Developer's Guide

to

the PARI library

(version 2.11.0)

The PARI Group

Institut de Mathématiques de Bordeaux, UMR 5251 du CNRS.
Université de Bordeaux, 351 Cours de la Libération
F-33405 TALENCE Cedex, FRANCE
`e-mail: pari@math.u-bordeaux.fr`

Home Page:
`http://pari.math.u-bordeaux.fr/`

Table of Contents

Chapter 1:
Work in progress

This draft documents private internal functions and structures for hard-core PARI developers. Anything in here is liable to change on short notice. Don't use anything in the present document, unless you are implementing new features for the PARI library. Try to fix the interfaces before using them, or document them in a better way. If you find an undocumented hack somewhere, add it here.

Hopefully, this will eventually document everything that we buried in `paripriv.h` or even more private header files like `anal.h`. Possibly, even implementation choices! Way to go.

1.1 The type `t_CLOSURE`.

This type holds closures and functions in compiled form, so is deeply linked to the internals of the GP compiler and evaluator. The length of this type can be 6, 7 or 8 depending whether the object is an "inline closure", a "function" or a "true closure".

A function is a regular GP function. The GP input line is treated as a function of arity 0.

A true closure is a GP function defined in a non-empty lexical context.

An inline closure is a closure that appears in the code without the preceding `->` token. They are generally attached to the prototype code 'E' and 'I'. Inline closures can only exist as data of other closures, see below.

In the following example,

```
f(a=Euler)=x->sin(x+a);
g=f(Pi/2);
plot(x=0,2*Pi,g(x))
```

`f` is a function, `g` is a true closure and both `Euler` and `g(x)` are inline closures.

This type has a second codeword `z[1]`, which is the arity of the function or closure. This is zero for inline closures. To access it, use

`long closure_arity(GEN C)`

• `z[2]` points to a `t_STR` which holds the opcodes. To access it, use

`GEN closure_get_code(GEN C)`.

`const char * closure_codestr(GEN C)` returns as an array of `char` starting at 1.

• `z[3]` points to a `t_VECSMALL` which holds the operands of the opcodes. To access it, use

`GEN closure_get_oper(GEN C)`

• `z[4]` points to a `t_VEC` which hold the data referenced by the `pushgen` opcodes, which can be `t_CLOSURE`, and in particular inline closures. To access it, use

`GEN closure_get_data(GEN C)`

• `z[5]` points to a `t_VEC` which hold extra data needed for error-reporting and debugging. See Section 1.1.1 for details. To access it, use

`GEN closure_get_dbg(GEN C)`

Additionally, for functions and true closures,

• `z[6]` usually points to a `t_VEC` with two components which are `t_STR`. The first one displays the list of arguments of the closure without the enclosing parentheses, the second one the GP code of the function at the right of the `->` token. They are used to display the closure, either in implicit or explicit form. However for closures that were not generated from GP code, `z[6]` can point to a `t_STR` instead. To access it, use

`GEN closure_get_text(GEN C)`

Additionally, for true closure,

• `z[7]` points to a `t_VEC` which holds the values of all lexical variables defined in the scope the closure was defined. To access it, use

`GEN closure_get_frame(GEN C)`

1.1.1 Debugging information in closure.

Every `t_CLOSURE` object `z` has a component `dbg=z[5]` which hold extra data needed for error-reporting and debugging. The object `dbg` is a `t_VEC` with 3 components:

`dbg[1]` is a `t_VECSMALL` of the same length than `z[3]`. For each opcode, it holds the position of the corresponding GP source code in the strings stored in `z[6]` for function or true closures, positive indices referring to the second strings, and negative indices referring to the first strings, the last element being indexed as -1. For inline closures, the string of the parent function or true closure is used instead.

`dbg[2]` is a `t_VECSMALL` that lists opcodes index where new lexical local variables are created. The value 0 denotes the position before the first offset and variables created by the prototype code 'V'.

`dbg[3]` is a `t_VEC` of `t_VECSMALL`s that give the list of `entree*` of the lexical local variables created at a given index in `dbg[2]`.

