Documentation Library

Gamma™ Reference Volume 2

Version 7.2

Cogent Real-Time Systems, Inc.

August 15, 2012

Gamma™ Reference Volume 2: Version 7.2

A dynamically-typed interpreted programming language specifically designed to allow rapid development of control and user interface applications. Gamma has a syntax similar to C and C++, but has a range of built-in features that make it a far better language for developing sophisticated real-time systems.

Published August 15, 2012
Cogent Real-Time Systems, Inc.
162 Guelph Street, Suite 253
Georgetown, Ontario
Canada, L7G 5X7

Toll Free: 1 (888) 628-2028
Tel: 1 (905) 702-7851
Fax: 1 (905) 702-7850

Information Email: info@cogent.ca
Tech Support Email: support@cogent.ca
Web Site: www.cogent.ca

Revision History

Revision 7.2-1 September 2007
Updated DataHub-related functions for 6.4 release of the DataHub.
Revision 6.2-1 February 2005
Simplified TCP connectivity.
Revision 4.1-1 August 2004
Compatible with Cogent DataHub Version 5.0.
Revision 4.0-2 October 2001
New functions in Input/Output, OSAPIs, Date, and Dynamic Loading reference sections.
Revision 4.0-1 September 2001
Source code compatible across QNX 4, QNX 6, and Linux.
Revision 3.2-1 August 2000
Renamed "Gamma", changed function syntax.
Revision 3.0 October 1999
General reorganization and update of Guide and Reference, released in HTML and QNX Helpviewer formats.
Revision 2.1 June 1999
Converted from Word97 to DocBook SGML.
Revision 2.0 June 1997
Initial release of hardcopy documentation.

Copyright, trademark, and software license information.

Copyright Notice

Trademark Notice

END-USER LICENSE AGREEMENT FOR COGENT SOFTWARE

IMPORTANT - READ CAREFULLY: This End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and Cogent Real-Time Systems Inc. ("Cogent") of 162 Guelph Street, Suite 253, Georgetown, Ontario, L7G 5X7, Canada (Tel: 905-702-7851, Fax: 905-702-7850), from whom you acquired the Cogent software product(s) ("SOFTWARE PRODUCT" or "SOFTWARE"), either directly from Cogent or through one of Cogent's authorized resellers.
The SOFTWARE PRODUCT includes computer software, any associated media, any printed materials, and any "online" or electronic documentation. By installing, copying or otherwise using the SOFTWARE PRODUCT, you agree to be bound by the terms of this EULA. If you do not agree with the terms of this EULA, Cogent is unwilling to license the SOFTWARE PRODUCT to you. In such event, you may not use or copy the SOFTWARE PRODUCT, and you should promptly contact Cogent for instructions on return of the unused product(s) for a refund.

SOFTWARE PRODUCT LICENSE

The SOFTWARE PRODUCT is protected by copyright laws and copyright treaties, as well as other intellectual property laws and treaties. The SOFTWARE PRODUCT is licensed, not sold.

1. **EVALUATION USE:** This software is distributed as "Free for Evaluation", and with a per-use royalty for Commercial Use, where "Free for Evaluation" means to evaluate Cogent's software and to do exploratory development and "proof of concept" prototyping of software applications, and where "Free for Evaluation" specifically excludes without limitation:
    i. use of the SOFTWARE PRODUCT in a business setting or in support of a business activity,
    ii. development of a system to be used for commercial gain, whether to be sold or to be used within a company, partnership, organization or entity that transacts commercial business,
    iii. the use of the SOFTWARE PRODUCT in a commercial business for any reason other than exploratory development and "proof of concept" prototyping, even if the SOFTWARE PRODUCT is not incorporated into an application or product to be sold,
    iv. the use of the SOFTWARE PRODUCT to enable the use of another application that was developed with the SOFTWARE PRODUCT,
    v. inclusion of the SOFTWARE PRODUCT in a collection of software, whether that collection is sold, given away, or made part of a larger collection.
    vi. inclusion of the SOFTWARE PRODUCT in another product, whether or not that other product is sold, given away, or made part of a larger product.

2. **COMMERCIAL USE:** COMMERCIAL USE is any use that is not specifically defined in this license as EVALUATION USE.

3. **GRANT OF LICENSE:** This EULA covers both COMMERCIAL and EVALUATION USE of the SOFTWARE PRODUCT. Either clause (A) or (B) of this section will apply to you, depending on your actual use of the SOFTWARE PRODUCT. If you have not purchased a license of the SOFTWARE PRODUCT from Cogent or one of Cogent's authorized resellers, then you may not use the product for COMMERCIAL USE.
    A. **GRANT OF LICENSE (EVALUATION USE):** This EULA grants you the following non-exclusive rights when used for EVALUATION purposes:

Software: You may use the SOFTWARE PRODUCT on any number of computers, either stand-alone, or on a network, so long as every use of the SOFTWARE PRODUCT is for EVALUATION USE. You may reproduce the SOFTWARE PRODUCT, but only as reasonably required to install and use it in accordance with this LICENSE or to follow your normal back-up practices.

Subject to the license expressly granted above, you obtain no right, title or interest in or to the SOFTWARE PRODUCT or related documentation, including but not limited to any copyright, patent, trade secret or other proprietary rights therein. All whole or partial copies of the SOFTWARE PRODUCT remain property of Cogent and will be considered part of the SOFTWARE PRODUCT for the purpose of this EULA.

Unless expressly permitted under this EULA or otherwise by Cogent, you will not:

i. use, reproduce, modify, adapt, translate or otherwise transmit the SOFTWARE PRODUCT or related components, in whole or in part;

ii. rent, lease, license, transfer or otherwise provide access to the SOFTWARE PRODUCT or related components;

iii. alter, remove or cover proprietary notices in or on the SOFTWARE PRODUCT, related documentation or storage media;

iv. export the SOFTWARE PRODUCT from the country in which it was provided to you by Cogent or its authorized reseller;

v. use a multi-processor version of the SOFTWARE PRODUCT in a network larger than that for which you have paid the corresponding multi-processor fees;

vi. decompile, disassemble or otherwise attempt or assist others to reverse engineer the SOFTWARE PRODUCT;

vii. circumvent, disable or otherwise render ineffective any demonstration time-outs, locks on functionality or any other restrictions on use in the SOFTWARE PRODUCT;

viii. circumvent, disable or otherwise render ineffective any license verification mechanisms used by the SOFTWARE PRODUCT;

ix. use the SOFTWARE PRODUCT in any application that is intended to create or could, in the event of malfunction or failure, cause personal injury or property damage; or

x. make use of the SOFTWARE PRODUCT for commercial gain, whether directly, indirectly or incidentally.

B. **GRANT OF LICENSE (COMMERCIAL USE):** This EULA grants you the following non-exclusive rights when used for COMMERCIAL purposes:

Software: You may use the SOFTWARE PRODUCT on one computer, or if the SOFTWARE PRODUCT is a multi-processor version - on one node of a network, either: (i) as a development systems for the purpose of creating value-added software applications in accordance with related Cogent documentation; or (ii) as a single run-time copy for use as an integral part of such an application. This includes reproduction and configuration of the SOFTWARE PRODUCT, but only as reasonably required to install and use it in association with your licensed processor or to follow your normal back-up practices.

Storage/Network Use: You may also store or install a copy of the SOFTWARE PRODUCT on one computer to allow your other computers to use the SOFTWARE PRODUCT over an internal network, and distribute the SOFTWARE PRODUCT to your other computers over an internal network. However, you must acquire and dedicate a license for the SOFTWARE PRODUCT for each computer on which the SOFTWARE PRODUCT is used or to which it is distributed. A license for the SOFTWARE PRODUCT may not be shared or used concurrently on different computers.

Subject to the license expressly granted above, you obtain no right, title or interest in or to the SOFTWARE PRODUCT or related documentation, including but not limited to any copyright, patent, trade secret or other proprietary rights therein. All whole or partial copies of the SOFTWARE PRODUCT remain property of Cogent and will be considered part of the SOFTWARE PRODUCT for the purpose of this EULA.

Unless expressly permitted under this EULA or otherwise by Cogent, you will not:

i. use, reproduce, modify, adapt, translate or otherwise transmit the SOFTWARE PRODUCT or related components, in whole or in part;

ii. rent, lease, license, transfer or otherwise provide access to the SOFTWARE PRODUCT or related components;

iii. alter, remove or cover proprietary notices in or on the SOFTWARE PRODUCT, related documentation or storage media;

iv. export the SOFTWARE PRODUCT from the country in which it was provided to you by Cogent or its authorized reseller;

v. use a multi-processor version of the SOFTWARE PRODUCT in a network larger than that for which you have paid the corresponding multi-processor fees;

vi. decompile, disassemble or otherwise attempt or assist others to reverse engineer the SOFTWARE PRODUCT;

vii. circumvent, disable or otherwise render ineffective any demonstration time-outs, locks on functionality or any other restrictions on use in the SOFTWARE PRODUCT;

viii. circumvent, disable or otherwise render ineffective any license verification mechanisms used by the SOFTWARE PRODUCT, or

ix. use the SOFTWARE PRODUCT in any application that is intended to create or could, in the event of malfunction or failure, cause personal injury or property damage.

4. **WARRANTY:** Cogent cannot warrant that the SOFTWARE PRODUCT will function in accordance with related documentation in every combination of hardware platform, software environment and SOFTWARE PRODUCT configuration. You acknowledge that software bugs are likely to be identified when the SOFTWARE PRODUCT is used in your particular application. You therefore accept the responsibility of satisfying yourself that the SOFTWARE PRODUCT is suitable for your intended use. This includes conducting exhaustive testing of your application prior to its initial release and prior to the release of any related hardware or software modifications or enhancements.

   Subject to documentation errors, Cogent warrants to you for a period of ninety (90) days from acceptance of this EULA (as provided above) that the SOFTWARE PRODUCT as delivered by Cogent is capable of performing the functions described in related Cogent user documentation when used on appropriate hardware. Cogent also warrants that any enclosed disk(s) will be free from defects in material and workmanship under normal use for a period of ninety (90) days from acceptance of this EULA. Cogent is not responsible for disk defects that result from accident or abuse. Your sole remedy for any breach of warranty will be either: i) terminate this EULA and receive a refund of any amount paid to Cogent for the SOFTWARE PRODUCT, or ii) to receive a replacement disk.

5. **LIMITATIONS:** Except as expressly warranted above, the SOFTWARE PRODUCT, any related documentation and disks are provided "as is" without other warranties or conditions of any kind, including but not limited to implied warranties of merchantability, fitness for a particular purpose and non-infringement. You assume the entire risk as to the results and performance of the SOFTWARE PRODUCT. Nothing stated in this EULA will imply that the operation of the SOFTWARE PRODUCT will be uninterrupted or error free or that any errors will be corrected. Other written or oral statements by Cogent, its representatives or others do not constitute warranties or conditions of Cogent.

   In no event will Cogent (or its officers, employees, suppliers, distributors, or licensors: collectively "Its Representatives") be liable to you for any indirect, incidental, special or consequential damages whatsoever, including but not limited to loss of revenue, lost or damaged data or other commercial or economic loss, arising out of any breach of this EULA, any use or inability to use the SOFTWARE PRODUCT or any claim made by a third party, even if Cogent (or Its Representatives) have been advised of the possibility of such damage or claim. In no event will the aggregate liability of Cogent (or that of Its Representatives) for any damages or claim, whether in contract, tort or otherwise, exceed the amount paid by you for the SOFTWARE PRODUCT.

   These limitations shall apply whether or not the alleged breach or default is a breach of a fundamental condition or term, or a fundamental breach. Some jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, or certain limitations of implied warranties. Therefore the above limitation may not apply to you.

6. **DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS:**

   Separation of Components. The SOFTWARE PRODUCT is licensed as a single product. Its component parts may not be separated for use on more than one computer.

Termination. Without prejudice to any other rights, Cogent may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In such an event, you must destroy all copies of the SOFTWARE PRODUCT and all of its component parts.

7. **UPGRADES:** If the SOFTWARE PRODUCT is an upgrade from another product, whether from Cogent or another supplier, you may use or transfer the SOFTWARE PRODUCT only in conjunction with that upgrade product, unless you destroy the upgraded product. If the SOFTWARE PRODUCT is an upgrade of a Cogent product, you now may use that upgraded product only in accordance with this EULA. If the SOFTWARE PRODUCT is an upgrade of a component of a package of software programs which you licensed as a single product, the SOFTWARE PRODUCT may be used and transferred only as part of that single product package and may not be separated for use on more than one computer.
8. **COPYRIGHT:** All title and copyrights in and to the SOFTWARE PRODUCT (including but not limited to any images, photographs, animations, video, audio, music, text and ’applets", incorporated into the SOFTWARE PRODUCT), any accompanying printed material, and any copies of the SOFTWARE PRODUCT, are owned by Cogent or its suppliers. You may not copy the printed materials accompanying the SOFTWARE PRODUCT. All rights not specifically granted under this EULA are reserved by Cogent.
9. **PRODUCT SUPPORT:** Cogent has no obligation under this EULA to provide maintenance, support or training.
10. **RESTRICTED RIGHTS:** Use, duplication, or disclosure by the U.S. Government is subject to restrictions as provided in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (OCT 1988), FAR 12.212(a)(1995), FAR 52.227-19, or FAR 52.227-14 (ALT III), as appropriate. Manufacturer is Cogent Real-Time Systems Inc. 162 Guelph Street, Suite 253, Georgetown, Ontario, L7G 5X7, Canada.
11. **GOVERNING LAW:** This Software License Agreement is governed by the laws of the Province of Ontario, Canada. You irrevocably attorn to the jurisdiction of the courts of the Province of Ontario and agree to commence any litigation that may arise hereunder in the courts located in the Judicial District of Peel, Province of Ontario.

Table of Contents

List of Tables

Chapter 1. What is Gamma?

Gamma is an interpreter, a high-level programming language that has been designed and optimized to reduce the time required for building applications. It has support for the Photon GIU in QNX, and the GTK GUI in Linux and QNX 6. It also has extensions that support HTTP and MySQL.

