NAME

io - Standard I/O server interface functions.

DESCRIPTION

This module provides an interface to standard Erlang I/O servers. The output functions all return *ok* if they are successful, or exit if they are not.

All functions in this module have an optional parameter *IoDevice*. If included, it must be the pid of a process that handles the I/O protocols. Normally, it is a *IoDevice* returned by *file:open/2*. If no *IoDevice* is given, *standard_io* is used.

For a description of the I/O protocols, see section The Erlang I/O Protocol in the User's Guide.

Warning:

As from Erlang/OTP R13A, data supplied to function *put_chars/2* is to be in the *unicode:chardata()* format. This means that programs supplying binaries to this function must convert them to UTF-8 before trying to output the data on an I/O device.

If an I/O device is set in binary mode, functions *get_chars/2,3* and *get_line/1,2* can return binaries instead of lists. The binaries are, as from Erlang/OTP R13A, encoded in UTF-8.

To work with binaries in ISO Latin-1 encoding, use the *file* module instead.

For conversion functions between character encodings, see the *unicode* module.

DATA TYPES

device() = atom() | pid()

An I/O device, either *standard_io*, *standard_error*, a registered name, or a pid handling I/O protocols (returned from *file:open/2*).

For more information about the built-in devices see Standard Input/Output and Standard Error.

opt_pair() =
{binary, boolean()} |
{echo, boolean()} |
{expand_fun, expand_fun()} |
{encoding, encoding()}

expand_fun() =
fun((term()) -> {yes | no, string(), [string(), ...]})

encoding() =
latin1 | unicode | utf8 | utf16 | utf32 |
{utf16, big | little} |
{utf32, big | little}

setopt() = binary | list | opt_pair()

format() = atom() | string() | binary()

location() = erl_anno:location()

prompt() = atom() | unicode:chardata()

server_no_data() = {error, ErrorDescription :: term()} | eof

What the I/O server sends when there is no data.

EXPORTS

columns() -> {ok, integer() >= 1} | {error, enotsup}

columns(IoDevice) -> {ok, integer() >= 1} | {error, enotsup}

Types:

IoDevice = device()

Retrieves the number of columns of the *IoDevice* (that is, the width of a terminal). The function succeeds for terminal devices and returns *{error, enotsup}* for all other I/O devices.

format(Format) -> ok

format(Format, Data) -> ok

format(IoDevice, Format, Data) -> ok

fwrite(Format) -> ok

fwrite(Format, Data) -> ok

fwrite(IoDevice, Format, Data) -> ok

Types:

IoDevice = device()
Format = format()
Data = [term()]

Writes the items in *Data* (*[]*) on the standard output (*IoDevice*) in accordance with *Format*. *Format* contains plain characters that are copied to the output device, and control sequences for formatting, see below. If *Format* is an atom or a binary, it is first converted to a list with the aid of *atom_to_list/1* or *binary_to_list/1*. Example:

```
1> io:fwrite("Hello world!~n", []).
Hello world!
ok
```

The general format of a control sequence is *~F.P.PadModC*.

The character *C* determines the type of control sequence to be used. It is the only required field. All of *F*, *P*, *Pad*, and *Mod* are optional. For example, to use a *#* for *Pad* but use the default values for *F* and *P*, you can write *~..#C*.

* *F* is the *field width* of the printed argument. A negative value means that the argument is left-justified within the field, otherwise right-justified. If no field width is specified, the required print width is used. If the field width specified is too small, the whole field is filled with * characters.

* *P* is the *precision* of the printed argument. A default value is used if no precision is specified. The interpretation of precision depends on the control sequences. Unless otherwise specified, argument *within* is used to determine print width.

* *Pad* is the padding character. This is the character used to pad the printed representation of the argument so that it conforms to the specified field width and precision. Only one padding character can be specified and, whenever applicable, it is used for both the field width and

precision. The default padding character is ’ ’ (space).

* *Mod* is the control sequence modifier. This is one or more characters that change the interpretation of *Data*. The current modifiers are *t*, for Unicode translation, and *l*, for stopping *p* and *P* from detecting printable characters.