1.2 The type `t_LIST`.

This type needs to go through various hoops to support GP's inconvenient memory model. Don't use `t_LIST`s in pure library mode, reimplement ordinary lists! This dynamic type is implemented by a `GEN` of length 3: two codewords and a vector containing the actual entries. In a normal setup (a finished list, ready to be used),

• the vector is malloc'ed, so that it can be realloc'ated without moving the parent `GEN`.

• all the entries are clones, possibly with cloned subcomponents; they must be deleted with `gunclone_deep`, not `gunclone`.

The following macros are proper lvalues and access the components

`long list_nmax(GEN L)`: current maximal number of elements. This grows as needed.

GEN list_data(GEN L): the elements. If v = list_data(L), then either v is NULL (empty list) or l = lg(v) is defined, and the elements are v[1], ..., v[l-1].

In most gerepile scenarios, the list components are not inspected and a shallow copy of the malloc'ed vector is made. The functions gclone, copy_bin_canon are exceptions, and make a full copy of the list.

The main problem with lists is to avoid memory leaks; in the above setup, a statement like a = List(1) would already leak memory, since List(1) allocates memory, which is cloned (second allocation) when assigned to a; and the original list is lost. The solution we implemented is

• to create anonymous lists (from List, gtolist, concat or vecsort) entirely on the stack, *not* as described above, and to set list_nmax to 0. Such a list is not yet proper and trying to append elements to it fails:

```
? listput(List(),1)
  ***   variable name expected: listput(List(),1)
  ***                                  ^----------------
```

If we had been malloc'ing memory for the List([1,2,3]), it would have leaked already.

• as soon as a list is assigned to a variable (or a component thereof) by the GP evaluator, the assigned list is converted to the proper format (with list_nmax set) previously described.

GEN listcopy(GEN L) return a full copy of the t_LIST L, allocated on the *stack* (hence list_nmax is 0). Shortcut for gcopy.

GEN mklistcopy(GEN x) returns a list with a single element x, allocated on the stack. Used to implement most cases of gtolist (except vectors and lists).

A typical low-level construct:

```
long l;
/* assume L is a t_LIST */
L = list_data(L); /* discard t_LIST wrapper */
l = L? lg(L): 1;
for (i = 1; i < l; i++) output( gel(L, i) );
for (i = 1; i < l; i++) gel(L, i) = gclone( ... );
```

1.2.1 Maps as Lists.

GP's maps are implemented on top of t_LISTs so as to benefit from their peculiar memory models. Lists thus come in two subtypes: t_LIST_RAW (actual lists) and t_LIST_MAP (a map).

GEN mklist_typ(long t) create a list of subtype t. GEN mklist(void) is an alias for

```
mklist_typ(t_LIST_RAW);
```

and GEN mkmap(void) is an alias for

```
mklist_typ(t_LIST_MAP);
```

long list_typ(GEN L) return the list subtype, either t_LIST_RAW or t_LIST_MAP.

void listpop(GEN L, long index) as listpop0, assuming that L is a t_LIST_RAW.

GEN listput(GEN list, GEN object, long index) as listput0, assuming that L is a t_LIST_RAW.

`GEN mapdomain(GEN T)` vector of keys of the map T.

`GEN mapdomain_shallow(GEN T)` shallow version of `mapdomain`.

`GEN maptomat(GEN T)` convert a map to a factorization matrix.

`GEN maptomat_shallow(GEN T)` shallow version of `maptomat`.

1.3 Protection of non-interruptible code.

GP allows the user to interrupt a computation by issuing SIGINT (usually by entering control-C) or SIGALRM (usually using alarm()). To avoid such interruption to occurs in section of code which are not reentrant (in particular `malloc` and `free`) the following mechanism is provided:

`BLOCK_SIGINT_START()` Start a non-interruptible block code. Block both `SIGINT` and `SIGARLM`.

`BLOCK_SIGALRM_START()` Start a non-interruptible block code. Block only `SIGARLM`. This is used in the `SIGINT` handler itself to delay an eventual pending alarm.