With Gamma a user can quickly implement algorithms that are far harder to express in other languages such as C. Gamma lets the developer take advantage of many time-saving features such as memory management and improved GUI support. These features, coupled with the ability to fully interact with and debug programs as they run, mean that developers can build, test and refine applications in a shorter time frame than when using other development platforms.

Gamma programs are small, fast and reliable. Gamma is easily embedded into today's smart appliances and web devices.

Gamma is an improved and expanded version of our previous Slang Programming Language for QNX and Photon. Gamma is available on QNX 4, QNX 6 and Linux, and is being ported to Microsoft Windows.

The implementation of Gamma is based on a powerful SCADALisp engine. SCADALisp is a dialect of the Lisp programming language which has been optimized for performance and memory usage, and enhanced with a number of internal functions. All references in this manual to Lisp are in fact to the SCADALisp dialect of Lisp.

You could say Gamma's object language is Lisp, just like Assembler is the object language for C. Knowing Lisp is not a requirement for using Gamma, but it can be helpful. All necessary information on Lisp and how it relates to Gamma is in the Input and Output chapter of this guide.

Chapter 2. System Requirements

QNX 6

- QNX 6.1.0 or later.

QNX 4

- QNX 4.23A or later.
- (For Gamma/Photon) Photon 1.14 or later.

Linux

- Linux 2.4 or later.
- (For Gamma/GTK) GTK 1.2.8.
- The SRR IPC kernel module, which includes a synchronous message passing library modeled on the QNX 4 send/receive/reply message-passing API. This module installs automatically, but requires a C compiler for the installation. You can get more information and/or download this module at the Cogent Web Site.

  

  This module may not be necessary for some Gamma applications, but it is required for any use of timers, event handling, or inter-process communication.

I. Input/Output

Table of Contents

close

`close` — closes an open file.

Syntax

`close (`*`file`*`)`

Arguments

`file`

A file pointer to a previously opened file. This may be either a file in the file system, or a string opened for read and write.

Returns

`t` if the *`file`* had been open and was closed successfully, else `nil`.

Description

This function closes a previously opened file. It is not strictly necessary, as the file will be closed when the garbage collector recognizes that there are no references to the file, but it is extremely good policy. This function will close a string opened for reading and writing as well.

Example

```
Gamma> fp = open("myfile.dat","r");
#<File:"myfile.dat">
Gamma> close(fp);
t
Gamma> fp;
#<Destroyed Instance>
Gamma>
```

See Also

`fd_close`, `open`, `open_string`

fd_close

`fd_close` — closes a file identified by a file descriptor.

Syntax

`fd_close (`*`fd`*`)`

Arguments

`fd`

A file descriptor as returned from `fd_open`.

Returns

`t`, if successful, otherwise `nil`.

Description

This function closes a file identifed by a file descriptor, ie. that was opened by `fd_open`.

Example

```
Gamma> require_lisp("const/filesys");
"const/filesys"
Gamma> fp = fd_open("/fd/ttyp8",O_WRONLY);
4
Gamma> fd_write(fp,"\nHello\n");
8
Gamma> fd_close(fp);
t
Gamma> fd_close(fp);
nil
```

See Also

`close`, `fd_open`, Referencing Files

fd_data_function

`fd_data_function` — attaches a write-activated callback to a file.

Syntax

```
fd_data_function (fd|file, code)
```

Arguments

`fd`|*`file`*

A file descriptor as returned from `fd_open`, or the name of a file pointer to a file that was opened by a call to `open` or `open_string`.

`code`

Any valid Gamma program, executable code block, or statement.

Returns

The return value of the executed *`code`*.

Description

This function acts as a callback, causing the *`code`* to execute whenever data is written to the file associated with the *`fd`* or *`file`* pointer.

See Also

`fd_open`, `fd_eof_function`, `fd_write`, `open`, `open_string`, `write`

fd_eof_function

fd_eof_function — attaches an eof-activated callback to a file.

Syntax

```
fd_eof_function (fd|file, code)
```

Arguments

fd|*file*

A file descriptor as returned from fd_open, or the name of a file pointer to a file that was opened by a call to open or open_string.

code

Any valid Gamma program, executable code block, or statement.

Returns

The return value of the executed *code*.

Description

This function acts as a callback, causing the *code* to execute whenever the end of the file (_eof_) is reached during a call to fd_read or one of the other read functions. The *fd*|*file* argument identifies the file.

See Also

fd_open, fd_data_function, fd_write, open, open_string, write

fd_open

`fd_open` — opens a file or device and assigns it a file descriptor.

Syntax

```
fd_open (name, mode)
```

Arguments

name

The name of a file, as a string.

mode

The mode for opening the file.

Returns

A non-negative integer representing the lowest numbered unused file descriptor if successful. If an error occurs, the function returns -1 and sets the errno.

Description

This function opens a file for reading and/or writing, and assigns it a file descriptor which is used as an argument by other functions such as `fd_read` and `fd_write`. The file that is opened could be a regular file, a directory, or a block or character device. Legal *mode* values are:

- **O_RDONLY** Read-only mode
- **O_WRONLY** Write-only mode
- **O_RDWR** Read-Write mode

Any combination of the following flags may be bitwise OR-ed with the open mode to modify how the file is accessed:

- **O_APPEND** Append (writes guaranteed at the end)
- **O_CREAT** Opens with file create
- **O_EXCL** Exclusive open
- **O_NOCTTY** Don’t assign a controlling terminal
- **O_NONBLOCK** Non-blocking I/O
- **O_TRUNC** Open with truncation
- **O_DSYNC** Data integrity synch
- **O_SYNC** File integrity synch
- **O_TEMP** Temporary file, don’t put to disk
- **O_CACHE** Cache sequential files too

If an error occurs -1 is returned and errno is set to one of the following:

- **EACCES** Search permission denied on a portion of the path prefix, or the file exists and the permissions required to open the file in the given mode so not exist.
- **EBADFSYS** The file or the path prefix to the file was found to be corrupted
- **EBUSY** The file is already open for writing.
- **EEXIST** O_CREAT and O_EXCL are set and the named file exists
- **EINTR** The function was interrupted by a signal
- **EISDIR** The named file is a directory
- **EMFILE** Too many file descriptors are currently in use by this process
- **ENAMETOOLONG** The length of the path to the file is too long.
- **ENFILE** Too many files are currently open on the system
- **ENOENT** O_CREAT is not set and the file does not exist
- **ENOSPC** The directory or file system which would create the new file cannot be extended
- **ENOTDIR** A component of the path to the file is not a directory
- **ENXIO** O_NONBLOCK is set, the file is a FIFO, O_WRONLY is set, and no process has the file open for reading
- **EROFS** The named file resides on a read-only file system.

Example

```
Gamma> require_lisp("const/filesys");
"/usr/cogent/lib/const/filesys.lsp"
Gamma> ptr = fd_open("/fd/ttyp8",O_WRONLY);
4
Gamma> fd_write(ptr,"\nhello\n");
7
```

See Also

`fd_close`, `fd_data_function`, `fd_eof_function`, `fd_read`, `fd_write` `ser_setup`, Referencing Files

fd_read

`fd_read` — reads a buffer or string from a file identified by a file descriptor.

Syntax

```
fd_read (fd, buffer|string, length?, offset?)
```

Arguments

`fd`

A file descriptor as returned from `fd_open`.

`buffer|string`

A buffer or string to be read from the file.

`length`

An integer specifying the length of the buffer or string.

`offset`

An integer specifying the position in the file to begin reading the buffer or string.

Returns

The number of bytes actually read from the file, or -1 on failure and the errno is set.

Description

This function reads a buffer or string from the specified file.

When an error occurs, the following errnos are possible:

- **EAGAIN** The O_NONBLOCK flag is set for the *`fd`* and the process would be delayed in the read operation.
- **EBADF** The passed *`fd`* is invalid or not open for writing.
- **EFBIG** File is too big.
- **EINTR** Read was interrupted by a signal.
- **EINVAL** iovcnt was less than or equal to 0, or greater than UIO_MAXIOV.
- **EIO** Physical I/O error.

Example

```
Gamma> x = fd_open("/fd/ser1",O_RDWR);
4
Gamma> fd_read(x,"hello\n");
6
Gamma> fd_close(x);
t
```

See Also

`fd_close`, `fd_open`, `fd_read`, `ser_setup`, Referencing Files

fd_to_file

`fd_to_file` — creates a file pointer from a descriptor.

Syntax

```
fd_to_file (fd, mode)
```

Arguments

`fd`

A file descriptor as returned from `fd_open`.

`mode`

A string indicating the mode for the file: `"r"` for read-only, `"w"` for writable, `"a"` for append.

Returns

`t`, if successful, otherwise `nil`.

Description

This function creates a file pointer from a file descriptor.

See Also

`fileno`, Referencing Files

fd_write

`fd_write` — writes a buffer or string to a file identified by a file descriptor.

Syntax

```
fd_write (fd, buffer|string, length?, offset?)
```

Arguments

`fd`

A file descriptor as returned from `fd_open`.

`buffer`|*`string`*

A buffer or string to write to the file.

`length`

An integer specifying the length of the buffer or string.

`offset`

An integer specifying the position in the file to begin writing the buffer or string.

Returns

The number of bytes actually written to the file, or -1 on failure and the errno is set.

Description

This function writes a buffer or string to the specified file.

When an error occurs, the following errnos are possible:

- **EAGAIN** The O_NONBLOCK flag is set for the *`fd`* and the process would be delayed in the write operation.
- **EBADF** The passed *`fd`* is invalid or not open for writing.
- **EFBIG** File is too big.
- **EINTR** Write was interrupted by a signal.
- **EINVAL** iovcnt was less than or equal to 0, or greater than UIO_MAXIOV.
- **EIO** Physical I/O error.
- **ENOSPC** No free space remaining on drive.
- **EPIPE** Attempt to write to a pipe (or FIFO) that is not open for write. SIGPIPE is also sent to process.

Example

```
Gamma> x = fd_open("/fd/ser1",O_RDWR);
4
Gamma> fd_write(x,"hello\n");
6
Gamma> fd_close(x);
t
```

See Also

`fd_close`, `fd_open`, `fd_read`, `ser_setup`, Referencing Files

`fileno`

`fileno` — creates a file descriptor from a pointer.

Syntax

```
fileno (file)
```

Arguments

`file`

A file pointer as returned from `open`.

Returns

`t`, if successful, otherwise `nil`.

Description

This function creates a file descriptor from a file pointer.

See Also

`fd_to_file`, Referencing Files

ioctl

`ioctl` — performs control functions on a file descriptor.

Syntax

```
ioctl (fd, request, value)
```

Arguments

`fd`

A file descriptor as returned from `fd_open`.

`request`

One of the functions listed below in Description.

`value`

A number that supplies additional information needed by the *`request`* function.

Returns

The return value of the *`request`* function.

Description

This function performs an `ioctl` call (C library `ioctl` subroutine) for the given *`fd`* file descriptor and *`request`*. The Gamma `ioctl` function currently only supports *`request`*s that take numeric arguments, ie. *`value`* must be a number. You may make operating-system specific `ioctl` calls by giving a numeric value for the *`request`* argument.

The currently supported *`request`*s are:

TCSBRK	TCXONC	TCFLSH	TIOCHPCL	TIOCEXCL	TIOXNXCL
TIOCFLUSH	TIOCDRAIN	TIOCSCTTY	TIOCMGET	TIOCMBIC	TIOCMBIS
TIOCMSET	TIOCSTART	TIOCSTOP	TIOCNOTTY	TIOCOUTQ	TIOCSPGRP
TIOCGPGRP	TIOCCDIR	TIOCSDIR	TIOCCBRK	TIOCSBRK	TIOCLGET
TIOCLSET	TIOCSETPGRP	TIOCGETPGRP	FIOCLEX	FIONCLEX	FIOGETOWN
FIOSETOWN	FIOASYNC	FIONBIO	FIONReAd	SIOCSHIWAT	SIOCGHIWAT
SIOCSLOWAT	SIOCGLOWAT	SIOCATMARK	SIOCSPGRP	SIOCGPGRP	

open

`open` — attempts to open a file.

Syntax

`open (`*`filename, mode, use_parser`*`?)`

Arguments

`filename`

A filename (possibly including the path), as a string.

`mode`

A string indicating the mode for the file: `"r"` for read-only, `"w"` for writable, `"a"` for append.

`use_parser`

Assume Lisp grammar regardless of the default grammar.

Returns

A file pointer, or `nil` if the request failed.

Description

This function attempts to open a file. If the file is opened for write (`"w"`), any previously existing file of the same name will be destroyed. If the file is opened for append (`"a"`) then a previously existing file will be lengthened with subsequent writes, but the data in that file will not be damaged. A file can only be opened read-only (`"r"`) if it already exists. The result of this function may be used as an argument to a variety of read and write operations.

When Gamma opens or creates a file, it creates an abstract *file pointer*. A printed representation of the file pointer looks like this: `#<File:`*`filename`*`>`. This representation cannot be read back in to Gamma, and so a symbol must be assigned to the file pointer in order to refer to or work with a file. In common language, we refer to this symbol as the file pointer. For instance, in the examples below, we would say the symbol `fp` is the file pointer. (See also Referencing Files.)

If *`use_parser`* is non-`nil`, then a call to read will parse the file according to its default grammar. If *`use_parser`* is `nil`, then a call to read will parse the file as if it were a Lisp expression. A file must be opened in Lisp format in order to use calls to `read_char`, `read_double`, `read_float`, `read_line`, `read_long`, `read_short` and `read_until`.

