and `release_janus_device` functions to reset a `janus_dev` internal fields to NULL values.

```
int reset_janus_device (struct janus_dev* idev);
```

You should never use it outside the `init/release_janus_device` scope. Resetting a `janus_dev` without initializing or releasing it leaves a malfunctioning janus device.

3.6 janus_comm

The `janus_comm` function is responsible for transmitting and receiving data to/from SPs through `josed`.

```
int janus_comm(struct janus_dev * idev, int Nbufs, int Bufsize, int Fsp, int Nsp,
               int DataAns, char * outbuffer, char * inbuffer);
```

The first argument is a valid initialized `janus_dev` structure object. The second argument `Nbufs` indicates the number of buffers we are going to send to Janus. `Bufsize` is the size of a single sent buffer. `Fsp` and `Nsp` indicate the first SP and the number of consecutive SPs to send data to, depending on the content of `idev->spvec`. `outbuffer` is an array of bytes containing all data to transmit to SPs. It has to be a valid pointer to an allocated memory portion of at least `Nbufs×Bufsize` chars. Note: `Nbufs` may only be `Nbufs= 1` or `Nbufs=Nsp`. In the first case, only one buffer will be sent to all selected SPs, and all of them will receive the same `Bufsize` bytes starting from the location pointed to by `outbuffer`. In the second case, `josed` will manage data in order to send a single buffer of `Bufsize` bytes per each selected SP: `outbuffer` must point to an array of `Nsp×Bufsize`.

`DataAns` is an integer (zero or positive) specifying the amount of bytes expected as a response; `DataAns` is not zero, `inbuffer` must point to an array of `DataAns` bytes, where returned data will be copied to. Please note that Janus' IOP does not support parallel reads, so specifying `Nsp ≥ 2` and `DataAns ≠ 0` results `janus_comm` returning with an error due to `josed` rejecting the request.

`janus_comm` return zero in successful completion, a negative value otherwise

3.6.1 SP data worms composition

The `janus_comm` function is the fundamental brick for SP communications. Each buffer sent to an SP must contain only data intelligible by the SP alone: the `janus_comm` send all data streams to the `josed` server which is responsible for subdividing and arranging it in order to be sent to the janus IOPs and target the specific devices in the `MultiDev`. A sketch of the process is depicted in Fig. 3.6.1 (see the `josed` manual for a detailed description of the communications).

`janus_comm` splits data beginning at the position pointed to by `outbuffer` in `Nbufs` blocks of equal size `Bufsize`. It passes such split buffers to `josed` along with some other information (`Nbufs`, `Bufsize`, `tttFsp`, `Nsp`, and `idev`). Such information is enough for `josed` to track what are exactly the SPs it has to send data to, and build up the correct SP masks along with the appropriate `VGBLen`. The so formed data worm is then headed with the SP interface id and sent to the IOP. If `josed` recognize that some SP do not reside in the same board (so that a single data worm and a single ethernet write does not suffice), it is capable to split that operation in more suboperations.