If *F*, *P*, or *Pad* is a * character, the next argument in *Data* is used as the value. For example:

```
1> io:fwrite("~*.*.0f~n",[9, 5, 3.14159265]).
003.14159
ok
```

To use a literal * character as *Pad*, it must be passed as an argument:

```
2> io:fwrite("~*.*.*f~n",[9, 5, $*, 3.14159265]).
**3.14159
ok
```

Available control sequences:

~:
Character ~ is written.

c: The argument is a number that is interpreted as an ASCII code. The precision is the number of times the character is printed and defaults to the field width, which in turn defaults to 1. Example:

```
1> io:fwrite("|~10.5c|~-10.5c|~5c|~n", [$a, $b, $c]).
|     aaaaa|bbbbb     |ccccc|
ok
```

If the Unicode translation modifier (*t*) is in effect, the integer argument can be any number representing a valid Unicode codepoint, otherwise it is to be an integer less than or equal to 255, otherwise it is masked with 16#FF:

```
2> io:fwrite("~tc~n",[1024]).
```

```
\x{400}
ok
3> io:fwrite("~c~n",[1024]).
^@
ok
```

f: The argument is a float that is written as *[-]ddd.ddd*, where the precision is the number of digits after the decimal point. The default precision is 6 and it cannot be < 1.

e: The argument is a float that is written as *[-]d.ddde+-ddd*, where the precision is the number of digits written. The default precision is 6 and it cannot be < 2.

g:
The argument is a float that is written as *f*, if it is >= 0.1 and < 10000.0. Otherwise, it is written in the *e* format. The precision is the number of significant digits. It defaults to 6 and is not to be < 2. If the absolute value of the float does not allow it to be written in the *f* format with the desired number of significant digits, it is also written in the *e* format.

s: Prints the argument with the string syntax. The argument is, if no Unicode translation modifier is present, an *iolist()*, a *binary()*, or an *atom()*. If the Unicode translation modifier (*t*) is in effect, the argument is *unicode:chardata()*, meaning that binaries are in UTF-8. The characters are printed without quotes. The string is first truncated by the specified precision and then padded and justified to the specified field width. The default precision is the field width.

This format can be used for printing any object and truncating the output so it fits a specified field:

```
1> io:fwrite("|~10w|~n", [{hey, hey, hey}]).
|**********|
ok
2> io:fwrite("|~10s|~n", [io_lib:write({hey, hey, hey})]).
|{hey,hey,h|
3> io:fwrite("|~-10.8s|~n", [io_lib:write({hey, hey, hey})]).
|{hey,hey  |
ok
```

A list with integers > 255 is considered an error if the Unicode translation modifier is not specified:

```
4> io:fwrite("~ts~n",[[1024]]).
\x{400}
ok
5> io:fwrite("~s~n",[[1024]]).
** exception error: bad argument
     in function  io:format/3
        called as io:format(<0.53.0>,"~s~n",[[1024]])
```

w:

Writes data with the standard syntax. This is used to output Erlang terms. Atoms are printed within quotes if they contain embedded non-printable characters. Atom characters > 255 are escaped unless the Unicode translation modifier (*t*) is used. Floats are printed accurately as the shortest, correctly rounded string.

p:

Writes the data with standard syntax in the same way as *~w*, but breaks terms whose printed representation is longer than one line into many lines and indents each line sensibly. Left-justification is not supported. It also tries to detect flat lists of printable characters and output these as strings. For example:

```
1> T = [{attributes,[[{id,age,1.50000},{mode,explicit},
{typename,"INTEGER"}], [{id,cho},{mode,explicit},{typename,'Cho'}]]},
{typename,'Person'},{tag,{'PRIVATE',3}},{mode,implicit}].
2> io:fwrite("~w~n", [T]).
[{attributes,[[{id,age,1.5},{mode,explicit},{typename,
[73,78,84,69,71,69,82]}],[{id,cho},{mode,explicit},{typena
me,'Cho'}]]},{typename,'Person'},{tag,{'PRIVATE',3}},{mode
,implicit}]
ok
3> io:fwrite("~62p~n", [T]).
[{attributes,[[{id,age,1.5},
              {mode,explicit},
              {typename,"INTEGER"}],
             [{id,cho},{mode,explicit},{typename,'Cho'}]]},
 {typename,'Person'},
 {tag,{'PRIVATE',3}},
 {mode,implicit}]
ok
```