Examples

An input file contains the following:

```
(setq y 5)
```

Calling `open` will produce:

```
Gamma> fp = open ("myopenfile.dat", "r", nil);
#<File:"myopenfile.dat">
Gamma> princ(read_line(fp), "\n");
(setq y 5)
t
Gamma> fp = open ("myopenfile.dat", "r", nil);
```

```
#<File:"myopenfile.dat">
Gamma> eval (read(fp));
5
Gamma> fp = open ("myopenfile.dat", "r", t);
#<File:"myopenfile.dat">
Gamma> princ(read_line(fp), "\n");
(setq y 5)
t
Gamma> fp = open ("myopenfile.dat", "r", t);
#<File:"myopenfile.dat">
Gamma> eval (read(fp));
Error: ./generate.slg: line 1: Malformed expression within ()
Error: ./generate.slg: line 1: Unexpected end of file
Macro read left extra stuff on the LISP stack: 8098478, 8098470
nil
nil
Gamma>
```

The following example opens and reads a file, if it exists. If not, it prints and error message.

```
if ( (fp=open("myfile","r")) != nil )
{
  local line;
  while((line = read_line(fp)) != _eof_)
    {
      princ(line, "\n");
    }
  close(fp);
}
else
{
 princ("Error : unable to open myfile for read\n");
}
```

See Also

close, fd_open, open_string, read, read_char, read_double, read_float, read_line, read_long, read_short, read_until, seek, tell, terpri, write, writec

pipe

`pipe` — creates a pipe.

Syntax

```
pipe ()
```

Arguments

none

Returns

A list of the read pipe and the write pipe, each as a file pointer.

Description

This function creates an un-named pipe.

Example

```
Gamma> pipe1 = pipe();
(#<File:"read_pipe"> #<File:"write_pipe">)
Gamma> pread = car(pipe1);
#<File:"read_pipe">
Gamma> pwrite = cadr(pipe1);
#<File:"write_pipe">
Gamma> write (pwrite, "This is a test");
t
Gamma> read (pread);
"This is a test"
Gamma>
```

princ, print, pretty_princ, pretty_print

princ, print, pretty_princ, pretty_print — write to the standard output file.

Syntax

```
princ (s_exp...)
print (s_exp...)
pretty_princ (s_exp...)
pretty_print (s_exp...)
```

Arguments

s_exp

Any Gamma or Lisp expression.

Returns

t

Description

These functions write to the standard output file, typically the screen. The princ and pretty_princ functions produce formatted output, which means that special characters are not escaped, and double quotes are not printed around character strings. Output generated by princ cannot be read by the Lisp reader.

print and pretty_print produce Lisp-readable output. The result of reading a printed expression using a call to read will generate an equal expression.

pretty_princ and pretty_print generate carriage returns and spaces with the intention of formatting the output to make long or complex Lisp expressions easier for a person to read.

Examples

```
Gamma> x = "hello";
"hello"
Gamma> print (x,"\n");
"hello""\n"t
Gamma> princ (x,"\n");
hello
t
Gamma> >

Gamma> class C {a; b; c;}
(defclass C nil [][a b c])
Gamma> princ (C);
(defclass C nil [][a b c])t
Gamma> pretty_princ (C);
(defclass C nil
[]
[a b c])t
Gamma>

Gamma> L = list (1,2,3,4,5,list(1,2,3,4,5,list(1,2,3
list(1,2,3,4,5,list(1,2,3,4,5,list(1,2,3,4,5,list(1,2
,list(1,2,3,4,5,list(1,2,3,4,5)))))))));
(1 2 3 4 5 (1 2 3 4 5 (1 2 3 4 5 (1 2 3 4 5 (1 2 3 4
4 5 (1 2 3 4 5 (1 2 3 4 5 (1 2 3 4 5 (1 2 3 4 5)))))
Gamma> princ (L);
```

```
(1 2 3 4 5 (1 2 3 4 5 (1 2 3 4 5 (1 2 3 4 5 (1 2 3 4
4 5 (1 2 3 4 5 (1 2 3 4 5 (1 2 3 4 5 (1 2 3 4 5)))))
Gamma> pretty_princ (L);

(1 2 3 4 5
   (1 2 3 4 5
      (1 2 3 4 5
         (1 2 3 4 5
            (1 2 3 4 5
               (1 2 3 4 5
                  (1 2 3 4 5 (1 2 3 4 5 (1 2 3 4 5 (
))))))))))t
Gamma>
```

See Also

write, writec, pretty_write

pty, ptytio

pty, ptytio — run programs in a pseudo-tty.

Syntax

```
pty (program, arguments...? = nil)
ptytio (termios, program, arguments...? = nil)
```

Arguments

program

A string containing the name of the program to be executed.

arguments

A string containing any command-line arguments for the program.

termios

A termios structure.

Returns

A list of:

```
(process_id file_for_stdin file_for_stdout file_for_stderr pty_name)
```

Where:

process_id

The process ID of the program called.

file_for_stdin

A pointer to the file used for STDIN.

file_for_stdout

A pointer to the file used for STDOUT.

file_for_stderr

A pointer to the file used for STDERR.

pty_name

The path and filename of this pseudo-tty, as a string.

Description

These functions run programs in a pseudo-tty. A Gamma program can read from either program's standard output by issuing a read or read_line call on *file_for_stdout*. The process can be reaped using wait.

The ptytio function is the same as pty, but the first argument is a termios structure. This is useful if particular terminal characteristics are required on the pty. The termios structure is only available through the gammatios.so dynamic library.

Example

This example calls `pty` on the following test program, called `testpty.g`:

```
#!/usr/cogent/bin/gamma
princ("Test output.\n");
princ(cadr(argv),"\n");
```

Here we call `pty` in interactive mode, and then read the output:

```
Gamma> ptylist = pty("testpty.g", "Argument");
(4760 #<File:"testpty.g-stdin"> #<File:"testpty.g-stdout">
#<File:"testpty.g-stderr"> "/dev/ptyp0")
Gamma> read_line(caddr(ptylist));
"This software is free for non-commercial use, and no valid commercial license"
Gamma> read_line(caddr(ptylist));
"is installed.  For more information, please contact info@cogent.ca."
Gamma> read_line(caddr(ptylist));
"Test output."
Gamma> read_line(caddr(ptylist));
"Argument"
Gamma>
```

read

`read` — reads a Lisp expression from a file.

Syntax

`read (`*`file`*`)`

Arguments

`file`

A file pointer to a previously opened file. This may be either a file in the file system, or a string opened for read and write.

Returns

The next Lisp expression in the file, or `_eof_`.

Description

This function reads one Lisp expression from the given *`file`*. The file must have been opened before this call with the `open` or `open_string` functions. White space and newline characters are ignored during a read. If the end of file is reached during the `read` call, the message "Unexpected end of file" is returned. `read` does not evaluate the expressions it reads.

Example

The file "myreadfile.dat" contains the following:

(a b (c d e))
"A message" (+ 2 3)

Successive calls to `read` will produce:

```
Gamma> fp = open ("myreadfile.dat", "r");
#<File:"myreadfile.dat">
Gamma> read (fp);
(a b (c d e))
Gamma> read (fp);
"A message"
Gamma> read (fp);
(+ 2 3)
Gamma> read (fp);
"Unexpected end of file"
Gamma>
```

See Also

`read_char`, `read_double`, `read_float`, `read_line`, `read_long`, `read_short`, `read_until`

read_char, read_double, read_float, read_long, read_short

read_char, read_double, read_float, read_long, read_short — read the next character, double, float, long or short value in binary representation from the input file.

Syntax

```
read_char (file)
read_double (file)
read_float (file)
read_long (file)
read_short (file)
```

Arguments

file

A file pointer to a previously opened file. This may be either a file in the file system, or a string opened for read and write.

Returns

read_char returns the decimal representation of a string of length 1 containing a single character, or -1 indicating end of file.

read_double, read_float return a floating point value, or nan indicating end of file.

read_long, read_short return an integer value, or -1 indicating end of file.

Description

These functions read the next character, double, float, long or short value in binary representation from the input file, regardless of Lisp expression syntax. This allows a programmer to read binary files constructed by other programs.

Example

The file "myfile.dat" contains the following:

ajz

Successive calls to read_char will produce:

```
Gamma> ft = open ("myreadcfile.dat","r");
#<File:"myreadcfile.dat">
Gamma> read_char(ft);
97
Gamma> read_char(ft);
106
Gamma> read_char(ft);
122
Gamma> read_char(ft);
-1 Gamma>
```

See Also

read, read_line, read_until

read_eval_file

`read_eval_file` — reads a file, evaluating and counting expressions.

Syntax

`read_eval_file (`*`file`*`)`

Arguments

`file`

A file pointer to a previously opened file. This may be either a file in the file system, or a string opened for read and write.

Returns

The number of expressions that were read and evaluated.

Description

This function reads from the current location in the file to the end, evaluating its contents as Lisp expressions and counting them.

Example

The file "myevalfile.dat" contains the following:

(+ 3 4) 3 + 2;

```
Gamma> ft = open ("myevalfile.dat", "r");
#<File:"myevalfile.dat">
Gamma> read_eval_file(ft);
4
Gamma> close(ft);
t
Gamma>
```

See Also

`require, load,` open

read_line

`read_line` — reads a single line of text.

Syntax

```
read_line (file)
```

Arguments

`file`

A file pointer to a previously opened file. This may be either a file in the file system, or a string opened for read and write.

Returns

All characters in the file up to the first newline character, as a string. If the end of file is reached, returns "Unexpected end of file".

Description

This function reads a single line of text from the given file, up to the first newline character, regardless of Lisp syntax. This allows a programmer to deal with text files constructed by other programs.

Example

An input file contains the following:

```
Lists can be
expressed as (a b c).
```

Successive calls to `read_line` will produce:

```
Gamma> ft = open ("myreadlfile.dat", "r");
#<File:"myreadlfile.dat">
Gamma> read_line(ft);
"Lists can be"
Gamma> read_line(ft);
"expressed as (a b c)."
Gamma> read_line(ft);
"Unexpected end of file"
Gamma>
```

See Also

`read`, `read_char`, `read_double`, `read_float`, `read_long`, `read_short`, `read_until`

read_n_chars

read_n_chars — reads and stores characters.

Syntax

read_n_chars (*file*, *nchars*)

Arguments

file

A file pointer to a previously opened file. This may be either a file in the file system, or a string opened for read and write.

nchars

The number of characters to read.

Returns

A buffer containing the characters read. The length of the return buffer is equal to the number of characters actually read. This function returns nil if no characters could be read.

Description

This function reads the given number of characters from the file, without any form of translation, and builds a new buffer object in which to store them. If this function reaches the end of the file before all characters are read, then the buffer will be shorter than the requested number of characters.

Example

An input file contains the following:

```
To be or not to be, that is the question.
```

Successive calls to read_n_chars will produce:

```
Gamma> ft = open ("myreadnfile.dat", "r");
#<File:"myreadnfile.dat">
Gamma> read_n_chars(ft,15);
#{To be or not to}
Gamma> read_n_chars(ft,18);
#{ be, that is the q}
Gamma> read_n_chars(ft,18);
#{uestion.}
Gamma> read_n_chars(ft,18);
nil
Gamma>
```

See Also

read_char

read_until

`read_until` — reads characters, constructing a string as it goes.

Syntax

`read_until (`*`file`*`, `*`delimiters`*`)`

Arguments

`file`

A file pointer to a previously opened file. This may be either a file in the file system, or a string opened for read and write.

`delimiters`

A string of delimiter characters. "" indicates white space.

Returns

All characters in the file up to the first occurrence of any of *`delimiter`* characters, or "Unexpected end of file".

Description

This function reads characters from the input file one at a time until it reaches any of the *`delimiter`* characters, constructing a string as it goes. Successive calls continue from the point of the previous `read_until`. If the end of file is reached, the function returns "Unexpected end of file".

Example

An input file contains the following:

```
Lists can be
expressed as (a b c).
```

Successive calls to `read_until` will produce:

```
Gamma> ft = open ("myreadlfile.dat","r");
#<File:"myreadlfile.dat">
Gamma> read_until(ft, "(");
"Lists can be\nexpressed as "
Gamma> read_until(ft,"x");
"a b c)."
Gamma> read_until(ft,"y");
"Unexpected end of file"
Gamma>
```

See Also

`read`, `read_char`, `read_double`, `read_float`, `read_line`, `read_long`, `read_short`

seek

`seek` — sets the file position for reading or writing.

Syntax

`seek (`*`file`*`, `*`offset`*`, `*`where`*`)`

Arguments

`file`

A file pointer to a previously opened file. This may be either a file in the file system, or a string opened for read and write.

`offset`

An integer specifying the number of characters into the file, starting from *`where`*.

`where`

A starting point, indicated by a number:

- **0** Beginning of the file.
- **1** Current position in the file.
- **2** End of the file.

Returns

`t` if successful, `nil` if unsuccessful.

Description

This function lets you specify a position in a file to start reading or writing.

Example

The file "myseekfile" contains the following:

```
Now is the time for all good men and women
to come to the aid of their world.
```

```
Gamma> msk = open("myseekfile.dat", "r",nil);
#<File:"myseekfile.dat">
Gamma> seek(msk, 5, 0);
t
Gamma> read_line(msk);
"s the time for all good men and women"
Gamma> seek(msk, 2, 1);
t
Gamma> read_line(msk);
" come to the aid of their world."
Gamma> seek(msk, -15, 2);
t
Gamma> read_line(msk);
"of their world."
Gamma> seek(msk, -3, 0);
nil
Gamma>
```

See Also

open, open_string, read, read_char, read_double, read_float, read_line, read_long, read_short, read_until, tell

ser_setup

`ser_setup` — sets parameters for a serial port device.

Syntax

`ser_setup (devno, baud, bits/char, parity, stopbits, min, time)`

Arguments

devno

A file ID as returned from a call to `fd_open`.

baud

A legal baud rate.

bits/char

Bits per character (6, 7 or 8).

parity

"none", "even", "odd", "mark" or "space"

stopbits

Stop bits (0, 1 or 2).

min

Default minimum number of characters for a read.

time

Default inter-character timeout for a read.

Returns

`t` on success or `nil` on failure.

Description

This function sets the most common parameters for a serial port device, as opened by a call to `fd_open`. The function is currently only available in QNX 4.