`BLOCK_SIGINT_END()` End a non-interruptible block code

The above macros make use of the following global variables:

`PARI_SIGINT_block`: set to 1 (resp. 2) by `BLOCK_SIGINT_START` (resp. `BLOCK_SIGALRM_START`).

`PARI_SIGINT_pending`: Either 0 (no signal was blocked), `SIGINT` (`SIGINT` was blocked) or `SIGALRM` (`SIGALRM` was blocked). This need to be set by the signal handler.

Within a block, an automatic variable `int block` is defined which records the value of `PARI_SIGINT_block` when entering the block.

1.3.1 Multithread interruptions.

To support multithreaded programs, `BLOCK_SIGINT_START` and `BLOCK_SIGALRM_START` call `MT_SIGINT_BLOCK(block)`, and `BLOCK_SIGINT_END` calls `MT_SIGINT_UNBLOCK(block)`.

`MT_SIGINT_BLOCK` and `MT_SIGINT_UNBLOCK` are defined by the multithread engine. They can calls the following public functions defined by the multithread engine.

`void mt_sigint_block(void)`

`void mt_sigint_unblock(void)`

In practice this mechanism is used by the POSIX thread engine to protect against asychronous cancellation.

1.4 $\mathbf{F}_{l^2}$ field for small primes l.

Let $l > 2$ be a prime `ulong`. A `Fl2` is an element of the finite field $\mathbf{F}_{l^2}$ represented (currently) by a `Flx` of degree at most 1 modulo a polynomial of the form $x^2 - D$ for some non square $0 \le D < p$. Below `pi` denotes the pseudo inverse of `p`, see `Fl_mul_pre`

`int Fl2_equal1(GEN x)` return 1 if $x = 1$, else return 0.

`GEN Fl2_mul_pre(GEN x, GEN y, ulong D, ulong p, ulong pi)` return xy.

`GEN Fl2_sqr_pre(GEN x, ulong D, ulong p, ulong pi)` return x^2.

`GEN Fl2_inv_pre(GEN x, ulong D, ulong p, ulong pi)` return x^{-1}.

`GEN Fl2_pow_pre(GEN x, GEN n, ulong D, ulong p, ulong pi)` return x^n.

`GEN Fl2_sqrtn_pre(GEN a, GEN n, ulong D, ulong p, ulong pi, GEN *zeta)` n-th root, as `Flxq_sqrtn`.

`GEN Fl2_norm_pre(GEN x, GEN n, ulong D, ulong p, ulong pi)` return the norm of x.

`GEN Flx_Fl2_eval_pre(GEN P, GEN x, ulong D, ulong p, ulong pi)` return $P(x)$.

1.5 Public functions useless outside of GP context.

These functions implement GP functionality for which the C language or other libpari routines provide a better equivalent; or which are so tied to the `gp` interpreter as to be virtually useless in `libpari`. Some may be generated by `gp2c`. We document them here for completeness.

1.5.1 Conversions.

`GEN toser_i(GEN x)` internal shallow function, used to implement automatic conversions to power series in GP (as in `cos(x)`). Converts a `t_POL` or a `t_RFRAC` to a `t_SER` in the same variable and precision `precdl` (the global variable corresponding to `seriesprecision`). Returns x itself for a `t_SER`, and `NULL` for other argument types. The fact that it uses a global variable makes it awkward whenever you're not implementing a new transcendental function in GP. Use `RgX_to_ser` or `rfrac_to_ser` for a fast clean alternative to `gtoser`.

`GEN listinit(GEN x)` a `t_LIST` (from `List` or `Map`) may exist in two different forms due to GP memory model:

• an ordinary *read-only* copy on the pari stack (as produced by `gtolist` or `gtomap`) to which one may not assign elements (`listput` will fail) unless the list is empty.

• a feature-complete GP list using (malloc'ed) `block`s to allow dynamic insertions. An empty list is automaticaly promoted to this status on first insertion.