The field width specifies the maximum line length. It defaults to 80. The precision specifies the initial indentation of the term. It defaults to the number of characters printed on this line in the *same* call to *write/1* or *format/1,2,3*. For example, using *T* above:

```
4> io:fwrite("Here T = ~62p~n", [T]).
Here T = [{attributes,[[{id,age,1.5},
                        {mode,explicit},
                        {typename,"INTEGER"}],
                       [{id,cho},
                        {mode,explicit},
                        {typename,'Cho'}]]},
          {typename,'Person'},
          {tag,{'PRIVATE',3}},
          {mode,implicit}]
ok
```

As from Erlang/OTP 21.0, a field width of value *0* can be used for specifying that a line is infinitely long, which means that no line breaks are inserted. For example:

```
5> io:fwrite("~0p~n", [lists:seq(1, 30)]).
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30]
ok
```

When the modifier *l* is specified, no detection of printable character lists takes place, for example:

```
6> S = [{a,"a"}, {b, "b"}], io:fwrite("~15p~n", [S]).
[{a,"a"},
 {b,"b"}]
ok
7> io:fwrite("~15lp~n", [S]).
[{a,[97]},
 {b,[98]}]
ok
```

The Unicode translation modifier *t* specifies how to treat characters outside the Latin-1 range of codepoints, in atoms, strings, and binaries. For example, printing an atom containing a

character > 255:

```
8> io:fwrite("~p~n",[list_to_atom([1024])]).
ok
9> io:fwrite("~tp~n",[list_to_atom([1024])]).
ok
```

By default, Erlang only detects lists of characters in the Latin-1 range as strings, but the *+pc unicode* flag can be used to change this (see *printable_range/0* for details). For example:

```
10> io:fwrite("~p~n",[[214]]).
"Ö"
ok
11> io:fwrite("~p~n",[[1024]]).
[1024]
ok
12> io:fwrite("~tp~n",[[1024]]).
[1024]
ok
```

but if Erlang was started with *+pc unicode*:

```
13> io:fwrite("~p~n",[[1024]]).
[1024]
ok
14> io:fwrite("~tp~n",[[1024]]).
"<?>"
ok
```

Similarly, binaries that look like UTF-8 encoded strings are output with the binary string syntax if the *t* modifier is specified:

```
15> io:fwrite("~p~n", [<<208,128>>]).
<<208,128>>
ok
```

```
16> io:fwrite("~tp~n", [<<208,128>>]).
<<"<?>"/utf8>>
ok
17> io:fwrite("~tp~n", [<<128,128>>]).
<<128,128>>
ok
```

W:

Writes data in the same way as *~w*, but takes an extra argument that is the maximum depth to which terms are printed. Anything below this depth is replaced with .... For example, using *T* above:

```
8> io:fwrite("~W~n", [T,9]).
[{attributes,[[{id,age,1.5},{mode,explicit},{typename,...}],
[{id,cho},{mode,...},{...}]]},{typename,'Person'},
{tag,{'PRIVATE',3}},{mode,implicit}]
ok
```

If the maximum depth is reached, it cannot be read in the resultant output. Also, the ,... form in a tuple denotes that there are more elements in the tuple but these are below the print depth.

P:

Writes data in the same way as *~p*, but takes an extra argument that is the maximum depth to which terms are printed. Anything below this depth is replaced with ..., for example:

```
9> io:fwrite("~62P~n", [T,9]).
[{attributes,[[{id,age,1.5},{mode,explicit},{typename,...}],
        [{id,cho},{mode,...},{...}]]},
 {typename,'Person'},
 {tag,{'PRIVATE',3}},
 {mode,implicit}]
ok
```

B:

Writes an integer in base 2-36, the default base is 10. A leading dash is printed for negative integers.

The precision field selects base, for example:

```
1> io:fwrite("~.16B~n", [31]).
1F
ok
2> io:fwrite("~.2B~n", [-19]).
-10011
ok
3> io:fwrite("~.36B~n", [5*36+35]).
5Z
ok
```

X:
Like *B*, but takes an extra argument that is a prefix to insert before the number, but after the leading dash, if any.

The prefix can be a possibly deep list of characters or an atom. Example:

```
1> io:fwrite("~X~n", [31,"10#"]).
10#31
ok
2> io:fwrite("~.16X~n", [-31,"0x"]).
-0x1F
ok
```

#:
Like *B*, but prints the number with an Erlang style #-separated base prefix. Example:

```
1> io:fwrite("~.10#~n", [31]).
10#31
ok
2> io:fwrite("~.16#~n", [-31]).
-16#1F
ok
```

b:
Like *B*, but prints lowercase letters.

x:
Like *X*, but prints lowercase letters.