Example

```
Gamma> id = fd_open("/dev/ser1",O_RDWR);
4
Gamma> ser_setup(id,9600,8,"none",1,1,0);
t
```

See Also

`fd_close`, `fd_open`

tell

`tell` — indicates file position.

Syntax

`tell (`*`file`*`)`

Arguments

`file`

A file pointer to a previously opened file. This may be either a file in the file system, or a string opened for read and write.

Returns

Current file position, as an integer.

Description

This function returns the current file position as an integer representing the number of characters from the beginning of the file.

Example

```
Gamma> msk = open("myseekfile.dat", "r",nil);
#<File:"myseekfile.dat">
Gamma> read_until(msk,"f");
"Now is the time "
Gamma> tell(msk);
17
Gamma> seek(msk, 18, 1);
t
Gamma> tell(msk);
35
Gamma> msk = open("myseekfile3.dat", "w",nil);
#<File:"myseekfile3.dat">
Gamma> write(msk,"hello");
t
Gamma> tell(msk);
7
Gamma> msk = open("myseekfile3.dat", "a",nil);
#<File:"myseekfile3.dat">
Gamma> write(msk,"goodbye");
t
Gamma> tell(msk);
16
Gamma>
```

See Also

open, open_string, read, read_char, read_double, read_float, read_line, read_long, read_short, read_until, seek

terpri

`terpri` — prints a newline to an open file.

Syntax

`terpri (`*`file`*`?)`

Arguments

`file`

A file pointer to a previously opened file. This may be either a file in the file system, or a string opened for read and write.

Returns

`t` if successful, otherwise `nil`.

Description

This function writes a newline to the open *`file`*. Any existing contents written to a file before it was opened will be deleted when the file is opened and written to.

Example

This example writes a file with the following contents, including the newline:

```
(chars (1 2 3))
(chars (4 5 6))
```

```
Gamma> fw = open("mytpfile.dat", "w");
#<File:"mytpfile.dat">
Gamma> write(fw,list(#chars,list(1,2,3)));
t
Gamma> terpri(fw);
t
Gamma> write(fw,list(#chars,list(4,5,6)));
t
Gamma> close(fw);
t
Gamma> fr = open("mytpfile.dat", "r", nil);
#<File:"mytpfile.dat">
Gamma> read_line(fr);
"(chars (1 2 3))"
Gamma> read_line(fr);
"(chars (4 5 6))"
Gamma> terpri();

t
Gamma>
```

unread_char

unread_char — attempts to replace a character to a file for subsequent reading.

Syntax

```
unread_char (file, character)
```

Arguments

file

A file pointer to a previously opened file. This may be either a file in the file system, or a string opened for read and write.

character

A single character.

Returns

t if the *character* could be replaced on the *file*, otherwise nil.

Description

This function attempts to place the given *character* back onto the *file* so that it can be read again by subsequent calls to any of the read family of functions. Only one character may be replaced onto a file between calls to read. At least one read call must have been made prior to calling this function.

Example

An input file contains the following:

```
ABCDE
```

A call to unread_char within a succession of calls to read_char will produce:

```
Gamma> fr = open("myunreadfile.dat", "r", t);
#<File:"myunreadfile.dat">
Gamma> read_char(fr);
65
Gamma> read_char(fr);
66
Gamma> unread_char(fr,'A');
t
Gamma> read_char(fr);
65
Gamma> read_char(fr);
67
Gamma> read_char(fr);
68
Gamma>
```

See Also

read

write, writec, pretty_write, pretty_writec

write, writec, pretty_write, pretty_writec — write an expression to a file.

Syntax

```
write (file, s_exp...)
writec (file, s_exp...)
pretty_write (file, s_exp...)
pretty_writec (file, s_exp...)
```

Arguments

file

A file pointer to a previously opened file. This may be either a file in the file system, or a string opened for read and write.

s_exp

Any Gamma or Lisp expression.

Returns

t on success, otherwise nil.

Description

Writes the given expressions to the file using the same format as print. See print for more information.

writec produces the same format as princ; pretty_write produces the same format as pretty_print; and pretty_writec produces the same format as pretty_printc. Any contents written to the file before it was opened will be deleted when any of these write functions are used.

Example

```
Gamma> fw = open("mywritefile.dat", "w");
#<File:"mywritefile.dat">
Gamma> write(fw,"This is on \n one line.");
t
Gamma> writec(fw,"This finishes on \n another line.");
t
Gamma> close(fw);
t
Gamma> fr = open("mywritefile.dat", "r", nil);
#<File:"mywritefile.dat">
Gamma> read_line(fr);
"\"This is on \n one line.\"This finishes on "
Gamma> read_line(fr);
" another line."
Gamma>
```

See Also

print

write_n_chars

`write_n_chars` — writes characters from a buffer to a file.

Syntax

`write_n_chars (`*`file`*`, `*`buffer`*`, `*`nchars`*`)`

Arguments

`file`

A file pointer to the open destination file for the characters.

`buffer`

The buffer that is the source of the characters.

`nchars`

The number of characters to write.

Returns

The number of characters successfully written to the file.

Description

This function reads a given number of characters from a buffer and writes them to an open file.

Example

The following example writes the characters ’e’, ’f’, and ’g’ to a file.

```
Gamma> fw = open("mywritencharsfile.dat", "w");
#<File:"mywritencharsfile.dat">
Gamma> buf = buffer (101, 102, 103, 104);
#{efgh}
Gamma> write_n_chars (fw, buf, 3);
3
Gamma>
```

See Also

`write, writec, pretty_write`

II. File System

Table of Contents

absolute_path

`absolute_path` — returns the absolute path of the given file.

Syntax

```
absolute_path (filename)
```

Arguments

`filename`

The name of a disk file.

Returns

The absolute path of the file.

Description

This function returns the absolute path of the given file, with extraneous ../ constructs removed, and with the full QNX 4 node number added. The filename can be relative or absolute, on any node on the network.

Example

```
Gamma> absolute_path(".profile");
"//1/home/andrewt/.profile"
```

access

`access` — checks a file for various permissions.

Syntax

```
access (filename, mode)
```

Arguments

`filename`

The name of a file on disk.

`mode`

The file mode to be tested. The legal modes are discussed below.

Returns

Zero is returned if the access *`mode`* is valid, otherwise -1 is returned and the errno is set.

Description

This function checks a file for the following permissions. Two or more permissions in bitwise OR combinations can be checked at one time.

- **R_OK** Test for read permission.
- **W_OK** Test for write permission.
- **X_OK** Test for execute permission.
- **F_OK** Test for existence of file.

The library "const/Filesys.lsp" must be required to use the constants listed above.

Example

```
Gamma> require_lisp("const/Filesys");
"/usr/cogent/lib/const/Filesys.lsp"
Gamma> system("touch /tmp/access_test");
0
Gamma> system("chmod a=rx /tmp/access_test");
0
Gamma> access("/tmp/access_test", R_OK|X_OK);
0
Gamma> access("/tmp/access_test", W_OK);
-1
Gamma>
```

See Also

`is_busy`, `is_file`, `is_readable`, `is_writable`, `errno`

basename

basename — gives the base of a filename.

Syntax

basename (*filename*, *suffix*?)

Arguments

filename

A file name as a string, as defined by the operating system.

suffix

Any ending part of the filename to exclude.

Returns

The base of the filename. If a suffix is specified, the base of the filename without the suffix.

Example

```
Gamma> x = basename("/usr/george/lib/misc/myfile.dat");
"myfile.dat"
Gamma> y = basename("misc/myfile.dat", ".dat");
"myfile"
Gamma>
```

See Also

dirname root_path

cd

`cd` — changes the working directory.

Syntax

`cd (`*`path`*`)`

Arguments

`path`

A character string which defines a directory path in the current operating system.

Returns

`t` if the operation is successful, otherwise `nil`.

Description

This function changes the current working directory for subsequent file system operations.

Example

```
Gamma> cd ("/usr/local/bin");
t
```

chars_waiting

`chars_waiting` — checks for characters waiting to be read on a file.

Syntax

```
chars_waiting (file)
```

Arguments

`file`

A file pointer to a previously opened file. This may be either a file in the file system, or a string opened for read and write.

Returns

The number of characters waiting on the open file.

Description

This function determines whether there are any characters waiting to be read on the given file. The file may be a string file (created by `open_string`), in which case the number of characters which have not yet been treated by a `read` or similar call will be returned.

If `chars_waiting` is to be called on a file after it has been partially read, the file must be unbuffered first with `unbuffer_file`. Otherwise characters will be read in buffer by buffer and held locally in groups of 1024. This will cause `chars_waiting` to return unexpected results.

Example

```
Gamma> ft = open("mytestfile.dat", "r");
#<File:"mytestfile.dat">
Gamma> unbuffer_file(ft);
#<File:"mytestfile.dat">
Gamma> chars_waiting(ft);
9
Gamma> char(read_char(ft));
"A"
Gamma> chars_waiting(ft);
8
Gamma> read_line(ft);
"BCDEFGHI"
Gamma>

Gamma> x = open_string("hello");
#<File:"String">
Gamma> chars_waiting(x);
5
Gamma> char(read_char(x));
"h"
Gamma> chars_waiting(x);
4
Gamma>
```

See Also

`open`, `open_string`, `unbuffer_file`

directory

`directory` — returns the contents of a directory.

Syntax

`directory (`*`path`*`, `*`filetypes`*`, `*`fullpaths`*`)`

Arguments

`path`

A path to a directory as defined by the operating system.

`filetypes`

A number in the range 0 to 2:

- **0** Find all files and directories.
- **1** Find all files.
- **2** Find all directories.

`fullpaths`

If non-`nil`, show the full pathname of the file by prepending the *`path`* to all filenames.

Returns

A list containing all of the requested directory entries as strings.

Example

```
Gamma> directory("/usr",0,nil);
("local" "lib" "bin" "readme")
Gamma> sort(directory("/usr",2,t),strcmp);
("/usr/bin" "/usr/lib" "/usr/local")
```

dirname

`dirname` — returns the directory path of a file.

Syntax

`dirname (`*`filename`*`)`

Arguments

`filename`

A file name as a string, including its directory path, as defined by the operating system.

Returns

The directory path of the *`filename`*, or if no path is entered, the *`filename`*.

Description

This function reads the *`filename`* and directory path as a string, returning the directory path as a string.

Example

```
Gamma> x = dirname("/usr/george/lib/misc/myfile.dat");
"/usr/george/lib/misc"
Gamma> y = dirname("misc/myfile.dat");
"misc"
Gamma> z = dirname("myfile.dat");
"myfile.dat"
Gamma>
```

See Also

`basename`, `root_path`

drain

`drain` — modifies end-of-file detection.

Syntax

`drain (`*`file`*`, `*`drain_p`*`)`

Arguments

`file`

An open file.

`drain_p`

A flag. If non-`nil`, sets the file to drain.

Returns

The previous state of the drain flag for this file.

Description

This function sets a flag on the file state such that if the *`drain_p`* flag is on, the first time that a read on that file finds no characters waiting, the read will return immediately with "Unexpected end of file". This is intended for use in situations where the operating system may never actually generate an end of file indication, but where it is known that no more input will be available once a read would block. This function does not affect `dev_read`.

For best results, the file should be unbuffered first with `unbuffer_file`. Otherwise characters will be read in buffer by buffer and held locally in groups of 1024. This could cause a `read` function to return "Unexpected end of file" even when there are still characters waiting to be read.

Example

```
Gamma> fp = open("mydrainfile.dat","r",nil);
#<File:"mydrainfile.dat">
Gamma> unbuffer_file(fp);
#<File:"mydrainfile.dat">
Gamma> drain(fp,t);
nil
Gamma> read_line(fp);
"This is my drain file."
Gamma> read_line(fp);
"Unexpected end of file"
Gamma>
```

file_date

`file_date` — gives the file modification date.

Syntax

```
file_date (filename)
```

Arguments

`filename`

A filename as defined by the operating system.

Returns

The modification date of the file as an integer if the file exists and is readable, else `nil`.

Example

```
Gamma> fd =(file_date("myfile.dat"));
936977583
Gamma> date_of(fd);
"Fri Sep 10 11:33:03 1999"
Gamma> file_date("nonexistent.file");
nil
Gamma> file_date("unreadable.file");
nil
Gamma>
```

See Also

`clock`, `date_of`

file_size

`file_size` — gives the file size.

Syntax

```
file_size (filename)
```

Arguments

`filename`

A file name as a string, as defined by the operating system.

Returns

The size of the file in bytes if the file exists and is readable, else `nil`.

Example

```
Gamma> file_size("myfile.dat");
1467
Gamma> file_size("non_existing.file");
nil
Gamma> file_size("unreadable.file");
nil
Gamma>
```

flush

`flush` — flushes any pending output on a file or string.

Syntax

`flush (`*`file`*`)`

Arguments

`file`

A file pointer to a previously opened file. This may be either a file in the file system, or a string opened for read and write.

Returns

`t`

Description

This function flushes any pending output on the file or string. This has the effect of printing output on the screen or updating a file on disk in the case of a file. `flush` has no effect on strings. `flush` is called automatically by `close`.

Example

```
Gamma> fp=open("myflushfile.dat","w",nil);
#<File:"myflushfile.dat">
Gamma> write(fp, "I am written.");
t
Gamma> fp=open("myflushfile.dat","r",nil);
#<File:"myflushfile.dat">
Gamma> read_line(fp);
"Unexpected end of file"

Gamma> fp=open("myflushfile.dat","w",nil);
#<File:"myflushfile.dat">
Gamma> write(fp, "I am written.");
t
Gamma> flush(fp);
t
Gamma> fp=open("myflushfile.dat","r",nil);
#<File:"myflushfile.dat">
Gamma> read_line(fp);
"\"I am written.\""
Gamma>
```

See Also

open, `open_string`

getcwd

`getcwd` — gets the current working directory.

Syntax

```
getcwd ()
```

Arguments

none

Returns

The current working directory as a string.

Example

```
Gamma> getcwd();
"/home/robert/w/devel/lisp"
Gamma>
```

is_busy

`is_busy` — determines if a file is busy.