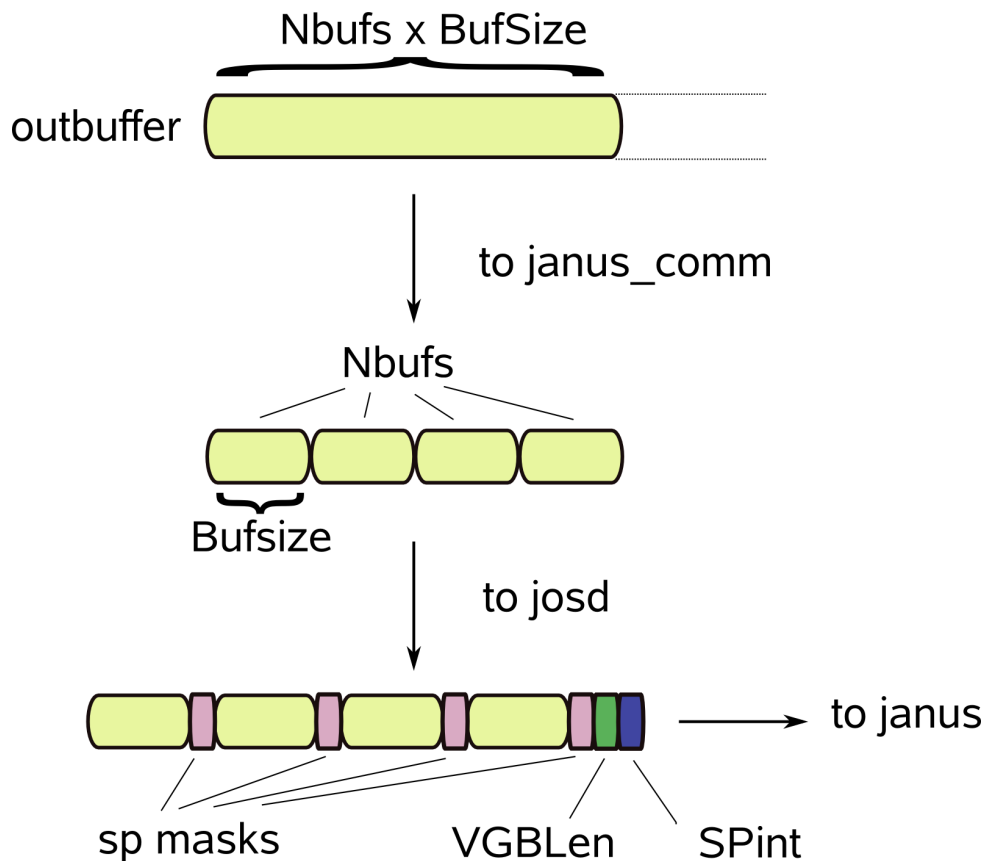


Figure 3.1: `janus_comm` function sketch

Usually (always) SP firmware receives data as a worm of 16 bit words. The SP interface in the IOP always forwards to SPs a minimum of two words. Just as an example, the following code sends the 16-bit command `0x103F` to SPs indexed as 2,3,4 in the `idev->spvec` array:

```
char buffer[4];

buffer[0]=0x3F; buffer[1]=0x10;
buffer[2]=buffer[3]=0xFF;
janus_comm( idev, 1, 4, Fsp, Nsp, 0, buffer, NULL);
```

Please note: i) a minimum of 4 bytes (two 16 bit words) is required: the two padding bytes are set to `0xFFFF` which is our conventional NOP for SP applications; ii) we are passing a `DataAns` value of zero and a `inbuffer=NULL` pointer just because we do not expect any response by the SPs we are controlling.

3.7 `janus_prog`

```
int janus_prog(struct janus_dev* idev, char* filename, int Fsp, int Nsp);
```

This function selectively loads the firmware whose pathname is specified as the `filename` argument, depending on the specified first SP index `Fsp` and number of consequent SPs `Nsp` in the `spvec` array member of the `idev` janus device structure. It is called by the `janus_load` function, which wraps it including the `spmask` mechanism for more versatile use.

`janus_prog` return zero in successful completion, a negative value otherwise

3.8 `janus_manage_answers`

```
int janus_manage_answers(struct janus_dev* idev);
```

This function interpretes responses by the `josed` daemon the `idev` device is connected to. It does not return, and in some case will recursively call itself, until some action is commanded by the daemon itself. It is opportunely wrapped into the `janus_comm` and `janus_prog` functions, and users should not refer it directly in any case.

3.9 Predefined firmware and library functions

Functions declared in the `janus.h` are all you need to build your own libraries for JANUS interaction, specially if you are a SP developer and want to use `josed` as interface to the machine. The following sections describe library functions for uploading and interacting with already available firmware for few specific applications

3.9.1 The two-replicas Ising 3d firmware and library

To access functions and default firmware specific for Ising 3d physics, include the `janusi3d.h` header (i3d library).

```
#include <janusi3d.h>
```

from `jos/usr/include/`. This header includes the `janus.h` header, so there's no need to include it explicitly. You need to declare a `janus_dev` structure and then initialize it exactly as described in the previous chapter. **Please note that all functions return negative integers upon failure, and that you should catch return values for program flow control.**

Loading

You may load your specific firmware by the functions `init_janus_device` od `janus_load` as described in section ??.

The i3d library also has its own specific load function. i3d functions are guaranteed to work with firmwares loaded this way.

```
int janus_i3d_load (struct janus_dev* jdev,
                   int phys, int size, int alg,
                   int Fsp, int Nsp, char* spmask);
```

The first argument is a JANUS device structure pointer.

The `phys` argument may be one of the macros

- DAFF for diluted antiferro
- RFIM for random field ising model
- DISN for site diluted Ising model

- DISG for site diluted EA Spin Glass
- SGEA for EA Spin Glass

At the time of writing one or more of them could not be supported, but they will all be in the near future. The choice of this parameter determines the features of the firmware that will be uploaded: if you select SGEA it is likely that a subsequent try in uploading random magnetic field data to the SPs will fail.

The `size` argument is the desired linear size of the 3d cubic lattice. Allowed values are multiples of 8, starting from 8 up to 96 ($L = 88$ still missing!!).

The `alg` argument may be either the macro HEATBATH or the macro METROPOLIS with consequent selection of the implemented Monte Carlo sequential update rule. A predefined binfile will be automatically selected by the choice of the arguments above.

You can select which SP nodes to program, in the `jdev->spvec` list, by means of the other three arguments. If a NULL is provided to `spmask`, then `Nsp` nodes starting from the `Fsp` element in the `jdev->spvec` list will be selected for uploading. If you provide a valid `spmask` char pointer, then `Fsp` and `Nsp` will be ignored. Example:

```
struct janus_dev JDEV[0];
int swaporder[] = {3,2,4,5,7,1,6,0};
char loadmask[] = {1,1,1,0,0,0,0,1}
init_janus_device(JDEV,8,IG_ANYWHERE,NULL);
janus_set_sp_order(JDEV,swaporder);
janus_i3d_load(JDEV,SGEA,40,HEATBATH,3,4,NULL);
janus_i3d_load(JDEV,SGEA,80,HEATBATH,0,0,loadmask);
...
```

given the `swaporder` values, after the two `janus_i3d_load` calls, SPs numbered 1,5,6,7 will be programmed with EA spin Glass firmware for a $L = 40$ 3D lattice, while SPs numbered 0,2,3,4 will be programmed with EA SG firmware for a $L = 80$ 3D lattice.

