7 Foreign Interface Reference

The Foreign Interface Reference documents each function and macro of the foreign interface in alphabetical order.

7.0.1 Reference Page Format

The header of each page of the reference contains the following information

- function or macro name
- category of function or macro

After the header, a number of paragraphs appear, the first of which is the **NAME** paragraph. Not all of the paragraphs appear for every foreign interface function or macro, but **NAME**, **FORMS**, and **DESCRIPTION** always appear. The following is a list of the paragraphs, and there associated purposes:

- **NAME** — gives the name of the function or macro, along with what the it does.
- **FORMS** — shows how to call the function or macro using prototypes or example usage.
- **DESCRIPTION** — gives a description of what the function or macro does. The description goes into more detail than the information given in the **NAME** paragraph.
- **EXAMPLES** — shows some examples of the function or macro being used.
- **NOTES** — shows unusual or surprising details that should be noted by the reader.
- **SEE ALSO** — gives references to other information.

The Foreign Reference List contains a list of all the functions and macros, and the page numbers of their documentation.

7.0.2 Naming Conventions

The ALS Prolog naming conventions for the foreign interface functions and macros documented here are as follows:

- each function name and macro name begins with the prefix `PI`.
- the names of functions consist entirely of lowercase alphabetical characters except for the `PI` prefix.
- the names of macros consist entirely of uppercase alphabetical characters.

NAME

`PI_BEGIN`	- begins the definition of the PI procedure table
`PI_MODULE`	- specifies module where subsequent PI_DEFINEs will be placed
`PI_DEFINE`	- defines an entry in the PI procedure table
`PI_END`	- finishes the definition of the PI procedure table

FORMS

```
PI_BEGIN
    PI_MODULE(const char *modname)
    PI_DEFINE(const char *name, int arity,
                  int (*addr)(void))
    .
    .
PI_END
```

DESCRIPTION

`PI_BEGIN` is used as the opening bracket of the foreign interface procedure table. `PI_MODULE` specifies the module in which entries generated by following `PI_DEFINE` expressions will be placed; multiple occurrences of `PI_MODULE` can be used to control placement of predicates in different modules. By default, `PI_DEFINE` places its entry in the `user` module. `PI_DEFINE` defines an entry in the foreign interface procedure table. `PI_END` is used as the closing bracket of the foreign interface procedure table.

EXAMPLES

```
static int hello(void)
{
  PI_printf("Hello world\n");
  PI_SUCCEED;
}

static int printit(void)
```

```
{
  PWord a1;
  int t1;

  PI_getan(&a1,&t1,1);
  if (t1 != PI_INTEGER)
       PI_FAIL;
  PI_printf("It: %d\n", a1);
  PI_SUCCEED;
}

PI_BEGIN
  PI_MODULE("examplemod")
  PI_DEFINE("hello", 0, hello)
  PI_DEFINE("printit", 1, printit)
PI_END
```

SEE ALSO

PI_INIT

NAME

`PI_DOUBLE` - identifies an object as a Prolog floating point number

FORMS

```
int type;

if (type == PI_DOUBLE)
    do_something();
```

DESCRIPTION

The value associated with `PI_DOUBLE` is a Prolog double-precision floating point number. In order to use this value in C, the Prolog floating point number must be converted to a C double. This is done by using the `PI_getdouble` function. If you have a C double and want to create a Prolog double with the same value, use the `PI_makedouble` function.

EXAMPLES

```
PWord val; int type;
double cdouble;

if (type == PI_DOUBLE) {
       PI_getdouble(&cdouble, val);
       cdouble = cdouble + 3.14;
}
```

SEE ALSO

`PI_makedouble`, `PI_getdouble`

NAME

`PI_FAIL` - causes the current C-defined predicate to fail

FORMS

```
PI_FAIL
```

DESCRIPTION

`PI_FAIL` makes the current predicate fail. Control is immediately returned to Prolog. Failure is usually made to occur in C-defined predicates when a passed-in argument is of the incorrect type or value. Other reasons for failure include runnning out of space (in a table, for example), and an error in a system call.