The `listinit` function creates a copy of existing `t_SER` x and makes sure it is of the second kind. Variants of this are automatically called by `gp` when assigning a `t_LIST` to a GP variable; the mecanism avoid memory leaks when creating a constant list, e.g. `List([1,2,3])` (read-only), without assigning it to a variable. Whereas after `L = List([1,2,3])` (GP list), we keep a pointer to the object and may delete it when L goes out of scope.

This `libpari` function allows `gp2c` to simulate this process by generating `listinit` calls at appropriate places.

1.5.2 Output.

`void print0(GEN g, long flag)` internal function underlying the `print` GP function. Prints the entries of the `t_VEC` g, one by one, without any separator; entries of type `t_STR` are printed without enclosing quotes. *flag*is one of `f_RAW`, `f_PRETTYMAT` or `f_TEX`, using the current default output context.

`void out_print0(PariOUT *out, const char *sep, GEN g, long flag)` as `print0`, using output context `out` and separator `sep` between successive entries (no separator if `NULL`).

`void printsep(const char *s, GEN g, long flag)` `out_print0` on `pariOut` followed by a newline.

`void printsep1(const char *s, GEN g, long flag)` `out_print0` on `pariOut`.

`char* pari_sprint0(const char *s, GEN g, long flag)` displays s, then `print0(g, flag)`.

`void print(GEN g)` equivalent to `print0(g, f_RAW)`, followed by a `\n` then an `fflush`.

`void printp(GEN g)` equivalent to `print0(g, f_PRETTYMAT)`, followed by a `\n` then an `fflush`.

`void print1(GEN g)` as above, without the `\n`. Use `pari_printf` or `output` instead.

`void printtex(GEN g)` equivalent to `print0(g, t_TEX)`, followed by a `\n` then an `fflush`. Use `GENtoTeXstr` and `pari_printf` instead.

`void write0(const char *s, GEN g)`

`void write1(const char *s, GEN g)` use `fprintf`

`void writetex(const char *s, GEN g)` use `GENtoTeXstr` and `fprintf`.

`void printf0(GEN fmt, GEN args)` use `pari_printf`.

`GEN Strprintf(GEN fmt, GEN args)` use `pari_sprintf`.

1.5.3 Input.

`gp`'s input is read from the stream `pari_infile`, which is changed using

`FILE* switchin(const char *name)`

Note that this function is quite complicated, maintaining stacks of files to allow smooth error recovery and `gp` interaction. You will be better off using `gp_read_file`.

1.5.4 Control flow statements.

`GEN break0(long n)`. Use the C control statement `break`. Since `break(2)` is invalid in C, either rework your code or use `goto`.

`GEN next0(long n)`. Use the C control statement `continue`. Since `continue(2)` is invalid in C, either rework your code or use `goto`.

`GEN return0(GEN x)`. Use `return`!

`void error0(GEN g)`. Use `pari_err(e_USER,)`

`void warning0(GEN g)`. Use `pari_warn(e_USER,)`

1.5.5 Accessors.

`GEN vecslice0(GEN A, long a, long b)` implements $A[a..b]$.

`GEN matslice0(GEN A, long a, long b, long c, long d)` implements $A[a..b, c..d]$.

1.5.6 Iterators.

`GEN apply0(GEN f, GEN A)` gp wrapper calling `genapply`, where f is a `t_CLOSURE`, applied to A. Use `genapply` or a standard C loop.

`GEN select0(GEN f, GEN A)` gp wrapper calling `genselect`, where f is a `t_CLOSURE` selecting from A. Use `genselect` or a standard C loop.

`GEN vecapply(void *E, GEN (*f)(void* E, GEN x), GEN x)` implements `[a(x)|x<-b]`.

`GEN veccatapply(void *E, GEN (*f)(void* E, GEN x), GEN x)` implements `concat([a(x)|x<-b])` which used to implement `[a0(x,y)|x<-b;y<-c(b)]` which is equal to `concat([[a0(x,y)|y<-c(b)]|x<-b])`.

`GEN vecselect(void *E, long (*f)(void* E, GEN x), GEN A)` implements `[x<-b,c(x)]`.

`GEN vecselapply(void *Epred, long (*pred)(void* E, GEN x), void *Efun, GEN (*fun)(void* E, GEN x), GEN A)` implements `[a(x)|x<-b,c(x)]`.