Syntax

`is_busy (`*`path`*`)`

Arguments

`path`

A character string defining a file path and file name in this file system.

Returns

`t` if the named file exists and is busy, otherwise `nil`.

Description

This function is supported only by certain operating system and hardware combinations that mark files as busy when they are opened for write by another task. You can check this using the **ls -l** shell command. If it shows a busy file with a ’B’ or ’b’ as the first bit in the bitmask, this function should be supported.

Example

```
Gamma> is_busy("/tmp/busyfile");
t
```

See Also

`is_writable`

is_dir

`is_dir` — determines if a file is a directory.

Syntax

```
is_dir (path)
```

Arguments

`path`

A character string defining a relative or absolute file path in this file system.

Returns

`t` if the named file exists and is a directory, otherwise `nil`.

Description

This function checks if a file is a directory. Relative file paths are relative to the current working directory.

Example

```
Gamma> is_dir("/home/robert/w/devel/lisp");
t
Gamma> is_dir("../../doc");
t
Gamma> is_dir("doc");
nil
Gamma>
```

See Also

`is_file`

is_file

`is_file` — determines if a file exists.

Syntax

```
is_file (path)
```

Arguments

path

A character string defining a file path and file name in this file system.

Returns

`t` if the named file exists and is a regular file, otherwise `nil`.

Example

```
Gamma> is_file("/usr/doc/FAQ/txt/FAQ");
t
Gamma>
```

See Also

is_dir

is_readable

`is_readable` — determines if a file is readable.

Syntax

```
is_readable (path)
```

Arguments

path

A character string defining a file path and file name in this file system.

Returns

`t` if the named file exists and is readable, otherwise `nil`. Existing files might not be readable because of settings on the files bitmask.

Example

```
Gamma> is_readable("/usr/doc/FAQ/txt/FAQ");
t
Gamma>
```

See Also

`is_writable`, `is_busy`

is_writable

`is_writable` — determines if a file is writable.

Syntax

```
is_writable (path)
```

Arguments

path

A character string defining a file path and file name in this file system.

Returns

`t` if the named file exists and is writable, otherwise `nil`.

Example

```
Gamma> is_writable("/usr/doc/FAQ/txt/FAQ");
nil
Gamma> is_writable("/home/robert/w/devel/lisp/mytestfile.dat");
t
Gamma>
```

See Also

`is_readable`

mkdir

`mkdir` — creates a new sub-directory.

Syntax

```
mkdir (dirname, mode)
```

Arguments

`dirname`

The name of the directory to create.

`mode`

The access permissions of the new directory, joined in sequence. If there are more than one, they are OR'ed by the | character in text format, or written consecutively in octal format. (See below.)

Returns

Zero if successful, otherwise non-zero, and the errno will be set.

Description

This function creates a new sub-directory whose path-name is *`dirname`*. The file permissions for the new sub-directory are determined from the *`mode`* argument. Valid modes are summarized here.

Table 1. User/owner permission modes

Text format	Octal format	Meaning
S_IRWXU	0o7	Read, write, execute/search
S_IRUSR	0o4	Read permission
S_IWUSR	0o2	Write permission
S_IXUSR	0o1	Execute/search permission

Table 2. Group permission modes

Text format	Octal format	Meaning
S_IRWXG	0o7	Read, write, execute/search
S_IRGRP	0o4	Read permission
S_IWGRP	0o2	Write permission
S_IXGRP	0o1	Execute/search permission

Table 3. Other permission modes

Text format	Octal format	Meaning
S_IRWXO	0o7	Read, write, execute/search
S_IROTH	0o4	Read permission
S_IWOTH	0o2	Write permission
S_IXOTH	0o1	Execute/search permission

Miscellaneous permissions.

- **S_IREAD** same as S_IRUSR
- **S_IWRITE** same as S_IWUSR
- **S_IEXEC** same as S_IXUSR

These flags are bitwise OR-ed together to get the desired mode.

Error constants for this function:

- **EACCES** Search permission for some component of the path denied.
- **EEXIST** The named file exists.
- **EMLINK** Maximum sub-dirs. reached.
- **ENAMETOOLONG** The name of the path or the new directory is too long.
- **ENOENT** The specified path does not exist.
- **ENOSPC** No space left on the file system.
- **ENOSYS** This function is not supported for this path.
- **ENOTDIR** A component of the passed path is not a directory.
- **EROFS** Tried to create a directory on a read-only file system.

Example

```
Gamma> require_lisp("const/Filesys");
"/usr/cogent/lib/const/Filesys.lsp"
Gamma> mkdir("/tmp/mydir", S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH);
0
Gamma> mkdir("/tmp/mydir2", 0o755);
0
Gamma>
```

See Also

unlink

path_node

`path_node` — gives the node number of a path in a QNX 2 path definition.

Syntax

`path_node (`*`path`*`)`

Arguments

`path`

Any legal QNX 2 path.

Returns

The node number portion of the *`path`* in a QNX 2 path definition. If the path does not contain an explicit node portion, the function returns `nil`.

Description

This function is for version 2 of the QNX operating system only.

Example

```
Gamma> path_node("[2]3:/user");
2
Gamma> path_node("3:/user");
0
```

rename

`rename` — renames a file.

Syntax

```
rename (filename, new_name)
```

Arguments

`filename`

The name of a file on disk.

`new_name`

The new name for the file.

Returns

`t` if the file could be renamed, otherwise `nil`.

Description

This function makes an operating system call rename a file on disk. The exact behavior of this function depends on the renaming facility for the operating system.

Example

```
Gamma> rename("myfile.dat","x/myrenamedfile.dat");
t
Gamma> rename("x/myrenamedfile.dat","myfile.dat");
t
Gamma>
```

root_path

`root_path` — strips the final file or directory name from a path.

Syntax

`root_path (`*`path`*`)`

Arguments

`path`

A file path name, as a string.

Returns

The portion of the *`path`* with the file name and any trailing directory separators removed.

Description

This function strips the final file or directory name from the *`path`* to produce its parent. Any trailing directory separators are also removed. If the *`path`* represents the root of the file system then it is unmodified.

Example

```
Gamma> x = "/usr/doc/FAQ";
"/usr/doc/FAQ"
Gamma> x = root_path(x);
"/usr/doc"
Gamma> x = root_path(x);
"/usr"
Gamma> x = root_path(x);
"/"
Gamma> x = root_path(x);
"/"
Gamma>
```

See Also

basename, dirname

tmpfile

`tmpfile` — generates temporary output file names.

Syntax

`tmpfile (`*`file_prefix`*`?)`

Arguments

`file_prefix`

A optional string specifying the beginning of a file name.

Returns

A string representing a file name which is guaranteed not to exist at the time that the function was called.

Description

This function is used to generate a temporary output file name. The *`file_prefix`* can specify any part of a file path. If the *`file_prefix`* is `nil`, then "`/tmp/lisp_t`" will be used. Typically the resulting file name will be the result of appending a number to the file prefix.

Example

```
Gamma> tmpfile();
"/tmp/lisp_t1"
Gamma> tmpfile("/tmp/atempfile");
"/tmp/atempfile2"
Gamma> tmpfile("/tmp/atempfile");
"/tmp/atempfile3"
Gamma> tmpfile("/tmp/atempfile");
"/tmp/atempfile4"
Gamma> tmpfile("anotherfile");
"anotherfile5"
Gamma>
```

unbuffer_file

`unbuffer_file` — causes a file to be treated as unbuffered on both input and output.

Syntax

```
unbuffer_file (file)
```

Arguments

`file`

A file pointer to a previously opened file. This may only be a file in the file system, and not a string opened for read and write.

Returns

The unbuffered file object on success, or `nil` on failure.

Description

This function causes a file to be treated as unbuffered on both input and output. The normal buffering mode of a file depends on whether it is in the file system, or to a character device such as a terminal or console.

When the file is unbuffered, all input and output to that file will occur immediately, without going through internal buffers. In general, an unbuffered file is much less efficient for file I/O. Unbuffering is temporary, as the file will revert to a buffered state when it is closed or reopened.

Example

```
Gamma> fu = open("mytestfile.dat","r",nil);
#<File:"mytestfile.dat">
Gamma> unbuffer_file(fu);
#<File:"mytestfile.dat">
Gamma>
```

unlink

`unlink` — deletes a file.

Syntax

```
unlink (filename)
```

Arguments

`filename`

A string representing a valid file name.

Returns

`t` if the file could be deleted, otherwise `nil`.

Description

This functioin deletes a file in the file system. It will fail if the given file does not exist, or the calling process does not have sufficient privilege to delete the file. Wild cards are not expanded in the file name.

Example

```
Gamma> fu = open("todeletefile.dat","r");
#<File:"todeletefile.dat">
Gamma> unlink("todeletefile.dat");
t
Gamma> unlink("todeletefile.dat");
nil
Gamma>
```

See Also

`open`

III. OS APIs

Table of Contents

atexit

`atexit` — evaluates code before exiting a program.

Syntax

`atexit (`*`code`*`)`

Arguments

`code`

Code to be evaluated.

Returns

The result of evaluating *`code`*.

Description

This function gives a program an opportunity to evaluate specified code before it exits. The *`code`* should be protected from evaluation using the quote operator #.

Example

Running this program...

```
#!/usr/cogent/bin/gamma

// Program name: exiting.g
// Demonstrates the atexit() function.

atexit(#princ("Exiting now.\n"));
princ("Started running...\n");
princ("Still running.\n");
exit_program(7);
princ("You missed this part.\n");
```

...gives these results:

```
[sh]$ exiting.g
Started running...
Still running.
Exiting now.
[sh]$
```

block_signal, unblock_signal

block_signal, unblock_signal — delimit signal blocking.

Syntax

```
block_signal (signo)
unblock_signal (signo)
```

Arguments

signo

The integer signal number as defined by the operating system. Symbols such as SIGINT are defined to provide an operating-system independent method for specifying this number. (see signal)

Returns

t

Description

block_signal causes a particular signal to be blocked until a call to unblock_signal is made. If the signal actually occurred while it was blocked, it will occur immediately when unblock_signal is called. Multiple occurrences of the signal while it was blocked will cause the signal to be reported multiple times when unblock_signal is called on most operating systems. Code that blocks signals should be surrounded by a call to unwind_protect.

Example

```
Gamma> block_signal(14);
t
Gamma> kill(getpid(),14);
t
Gamma> unblock_signal(14);
Alarm clock

Gamma> block_signal (SIGINT);
t
Gamma> critical_function();
<function return>
Gamma> unblock_signal (SIGINT);
t
```

See Also

block_timers, unblock_timers

errno

`errno` — detects and numbers errors.

Syntax

```
errno ();
```

Arguments

none

Returns

The system error number.

Description

When a function fails it returns a value and optionally sets the system error number. The `errno` function can be used to check the current error number. To check error numbers against constant error code in your program remember to include the file with: `require_lisp` ("Errno.lsp");.

Calling the `errno` function in interactive mode does not return a valid number since you are retrieving the errno of the C function `printf` of the error to the screen (which will usually be 0).

Example

In this example, we first define a function to remove a file. Then we call that function on a non-existing file to generate an error. Finally, we check the returned error code to get the error message.

```
function remove_file(file)
{
   unlink(file);
   errno();
}

Gamma> ret_val = remove_file("/tmp/xyz");
2
Gamma> strerror(2);
"No such file or directory"
Gamma>
```

See Also

`error`, `strerror`

exec

`exec` — executes a program.

Syntax

`exec (`*`program, arguments`*`?...)`

Arguments

`program`

The name of a program to execute, as a string.

`arguments`

The arguments to the program, each as a string.

Returns

Does not return if successful, or -1 if an error occurs.

Description

This function is a binding for the C function `execvp`, which causes the interpreter to terminate immediately, and to run the named program in its place. The "p" in the `execvp` function indicates that a search is made for the named executable in the current path, as defined by the PATH shell variable. Unlike the C `execvp` function, the first argument in Gamma's `exec` function does not repeat the program name—it is automatically inserted for you.

Example

```
Gamma> exec("/bin/ls","-l","/usr/bin");
-rwxr-xr-x    1 root     root        98844 Aug  7  2000 a2p*
-rwxr-xr-x    1 root     root         4080 Jul 19  2000 access*
-rwxr-xr-x    1 root     root        10256 Jul 12  2000 aclocal*
...
```

See Also

`fork`, `wait`

exit_program

`exit_program` — terminates the interpreter.

Syntax

`exit_program (`*`return_value`*`)`

Arguments

`return_value`

An integer value to be returned to the operating system when the program exits.

Returns

This function does not return.

Description

Terminate the interpreter immediately and return the provided integer value to the operating system as an exit code.

Example

Running this program...

```
#!/usr/cogent/bin/gamma

//Program: exiting.g

atexit(#princ("Exiting now.\n"));
princ("Started running...\n");
princ("Still running.\n");
exit_program(7);
princ("You missed this part.\n");
```

...gives these results:

```
[]$ gamma exiting.g
Started running...
Still running.
Exiting now.
[]$
Exit showing abnormal termination of -1 (255) to the operating system.

exit_program(-1);

/>echo $?
255
/>
```

fork

`fork` — duplicates a process.

Syntax

```
fork ()
```

Arguments

none

Returns

A positive task id that identifies the child process, and 0 that identifies the parent process to the child; or -1 if an error occurred. The errno is set if an error occurs.

Description

The `fork` function creates a new process identical to the calling (parent) process except for a unique process ID. The child process has a different parent process ID and its own copy of the parent file descriptors. The child process does not inherit outstanding signals.