EXAMPLES

```
if (type != PI_INTEGER)
        PI_FAIL;
```

NOTES

`PI_FAIL` is implemented as a macro which executes a `return()`. For this reason, `PI_FAIL` should only be used from the top level.

SEE ALSO

PI_SUCCEED

NAME

`PI_forceuia` - turn an uninterned atom into a symbol table entry

FORMS

```
char *PI_forceuia(PWord *val, int *type);
```

DESCRIPTION

Turns the specified uninterned atom into a symbol table entry. In other words, the atom becomes interned if it wasn't already. If the interning of the UIA was successful:

- the symbol value is placed in `val`
- `PI_SYM` is placed in `type`
- a pointer to the symbol table entry is returned

If the interning of the UIA was unsuccessful, a null pointer returned. `PI_forceuia` is useful if you want to pass the symbol's string to another C routine. You can't do this with a UIA because you can't get a pointer to the string. This is because UIAs are stored in the Prolog heap, and can be moved to other locations during garabage collection. Another problem that might occur would be if the UIA was no longer needed, Prolog would garbage collect that space. Unfortunately, your function (or a system function) would be pointing into the Prolog heap at the location of the old UIA. This could make for some mysterious bugs. The point is, you can use the Prolog symbol table as a refuge from the volatility of UIAs. However, be careful not to fill the symbol table with interned UIAs.

EXAMPLES

```
if (type == PI_UIA) {
        str = (PI_forceuia(&val,&type))
        makewindow(str);
}
```

SEE ALSO

PI_UIA, PI_makeuia, PI_getuianame, PI_makesym, PI_getsymname

NOTES

Currently, if the UIA is longer than 256 bytes, PI_forceuia will return NULL.

NAME

`PI_getan` - get an argument passed in from Prolog

FORMS

```
void PI_getan(PWord *val, int *type, int argnum);
```

DESCRIPTION

`PI_getan` is used to get the current values of the arguments in the Prolog goal. `argnum` is an integer designating the number of the argument in the Prolog goal to get. If you want to get the first argument in the goal, you should make the following call:

```
PI_getan(&val,&type,1);
```

`PI_getan` will place one of the following type constants in `type` :

```
PI_DOUBLE
PI_INT
PI_LIST
PI_STRUCT
PI_SYM
PI_UIA
PI_VAR
```

The contents of `val` will depend upon the associated type. Usually, this value will have to be taken apart to be useful.

EXAMPLES

```
static int test(void)
{
    PWord v1, v2;
    int t1, t2;

    PI_getan(&v1,&t1,1);
    PI_getan(&v2,&t2,2);
```

```
    if (t1 != PI_INT || t2 != PI_INT)
        PI_FAIL;
    PI_printf("Numbers: %d, %d\n", v1, v2);

    PI_SUCCEED;
}
```

SEE ALSO

PI_DOUBLE, PI_INT, PI_LIST, PI_STRUCT, PI_SYM, PI_UIA, PI_VAR

NAME

`PI_getargn` - get an argument of a structure

FORMS

```
void PI_getargn(PWord *val, int *type,
                PWord structval, int argnum);
```

DESCRIPTION

`PI_getargn` is used to get an argument of a structure. `structval` specifies the structure and `argnum` specifies the number of the argument to get. The value of the structure argument is placed in `value`, and the associated type is placed in type.

EXAMPLES

The following example gets the argument from the Prolog goal, and unifies a new structure with the argument. The new structure will look like:

```
fried(chris)
```

After the structure is created and unified with the incoming argument, `PI_getargn` is used to get the argument of the structure. Then the argument is unified with the symbol `chris`. This is how arguments are installed in structures.

```
static int test(void)
{
  PWord val, func, chris, struc, arg;
  int type, functype, christype, structype, argtype;

  PI_getan(&val, &type, 1);
  PI_makesym(&func, &functype, "fried");
  PI_makesym(&chris, &christype, "chris");
  PI_makestruct(&struc, &structype, func, 1);

  if (!PI_unify(struc, structype, val, type))
```

```
            PI_FAIL;
        PI_getargn(&arg, &argtype, struc, 1);
        if (!PI_unify(arg, argtype, chris, christype))
            PI_FAIL;
        PI_SUCCEED;
    }
```

SEE ALSO

Cf. `PI_getan` for the possible types returned by `PI_getargn`.