1.5.7 Local precision.

Theses functions allow to change `realprecision` locally when calling the GP interpretor.

`void push_localprec(long p)` set the local precision to p.

`void push_localbitprec(long b)` set the local precision to b bits.

`void pop_localprec(void)` reset the local precision to the previous value.

`long get_localprec(void)` returns the current local precision.

`long get_localbitprec(void)` returns the current local precision in bits.

`void localprec(long p)` trivial wrapper around `push_localprec` (sanity checks and convert from decimal digits to a number of codewords). Use `push_localprec`.

`void localbitprec(long p)` trivial wrapper around `push_localbitprec` (sanity checks). Use `push_localbitprec`.

1.5.8 Functions related to the GP evaluator.

The prototype code `C` instructs the GP compiler to save the current lexical context (pairs made of a lexical variable name and its value) in a `GEN`, called `pack` in the sequel. This `pack` can be used to evaluate expressions in the corresponding lexical context, providing it is current.

`GEN localvars_read_str(const char *s, GEN pack)` evaluate the string s in the lexical context given by `pack`. Used by `geval_gp` in GP to implement the behavior below:

```
? my(z=3);eval("z=z^2");z
%1 = 9
```

`long localvars_find(GEN pack, entree *ep)` does `pack` contain a pair whose variable corresponds to `ep`? If so, where is the corresponding value? (returns an offset on the value stack).

1.5.9 Miscellaneous.

`char* os_getenv(const char *s)` either calls `getenv`, or directly return `NULL` if the `libc` does not provide it. Use `getenv`.

`sighandler_t os_signal(int sig, pari_sighandler_t fun)` after a

```
typedef void (*pari_sighandler_t)(int);
```

(private type, not exported). Installs signal handler `fun` for signal `sig`, using `sigaction` with flag `SA_NODEFER`. If `sigaction` is not available use `signal`. If even the latter is not available, just return `SIG_IGN`. Use `sigaction`.

1.6 Embedded GP interpretor.

These function provide a simplified interface to embed a GP interpretor in a program.

`void gp_embedded_init(long rsize, long vsize)` Initialize the GP interpretor (like `pari_init` does) with `parisize=rsize rsize` and `parisizemax=vsize`.

`char * gp_embedded(const char *s)` Evaluate the string `s` with GP and return the result as a string, in a format similar to what GP displays (with the history index). The resulting string is allocated on the PARI stack, so subsequent call to `gp_embedded` will destroy it.

1.7 Readline interface.

Code which wants to use libpari readline (such as the Jupyter notebook) needs to do the following:

```
#include <readline.h>
#include <paripriv.h>
pari_rl_interface S;
...
pari_use_readline(S);
```

The variable S, as initialized above, encapsulates the libpari readline interface. (And allow us to move gp's readline code to libpari without introducing a mandatory dependency on readline in libpari.) The following functions then become available:

`char** pari_completion_matches(pari_rl_interface *pS, const char *s, long pos, long *wordpos)` given a command string s, where the cursor is at index `pos`, return an array of completion matches.

If `wordpos` is not `NULL`, set `*wordpos` to the index for the start of the expression we complete.

`char** pari_completion(pari_rl_interface *pS, char *text, int start, int end)` the low-level completer called by `pari_completion_matches`. The following wrapper

```
char**
gp_completion(char *text, int START, int END)
{ return pari_completion(&S, text, START, END);)
```

is a valid value for `rl_attempted_completion_function`.