Status and Wait

```
int janus_status_safe (struct janus_dev* idev, char* STATUS,
                      unsigned long long *NITER,
                      int Fsp, int Nsp, char* SPmask);
int janus_status      (struct janus_dev* idev, char* STATUS,
                      unsigned long long *NITER,
                      int Fsp, int Nsp, char* SPmask);
int janus_wait_safe   (struct janus_dev* idev, char* SPmask);
int janus_wait        (struct janus_dev* idev, char* SPmask);
```

These functions permit polling the status of the run on SP nodes. As usual, the first argument is a JANUS device structure pointer. The second argument to `janus_status` is an array of `idev->splen` chars. The third argument is an array of `idev->splen` unsigned long long values. These two arrays will be eventually overwritten by the function itself. The choice of which SPs to check for is as usual performed specifying a valid SP mask pointer as the sixth argument (in this case `Fsp` and `Nsp` will be ignored). Equivalently, one may specify a NULL SP mask and specify `Nsp` nodes to check for, starting with the position `Fsp` in the `idev->spvec` array. The function returns negative value in case of problems. On success, 0 is returned and arrays `STATUS` and `NITER` are set. A `STATUS[n]` value of 1 indicates that SP corresponding to `idev->spvec[n]` is running. A `STATUS[n]` value of 0 means that the SP corresponding to `idev->spvec[n]` is either idle or was not included in the input selection by `Fsp` and `Nsp` or `spmask`. For each checked SP, the corresponding value of `NITER` is set to the value of the internal Monte Carlo steps counter.

The `janus_wait` function polls all SPs specified by the `spmask` not returning until all are idle. A return value less than zero means problems during polling, and some selected SPs might still be running.

IMPORTANT: The `_safe` versions of these functions are mandatory if you are uploading firmwares with the `janus_i3d_load` function. If you are uploading older firmware or firmware that does not include the DDR fault correction, normal versions are recommended.

Run control

```
int janus_start      (struct janus_dev* idev, int Fsp, int Nsp, char* SPmask);
int janus_stop      (struct janus_dev* idev, int Fsp, int Nsp, char* SPmask);
int janus_reset     (struct janus_dev* idev, int Fsp, int Nsp, char* SPmask);
int janus_continue  (struct janus_dev* idev, int Fsp, int Nsp, char* SPmask);
int janus_wrmsfr    (struct janus_dev* idev, unsigned long long niter,
                    int Fsp, int Nsp, char* SPmask)
```

These functions allow run control by starting and stopping the update machinery in the i3d firmware. Selection of SPs works as usual with the `int Fsp, int Nsp, char SPmask` arguments (see above). `janus_reset` will reset all counters (e.g. the total Monte Carlo steps, lattice span indexes, etc.). You better reset only idle SPs (after they have been checked idle).

`janus_start` will start the update machinery in the selected SPs. An SP will be running until the internal Monte Carlo steps counter match the Measure Frequency register (see the `janus_wrmsfr` function below).

The `janus_stop` and `janus_continue` functions respectively stop and resume the Monte Carlo update with no internal counter reset. Their use is deprecated.

`janus_wrmsfr` writes the `niter` argument in the Measure Frequency register inside each SP selected by the `int Fsp, int Nsp, char SPmask` arguments. After an SP is started, it won't stop until the internal Monte Carlo step counter will match this register, or a `janus_stop` or a `janus_reset` is called. IMPORTANT: Pay attention in not setting the Measure Frequency register to a value equal or lesser than the actual value of the Monte Carlo step counter, as it will result in very long runs (`janus_wait_safe` may wait a very very long time before returning).

Writing fields data

```
int janus_ws (struct janus_dev* idev, char* Sa, char* Sb,
              int Size, int Fsp, int Nsp, char* spmask, int Pflag);
int janus_rs (struct janus_dev* idev, char* Sa, char* Sb,
              int Size, int Fsp, int Nsp, char* spmask, int Pflag);
int janus_wj (struct janus_dev* idev, char* Jx, char* Jy, char* Jz,
              int Size, int Fsp, int Nsp, char* spmask, int Pflag);
int janus_wh (struct janus_dev* idev, char* HH,
              int Size, int Fsp, int Nsp, char* spmask, int Pflag);
int janus_wd (struct janus_dev* idev, char* DD,
              int Size, int Fsp, int Nsp, char* spmask, int Pflag);
int janus_wr (struct janus_dev* idev, unsigned int* RNG,
              int Size, int Fsp, int Nsp, char* SPmask, int Pflag);
```

These function permits uploading of respectively, spin, coupling, random fields orientations, random site occupations and random number generator seeds. The `janus_rs` permits downloading of spin configurations. All of them accept as first argument `idev` is a pointer to valid initialized `struct janus_dev`.

The `janus_ws` function requires two char buffer pointers, `Sa` and `Sb`, corresponding to the two replicas. The linear size of the cubic lattice must be passed as the `Size` argument. SP selection is performed by means of either the couple of arguments `Fsp, Nsp` or specifying a non-NULL pointer as `spmask`; if `spmask`

is non-NULL, the function ignores the passed values for `Fsp`, `Nsp`. If `spmask` is NULL, then the functions check for which SP id corresponds to position `Fsp` in the `idev->spvec` array. Then it will mark as active all subsequent SP id corresponding to positions `Fsp+1`, `Fsp+1` up to `Fsp+Nsp`. If `spmask` is non-NULL, it must be a pointer to an array of `idev->splen` chars; an `spmask[i]` value of 1 marks the SP whose id resides in `idev->spvec[i]` as active; a value of zero marks it as inactive. The behavior is finally determined by the `Pflag` argument.