NAME

`PI_getdouble` - returns a C double given a Prolog double

FORMS

```
void PI_getdouble(double *cdouble, PWord doubleval);
```

DESCRIPTION

`PI_getdouble` is used to put a Prolog double precision floating point number into a C double. `doubleval` is a PWord specifying the Prolog double, and `cdouble` is used to store the C double. `PI_getdouble` is most often used so that the C-defined predicate can perform some arithmetic with the number. The floating point number can be translated back into Prolog using the `PI_makedouble` function.

EXAMPLES

The following computes the circumference of a circle given the radius:

```
static int circum(void)
{
  PWord v1, v2, pdouble;
  int t1, t2, pdoubletype;
  double radius;

  PI_getan(&v1, &t1, 1);
  PI_getan(&v2, &t2, 2);
  if (type == PI_INT)
       radius = v1;
  else if (type == PI_DOUBLE)
       PI_getdouble(&radius, v1);
  else
       PI_FAIL;
  PI_makedouble(&pdouble, &pdoubletype,
                 radius*2*3.14 );
  if (!PI_unify(pdouble, pdoubletype, v2, t2))
```

```
          PI_FAIL;
    PI_SUCCEED;
  }
```

SEE ALSO

PI_makedouble

NAME

`PI_gethead` - get the head of a list

FORMS

```
void PI_gethead(PWord *val, int *type, PWord listval);
```

DESCRIPTION

`PI_gethead` is used to get the head of the list cell specified by `listval`. The value of the head of the list is stored in `val`, and the associated type is stored in `type`.

EXAMPLES

```
static int gethead(void)
{
  PWord v1, v2, head;
  int t1, t2, headtype;

  PI_getan(&v1, &t1, 1);
  PI_getan(&v2, &t2, 2);

  if (t1 != PI_LIST)
       PI_FAIL;
  PI_gethead(&head, &headtype, v1);
  if (!PI_unify(head, headtype, v2, t2))
       PI_FAIL;

  PI_SUCCEED;
}
```

SEE ALSO

PI_makelist

NAME

`PI_getstruct` - get the functor and arity of a structure

FORMS

```
void PI_getstruct(PWord *func, int *arity,
                  PWord structval);
```

DESCRIPTION

`PI_getstruct` is used to get the functor and the arity of a structure specified by `structval`. The functor of the structure is stored in `func`, and is of type `PI_SYM`. The arity, an integer, is stored in `arity`. The arity is the number of arguments in the structure. The string associated with the functor can be found using the `PI_getsymname` function.

EXAMPLES

```
static int printstruct(void)
{
  PWord val, func;
  int type, arity;
  const char *name;

  PI_getan(&val, &type, 1);
  if (type != PI_STRUCT)
       PI_FAIL;
  PI_getstruct(&func, &arity, val);
  name = PI_getsymname(0, func, 0);
  PI_printf("Struct is: %s/%d\n", name, arity);
  PI_SUCCEED;
}
```

SEE ALSO

`PI_makestruct`, `PI_getsymname`

NAME

`PI_getsymname` - get the string representing the specified symbol

FORMS

```
char *PI_getsymname(char *buf, PWord symval,
                    int bufsize);
```

DESCRIPTION

`PI_getsymname` places the print name of the symbol specified by `symval` into the buffer pointed to by `buf`. Normally, `buf` is returned as the result of the function call. If the symbol name is longer than `bufsize` bytes, the buffer will not be filled, and 0 will be returned.