+:
Like #, but prints lowercase letters.

n:
Writes a new line.

i: Ignores the next term.

The function returns:

ok:
The formatting succeeded.

If an error occurs, there is no output. Example:

```
1> io:fwrite("~s ~w ~i ~w ~c ~n",[’abc def’, ’abc def’, {foo, 1},{foo, 1}, 65]).
abc def ’abc def’  {foo,1} A
ok
2> io:fwrite("~s", [65]).
** exception error: bad argument
    in function  io:format/3
      called as io:format(<0.53.0>,"~s","A")
```

In this example, an attempt was made to output the single character 65 with the aid of the string formatting directive *"~s"*.

fread(Prompt, Format) -> Result

fread(IoDevice, Prompt, Format) -> Result

Types:

```
IoDevice = device()
Prompt = prompt()
Format = format()
Result =
   {ok, Terms :: [term()]} |
   {error, {fread, FreadError :: io_lib:fread_error()}} |
```

server_no_data()
server_no_data() = {error, ErrorDescription :: term()} | eof

Reads characters from the standard input (*IoDevice*), prompting it with *Prompt*. Interprets the characters in accordance with *Format*. *Format* contains control sequences that directs the interpretation of the input.

Format can contain the following:

* Whitespace characters (*Space*, *Tab*, and *Newline*) that cause input to be read to the next non-whitespace character.

* Ordinary characters that must match the next input character.

* Control sequences, which have the general format ~**FMC*, where:

  * Character * is an optional return suppression character. It provides a method to specify a field that is to be omitted.

  * *F* is the *field width* of the input field.

  * *M* is an optional translation modifier (of which *t* is the only supported, meaning Unicode translation).

  * *C* determines the type of control sequence.

  Unless otherwise specified, leading whitespace is ignored for all control sequences. An input field cannot be more than one line wide.

  Available control sequences:

  ~:
  A single ~ is expected in the input.

  d:
  A decimal integer is expected.

  u:
  An unsigned integer in base 2-36 is expected. The field width parameter is used to specify base. Leading whitespace characters are not skipped.

-: An optional sign character is expected. A sign character - gives return value *-1*. Sign character + or none gives *1*. The field width parameter is ignored. Leading whitespace characters are not skipped.

#:
An integer in base 2-36 with Erlang-style base prefix (for example, *"16#ffff"*) is expected.

f: A floating point number is expected. It must follow the Erlang floating point number syntax.

s: A string of non-whitespace characters is read. If a field width has been specified, this number of characters are read and all trailing whitespace characters are stripped. An Erlang string (list of characters) is returned.

If Unicode translation is in effect (*~ts*), characters > 255 are accepted, otherwise not. With the translation modifier, the returned list can as a consequence also contain integers > 255:

```
1> io:fread("Prompt> ","~s").
Prompt> <Characters beyond latin1 range not printable in this medium>
{error,{fread,string}}
2> io:fread("Prompt> ","~ts").
Prompt> <Characters beyond latin1 range not printable in this medium>
{ok,[[1091,1085,1080,1094,1086,1076,1077]]}
```

a: Similar to *s*, but the resulting string is converted into an atom.

c: The number of characters equal to the field width are read (default is 1) and returned as an Erlang string. However, leading and trailing whitespace characters are not omitted as they are with *s*. All characters are returned.

The Unicode translation modifier works as with *s*:

```
1> io:fread("Prompt> ","~c").
Prompt> <Character beyond latin1 range not printable in this medium>
{error,{fread,string}}
2> io:fread("Prompt> ","~tc").
Prompt> <Character beyond latin1 range not printable in this medium>
{ok,[[1091]]}
```

l: Returns the number of characters that have been scanned up to that point, including whitespace characters.

The function returns:

{ok, Terms}:
The read was successful and *Terms* is the list of successfully matched and read items.

eof:
End of file was encountered.

{error, FreadError}:
The reading failed and *FreadError* gives a hint about the error.

{error, ErrorDescription}:
The read operation failed and parameter *ErrorDescription* gives a hint about the error.