Example

The following example illustrates using the `fork` function with `if` syntax. This is a useful way of separating the two, identical processes produced from `fork`. The first block of code applies to the parent process, while the second block applies to the child.

```
#!/usr/cogent/bin/gamma

if ((childID = fork()) > 0)
{
  princ("P> My ID is: ", getpid(),"\n");
  princ("P> My child's ID is: ", childID, "\n");
  signal(SIGCHLD, #princ("P> Signal received that my child -- ",
                         childID, " -- has died.\n"));
  princ("P> Waiting for my child.\n");
  w = wait(childID);
  princ("P> wait() returned this: ", w, "\n");
}
else
{
  sleep(2);
  if (childID == -1)
    error("C> An error occurred.\n");
  else
    {
      princ("C> I am the child process.\nC> My process ID is: ",
            getpid(), "\n");
      sleep(2);
      princ("C> Time to exit.\n");
      exit_program(3);
    }
}
```

Will produce these results:

```
P> My ID is: 1225
P> My child's ID is: 1226
P> Waiting for my child.
```

(after 2 seconds)

```
C> I am the child process.
C> My process ID is: 1226
```

(after 2 more seconds)

```
C> Time to exit.
P> Signal received that my child -- 1226 -- has died.
P> wait() returned this: (1226 3 nil nil)
```

See Also

exec, wait

getenv

`getenv` — retrieves the value of an environment variable.

Syntax

`getenv (`*`envar`*`)`

Arguments

`envar`

A string.

Returns

A string containing the value of the given environment variable, or `nil` if the environment variable is not defined.

Description

This function retrieves the value of an environment variable from the current process's environment. The environment variable must have been set or defined previously by a call to `setenv`.

Example

```
Gamma> setenv("high", "40");
t
Gamma> getenv("high");
"40"
Gamma> low = 20;
20
Gamma> getenv("low");
nil
Gamma>
```

See Also

`setenv`

gethostname

`gethostname` — gets the computer's host name.

Syntax

```
gethostname ()
```

Arguments

none

Returns

The host name of this computer, as a string.

Example

```
Gamma> gethostname();
"rex"
Gamma>
```

getnid

`getnid` — returns the local node number.

Syntax

```
getnid ()
```

Arguments

none

Returns

The node number

Example

```
Gamma> getnid();
2
```

See Also

`getpid`

getpid

`getpid` — returns the program ID.

Syntax

```
getpid ()
```

Arguments

none

Returns

The program ID of the current session of the interpreter.

Example

```
Gamma> getpid();
8081
Gamma>
```

See Also

`getnid`

`getsockopt`, `setsockopt`

`getsockopt, setsockopt` — get and set a socket option.

Syntax

```
getsockopt (socket, option)
setsockopt (socket, option, value1, value2? = nil)
```

Arguments

`socket`

The file descriptor of a socket.

`option`

The option being queried. Supported options and their possible values are listed below.

`value`

The value to set the socket option to. There may be one or two values, depending on the option. If a socket option requires two values, both must be specified.

Returns

`getsockopt` returns the socket option value(s) on success, as shown below, or `nil` on failure. When the option has two values, they are returned as a list.

`setsockopt` returns `0` on success, otherwise `-1`.

Description

These functions get and set a socket option, using the socket's file descriptor. The supported socket options are given below.

`SO_SNDTIMEO`, `SO_RCVTIMEO`, `SO_SNDLOWAT` and `SO_RCVLOWAT` are not supported by all operating systems.

Option	Possible Values	Comments
`SO_BROADCAST`	`0` for off, non-zero for on.	Allows for broadcasting datagrams from the socket.
`SO_DEBUG`	`0` for off, non-zero for on.	Records debugging information.
`SO_DONTROUTE`	`0` for off, non-zero for on.	Sends messages directly to the network interface instead of using normal message routing.
`SO_ERROR`	A number.	Resets the error status (for `getsockopt` only).
`SO_KEEPALIVE`	`0` for off, non-zero for on.	Transmits messages periodically on a connected socket. No response means the connection is broken.
`SO_LINGER`	Two values: `on_or_off` and `linger_time`, where `on_or_off` is `0` for off, non-zero for on. If on, a value for `linger_time` is required.	Keeps the socket open after a `close()` call, to deliver untransmitted messages. If `on_or_off` is non-zero, the socket will block for the duration of the `linger_time` or until all messages have been sent.

Option	Possible Values	Comments
TCP_NODELAY	0 for enable, non-zero for disable.	Disables the Nagle algorithm for sending data.
SO_OOBINLINE	0 for off, non-zero for on.	Puts out-of-band data in the normal input queue.
SO_REUSEADDR	0 for off, non-zero for on.	Permits the reuse of local addresses for this socket.
SO_RCVBUF	A number.	The size of the input buffer.
SO_RCVLOWAT	A number.	Sets the minimum count for input operations.
SO_RCVTIMEO	Two values: seconds and nanoseconds.	Sets a timeout value for input.
SO_SNDBUF	A number.	The size of the output buffer.
SO_SNDLOWAT	A number.	Sets the minimum count for output operations.
SO_SNDTIMEO	Two values: seconds and nanoseconds.	Sets a timeout value for output.
SO_TYPE	A number.	The type of socket (for getsockopt only).

Example

```
Gamma> skt = tcp_connect("localhost", 22);
8
Gamma> getsockopt(skt, SO_KEEPALIVE);
0
Gamma> setsockopt(skt, SO_KEEPALIVE, 1);
0
Gamma> getsockopt(skt, SO_KEEPALIVE);
1
Gamma> getsockopt(skt, SO_DEBUG);
0
Gamma> setsockopt(skt, SO_DEBUG, 1);
-1
Gamma> getsockopt(skt, SO_DEBUG);
0
Gamma>
```

kill

`kill` — sends a signal to a process.

Syntax

`kill (`*`pid`*`, `*`signo`*`)`

Arguments

`pid`

The process id number.

`signo`

The signal number, normally one of the built-in signal values.

Returns

`t`

Description

This process functions similarly to the `kill` shell command. Signals and their descriptions can be found in `signal`.

Example

Process 1:

```
Gamma> getpid();
8299
Gamma>
```

Process 2:

```
Gamma> kill(8299,9);
t
Gamma>
```

Process 1:

```
Gamma> Killed
```

Process 3:

```
Gamma> getpid();
9041
Gamma> kill (9041,14);
Alarm clock
```

See Also

`signal`

nanosleep

`nanosleep` — pauses the interpreter for seconds and nanoseconds.

Syntax

```
nanosleep (seconds,nanosecs)
```

Arguments

seconds

The number of seconds to pause.

nanosecs

The number of nanoseconds to pause.

Returns

`t` after the time has elapsed.

Description

This function will pause the interpreter for the total time of seconds + nanoseconds

Example

```
Gamma> time(1,nanosleep( 0, 999999999 ));
1.0009529590606689453
//this example is done with the ticksize at 0.5 ms.
```

See Also

sleep

setenv

`setenv` — sets an environment variable for the current process.

Syntax

```
setenv (envar, value)
```

Arguments

envar

The name of the environment variable to set.

value

The string value for this environment variable.

Returns

`t` on success, or `nil` on failure.

Description

This function sets an environment variable for the current process. Both arguments are strings. The value of an environment variable can be acquired using the function `getenv`.

Example

```
Gamma> setenv("high", "40");
t
Gamma> getenv("high");
"40"
Gamma> low = 20;
20
Gamma> getenv("low");
nil
Gamma>
```

See Also

getenv

shm_open

`shm_open` — opens shared memory objects.

Syntax

```
shm_open (share_name, open_flags, create_mode, size?)
```

Arguments

`share_name`

The name of the shared memory object.

`open_flags`

Open control flags.

`create_mode`

Creation mode.

`size`

The size of the shared object in bytes.

Returns

A handle to the shared memory object, or `nil` on failure.

Description

This function is a wrapper for the C function `shm_open`. It is currently only available in QNX 4.

The name of the shared memory object is usually a name found under the /dev/shmem directory. Direct shared memory access to devices is acheived through a `shm_open` call to the existing `Physical` shared memory.

If you are accessing the existing `Physical` shared memory region (`/dev/shmem/Physical`) DO NOT use the *`size`* argument, as you may inadvertantly resize this shared memory. The *`size`* argument is added as a convenience, and can be used to specify the size of a newly created object.

Valid open-flags are OR-ed combinations of:

- **O_RDONLY** open for read-only
- **O_RDWR** open for read and write access
- **O_CREAT** create a new shared memory segment with access privledges governed by the *`create_mode`* parameter
- **O_EXCL** Exclusive mode. If O_EXCL and O_CREAT are set then shm_open will fail if the shared memory segment exists.
- **O_TRUNC** If the shared memory object exists, and it is successfully opened O_RDWR, the object is truncated to zero length and the mode and owner are unchanged.

The creation mode is usually an octal number in the range 0o000 - 0o777 defining the access privledges for the shared memory object. Require the 'const/filesys' file to load constants to make this arg easier

Possible errno values are:

- **EACCESS** permission to create the shared memory object denied
- **EEXIST** O_CREAT and O_EXCL are set and the named shared memory object already exists
- **EINTR** The function call was interrupted by a signal
- **EMFILE** Too many file descriptors in use by this process
- **ENAMETOOLONG** The lengthof the name arg is too long
- **ENFILE** To many shared memory objects are currently open in the system
- **ENOENT** O_CREAT is not set and the named shared memory object does not exist, or O_CREAT is set and either the name prefix does not exist or the name arg is an empty string
- **ENOSPC** Not enough space for the creation of the new shared memory object
- **ENOSYS** This function is not supported by this implementation.

Example

```
//This code maps the first 1000 bytes from video
//memory (0xA0000) into a buffer named buf.

require_lisp("const/filesys");
require_lisp("const/mman");
fd = shm_open("Physical",O_RDONLY, 0o777);
buf = mmap(1000, PROT_READ , MAP_SHARED, fd, 0xA0000);
```

See Also

mmap, shm_unlink

shm_unlink

`shm_unlink` — removes shared memory objects.

Syntax

```
shm_unlink (share_name)
```

Arguments

`share_name`

The name of the shared object to delete.

Returns

`t` on success, or `nil` on failure, with `errno` set.

Description

This function is currently only available in QNX 4. It attempts to remove the shared object, *`share_name`*. If more than one process or link into the shared memory area exists the shared object will not be removed.

Possible values of errno are:

- **EACCESS** Permission to unlink the object is denied
- **ENAMETOOLONG** The lenght of the name of the object is too long
- **ENOENT** The named shared memory object does not exist.
- **ENOSYS** This function is not supported by this implementation.

Example

```
Gamma> shm_unlink("card_mem");
t
Gamma>
```

See Also

`shm_open`, `mmap`

signal

`signal` — defines an expression to be evaluated at an OS generated signal.

Syntax

```
signal (signal, action[, action]...)
```

Arguments

`signal`

A signal number. Normally one of the built-in signal values.

`action`

Any Gamma or Lisp expression.

Returns

`t`

Description

This function defines an expression to be evaluated whenever the operating system generates signal number signal to this process. A signal handler may be of any complexity, though it is advisable to keep signal handlers as simple as possible. All signals and timers are blocked for the duration of the signal handler. In addition, the signal handler runs in a separate, smaller heap. If the signal handler is large, this could result in memory inefficiency. Signal handlers are typically used to ensure that the Gamma application does not exit when a signal occurs.

Table 1. Signals

Signal	Description
`SIGABRT`	Abort signal from the `abort()` C function.
`SIGALRM`	A timer has occurred. This signal is reserved in most operating system implementations of Gamma for use with the `after`, `at` and `every` functions. This signal is not available in Linux because it is used by the timer processing internally to Gamma. In QNX, it is the timer signal from the `alarm()` C function.
`SIGBUS`	Bus error.
`SIGCHLD`	Child died. Generated when a child process of the current process has died.
`SIGCONT`	Continue. Causes the task to restart after a SIGSTP.
`SIGFPE`	Floating point exception. Generated by an illegal mathematical function call (such as division by zero).
`SIGHUP`	Hangup. Typically generated when a terminal session disconnects.
`SIGILL`	Illegal instruction. This is an internal error.
`SIGINT`	Keyboard interrupt. Generated by **`CTRL-C`**.
`SIGIO`	I/O processing is required. This signal is generated when a socket or file descriptor has incoming data which must be processed.
`SIGIOT`	IOT trap. A synonym for `SIGABRT`.

Signal	Description
SIGKILL	Killed. Kills the process with extreme prejudice. This signal cannot be caught.
SIGPIPE	Broken pipe. This occurs when a TCP/IP socket or a pipe to an inferior process is broken.
SIGPOLL	A pollable event. Synonym of SIGIO.
SIGPWR	Power failure. This is generated by a power monitor program to indicate that a power loss is imminent.
SIGQUIT	Quit.
SIGSEGV	Segmentation fault. This signal is generated by an attempt to access illegal memory. If this signal occurs, it represents a fault in the LISP interpreter and should be reported along with the corresponding memory address of the fault.
SIGSTOP	Stop execution immediately. This is used by the operating system to implement multi-tasking. This signal cannot be caught.
SIGSYS	Bad argument to a system routine. This happens very rarely.
SIGTERM	Terminated. This is generated by other programs which wish to terminate the job.
SIGTRAP	Trace/breakpoint trap.
SIGTSTP	Terminal stop. This signal is generated when the user attempts to stop a process (in operating systems which support job control).
SIGTTIN	Terminal input is available.
SIGTTOU	Terminal output is required.
SIGURG	Urgent. An urgent condition has occurred.
SIGUSR1	User-defined signal 1.
SIGUSR2	User-defined signal 2.
SIGWINCH	Window change. This is used to indicate that a change has been made to the size or position of the window in which the process is running.

Example

```
Gamma> getpid();
10341
Gamma> signal(SIGUSR1,#princ("Got the signal.\n"));
t
Gamma> kill(10341,SIGUSR1);
Got the signal.
t
Gamma>
```

See Also

after, at, every

sleep, usleep

sleep, usleep — suspend execution.

Syntax

```
sleep (seconds)
usleep (microseconds)
```

Arguments

seconds

The integer number of seconds to sleep.

microseconds

The integer number of microseconds to sleep.

Returns

t

Description

These functions suspend execution for the given number of seconds or microseconds, after which time the task continues. Signals and timers will still be processed during this time.

Example

```
Gamma> sleep (3);

(after 3 seconds...)

t
Gamma> usleep (500000);

(after 1/2 second...)

t
Gamma>
```

See Also

nanosleep

strerror

`strerror` — retrieves an error message.