EXAMPLES

```
#define BUFSIZE 256
static int printsymbol(void)
{
  PWord val; int type;
  char name[BUFSIZE];

  PI_getan(&val, &type, 1);
  if (type != PI_SYM)
       PI_FAIL;
  if (!PI_getsymname(name, val, BUFSIZE))
       PI_FAIL;
  PI_printf("Symbol is: %s\n", name);
  PI_SUCCEED;
}
```

SEE ALSO

`PI_makesymname`, `PI_getuianame`

NAME

`PI_gettail` - get the tail of a list cell

FORMS

```
void PI_gettail(PWord *val, int *type, PWord listval);
```

DESCRIPTION

`PI_gettail` is used to get the tail of a list cell specified by `listval`. `val` is used to store the value of the tail, and `type` is used to store the associated type. The type is one of the ALS Prolog type constants.

SEE ALSO

`PI_getan` for the Prolog types.

NAME

`PI_getuianame` - get string associated the specified uninterned atom

FORMS

```
char *PI_getuianame(char *buf, PWord uiaval,
                    int bufsize);
```

DESCRIPTION

`PI_getuianame` copies the string associated with the specified uninterned atom, `uiaval`, into the buffer pointed to by `buf`. `bufsize` should be the size of the buffer referenced by `buf`. Normally, `buf` is returned as the result of the function. If the UIA name is longer than `bufsize` bytes, the buffer will not be filled, and 0 will be returned.

EXAMPLES

```
#define BUFSIZE 256
static int printuia(void)
{
  PWord val; int type;
  char buf[BUFSIZE];

  PI_getan(&val, &type, 1);
  if (type != PI_UIA)
       PI_FAIL;
  if (!PI_getuianame(buf, val, BUFSIZE))
       PI_FAIL;
  PI_printf("UIA is: %s\n", buf);
  PI_SUCCEED;
}
```

SEE ALSO

`PI_makeuia`, `PI_getsymname`

NAME

`PI_INIT` - introduces C-defined predicates to the Prolog system

FORMS

`PI_INIT`

DESCRIPTION

`PI_INIT` puts the procedure table entries defined using `PI_BEGIN`, `PI_DEFINE`, and `PI_END` into the ALS Prolog procedure table. Currently all C-defined Prolog procedures are exported from the module in which they are placed. The module is determined by `PI_MODULE` entries in the table defined by `PI_BEGIN` and `PI_END`. By default, `PI_DEFINE` places its entry in module `user`.

EXAMPLES

In the following example,pretend that `my_init()` is a function with some application specific initializations in it, and fried() is a C-defined predicate.

```
PI_BEGIN
  PI_DEFINE("fried", 1, fried)
PI_END

static void my_init(void)
{
  PI_INIT;
}
```

SEE ALSO

`PI_BEGIN`, `loadforeign/2`, `loadforeign/3`

NAME

`PI_INT` - identifies an object as a Prolog integer

FORMS

```
if (type == PI_INT)
    do_something();
```

DESCRIPTION

The value associated with `PI_INT` is a C integer. The `PI_INT` type is the only Prolog type which doesn't have to be taken apart or created. There are no `PI_getint` and `PI_makeint` functions.

EXAMPLES

```
static int printint(void)
{
  PWord val; int type;
  PI_getan(&val,&type,1);
  if (type != PI_TYPE)
       PI_FAIL;
  PI_printf("The integer is: %d\n", val);
  PI_SUCCEED;
}
```

NOTES

Integers which are too big might cause problems.

NAME

`PI_LIST` - identifies an object as a Prolog list

FORMS

```
if (type == PI_LIST)
    do_something();
```

DESCRIPTION

The corresponding value for `PI_LIST` is a Prolog list cell. The `PI_gethead` function gets the head of the list cell, and `PI_gettail` gets the tail of the list cell. A list cell can be created by using the `PI_makelist` function.

SEE ALSO

`PI_gethead`, `PI_gettail`, `PI_makelist`

NAME

`PI_main` - Starts the ALS Prolog development system

FORMS

```
int PI_main(int argc, char **argv,
            void (*init)(void));
```

DESCRIPTION

`PI_main` starts the ALS Prolog development system. `argc` and `argv` are the standard C command line arguments. `init` is a function pointer for initializing C defined predicates, it may be NULL. `PI_main` is primarily used for static linking of C defined predicates with ALS Prolog.