Examples:

```
20> io:fread('enter>', "~f~f~f").
enter>1.9 35.5e3 15.0
{ok,[1.9,3.55e4,15.0]}
21> io:fread('enter>', "~10f~d").
enter>     5.67899
{ok,[5.678,99]}
22> io:fread('enter>', ":~10s:~10c:").
enter>:   alan   :   joe    :
{ok, ["alan", "   joe    "]}
```

get_chars(Prompt, Count) -> Data | server_no_data()

get_chars(IoDevice, Prompt, Count) -> Data | server_no_data()

Types:

IoDevice = device()
Prompt = prompt()
Count = integer() >= 0

Data = string() | unicode:unicode_binary()
server_no_data() = {error, ErrorDescription :: term()} | eof

Reads *Count* characters from standard input (*IoDevice*), prompting it with *Prompt*.

The function returns:

Data:
The input characters. If the I/O device supports Unicode, the data can represent codepoints > 255 (the *latin1* range). If the I/O server is set to deliver binaries, they are encoded in UTF-8 (regardless of whether the I/O device supports Unicode).

eof:
End of file was encountered.

{error, ErrorDescription}:
Other (rare) error condition, such as *{error, estale}* if reading from an NFS file system.

get_line(Prompt) -> Data | server_no_data()

get_line(IoDevice, Prompt) -> Data | server_no_data()

Types:

IoDevice = device()
Prompt = prompt()
Data = string() | unicode:unicode_binary()
server_no_data() = {error, ErrorDescription :: term()} | eof

Reads a line from the standard input (*IoDevice*), prompting it with *Prompt*.

The function returns:

Data:
The characters in the line terminated by a line feed (or end of file). If the I/O device supports Unicode, the data can represent codepoints > 255 (the *latin1* range). If the I/O server is set to deliver binaries, they are encoded in UTF-8 (regardless of if the I/O device supports Unicode).

eof:
End of file was encountered.

{error, ErrorDescription}:
Other (rare) error condition, such as *{error, estale}* if reading from an NFS file system.

getopts() -> [opt_pair()] | {error, Reason}

getopts(IoDevice) -> [opt_pair()] | {error, Reason}

Types:

IoDevice = device()
Reason = term()

Requests all available options and their current values for a specific I/O device, for example:

```
1> {ok,F} = file:open("/dev/null",[read]).
{ok,<0.42.0>}
2> io:getopts(F).
[{binary,false},{encoding,latin1}]
```

Here the file I/O server returns all available options for a file, which are the expected ones, *encoding* and *binary*. However, the standard shell has some more options:

```
3> io:getopts().
[{expand_fun,#Fun<group.0.120017273>},
 {echo,true},
 {binary,false},
 {encoding,unicode}]
```

This example is, as can be seen, run in an environment where the terminal supports Unicode input and output.

nl() -> ok

nl(IoDevice) -> ok

Types:

IoDevice = device()

Writes new line to the standard output (*IoDevice*).

parse_erl_exprs(Prompt) -> Result

parse_erl_exprs(IoDevice, Prompt) -> Result

parse_erl_exprs(IoDevice, Prompt, StartLocation) -> Result

parse_erl_exprs(IoDevice, Prompt, StartLocation, Options) ->
Result

Types:

```
IoDevice = device()
Prompt = prompt()
StartLocation = location()
Options = erl_scan:options()
Result = parse_ret()
parse_ret() =
   {ok,
    ExprList :: [erl_parse:abstract_expr()],
    EndLocation :: location()} |
   {eof, EndLocation :: location()} |
   {error,
    ErrorInfo :: erl_scan:error_info() | erl_parse:error_info(),
    ErrorLocation :: location()} |
   server_no_data()
server_no_data() = {error, ErrorDescription :: term()} | eof
```

Reads data from the standard input (*IoDevice*), prompting it with *Prompt*. Starts reading at location *StartLocation* (*1*). Argument *Options* is passed on as argument *Options* of function *erl_scan:tokens/4*. The data is tokenized and parsed as if it was a sequence of Erlang expressions until a final dot (.) is reached.

The function returns:

{ok, ExprList, EndLocation}:
The parsing was successful.

{eof, EndLocation}:
End of file was encountered by the tokenizer.

eof:
End of file was encountered by the I/O server.

{error, ErrorInfo, ErrorLocation}:
An error occurred while tokenizing or parsing.

{error, ErrorDescription}:
Other (rare) error condition, such as *{error, estale}* if reading from an NFS file system.