Syntax

`strerror (`*`errno`*`)`

Arguments

`errno`

The error number as returned by `errno`.

Returns

An error message as a string.

Description

This function looks up error messages associated with error numbers.

Example

In this example, we first define a function to remove a file. Then we call that function on a non-existing file to generate an error. Finally, we check the returned error code to get the error message.

```
function remove_file(file)
{
   unlink(file);
   errno();
}

Gamma> ret_val = remove_file("/tmp/xyz");
2
Gamma> strerror(2);
"No such file or directory"
Gamma>
```

See Also

`errno`

system

`system` — treats its argument as a system command.

Syntax

`system (`*`command_line`*`)`

Arguments

`command_line`

A string.

Returns

A numerical return code as generated by the operating system.

Description

This function treats its argument as a command to be run in the native operating system. This function will wait until the command completes before returning with the command's exit status. In UNIX and QNX 4, the command may be run in the background by using an *`&`* symbol after the *`command_line`* argument.

Example

```
Gamma> system("ps");
 PID TTY          TIME CMD
7856 pts/4    00:00:00 bash
8335 pts/4    00:00:00 Gamma
8336 pts/4    00:00:00 ps
0
Gamma> system("ls *ty*");
li_type.c  li_type.o  privity.c  pty.lsp
0
Gamma> system("mysubtask &");
0
```

tcp_accept

`tcp_accept` — forks a new TCP socket on the server side to accept a new connection.

Syntax

```
tcp_accept (socket)
```

Arguments

`socket`

A descriptor for a listening socket, as returned by a call to `tcp_listen`.

Returns

A file descriptor for a new, connected socket.

Description

This function allows a passive, listening socket to accept a connection, by spawning a new socket that maintains the connection. It is essentially the same as the C `accept` function, but returns a socket descriptor instead of an address.

Example

```
            CLIENT SIDE:                                SERVER SIDE:

                                                Gamma> tcp_listen(51715);
                                                5
                                                Gamma> tcp_accept(51715);

    Gamma> tcp_connect("localhost", 51715);     6
    5

    Gamma> fd_write(5, "Hi there");             Gamma> fd_read(6, "Hi there");
    8                                           8
```

See Also

`tcp_connect`, `tcp_listen`

tcp_connect

`tcp_connect` — creates a client-side TCP socket connection.

Syntax

```
tcp_connect (host, port)
```

Arguments

`host`

The IP address of the host machine.

`port`

The port to connect to.

Returns

A file descriptor for a new, connected socket.

Description

This function creates a connected socket. This can be accessed with Gamma `fd_*` functions such as `fd_write` and `fd_read`.

Example

See the example for `tcp_accept`.

See Also

`tcp_accept`, `tcp_listen`

tcp_listen

`tcp_listen` — creates a server-side TCP socket connection.

Syntax

```
tcp_listen (port, backlog?)
```

Arguments

`port`

The port to connect to.

`backlog`

The IP address of the host machine.

Returns

A file descriptor for a new, connected socket.

Description

This function creates a connected socket. This can be accessed with Gamma `fd_*` functions such as `fd_write` and `fd_read`.

Example

See the example for `tcp_accept`.

See Also

`tcp_accept`, `tcp_connect`

wait

`wait` — waits for process exit status.

Syntax

```
wait (taskid?, options?)
```

Arguments

`taskid`

A process ID number. A value of `0` indicates any process.

`options`

Wait option `WNOHANG` or `WUNTRACED`.

Returns

One of three possibilities:

- A list of four items:
  1. The process ID.
  2. The process exit status (`WEXITSTATUS`), or `nil`.
  3. A termination signal (`WTERMSIG`) if the process exited due to a signal, or `nil`.
  4. A stopped signal (`WSTOPSIG`) if the process stopped due to a signal, or `nil`.
- `t` if the `WNOHANG` option had been set and there was a child with a status change.
- `nil` if there was a failure due to error.

Description

This function combines and simplifies the C functions `wait` and `waitpid` in a single function. If *`taskid`* is provided, then the function acts as `waitpid`, and will not return until the given child task has died.

The `WNOHANG` option allows the calling process to continue if the status of specified child process is not immediately available. The `WUNTRACED` option allows the calling process to return if the child process has stopped and its status has not been reported. Both of these can be specified using the `OR` (`||`) operator.

Example

Process 1:

```
Gamma> child = fork();
9089
Gamma> 0
Gamma> if (child > 0) wait(); else exit_program(5);
```

Process 2:

```
Gamma> kill(9089,14);
```

```
t
Gamma>
```

Process 1:

```
(9089 nil 14 nil)
Gamma>
```

See Also

`fork`, `exec`

IV. Dynamic Loading

Table of Contents

AutoLoad

AutoLoad — allows for run-time symbol lookup.

Syntax

```
AutoLoad ("pattern", 'action)
```

Arguments

pattern

A shell style pattern.

action

An action to be taken when the *pattern* is matched.

Returns

The _auto_load_alist_, which is a list of all currently stored AutoLoad rules. Each rule is itself formatted as a list. This is the _auto_load_alist_ syntax:

```
((pattern action_func action_arg ...) ...)
```

The members of each rule list are as follows:

pattern

The AutoLoad *pattern* parameter.

action_func

The function specified in the AutoLoad *action* parameter.

action_arg

The function argument(s) specified in the AutoLoad *action* parameter.

For example, the AutoLoad rules in AutoLoadLib.g (at the time of this writing) would be returned as follows:

```
(("P[Tthg]*" DllLoad "libgammaph.so") ("gl[A-Z]*" DllLoad "libgammagl.so")
 ("GLUT_*" DllLoad "libgammagl.so") ("GLU_*" DllLoad "libgammagl.so")
 ("GL_*" DllLoad "libgammagl.so") ("ASCII_*" DllLoad "libgammamgl.so")
 ("KB_*" DllLoad "libgammamgl.so") ("GM_*" DllLoad "libgammamgl.so")
 ("EVT_*" DllLoad "libgammamgl.so") ("[mM][gG][lL]*" DllLoad "libgammamgl.so")
 ("[gG]tk*" DllLoad "gammagtk.so"))
```

Description

This function gives Gamma a way to look up symbols during run-time. If Gamma comes across an undefined symbol while executing a program, and if the symbol matches the *pattern*, then Gamma executes the *action*. Normally the *action* is either a direct definition of the symbol, or an attempt to load a DLL that defines the symbol, using DllLoad, for example.

The available *pattern*s are as follows:

- ***** matches any number of characters, including zero.
- **[*c*]** matches a single character which is a member of the set contained within the square brackets.
- **[^*c*]** matches any single character which is not a member of the set contained within the square brackets.

- **?** matches a single character.
- ***{xx,yy}*** matches either of the simple strings contained within the braces.
- ***\c*** (a backslash followed by a character) - matches that character.

This function is not part of the base Gamma executable. It is provided by a Gamma library `AutoLoadLib.g` which can be accessed using the Gamma `require` function like this:

```
require ("/usr/cogent/require/AutoLoadLib.g");
```

Example

- In this example, we use the `ClearAutoLoad` function to clear the `AutoLoad` list just to make the steps easier to follow.
- Once a library is loaded or a symbol is defined, Gamma no longer sends a "`Looking for symbol`" message.
- Notice how although `NoAutoLoad` and `ClearAutoLoad` remove a pattern from future consideration, any symbols defined or any libraries loaded before they were called remain valid.

```
Gamma> require ("/usr/cogent/require/AutoLoadLib.g");
t
Gamma> ClearAutoLoad();
nil
Gamma> AutoLoad ("[gG]tk*", 'DllLoad ("gammagtk.so"));
(("[gG]tk*" DllLoad "gammagtk.so"))
Gamma> gtk_arg_new;
Looking for gtk_arg_new
(defun gtk_arg_new (arg_type) ...)
Gamma> gtk_main;
(defun gtk_main () ...)
Gamma> testvar;
Looking for testvar
Symbol is undefined: testvar
debug 1> (Ctrl - D)
Gamma> AutoLoad("testvar", 'testvar = 5);
(("testvar" setq testvar 5) ("[gG]tk*" DllLoad "gammagtk.so"))
Gamma> testvar;
Looking for testvar
5
Gamma> NoAutoLoad("testvar");
(("[gG]tk*" DllLoad "gammagtk.so"))
Gamma> testvar;
5
Gamma> ClearAutoLoad();
nil
Gamma> gtk_main;
(defun gtk_main () ...)
Gamma> gtk_false;
(defun gtk_false () ...)
Gamma>
```

See Also

`ClearAutoLoad`, `NoAutoLoad`, `DllLoad`

autoload_undefined_symbol

autoload_undefined_symbol — checks undefined symbols for AutoLoad.

Syntax

autoload_undefined_symbol (!*sym*)

Arguments

sym

A symbol.

Returns

nil on success, else error.

Description

This function is generally used internally by the AutoLoadLib.g program. It is the default function that is called when an undefined symbol is encountered at run-time, if the AutoLoad.g library has been required into the program. This is normally done by startup.g, which is automatically loaded by the Gamma executable at startup.

Example

In this example, the first symbol (test1) is checked by autoload_undefined_symbol from within the AutoLoadLib.g program. We know this because the message "Looking for *symbol*" indicates that Gamma had to use AutoLoadLib.g to get the definition of the symbol. For the second symbol (test2), we make the autoload_undefined_symbol call ourselves, and the "Looking for *symbol*" doesn't appear. This indicates that Gamma knew the value of the symbol and didn't have to use AutoLoadLib.g to look it up.

```
Gamma> AutoLoad("test1", 'test1 = 9);
(("test1" setq test1 9) ("P[Tthg]*" DllLoad "libgammaph.so")...)
Gamma> test1;
Looking for test1
9
Gamma> AutoLoad("test2", 'test2 = 8);
(("test2" setq test2 8) ("test1" setq test1 9) ("P[Tthg]*" DllLoad "libgammaph.so")...)
Gamma> autoload_undefined_symbol(test2);
nil
Gamma> test2;
8
Gamma>
```

See Also

AutoLoad

AutoMapFunction

`AutoMapFunction` — maps a C function to a Gamma function.

Syntax

`AutoMapFunction (`*`name`*`, `*`rettype`*`, `*`args`*`)`

Arguments

`name`

The name of a C function.

`rettype`

Not yet documented.

`args`

Not yet documented.

Returns

Not yet documented.

Description

This function checks to see if the C function *`name`* exists in (is linked into) the Gamma executable. If so, it then maps it to a Gamma function according to the *`rettype`* and *`args`*. The details this function have not yet been documented.

See Also

`AutoLoad`

ClearAutoLoad

`ClearAutoLoad` — removes all `AutoLoad` rules.

Syntax

```
ClearAutoLoad ()
```

Arguments

None.

Returns

`nil`.

Description

This function removes all `AutoLoad` rules by setting the `_auto_load_alist_` to `nil`.

This function is not part of the base Gamma executable. It is provided by a Gamma library `AutoLoadLib.g` which can be accessed using the Gamma `require` function like this:

```
require ("/usr/cogent/require/AutoLoadLib.g");
```

Example

See the example for AutoLoad.

See Also

`AutoLoad`, `NoAutoLoad`

dlclose

`dlclose` — closes an open dynamic library.

Syntax

```
dlclose (handle)
```

Arguments

`handle`

The "handle" returned by `dlopen`.

Returns

0 when successful, else -1.

Description

This function is a wrapper for the `dlclose` shell command. Each call decrements the link count in the dl library created by `dlopen`. When this count reaches zero and no other loaded libraries use symbols in it, the library is unloaded.

If the library exports a routine named `_fini`, that will be called just before the library is unloaded.

Example

```
Gamma> dlopen("libform.so",RTLD_NOW|RTLD_GLOBAL);
134940024
Gamma> a = dlopen("libform.so",RTLD_NOW|RTLD_GLOBAL);
134940024
Gamma> dlclose(a);
0
Gamma> dlclose(a);
0
Gamma> dlclose(a);
-1
Gamma>
```

See Also

`dlopen`

dlerror

`dlerror` — reports errors in dl functions.

Syntax

```
dlerror ()
```

Arguments

none

Returns

An error message, or 0 if no error has occurred since it was last called.

Description

This function returns an error message for the most recent error in `dlopen` or `dlclose`. If several errors have occured since the last call to `dlerror`, only the first will return an error message.

Example

```
Gamma> dlopen("nolibraryhere",RTLD_LAZY);
0
Gamma> dlopen("norhere",RTLD_LAZY);
0
Gamma> dlerror();
"norhere: cannot open shared object file: No such file or directory"
Gamma> dlerror();
nil
Gamma>
```

See Also

`dlopen`, `dlclose`

dlfunc

`dlfunc` — reserved for future use.

Syntax

```
dlfunc (handle symname rettype args)
```

Arguments

Returns

Description

Example

See Also

DllLoad

`DllLoad` — loads dynamic libraries.

Syntax

```
DllLoad ("filename", verbose? = nil)
```

Arguments

`filename`

The name of the dynamic library to be loaded.

`verbose`

When set to `t`, shows the paths of load attempts.

Returns

An integer "handle" on success, or an error message.

Description

This function loads a DLL if the system supports it (Linux, QNX 6, and MS-Windows). The first search path for the DLL is taken to be `./`, next is `/usr/cogent/dll/`, and finally the system DLL search path, if any.

Example

Without using the optional *`verbose`* parameter:

```
Gamma> DllLoad("gammagtk.so");
135024608
Gamma>
```

Using the optional *`verbose`* parameter:

```
Gamma> DllLoad("gammagtk.so", t);
DllLoad: attempting to load: ./gammagtk.so
DllLoad: attempting to load: /usr/cogent/dll/gammagtk.so
135024608
Gamma>
```

See Also

AutoLoad

dlmethod

`dlmethod` — reserved for future use.

Syntax

`dlmethod (`*`handle class methodname symname rettype args`*`)`

Arguments

Returns

Description

Example

See Also

NoAutoLoad

`NoAutoLoad` — removes selected `AutoLoad` rules.