EXAMPLES

```
int main(int argc, char **argv)
{
  return PI_main(argc, argv, NULL);
}
```

NAME

`PI_makedouble` - creates a Prolog floating point number

FORMS

```
void PI_makedouble(PWord *val, int *type,
                   double cdouble);
```

DESCRIPTION

`PI_makedouble` creates a Prolog double precision floating point number given a C double. If the value of `cdouble` is an integer, then the integer is placed in `val`, and `type` is set to `PI_INT`. If the value of `cdouble` is not an integer, then the double is placed in `val`, and `type` is set to `PI_DOUBLE`.

EXAMPLES

```
static int makepi(void)
{
  PWord val, dbl;
  int type, dbltype;

  PI_getan(&val, &type, 1);
  PI_makedouble(&dbl, &dbltype, 3.14);
  if (!PI_unify(val, type, dbl, dbltype))
       PI_FAIL;
  PI_SUCCEED;
}
```

SEE ALSO

PI_getdouble

NAME

`PI_makelist` - creates a Prolog list

FORMS

```
void PI_makelist(PWord *listval, int *listtype);
```

DESCRIPTION

`PI_makelist` creates a new list cell and stores the value in `listval`. The Prolog type `PI_LIST` is stored in `type`. Both the head and the tail of the list are initialized to unbound variables.

SEE ALSO

`PI_gethead`, `PI_gettail`

NAME

`PI_makestruct` - creates a Prolog structure

FORMS

```
void PI_makestruct(PWord *strucval, int *structype,
                   PWord func, int arity);
```

DESCRIPTION

`PI_makestruct` creates a Prolog structure with the specified functor and arity. The functor (`func`) should be a symbol of type `PI_SYM`. `PI_makesym` can be used to create a symbol that can be used as the functor. `arity` should be an integer specifying the number of arguments in the structure. `PI_makestruct` stores the value of the structure in `strucval`, and it stores the Prolog type `PI_STRUCT` in `structype`. The arguments of the newly created structure are initialized with unbound variables.

NAME

`PI_makesym` - creates a symbol in the Prolog symbol table

FORMS

```
void PI_makesym(PWord *symval, int *symtype,
                const char *str);
```

DESCRIPTION

`PI_makesym` returns the value of the symbol specified by the string in `str`. The value of the symbol is stored in `symval`, and the Prolog type `PI_SYM` is stored in `symtype`. If the specified symbol is not already in the symbol table, it will be entered there.

EXAMPLES

```
static int makefuzz(void)
{
  PWord val, sym;
  int type, symtype;

  PI_getan(&val, &type, 1);
  PI_makesym(&sym, &symtype, "fuzz");
  if (!PI_unify(sym, symtype, val, type))
       PI_FAIL;
  PI_SUCCEED;
}
```

SEE ALSO

`PI_getsymname`, `PI_makeuia`, `PI_SYM`

NAME

`PI_makeuia` - creates a symbol on the Prolog heap (UIA)

FORMS

```
void PI_makeuia(PWord *val, int *type,
                 const char *str);
```

DESCRIPTION

`PI_makeuia` attempts to create an uninterned atom (UIA) specified by the string contained in `str`. If the specified symbol is already in the symbol table, `val` is used to stored the symbol value, and `type` is used to store the Prolog type `PI_SYM`. If the symbol is not in the symbol table, `val` is used to store the UIA, and type is used to store the Prolog type `PI_UIA`.

EXAMPLES

```
static int makebuzz(void)
{
  PWord val, sym;
  int type, symtype;

  PI_getan(&val, &type, 1);
  PI_makeuia(&sym, &symtype, "buzz");
  if (!PI_unify(sym, symtype, val, type))
       PI_FAIL;
  PI_SUCCEED;
}
```

SEE ALSO

`PI_getuianame`, `PI_makesym`

NAME

`PI_printf` - prints to the Prolog standard output

FORMS

```
int PI_printf(const char *format, ...);
```

DESCRIPTION

`PI_printf` is like the standard `printf` function except it prints to the Prolog standard output.