If `Pflag` is `PARALLEL`, the functions expects `Size×Size×Size` bytes in both the `Sa` and `Sb` buffers, which have been previously initialized by the user to the values of the spins (one spin per byte) with values 0 for down spins and value 1 for up ones. The same spin configurations will be then uploaded to all active SPs.

If `Pflag` is set to `SEQUENTIAL`, the function expects `Size×Size×Size×NUM` bytes, with `NUM` the number of active SPs (i.e. either `Nsp` or the number of ones in the `spmask` array), in both the `Sa` and `Sb` buffers. The function then will upload the first `Size×Size×Size` bytes of both `Sa` and `Sb` to the first active SP, the second block of `Size×Size×Size` bytes to the second active SP and so on.

Inside each block, the index $i = 0, \dots, \text{Size} \times \text{Size} \times \text{Size} - 1$ of the spin maps to conventional cubic 3d lattice coordinates as $i = x + \text{Size} \times y + \text{Size} \times \text{Size} \times z$.

`PARALLEL` and `SEQUENTIAL` and their shorthands `PAR` and `SEQ` are macros defined in `janusi3d.h`

`janus_rs` performs the inverse operation. As for `janus_ws`, `Fsp`, `Nsp`, `spmask` selectively activate SPs in the `idev->spvec` array. `Sa` and `Sb` must point to buffers of `Size×Size×Size×NUM` bytes, with `NUM` the number of active SPs (i.e. either `Nsp` or the number of ones in the `spmask` array). The `Pflag` argument is completely ignored as all reads are intrinsically sequential. Upon successful returning, the buffers `Sa` and `Sb` will contain spin configurations for the two replicas of all active SPs, in consecutive blocks of `Size×Size×Size` bytes, one byte per spin.

The other functions behaves exactly the same as `janus_ws`, apart for the number and meaning of supplied buffers.

`janus_wj` uploads quenched coupling configurations. The three buffers `Jx`, `Jy` and `Jz` respectively contain couplings along the positive direction of the conventional axis x , y , z . Byte-order/lattice-site relations is the same as for spin configurations. A char value of 1 represents a ferro coupling, while a value of 0 represents an antiferro.

`janus_wh` uploads quenched random field orientations. Only one buffer `HH` is needed. Byte-order/lattice-site relations is the same as for spin configurations. A char value of 1 represents spin-up favoring, while a value of 0 represents a spin-down favoring field. If you are loading firmwares with the `janus_i3d_load` function, only SPs loaded with the `RFIM` mode will support this feature.

`janus_wd` uploads quenched random site dilution. Only one buffer `DD` is needed. Byte-order/lattice-site relations is the same as for spin configurations. A char value of 1 represents an occupied site, while a value of 0 represents an unoccupied one. If you are loading firmwares with the `janus_i3d_load` function, only SPs loaded with the `DAFF` or `DISN` or `DISG` mode will support this feature.

All two-replicas Ising 3D firmwares use one or more shift registers for 32-bit random numbers generations. You have to provide an array of random bits to SPs as seeds. In order to keep safe, the receipt is: i) Reserve an array of `RNG_SAFE_SIZE` 32 bit integer seeds somewhere in your program; ii) Fill it with random 32 bit numbers; iii) use the `janus_wr` function to upload the seeds to SPs. The syntax and arguments are mostly the same as the above functions, except for the data buffer `RNG`, which has to be the pointer to your pre-generated array of random seeds. The `Size` argument is needed to instruct internally the function on how many seeds to send to SPs. The relation between number of seed and lattice size may change when firmware is upgraded, so better `RNG` be an array of `RNG_SAFE_SIZE` 32 bit integers, and let the function manage such issue internally. `RNG_SAFE_SIZE` is a predefined macro. Even if it is not a so big number, no Janus machine will ever need more than `RNG_SAFE_SIZE` 32 bit random seeds.

Update algorithm LUTs

```
int janus_wl (struct janus_dev* idev, unsigned int* Table,
```

```

        int Fsp, int Nsp, char* SPmask, int Pflag);
int janus_generate_lut      (unsigned int* Table, double Beta, double Hfield);
int janus_generate_lut_diluted (unsigned int* Table, double Beta, double Hfield);

```

Transition probabilities are represented as arrays of LUTSIZE 32-bit integers (LUTs) (renormalized to $2^{32} - 1$). The macro LUTSIZE (usually 16) is defined in `janusi3d.h`. Not all LUTSIZE values are needed: SGEA applications, for instance, only use the first 7 values, while DAFF applications use 13 values. Each SP needs LUTSIZE 32-bit values, and is internally configured to use only the proper fraction of them. If you do not know how to correctly build LUTs, use the `janus_generate_lut` and `janus_generate_lut_diluted` functions.