Example:

```
25> io:parse_erl_exprs('enter>').
enter>abc(), "hey".
{ok, [{call,1,{atom,1,abc},[]},{string,1,"hey"}],2}
26> io:parse_erl_exprs ('enter>').
enter>abc("hey".
{error,{1,erl_parse,["syntax error before: ",["'.'"]]},2}
```

parse_erl_form(Prompt) -> Result

parse_erl_form(IoDevice, Prompt) -> Result

parse_erl_form(IoDevice, Prompt, StartLocation) -> Result

parse_erl_form(IoDevice, Prompt, StartLocation, Options) -> Result

Types:

IoDevice = device()
Prompt = prompt()

```
StartLocation = location()
Options = erl_scan:options()
Result = parse_form_ret()
parse_form_ret() =
    {ok,
     AbsForm :: erl_parse:abstract_form(),
     EndLocation :: location()} |
    {eof, EndLocation :: location()} |
    {error,
     ErrorInfo :: erl_scan:error_info() | erl_parse:error_info(),
     ErrorLocation :: location()} |
    server_no_data()
server_no_data() = {error, ErrorDescription :: term()} | eof
```

Reads data from the standard input (*IoDevice*), prompting it with *Prompt*. Starts reading at location *StartLocation* (*1*). Argument *Options* is passed on as argument *Options* of function *erl_scan:tokens/4*. The data is tokenized and parsed as if it was an Erlang form (one of the valid Erlang expressions in an Erlang source file) until a final dot (.) is reached.

The function returns:

{ok, AbsForm, EndLocation}:
End of file was encountered by the tokenizer.

{eof, EndLocation}:
End of file was encountered by the tokenizer.

eof:
End of file was encountered by the I/O server.

{error, ErrorInfo, ErrorLocation}:
An error occurred while tokenizing or parsing.

{error, ErrorDescription}:
Other (rare) error condition, such as *{error, estale}* if reading from an NFS file system.

printable_range() -> unicode | latin1

Returns the user-requested range of printable Unicode characters.

The user can request a range of characters that are to be considered printable in heuristic detection of strings by the shell and by the formatting functions. This is done by supplying *+pc <range>* when starting Erlang.

The only valid values for *<range>* are *latin1* and *unicode*. *latin1* means that only code points < 256 (except control characters, and so on) are considered printable. *unicode* means that all printable characters in all Unicode character ranges are considered printable by the I/O functions.

By default, Erlang is started so that only the *latin1* range of characters indicate that a list of integers is a string.

The simplest way to use the setting is to call *io_lib:printable_list/1*, which uses the return value of this function to decide if a list is a string of printable characters.

Note:

In a future release, this function may return more values and ranges. To avoid compatibility problems, it is recommended to use function *io_lib:printable_list/1*.

put_chars(CharData) -> ok

put_chars(IoDevice, CharData) -> ok

Types:

IoDevice = device()
CharData = unicode:chardata()

Writes the characters of *CharData* to the I/O server (*IoDevice*).

read(Prompt) -> Result

read(IoDevice, Prompt) -> Result

Types:

IoDevice = device()
Prompt = prompt()
Result =

{ok, Term :: term()} | server_no_data() | {error, ErrorInfo}
ErrorInfo = erl_scan:error_info() | erl_parse:error_info()
server_no_data() = {error, ErrorDescription :: term()} | eof

Reads a term *Term* from the standard input (*IoDevice*), prompting it with *Prompt*.

The function returns:

{ok, Term}:
The parsing was successful.

eof:
End of file was encountered.

{error, ErrorInfo}:
The parsing failed.

{error, ErrorDescription}:
Other (rare) error condition, such as *{error, estale}* if reading from an NFS file system.

read(IoDevice, Prompt, StartLocation) -> Result

read(IoDevice, Prompt, StartLocation, Options) -> Result

Types:

IoDevice = device()
Prompt = prompt()
StartLocation = location()
Options = erl_scan:options()
Result =
{ok, Term :: term(), EndLocation :: location()} |
{eof, EndLocation :: location()} |
server_no_data() |
{error, ErrorInfo, ErrorLocation :: location()}
ErrorInfo = erl_scan:error_info() | erl_parse:error_info()
server_no_data() = {error, ErrorDescription :: term()} | eof

Reads a term *Term* from *IoDevice*, prompting it with *Prompt*. Reading starts at location

StartLocation. Argument *Options* is passed on as argument *Options* of function *erl_scan:tokens/4*.