EXAMPLES

The following functions aren't meant to be predicates. They could be supporting routines for predicates. Therefore, PI_SUCCEED is not necessary.

```
static void printhello(void)
{
  PI_printf("hello");
}

static void printnumbers(void)
{
  int x = 1, y = 2, z = 3;
  PI_printf("Count: %d %d %d\n", x, y, z);
}

static void printstr(void)
{
  char *str = "play ball!";
  PI_printf("The string is: %s\n", str);
}
```

NAME

`PI_prolog_init` - initializes the prolog system

FORMS

```
int PI_prolog_init(int argc, char **argv);
```

DESCRIPTION

`PI_prolog_init` initializes the prolog system. `argc` and `argv` are the standard C command line arguments.

EXAMPLES

```
void main(int argc, char **argv)
{
  PI_prolog_init(argc, argv);

  PI_top_level();

  PI_shutdown();
}
```

NAME

`PI_rungoal` - runs the specified Prolog goal
`PI_rungoal_with_update`- runs the specified Prolog goal

FORMS

```
int PI_rungoal(PWord module,
                  PWord goalval, int goaltype);
int PI_rungoal_with_update(PWord module,
                  PWord *goalval, int *goaltype);
```

DESCRIPTION

Given the module, and the goal to run, `PI_rungoal` submits the goal returning '1' for success, and '0' for failure; `module` is assumed to be of type `PI_SYM`, `goalval` should be of type `PI_SYM` or of type `PI_STRUCT`, and `goaltype` should be one of `PI_SYM` or `PI_STRUCT`.

`PI_rungoal_with_update` does what `PI_rungoal` does (runs a prolog goal with structure and type, `goalval` and `goaltype`, supplied from C), but it performs the additional step of updating the goal structure `goalval` upon return. An application may then safely inspect this structure for bindings created while running the goal. Doing these inspections with `PI_rungoal` is unsafe as the garbage collector may move structures around thus invalidating any previously held pointers.

EXAMPLES

```
static int test(void)
{
  PWord user, hello;
  int usertype, hellotype;
  PI_makesym(&user, &usertype, "user");
  PI_makesym(&hello, &hellotype, "hello");
  if (PI_rungoal(user, hello, hellotype))
       PI_printf("Goal worked!");
  else
```

```
        PI_printf("Goal didn't work.");
  PI_SUCCEED;
}
```

SEE ALSO

PI_makesym, PI_makestruct

NAME

`PI_shutdown` - shutdown ALS Prolog

FORMS

```
void PI_shutdown(void);
```

DESCRIPTION

The `PI_shutdown()` function shuts down ALS Prolog, allowing it to do cleanup processing.

NAME

`PI_startup` - initializes the prolog system

FORMS

```
typedef struct {
  unsigned long heap_size;
  unsigned long stack_size;
  unsigned long icbuf_size;
  const char *alsdir;
  const char *saved_state;

  int argc;
  char **argv;
  ...
} PI_system_setup;

int PI_startup(const PI_system_setup *setup);
```

DESCRIPTION

`PI_startup` initilizes the Prolog system with an option set of parameters. PI_startup performs the same initilization as PI_prolog_init, except that it allows some system parameters to be set, rather than use the defaults.

The PI_system_setup structure contains settings that can be passed to PI_startup function through the setup parameter. If PI_startup is passed a NULL, the system defaults are used.

Each field PI_system_setup has a default value which causes the system to use the system default value. PI_system_setup may have extra fields for OS specific parameters.

Parameter	Description	Default Value
heap_size	Size of the Prolog heap in bytes	0

Table 16: PI_system_setup fields

stack_size	Size of the Prolog Stack in bytes	0
icbuf_size	Size of the intermediate code buffer in bytes	0
alsdir	File path of the alsdir directory	NULL
save_state	File path to the a saved state file to load	NULL
argc	main() argument count	0
argv	main() arguments	NULL

Table 16: PI_system_setup fields

EXAMPLES

```
int main(int argc, char **argv)
{
  PI_system_setup my_setup;

  /* Set the heap size to 10Mb, all other settings
     are default. */
  my_setup.heap_size = 10000000;
  my_setup.stack_size = my_setup.icbuf_size = 0;
  my_setup.alsdir = my_setup.save_state = NULL;
  my_setup.argc = argc;
  my_setup.argv = argv;

  PI_setup(&my_setup);

  ...
}
```