The `janus_generate_lut` accepts as first argument `Table` a pointer to an array of LUTSIZE unsigned ints. The second and third arguments are, respectively, the values of inverse temperature and (signed) magnetic field intensity from which the Heat-Bath transition probabilities are computed. On returning, the `Table` array is correctly set and ready to be fed to SPs.

The `janus_generate_lut_diluted` performs the same action but taking into account that local energy for a diluted system may take 13 different values. Use this version if you are loading firmwares with the `janus_i3d.load` function and with the DAFF or DISN or DISG mode.

To upload the LUTs to Janus, use the `janus_wl` function. `janus_wl` requires an unsigned int buffer pointer `Table`, SP selection is performed by means of either the couple of arguments `Fsp`, `Nsp` or specifying a non-NULL pointer as `spmask`; if `spmask` is non-NULL, the function ignores the passed values for `Fsp`, `Nsp`. If `spmask` is NULL, then the functions check for which SP id corresponds to position `Fsp` in the `idev->spvec` array. Then it will mark as active all subsequent SP id corresponding to positions `Fsp+1`, `Fsp+1` up to `Fsp+Nsp`. If `spmask` is non-NULL, it must be a pointer to an array of `idev->splen` chars; an `spmask[i]` value of 1 marks the SP whose id resides in `idev->spvec[i]` as active; a value of zero marks it as inactive. The behavior is finally determined by the `Pflag` argument.

If `Pflag` is PARALLEL, the functions expects LUTSIZE unsigned ints in the `Table` buffer, which have been previously initialized by the user to the correct LUT values. The same LUT will be then uploaded to all active SPs.

If `Pflag` is set to SEQUENTIAL, the function expects LUTSIZE×NUM unsigned ints in the `Table` array, with NUM the number of active SPs (i.e. either `Nsp` or the number of ones in the `spmask` array). The function then will upload the first block of LUTSIZE unsigned ints to the first active SP, the second block of 16 unsigned ints to the second active SP and so on.

3.9.2 The Ising and Potts 3D with Parallel Tempering firmware and libraries

At the time of writing, some preliminary but stable versions of firmwares for Ising and Potts 3D Monte Carlo with Parallel Tempering simulation are available. An SP loaded with such a firmware is able to simulate a variable number R of replicas (up to 64, run-time configurable) every one with its own local update transition probabilities (LUTs). The R replicas are also periodically interchanged (run-time configurable frequency) following the Parallel Tempering scheme. User must load LUTs in either increasing or decreasing temperature order, and the internal machinery is responsible for tracking replicas and swapping them sequentially in increasing or decreasing temperature order, depending on the user initial upload choice. The firmware **does not** simulate two replicas per temperature: to have more real replicas, you must multiply your run on several SP nodes and configure them with the same quenched disorder.

Both Ising and generic Potts library functions are declared in `janusppt3d.h`, which is available in the `jos/usr/include` directory of the main `jost` installation (usually `/usr/local/jos/usr/include`).

User program must declare and initialize before accessing library functions a variable whose type is a `struct janus_Pconf`

```
struct janus_Pconf pconf;
```

or equivalently

```
struct janus_Pconf pconf[1];
```

Such structure is defined in the `janusppt3d.h` header.

```
struct janus_Pconf {
    int Size;
    int Pstates;
    int Nbetas;
};
```

Its members have auto-explaining names: the linear size of the lattice, the number of states of Potts variable (2 for Ising-like systems), the number of temperature replicas for parallel tempering. A pointer to such a structure has to be passed to many of the control functions described below.

Loading

You may load your specific firmware by the functions `init_janus_device` or `janus_load` as described in section ???. However, the library functions are guaranteed to work for firmwares you load by means of the following function:

```
int janus_ppt3d_load (struct janus_dev* idev, struct janus_Pconf* pconf,
                    int Fsp, int Nsp, char* spmask);
```

The first argument is a pointer to a valid initialized janus device (see sec. 3.2). The second argument is a pointer to a `janus_Pconf` structure: the specific firmware will be automatically selected depending on the contents of the `pconf` structure. The remaining arguments `Fsp`, `Nsp`, `spmask` allow for SP inclusion and skipping as described in sections ???.

Status and Wait

```
int janus_status_safe (struct janus_dev* idev, char* STATUS,
                     unsigned long long *NITER,
                     int Fsp, int Nsp, char* SPmask);
int janus_status      (struct janus_dev* idev, char* STATUS,
                     unsigned long long *NITER,
                     int Fsp, int Nsp, char* SPmask);
int janus_wait_safe   (struct janus_dev* idev, char* SPmask);
int janus_wait        (struct janus_dev* idev, char* SPmask);
```

These functions permit polling the status of the run on SP nodes. As usual, the first argument is a JANUS device structure pointer. The second argument to `janus_status` is an array of `idev->splen` chars. The third argument is an array of `idev->splen` unsigned long long values. These two arrays will be eventually overwritten by the function itself. The choice of which SPs to check for is as usual performed specifying a valid SP mask pointer as the sixth argument (in this case `Fsp` and `Nsp` will be ignored). Equivalently, one may specify a NULL SP mask and specify `Nsp` nodes to check for, starting with the position `Fsp` in the `idev->spvec` array. The function returns negative value in case of problems. On success, 0 is returned and arrays `STATUS` and `NITER` are set. A `STATUS[n]` value of 1 indicates that SP corresponding to `idev->spvec[n]` is running. A `STATUS[n]` value of 0 means that the SP corresponding to `idev->spvec[n]` is either idle or was not included in the input selection by `Fsp` and `Nsp` or `spmask`. For each checked SP, the corresponding value of `NITER` is set to the value of the internal Parallel Tempering steps counter.