1.8 Constructors called by `pari_init` functions.

`void pari_init_buffers()`

`void pari_init_compiler()`

`void pari_init_defaults()`

`void pari_init_evaluator()`

`void pari_init_files()`

`void pari_init_floats()`

`void pari_init_graphics()`

`void pari_init_homedir()`

`void pari_init_parser()`

`void pari_init_paths()`

`void pari_init_primetab()`

`void pari_init_rand()`

`void pari_init_seadata()`

1.9 Destructors called by `pari_close`.

`void pari_close_compiler()`

`void pari_close_evaluator()`

`void pari_close_files()`

`void pari_close_floats()`

`void pari_close_homedir()`

`void pari_close_mf()`

`void pari_close_parser()`

`void pari_close_paths()`

`void pari_close_primes()`

1.10 Constructors and destructors used by the `pthreads` interface.

- Called by `pari_thread_sync`

`void pari_pthread_init_primetab()`

`void pari_pthread_init_seadata()`

`void pari_pthread_init_varstate()`

- Called by `pari_thread_start`

`void pari_thread_init_primetab()`

`void pari_thread_init_seadata()`

`void pari_thread_init_varstate()`

- Called by `pari_thread_close`

`void pari_thread_close_files()`

Chapter 2: Regression tests, benches

This chapter documents how to write an automated test module, say `fun`, so that `make test-fun` executes the statements in the `fun` module and times them, compares the output to a template, and prints an error message if they do not match.

• Pick a *new* name for your test, say `fun`, and write down a GP script named `fun`. Make sure it produces some useful output and tests adequately a set of routines.

• The script should not be too long: one minute runs should be enough. Try to break your script into independent easily reproducible tests, this way regressions are easier to debug; e.g. include `setrand(1)` statement before a randomized computation. The expected output may be different on 32-bit and 64-bit machines but should otherwise be platform-independent. If possible, the output shouldn't even depend on `sizeof(long)`; using a `realprecision` that exists on both 32-bit and 64-bit architectures, e.g. `\p 38` is a good first step. You can use `sizebyte(0)==16` to detect a 64-bit architecture and `sizebyte(0)==8` for 32-bit.

• Dump your script into `src/test/in/` and run `Configure`.

• `make test-fun` now runs the new test, producing a `[BUG]` error message and a `.dif` file in the relevant object directory `Oxxx`. In fact, we compared the output to a non-existing template, so this must fail.

• Now

```
patch -p1 < Oxxx/fun.dif
```

generates a template output in the right place `src/test/32/fun`, for instance on a 32-bit machine.

• If different output is expected on 32-bit and 64-bit machines, run the test on a 64-bit machine and patch again, thereby producing `src/test/64/fun`. If, on the contrary, the output must be the same (preferred behavior!), make sure the output template land in the `src/test/32/` directory which provides a default template when the 64-bit output file is missing; in particular move the file from `src/test/64/` to `src/test/32/` if the test was run on a 64-bit machine.

• You can now re-run the test to check for regressions: no `[BUG]` is expected this time! Of course you can at any time add some checks, and iterate the test / patch phases. In particular, each time a bug in the `fun` module is fixed, it is a good idea to add a minimal test case to the test suite.

• By default, your new test is now included in `make test-all`. If it is particularly annoying, e.g. opens tons of graphical windows as `make test-ploth` or just much longer than the recommended minute, you may edit `config/get_tests` and add the `fun` test to the list of excluded tests, in the `test_extra_out` variable.

• You can run a subset of existing tests by using the following idiom:

```
cd Oxxx     # call from relevant build directory
make TESTS="lfuntype lfun gamma" test-all
```

will run the `lfuntype`, `lfun` and `gamma` tests. This produces a combined output whereas the alternative

```
make test-lfuntype test-lfun test-gamma
```

would not.

• By default, the test is run on both the `gp-sta` and `gp-dyn` binaries, making it twice as slow. If the test is somewhat long, it can be annoying; you can restrict to one binary only using the `statest-all` or `dyntest-all` targets. Both accept the `TESTS` argument seen above.

```
make test-lfuntype test-lfun gamma
```

would not.

• Finally, the `get_tests` script also defines the recipe for `make bench` timings, via the variable `test_basic`. A test is included as `fun` or `fun_`n, where n is an integer ≤ 1000; the latter means that the timing is weighted by a factor $n/1000$. (This was introduced a long time ago, when the `nfields` bench was so much slower than the others that it hid slowdowns elsewhere.)

2.1 Functions for GP2C.

2.1.1 Functions for safe access to components.

Theses function returns the adress of the requested component after checking it is actually valid. This is used by GP2C -C.