NAME

`PI_STRUCT` - identifies an object as a Prolog structure

FORMS

```
if (type == PI_STRUCT)
    do_something();
```

DESCRIPTION

The value associated with `PI_STRUCT` is a Prolog structure. `PI_getstruct` is used to retrieve the functor and the arity of the structure, while `PI_getargn` is used to retrieve the individual arguments of the structure. A Prolog structure can be created by using the `PI_makestruct` function.

SEE ALSO

`PI_getstruct`, `PI_getargn`, `PI_makestruct`

NAME

`PI_SUCCEED` - causes the current C-defined predicate to succeed

FORMS

`PI_SUCCEED`

DESCRIPTION

`PI_SUCCEED` makes the current predicate succeed. Control is returned immediately to Prolog.

EXAMPLES

```
static int mytrue(void)
{
  PI_SUCCEED;
}
```

NOTES

`PI_SUCCEED` is implemented as a macro, which executes a `return()`. For this reason, `PI_SUCCEED` should only be used from the top level.

SEE ALSO

`PI_FAIL`

NAME

`PI_SYM` - identifies an object as a Prolog symbol

FORMS

```
if (type == PI_SYM)
    do_something();
```

DESCRIPTION

The values associated with `PI_SYM` are Prolog symbols contained in the symbol table. `PI_getsymname` can be used to get the string associated with a symbol. `PI_makesym` can be used to create a symbol.

SEE ALSO

`PI_UIA`, `PI_getsymname`, `PI_makesym`

NAME

`PI_toplevel` - invokes a prolog shell

FORMS

```
void PI_toplevel(void);
```

DESCRIPTION

`PI_toplevel` starts a prolog shell. Control does not return until the shell is exited.

EXAMPLES

```
void main(int argc, char **argv)
{
  PI_prolog_init(argc, argv);

  PI_top_level();

  PI_shutdown();
}
```

NAME

`PI_UIA` - identifies an object as an uninterned atom

FORMS

```
if (type == PI_UIA)
    do_something();
```

DESCRIPTION

The value associated with a `PI_UIA` is an uninterned atom (UIA). The function `PI_getuianame` can be used to get the string associated with a UIA. The function `PI_makeuia` can be used to create a UIA.

SEE ALSO

`PI_SYM`, `PI_getuianame`, `PI_makeuia`

NAME

`PI_unify` - attempts to match/bind two Prolog objects

FORMS

```
int PI_unify(PWord val1, int type1,
             PWord val2, int type2);
```

DESCRIPTION

`PI_unify` will attempt to unify the object specified by `val1–type1` with the object specified by `val2–type2`. If unification is successful, this function will return `1`. If not, it will return `0`.

NOTES

In previous versions of ALS Prolog, `PI_unify` did not return to the C function that called it if failure occured. This has been changed so that an unsuccessful unification returns `0` as the result. Always test for `0` as the result of `PI_unify`, and if it occurs, call `PI_FAIL`:

```
    if (!PI_unify(v1, t1, v2, t2))
        PI_FAIL;
```

instead of

```
    PI_unify(v1, t1, v2, t2);
```

SEE ALSO

=/2

NAME

`PI_VAR` - identifies an object as being a Prolog unbound variable

FORMS

```
if (type == PI_VAR)
    do_something();
```

DESCRIPTION

The value associated with `PI_VAR` is an unbound Prolog variable. An unbound variable can be instantiated by calling `PI_unify` with the unbound variable value as an argument.

EXAMPLES

Note that `PI_unify` shouldn't fail in the following predicate, because we know that `val` is an unbound variable.

```
static int boundvar(void)
{
  PWord val; int type;

  PI_getan(&val, &type, 1);
  if (type != PI_VAR)
       PI_SUCCEED;
  PI_unify(val, type, 1, PI_INT);
  PI_SUCCEED;
}
```

SEE ALSO

PI_unify