The `janus_wait` function polls all SPs specified by the `spmask` not returning until all are idle. A return value less than zero means problems during polling, and some selected SPs might still be running.

IMPORTANT: The `_safe` versions of these functions are mandatory if you are uploading firmwares with the `janus_i3d_load` function. If you are uploading older firmware or firmware that does not include the DDR fault correction, normal versions are recommended.

Run control

```
int janus_start      (struct janus_dev* idev, int Fsp, int Nsp, char* SPmask);
int janus_stop      (struct janus_dev* idev, int Fsp, int Nsp, char* SPmask);
int janus_reset     (struct janus_dev* idev, int Fsp, int Nsp, char* SPmask);
int janus_continue  (struct janus_dev* idev, int Fsp, int Nsp, char* SPmask);
int janus_wrptfr    (struct janus_dev* idev, unsigned long long niter,
                    int Fsp, int Nsp, char* SPmask)
int janus_wrptn     (struct janus_dev* idev, unsigned int niter,
                    int Fsp, int Nsp, char* SPmask)
```

These functions allow run control by starting and stopping the update machinery in the i3d firmware. Selection of SPs works as usual with the `int Fsp, int Nsp, char SPmask` arguments (see above). `janus_reset` will reset all counters (e.g. the total Monte Carlo steps, lattice span indexes, beta indexes etc.). You better reset only idle SPs (after they have been checked idle).

`janus_start` will start the update machinery in the selected SPs. An SP will be running until the internal Parallel Tempering (PT) steps counter match the Measure Frequency register (see the `janus_wrptn` function below).

The `janus_stop` and `janus_continue` functions respectively stop and resume the Monte Carlo update with no internal counter reset. Their use is deprecated.

`janus_wrptfr` writes the `niter` argument in the PT Frequency register inside each SP selected by the `int Fsp, int Nsp, char SPmask` arguments. This is the number of local update lattice sweeps (HeatBath for Ising systems, Metropolis for Potts with more than 2 states) between two consecutive Parallel Tempering steps.

`janus_wrptn` writes the internal Measure Frequency register of selected SPs. After an SP is started, it won't stop until the internal PT step counter will match this register, or a `janus_stop` or a `janus_reset` is called. **IMPORTANT:** Pay attention in not setting the Measure Frequency register to a value equal or lesser than the actual value of the PT step counter, as it will result in very long runs (`janus_wait_safe` may wait a very very long time before returning).

Very Very Important: always issue a `janus_reset` before any `janus_start`. This ensures that all internal machinery counters are properly reset. As a side consequence, The Monte Carlo step counter and the PT step counter will be reset to zero.

Uploading fields and Doenloading spin configurations

After initializing the janus device and loading SPs with the proper firmware, you have to tell them how many temperature replicas you are going to simulate, before any other operation. This is done by the function

```
int janus_wnbconf (struct janus_dev* idev, struct janus_Pconf* Pconf,
                  int Fsp, int Nsp, char* SPmask);
```

The first argument is a valid initialized janus device structure pointer; the second argument is a pointer to a `janus_Pconf` structure, containing the information about the number of temperatures. As usual `Fsp, Nsp, SPmask` allow for action on a subset of reserved SPs. If `SPmask` is not `NULL`, the function ignores `Fsp, Nsp`. Next, you have to initialize the Parallel Tempering beta array. The function

```
int janus_wbetas (struct janus_dev* idev, float* Betas, struct janus_Pconf* Pconf,
                 int Fsp, int Nsp, char* SPmask, int Pflag);
```

accepts a valid initialized janus device as first argument. The second argument is an array of float (32 bit) floating point numbers, representing inverse temperatures. As usual, the target SPs amongst the ones in the `idev->spvec` array are selected by means of the `Fsp`, `Nsp`, `spmask` arguments. If `SPmask` is not NULL, the function ignores `Fsp`, `Nsp`. Being either `NUM=Nsp` or the number of non-zero elements of `spmask`, the behavior is governed by the `Pflag` argument. Possible values to be passed to `Pflag` are the macros `PARALLEL` and `SEQUENTIAL`. With `PARALLEL`, all selected SPs will receive the same inverse temperatures set. The `Betas` array must then contain `pconf->Nbetas` 32-bit floating point numbers. If `SEQUENTIAL` is selected, then `pconf->Nbetas×NUM` values are expected, and the first `pconf->Nbetas` values will be sent to the first selected SPs, the second block of `pconf->Nbetas` values to the second selected SP and so on.

`PARALLEL` and `SEQUENTIAL` and their shorthands `PAR` and `SEQ` are macros defined in `janusppt3d.h`

Important: each SP should receive the set of beta values in increasing or decreasing order, depending on what you prefer for the Parallel Tempering algorithm. The SP does not provide an internal ordering feature for the beta array.

You should also call the `janus_init_bindex` before any further operation (see sec. 3.9.2).