The function returns:

{ok, Term, EndLocation}:
The parsing was successful.

{eof, EndLocation}:
End of file was encountered.

{error, ErrorInfo, ErrorLocation}:
The parsing failed.

{error, ErrorDescription}:
Other (rare) error condition, such as *{error, estale}* if reading from an NFS file system.

rows() -> {ok, integer() >= 1} | {error, enotsup}

rows(IoDevice) -> {ok, integer() >= 1} | {error, enotsup}

Types:

IoDevice = device()

Retrieves the number of rows of *IoDevice* (that is, the height of a terminal). The function only succeeds for terminal devices, for all other I/O devices the function returns *{error, enotsup}*.

scan_erl_exprs(Prompt) -> Result

scan_erl_exprs(Device, Prompt) -> Result

scan_erl_exprs(Device, Prompt, StartLocation) -> Result

scan_erl_exprs(Device, Prompt, StartLocation, Options) -> Result

Types:

Device = device()

Prompt = prompt()
StartLocation = location()
Options = erl_scan:options()
Result = erl_scan:tokens_result() | server_no_data()
server_no_data() = {error, ErrorDescription :: term()} | eof

Reads data from the standard input (*IoDevice*), prompting it with *Prompt*. Reading starts at location *StartLocation* (*1*). Argument *Options* is passed on as argument *Options* of function *erl_scan:tokens/4*. The data is tokenized as if it were a sequence of Erlang expressions until a final dot (.) is reached. This token is also returned.

The function returns:

{ok, Tokens, EndLocation}:
The tokenization succeeded.

{eof, EndLocation}:
End of file was encountered by the tokenizer.

eof:
End of file was encountered by the I/O server.

{error, ErrorInfo, ErrorLocation}:
An error occurred while tokenizing.

{error, ErrorDescription}:
Other (rare) error condition, such as *{error, estale}* if reading from an NFS file system.

Example:

```
23> io:scan_erl_exprs('enter>').
enter>abc(), "hey".
{ok,[{atom,1,abc},{'(',1},{')',1},{',',1},{string,1,"hey"},{dot,1}],2}
24> io:scan_erl_exprs('enter>').
enter>1.0er.
{error,{1,erl_scan,{illegal,float}},2}
```

scan_erl_form(Prompt) -> Result

scan_erl_form(IoDevice, Prompt) -> Result

scan_erl_form(IoDevice, Prompt, StartLocation) -> Result

scan_erl_form(IoDevice, Prompt, StartLocation, Options) -> Result

Types:

IoDevice = device()
Prompt = prompt()
StartLocation = location()
Options = erl_scan:options()
Result = erl_scan:tokens_result() | server_no_data()
server_no_data() = {error, ErrorDescription :: term()} | eof

Reads data from the standard input (*IoDevice*), prompting it with *Prompt*. Starts reading at location *StartLocation* (*1*). Argument *Options* is passed on as argument *Options* of function *erl_scan:tokens/4*. The data is tokenized as if it was an Erlang form (one of the valid Erlang expressions in an Erlang source file) until a final dot (.) is reached. This last token is also returned.

The return values are the same as for *scan_erl_exprs/1,2,3,4*.

setopts(Opts) -> ok | {error, Reason}

setopts(IoDevice, Opts) -> ok | {error, Reason}

Types:

IoDevice = device()
Opts = [setopt()]
Reason = term()

Set options for the standard I/O device (*IoDevice*).

Possible options and values vary depending on the I/O device. For a list of supported options and their current values on a specific I/O device, use function *getopts/1*.

The options and values supported by the OTP I/O devices are as follows:

binary, *list*, or *{binary, boolean()}*:

If set in binary mode (*binary* or *{binary, true}*), the I/O server sends binary data (encoded in UTF-8) as answers to the *get_line*, *get_chars*, and, if possible, *get_until* requests (for details, see section The Erlang I/O Protocol) in the User's Guide). The immediate effect is that *get_chars/2,3* and *get_line/1,2* return UTF-8 binaries instead of lists of characters for the affected I/O device.

By default, all I/O devices in OTP are set in *list* mode. However, the I/O functions can handle any of these modes and so should other, user-written, modules behaving as clients to I/O servers.

This option is supported by the standard shell (*group.erl*), the 'oldshell' (*user.erl*), and the file I/O servers.