`GEN* safegel(GEN x, long l)`, safe version of `gel(x,l)` for `t_VEC`, `t_COL` and `t_MAT`.

`long* safeel(GEN x, long l)`, safe version of `x[l]` for `t_VECSMALL`.

`GEN* safelistel(GEN x, long l)` safe access to `t_LIST` component.

`GEN* safegcoeff(GEN x, long a, long b)` safe version of `gcoeff(x,a, b)` for `t_MAT`.

Chapter 3:
Parallelism

3.1 The PARI MT interface.

PARI provides an abstraction for doing parallel computations.

`void mt_queue_start(struct pari_mt *pt, GEN worker)` Let `worker` be a `t_CLOSURE` object of arity 1. Initialize the structure `pt` to evaluate `worker` in parallel.

`void mt_queue_start_lim(struct pari_mt *pt, GEN worker, long lim)` as `mt_queue_start`, where `lim` is an upper bound on the number of `tasks` to perform. Concretly the number of threads will be `min(lim,nbthreads`. The values 0 and 1 of `lim` are special:

- 0: no limit, equivalent to `mt_queue_start`.
- 1: no parallelism. Evaluate the tasks sequentially.

`void mt_queue_submit(struct pari_mt *pt, long taskid, GEN task)` Submit `task` to be evaluated by `worker`, or `NULL` if no further task is left to be submitted. The value `taskid` is user-specified and allows to later match up results and submitted tasks.

`GEN mt_queue_get(struct pari_mt *pt, long *taskid, long *pending)` Return the result of the evaluation by `worker` of one of the previously submitted tasks. Set `pending` to the number of remaining pending tasks. Set `taskid` to the value associate to this task by `mt_queue_submit`. Returns `NULL` if more tasks need to be submitted.

`void mt_queue_end(struct pari_mt *pt)` End the parallel execution.

Calls to `mt_queue_submit` and `mt_queue_get` must alternate: each call to `mt_queue_submit` must be followed by a call to `mt_queue_get` before any other call to `mt_queue_submit`, and conversely.

The first call to `mt_queue_get` will return `NULL` until a sufficient number of tasks have been submitted. If no more tasks are left to be submitted, use

```
mt_queue_submit(handle, id, NULL)
```

to allow further calls to `mt_queue_get`. If `mt_queue_get` sets `pending` to 0, then no more tasks are pending and it is safe to call `mt_queue_end`.

The parameter `taskid` can be chosen arbitrarily. It is attached to a task but is not available to `worker`. It provides an efficient way to match a tasks and results. It is ignored when the parameter `task` is `NULL`.

3.1.1 Miscellaneous.

`void mt_broadcast(GEN code)`: do nothing unless the MPI threading engine is in use. In that case, it evaluates the closure `code` on all secondary nodes. This can be sued to change the states of the MPI child nodes. This is used by `install`.

3.2 Initialization.

This section is technical.

`void pari_mt_init(void)` When using MPI, it is sometimes necessary to run initialization code on the child nodes after PARI is initialized. This can be done as follow:

• call `pari_init_opts` with the flag `INIT_noIMTm`. This initializes PARI, but not the MT engine.

• call the required initialization code.

• call `pari_mt_init` to initialize the MT engine. Note that under MPI, this function only returns on the master node. On the child nodes, it enters slave mode. Thus it is no longer possible to run initialization code on the child nodes.

See the file `examples/pari-mt.c` for an example.

`void pari_mt_close(void)` When using MPI, calling `pari_close` will terminate the MPI execution environment. If this is undesirable, you should call `pari_close_opts` with the flag `INIT_noIMTm`. This closes PARI without terminating the MPI execution environment It is allowed to call `pari_mt_close` later to terminate it. Note that the once MPI is terminated it cannot be restarted, and that it is considered an error for a program to end without having terminated the MPI execution environment.

Index

SomeWord refers to PARI-GP concepts.
`SomeWord` is a PARI-GP keyword.
SomeWord is a generic index entry.

A

B

C

E

F

G

I

L

M

N