The following function manage spin and coupling configurations.

```
int janus_ws      (struct janus_dev* idev, char* Sa, struct janus_Pconf* Pconf,
                  int Fsp, int Nsp, char* spmask, int Pflag);
int janus_rs      (struct janus_dev* idev, char* Sa, struct janus_Pconf* Pconf,
                  int Fsp, int Nsp, char* spmask, int Pflag);
int janus_wj      (struct janus_dev* idev, char* Jx, char* Jy, char* Jz, struct janus_Pconf* pconf,
                  int Fsp, int Nsp, char* spmask, int Pflag);
```

`janus_ws` and `janus_wj` respectively upload spin and coupling configurations. The `janus_rs` permits downloading of spin configurations. All of them accept as first argument `idev` is a pointer to valid initialized `struct janus_dev`.

The `janus_ws` function requires only one char buffer pointer, `Sa`, The linear size of the cubic lattice, number of spin states and number of beta values must be passed as the `Pconf` argument. SP selection is performed by means of either the couple of arguments `Fsp`, `Nsp` or specifying a non-NULL pointer as `spmask`; if `spmask` is non-NULL, the function ignores the passed values for `Fsp`, `Nsp`. If `spmask` is NULL, then the functions check for which SP id corresponds to position `Fsp` in the `idev->spvec` array. Then it will mark as active all subsequent SP id corresponding to positions `Fsp+1`, `Fsp+1` up to `Fsp+Nsp`. If `spmask` is non-NULL, it must be a pointer to an array of `idev->splen` chars; an `spmask[i]` value of 1 marks the SP whose id resides in `idev->spvec[i]` as active; a value of zero marks it as inactive. The behavior is finally determined by the `Pflag` argument.

If `Pflag` is `PARALLEL`, the functions expects `Pconf->Size×Pconf->Size×Pconf->Size×Pconf->Nbetas` bytes in the `Sa` buffer, which has been previously initialized by the user to the values of the spins (one spin per byte) with values 0 for down spins and value 1 for up ones. The same spin configurations for all beta values will be then uploaded to all active SPs.

If `Pflag` is set to `SEQUENTIAL`, the function expects `Pconf->Size×Pconf->Size×Pconf->Size×Pconf->Nbetas×NUM` bytes, with `NUM` the number of active SPs (i.e. either `Nsp` or the number of ones in the `spmask` array), in the `Sa` buffer. The function then will upload the first `Size×Size×Size` bytes of `Sa` to the first active SP, the second block of `Size×Size×Size` bytes to the second active SP and so on.

Inside each block, spins in the same temperature replica are consecutive bytes; inside each temperature replica portion of the buffer, the index $i = 0, \dots, Pconf->Size \times Pconf->Size \times Pconf->Size - 1$ of the byte maps to conventional cubic 3d lattice coordinates as $i = x + Pconf->Size \times y + Pconf->Size \times Pconf->Size \times z$.

`PARALLEL` and `SEQUENTIAL` and their shorthands `PAR` and `SEQ` are macros defined in `janusppt3d.h`

`janus_rs` performs the inverse operation. As for `janus_ws`, `Fsp`, `Nsp`, `spmask` selectively activate SPs in the `idev->spvec` array. `Sa` must point to a buffer of `Pconf->Size×Pconf->Size×Pconf->Size×Pconf->Nbetas×NUM` bytes, with `NUM` the number of active SPs (i.e. either `Nsp` or the number of ones in the `spmask` array).

The `Pflag` argument is completely ignored as all reads are intrinsically sequential. Upon successful returning, the buffer `Sa` will contain spin configurations for the `Pconf->Nbetas` replicas of all active SPs, in consecutive blocks of `Pconf->Size×Pconf->Size×Pconf->Size×Pconf->Nbetas` bytes, one byte per spin. **Important: after performing Parallel Tempering, the SP does not reorder configurations depending on the swapped temperatures. Each replica maintain its replica index and ordering inside the SP. What the SP interchange are the beta values, whose new order can be fetched at any time with the `janus_rd_beta_index` function described below in the following section.**

`janus_wj` functions behaves exactly the same as `janus_ws`, apart for the number and meaning of supplied buffers; it uploads quenched coupling configurations. The three buffers `Jx`, `Jy` and `Jz` respectively contain couplings along the positive direction of the conventional axis x, y, z . You have to provide only one set of `Pconf->Size×Pconf->Size×Pconf->Size` bytes per SP, as (no surprise) all the `Pconf->Nbetas` replicas inside each SPs share the same coupling configurations.

To upload random number generator seeds, use the function:

```
int janus_wr      (struct janus_dev* idev, JANUSPPT3D_RANDOMWORD* RNG,
                  int Fsp,      int Nsp, char* SPmask, int Pflag);
```

The semantics is identical to the one described at the end of sec. 3.9.1, except that the array passed as `RNG` must be declared as an array of `RNG_SAFE_SIZE` elements of type `JANUSPPT3D_RANDOMWORD`. Byte size of this internal data type may be obtained by usual `sizeof` directives.