{echo, boolean()}:

Denotes if the terminal is to echo input. Only supported for the standard shell I/O server (*group.erl*)

{expand_fun, expand_fun()}:

Provides a function for tab-completion (expansion) like the Erlang shell. This function is called when the user presses the *Tab* key. The expansion is active when calling line-reading functions, such as *get_line/1,2*.

The function is called with the current line, up to the cursor, as a reversed string. It is to return a three-tuple: *{yes|no, string(), [string(), ...]}*. The first element gives a beep if *no*, otherwise the expansion is silent; the second is a string that will be entered at the cursor position; the third is a list of possible expansions. If this list is not empty, it is printed and the current input line is written once again.

Trivial example (beep on anything except empty line, which is expanded to *"quit"*):

```
fun("") -> {yes, "quit", []};
  (_) -> {no, "", ["quit"]} end
```

This option is only supported by the standard shell (*group.erl*).

{encoding, latin1 | unicode}:

Specifies how characters are input or output from or to the I/O device, implying that, for example, a terminal is set to handle Unicode input and output or a file is set to handle UTF-8

data encoding.

The option *does not* affect how data is returned from the I/O functions or how it is sent in the I/O protocol, it only affects how the I/O device is to handle Unicode characters to the "physical" device.

The standard shell is set for *unicode* or *latin1* encoding when the system is started. The encoding is set with the help of the *LANG* or *LC_CTYPE* environment variables on Unix-like system or by other means on other systems. So, the user can input Unicode characters and the I/O device is in *{encoding, unicode}* mode if the I/O device supports it. The mode can be changed, if the assumption of the runtime system is wrong, by setting this option.

The I/O device used when Erlang is started with the "-oldshell" or "-noshell" flags is by default set to *latin1* encoding, meaning that any characters > codepoint 255 are escaped and that input is expected to be plain 8-bit ISO Latin-1. If the encoding is changed to Unicode, input and output from the standard file descriptors are in UTF-8 (regardless of operating system).

Files can also be set in *{encoding, unicode}*, meaning that data is written and read as UTF-8. More encodings are possible for files, see below.

{encoding, unicode | latin1} is supported by both the standard shell (*group.erl* including *werl* on Windows), the ’oldshell’ (*user.erl*), and the file I/O servers.

{encoding, utf8 | utf16 | utf32 | {utf16,big} | {utf16,little} | {utf32,big} | {utf32,little}}:
For disk files, the encoding can be set to various UTF variants. This has the effect that data is expected to be read as the specified encoding from the file, and the data is written in the specified encoding to the disk file.

{encoding, utf8} has the same effect as *{encoding, unicode}* on files.

The extended encodings are only supported on disk files (opened by function *file:open/2*).

write(Term) -> ok

write(IoDevice, Term) -> ok

Types:

IoDevice = device()
Term = term()

Writes term *Term* to the standard output (*IoDevice*).

STANDARD INPUT/OUTPUT

All Erlang processes have a default standard I/O device. This device is used when no *IoDevice* argument is specified in the function calls in this module. However, it is sometimes desirable to use an explicit *IoDevice* argument that refers to the default I/O device. This is the case with functions that can access either a file or the default I/O device. The atom *standard_io* has this special meaning. The following example illustrates this:

```
27> io:read('enter>').
enter>foo.
{ok,foo}
28> io:read(standard_io, 'enter>').
enter>bar.
{ok,bar}
```

standard_io is an alias for *group_leader/0*, so in order to change where the default input/output requests are sent you can change the group leader for the current process using *group_leader(NewGroupLeader, self()).*

There is always a process registered under the name of *user*. This can be used for sending output to the user.

STANDARD ERROR

In certain situations, especially when the standard output is redirected, access to an I/O server specific for error messages can be convenient. The I/O device *standard_error* can be used to direct output to whatever the current operating system considers a suitable I/O device for error output. Example on a Unix-like operating system:

```
$ erl -noshell -noinput -eval 'io:format(standard_error,"Error: ~s~n",["error 11"]),'\
'init:stop().' > /dev/null
Error: error 11
```

ERROR INFORMATION

The *ErrorInfo* mentioned in this module is the standard *ErrorInfo* structure that is returned from all I/O

modules. It has the following format:

{ErrorLocation, Module, ErrorDescriptor}

A string that describes the error is obtained with the following call:

Module:format_error(ErrorDescriptor)