Monitoring the Parallel Tempering procedure

Some functions permit managing and monitoring the status of the Parallel Tempering internal counters.

```
int janus_rd_beta_index (struct janus_dev* idev, short int* beta, struct janus_Pconf* Pconf,
                        int Fsp, int Nsp, char* SPmask);
```

This function permits keeping track of the beta value of each beta replica. The SP does not swap configuration, nor actually swaps beta values. It keeps an internal record of “who is where”. The index of the record is the index of the replica, and reflects the order in which they have been initially uploaded to the FPGA. The value k stored in the record at index j says that the replica number j is at the moment visiting the beta value whose index is k . Such beta-index array is returned by the function in the `beta` argument array. Such array must be large enough to contain `Pconf->Nbetas×NUM short int` values. `NUM` is determined depending on the SP selection performed by specifying the `Fsp`, `Nsp`, `SPmask` arguments as explained in the previous section.

The following function returns more information on what’s the Parallel Tempering is going on

```
int janus_sumindex (struct janus_dev* idev, unsigned int *s, unsigned long long int *s2, unsigned long long int *s3,
                   struct janus_Pconf* Pconf, int Fsp, int Nsp, char* SPmask);
```

The semantics is identical as in the case of the `janus_rd_beta_index` function above, apart that returns data in three different buffers: i) in the buffer `s`, in order of replica indexes, returns the sum of all visited betas indexes; ii) in the buffer `s2`, in order of replica indexes, returns the sum of all visited betas indexes squared; iii) in the buffer `s3`, in order of replica indexes, returns the sum of all visited betas indexes cubed;

At any time you may want to reset the internal “who is where” record and the sum buffers. (you are changing quenched disorder sample, for instance). The following function accomplishes the task:

```
int janus_init_bindex (struct janus_dev* idev,
                      int Fsp, int Nsp, char* SPmask);
```

`Fsp`, `Nsp`, `SPmask` arguments as usual permit action on a subset of reserved SPs as listed in `idev->spvec`. **Important: a call of this function for all reserved SPs is mandatory after setting the number of temperature replicas and uploading the beta values arrays.**

Update algorithm LUTs

Transition probabilities are represented as arrays of LUTSIZE integers (LUTs) the size of such integers is the size of the internal type JANUSPPT3D_LUTWORD. The macro LUTSIZE (usually 16) is defined in janusi3d.h.

```
int janus_wl      (struct janus_dev* idev, JANUSPPT3D_LUTWORD* Table, struct janus_Pconf* Pconf,
                  int Fsp, int Nsp, char* SPmask, int Pflag);
```

The `janus_wl` function behaves similarly as the one described in sec. ???. The first argument is a pointer to a valid initialized janus device. the third argument `Pconf` is a pointer to a `janus_Pconf` structure. The second argument is a `JANUSPPT3D_LUTWORD` buffer pointer `Table`; SP selection is performed by means of either the couple of arguments `Fsp`, `Nsp` or specifying a non-NULL pointer as `spmask`; if `spmask` is non-NULL, the function ignores the passed values for `Fsp`, `Nsp`. If `spmask` is NULL, then the functions check for which SP id corresponds to position `Fsp` in the `idev->spvec` array. Then it will mark as active all subsequent SP id corresponding to positions `Fsp+1`, `Fsp+1` up to `Fsp+Nsp`. If `spmask` is non-NULL, it must be a pointer to an array of `idev->splen` chars; an `spmask[i]` value of 1 marks the SP whose id resides in `idev->spvec[i]` as active; a value of zero marks it as inactive. The behavior is finally determined by the `Pflag` argument.

If `Pflag` is `PARALLEL`, the functions expects `LUTSIZE×Pconf->Nbetas` `JANUSPPT3D_LUTWORD` integers in the `Table` buffer, which have been previously initialized by the user to the correct LUT values. The same `Pconf->Nbetas` LUTs will be then uploaded to all active SPs. The LUTs must be calculated and ordered following the order in which beta values have been uploaded to the SPs, as described for the `janus_wbetas` function above. The SP does not perform any check for the LUT being ordered as the beta array.

Important: just as `JANUSPPT3D_LUTWORD` is an internal type and its size is not known a priori, a way for defining the normalization constant when computing [0,1] probabilities and renormalizing in a transparent way is:

```
JANUSPPT3D_LUTWORD Lnorm;
Lnorm=~((JANUSPPT3D_LUTWORD)0);
```

Looking at `janusppt3d.h` and figuring out what the typedef for `JANUSPPT3D_LUTWORD` is and using it directly will be considered as cheating, and the penalty is your program not functioning next week when we'll change that typedef.

If `Pflag` is set to `SEQUENTIAL`, the function expects `LUTSIZE×Pconf->Nbetas×NUM` `JANUSPPT3D_LUTWORD` integers in the `Table` array, with `NUM` the number of active SPs (i.e. either `Nsp` or the number of ones in the `spmask` array). The function then will upload the first block of `LUTSIZE×Pconf->Nbetas` `JANUSPPT3D_LUTWORD` integers to the first active SP, the second block of `LUTSIZE` `JANUSPPT3D_LUTWORD` integers to the second active SP and so on.

It would be very nice to have a function like that:

```
int janus_generate_lut      (unsigned int* Table, float* Beta);
```

which will be next coming. Meanwhile, ask us for tested routines to compute well-ordered LUTs.

3.10 Reserved names

This section lists all names that you should avoid in your programs as they are names for macros defined in the `jlib` or `josd` headers. Blah Blah (until we complete this section you may find them in `.h` files under `/usr/local/jos/usr/include`)