Syntax

```
NoAutoLoad ("pattern")
```

Arguments

`pattern`

A shell style pattern.

Returns

The `_auto_load_alist_` (a list of all currently stored `AutoLoad` rules) with the rules corresponding to the *`pattern`* removed.

Description

This function removes from future consideration any `AutoLoad` rules that correspond to the *`pattern`*.

The available *`pattern`*s are as follows:

- ***** matches any number of characters, including zero.
- **[*c*]** matches a single character which is a member of the set contained within the square brackets.
- **[^*c*]** matches any single character which is not a member of the set contained within the square brackets.
- **?** matches a single character.
- **{*xx*,*yy*}** matches either of the simple strings contained within the braces.
- ***c*** (a backslash followed by a character) - matches that character.

This function is not part of the base Gamma executable. It is provided by a Gamma library `AutoLoadLib.g` which can be accessed using the Gamma `require` function like this:

```
require ("/usr/cogent/require/AutoLoadLib.g");
```

Example

See the example for AutoLoad.

See Also

`AutoLoad`, `ClearAutoLoad`

dlopen

`dlopen` — loads a dynamic library from a file.

Syntax

```
dlopen (filename flags)
```

Arguments

`filename`

The name of the file to open, as a string. If no absolute path is given, the file is searched for in the user's `LD_LIBRARY` path, the `/etc/ld.so.cache` list of libraries, and the `/usr/lib/` directory.

`flags`

Must be either `RTLD_LAZY` or `RTLD_NOW`, optionally OR'ed with `RTLD_GLOBAL`.

- **`RTLD_LAZY`** causes undefined symbols to be resolved as the dynamic library code executes.
- **`RTLD_NOW`** forces undefined symbols to be resolved before `dlopen` returns, otherwise `dlopen` fails.
- **`RTLD_GLOBAL`** makes any external symbols defined in the library available to subsequently loaded libraries.

Returns

An integer "handle" if successful, else 0.

Description

This function is a wrapper for the `dlopen` shell command. It loads a dynamic library from the file and returns a "handle", which is an integer uniquely associated with the file for this application. The same handle is returned each time the same library is opened, and the dl library counts the number of links created for each handle.

If the library exports a routine named `_init`, that will be executed before `dlopen` returns.

Example

```
Gamma> dlopen("libform.so",RTLD_LAZY|RTLD_GLOBAL);
134936808
Gamma> dlopen("libconsole.so",RTLD_NOW);
0
Gamma> dlopen("libconsole.so",RTLD_LAZY);
134935848
Gamma> dlopen("libconsole.so",RTLD_LAZY);
134935848
```

See Also

`dlclose`

V. Profiling and Debugging

Table of Contents

allocated_cells

`allocated_cells` — gives the number of allocated and free cells.

Syntax

```
allocated_cells ()
```

Arguments

none

Returns

A list containing the number of allocated cells and the number of free cells currently held by the memory management system.

Description

The memory management system allocates cells as required to continue execution, limited only by operating system memory. Once cells have been allocated, they are placed on the heap by the garbage collector and re-used. New cells are only allocated from the operating system if the garbage collector is unable to fulfill a request for more memory from the running Gamma or Lisp program. This function returns the number of cells which are currently in use, and the number of free cells remaining on the heap. The sum of these numbers is the total number of cells allocated by the interpreter.

Example

This example shows 380 cells in use and 1620 cells free on the heap, for a total of 2000 cells available.

```
Gamma> allocated_cells();
(380 1620 0 0 0 0 0 0)
Gamma>
```

See Also

`free_cells`, `gc`

eval_count

`eval_count` — counts evaluations made since a program started.

Syntax

```
eval_count ()
```

Arguments

none

Returns

A list of three values. First is the number of times any symbol has been evaluated. Second is the number of times any function has been evaluated. Third is the number of times any other Gamma expression has been evaluated.

Description

This function counts the number of evaluations of symbols, functions, and other Gamma expressions. All of these are counted from the time the program started.

Example

```
Gamma> gc();
1
Gamma> eval_count();
(0 2 0)
Gamma> a = 5;
5
Gamma> eval_count();
(0 4 1)
Gamma> a;
5
Gamma> eval_count();
(1 5 1)
Gamma>
```

free_cells

`free_cells` — returns the number of available memory cells.

Syntax

```
free_cells ()
```

Arguments

none

Returns

The number of free memory cells available on the memory heap.

Example

```
Gamma> free_cells();
1620
Gamma>
```

See Also

`allocated_cells`

function_calls

`function_calls` — tells how often a function was called during profiling.

Syntax

`function_calls (`*`function`*`)`

Arguments

`function`

A function.

Returns

The number of times this function has been called while profiling was active.

Description

This function queries the system to determine the number of times that a function was called while profiling was active (using the profile function).

Example

```
Gamma> profile(t);
t
for(i=0;i<10;i++)
{
 princ("i:",i,"\n");
}
>> i:0
>> i:1
>> i:2
>> i:3
>> i:4
>> i:5
>> i:6
>> i:7
>> i:8
>> i:9
Gamma> i;
9
Gamma> profile(nil);
t
Gamma> function_calls(princ);
10
```

See Also

`profile`, `function_runtime`

function_runtime

`function_runtime` — gives the time a function has run during profiling.

Syntax

```
function_runtime (function)
```

Arguments

`function`

A function.

Returns

The total number of seconds that the *`function`* has run.

Description

This function returns the number of seconds (as a floating point number) that a function has run during all complete invocations of the function while profiling has been active. The number of seconds is measured using the QNX 4 tick clock, and thus represents elapsed time rather than CPU time, with a granularity of one tick (typically 10ms). Invocations of the function which have not completed at the time of the call to `function_runtime` are not included in the calculation.

Example

```
Gamma> function_runtime(cdr);
0.05341
```

See Also

`function_calls`, `profile`

gc

`gc` — runs the garbage collector.

Syntax

```
gc ()
```

Arguments

none

Returns

The number of cells freed by the garbage collector.

Description

Causes the garbage collector to run if possible. The garbage collector will not run during a timer or signal handler, but it will flag the need for garbage collection, causing the garbage collector to run immediately after the timer or signal handler exits.

Example

```
Gamma> gc();
68
Gamma> gc();
17
Gamma> fp = open("myfile.dat", "r", nil);
#<File:"myfile.dat">
Gamma> close(fp);
t
Gamma> gc();
67
Gamma> gc();
17
Gamma>
```

See Also

`allocated_cells`, `free_cells`

gc_blocksize

`gc_blocksize` — for internal use only.

Syntax

```
gc_blocksize (ncells)
```

gc_enable

`gc_enable` — for internal use only.

Syntax

```
gc_enable (enable_p)
```

gc_newblock

`gc_newblock` — for internal use only.

Syntax

```
gc_newblock ()
```

gc_trace

`gc_trace` — controls the tracing of garbage collection.

Syntax

```
gc_trace (on_flag)
```

Arguments

`on_flag`

If non-`nil`, turn on garbage collector tracing, else turn it off.

Returns

The new status of garbage collector tracing.

Description

This function turns on (on-flag is non-`nil`) or off (on-flag is `nil`) the tracing of garbage collection. When garbage collection tracing is on, statistics are collected concerning the number of allocated cells, number of collection calls, and the elapsed time spent within the garbage collector. These statistics can be accessed using a call to allocated-cells.

Example

```
Gamma> gc_trace (t);
nil
```

See Also

`allocated_cells`

profile

`profile` — collects statistics on function usage and run time.

Syntax

`profile (`*`on_p`*`, `*`tick_nanosecs`*`?)`

Arguments

`on_p`

If non-`nil`, start profiling, else stop profiling.

`tick_nanosecs`

Reset the QNX 4 tick size to this many nanoseconds before beginning to profile.

Returns

The previous state of profiling.

Description

This function starts (or stops) collecting statistics on the usage and run time of all functions in the system. The profile mechanism uses an interrupt on the QNX 4 tick clock, and so must run with root permissions. If the optional *`tick_nanosecs`* argument is provided, this function will reset the tick size. Otherwise, it will profile using the current tick size. The smaller the tick size, the more precise is the profile result.

Example

The following program gives the output shown below.

```
#!/usr/cogent/bin/gamma

require_lisp("Profile.lsp");

e_list = list();
j = 0;

function print_reverse()
{
  with i in cdr(argv) do
    {
      e_list = cons(i, e_list);
      j++;
    }
  princ("The numbers in reverse order are:\n", e_list, "\n");
}

function main()
{
  profile(t);
  print_reverse();
  profile(nil);

  profiled_functions();
  princ("Function calls: ", function_calls(cons),"\n");
  princ("Function runtime: ", function_runtime(cons),"\n");
}
```

Entered on command line:

```
[sh]$ ex_profile.g 1 2 3 4 5
```

Output:

```
The numbers in reverse order are:
(5 4 3 2 1)
                  Function     Calls  Total Time
                       +++         5  4e-06
                       cdr         1  0
                      cons         5  2e-06
                       for         1  2.3e-05
                   profile         1  1e-06
                      setq         5  3e-06
                     princ         1  0.006502
             print_reverse         1  0.006534
                     progn         6  0.006546
Function calls: 5
Function runtime: 1.9999999999999999095e-06
```

See Also

function_calls, function_runtime

set_autotrace

`set_autotrace` — is reserved for future use.

Syntax

```
set_autotrace (state, functions...)
```

Arguments

state

functions

Returns

Description

Example

See Also

`set_breakpoint`

`set_breakpoint` — is reserved for future use.

Syntax

```
set_breakpoint (state, functions...)
```

Arguments

`state`

`functions`

Returns

Description

Example

See Also

time

`time` — gives command execution times.

Syntax

`time (`*`iterations`*`, !`*`command`*`)`

Arguments

`iterations`

The number of times to execute the command.

`command`

Any Gamma or Lisp command.

Returns

The number of seconds consumed performing the *`command`* for the given number of *`iterations`*.

Description

This function performs the *`command`* for the given number of *`iterations`* and returns the clock time consumed. This does not break down the time into user and system time. Times on successive calls to this function will differ slightly due to operating system requirements, garbage collection and active timers.

Example

```
Gamma> time(10, list(1,2,3));
3.297225339338183403e_05
Gamma>
```

trace, notrace

`trace, notrace` — turn tracing on or off.

Syntax

```
trace (!code?)
notrace (!code?)
```

Arguments

code

If provided, limits the scope to this code.

Returns

With no argument, `t`, or with a *code* argument, the result of evaluating code.

Description

These functions turn tracing (execution tracking to standard output) on or off, either at the global level, or for the duration of the evaluation of *code*, if provided.

Example

```
#!/usr/local/bin/gamma -d

/* here is an example of a troublesome function and its
   return being debugged with the aid of trace() and
   notrace() functions.
*/

a = 0;
b = 1;
trace();
function trouble_function(x,y) {y/x;}
results = trouble_function(a,b);
notrace();
princ(results, "\n");
```

Gamma generates the following:

```
(defun trouble_function (x y) (/ y x))
--> trouble_function
(trouble_function a b)
  (/ y x)
  --> inf
--> inf
(setq results (trouble_function a b))
--> inf
(notrace)
inf
```

VI. Miscellaneous

Table of Contents

apropos

`apropos` — finds all defined symbols in the current interpreter environment.

Syntax

`apropos (`*`pattern`*`, `*`predicate`*`?)`

Arguments

`pattern`

A character string which specifies a search pattern

`predicate`

A function taking one argument which will return either `nil` or non-`nil`.

Returns

A list of all symbols defined in the system which match the given *`pattern`*, and if the *`predicate`* is supplied, whose values are true under that predicate.

Description

This function searches the names of all defined symbols in the currently running interpreter environment. The *`pattern`* can contain the following special characters:

- ***** matches any number of characters, including zero.
- **[*`c`*]** matches a single character which is a member of the set contained within the square brackets.
- **[^*`c`*]** matches any single character which is not a member of the set contained within the square brackets.
- **?** matches a single character.
- **{*`xx`*,*`yy`*}** matches either of the simple strings contained within the braces.
- ******`c`*** (a backslash followed by a character) - matches that character.

The *`predicate`* is any function which accepts a single argument. If the *`predicate`* evaluates to non-`nil` when given the value of a symbol, and if the symbol matches the pattern, then the symbol will be reported by apropos. If the *`predicate`* is not supplied, then all symbols which match the *`pattern`* will be reported. The *`pattern`* is case-sensitive.

Example

```
Gamma> apropos("s*", function_p);
(setq strchr string symbol)
Gamma> apropos("?[sli]{igc,er}*");
(SIGCHLD SIGCONT dlerror)
```

create_state, enter_state, exit_state

create_state, enter_state, exit_state — are part of the SCADALisp exception-driven state machine mechanism.

Syntax

```
create_state (state_function, symbol?...)
enter_state (state_machine, state)
exit_state (state_machine, state)
```

Arguments

state_function

The function to call upon entering this state.

symbol

One or more symbols which will act as triggers to cause this state to be re-evaluated.

state_machine

A state machine created through a call to (new StateMachine)

state

A state created through a call to (create-state...)

Returns

create_state: The new state definition.

enter_state: A status value.

exit_state: A status value.

Description

These functions are part of the exception-driven state machine mechanism built into SCADALisp. This mechanism is not fully supported, and will not be documented for this release. The reader may find the library file StateMachine.lsp helpful in determining how to use state machines. In general, this function should not be called directly from user code, as it is designed to provide support for the StateMachine library functions.

Example

none

gensym

`gensym` — generates a unique symbol.

Syntax

`gensym (`*`prefix_string`*`?)`

Arguments

`prefix_string`

A character string which will be used as the prefix for the newly generated symbol.

Returns

A unique symbol.

Description

This function generates a symbol which does not currently exist by attaching a unique number to the end of the *`prefix_string`*. If the *`prefix_string`* is `nil`, use a default prefix.

Example

```
Gamma> gensym("tag");
tag1
Gamma> gensym();
tmp_sym2
Gamma> tag3 = 1;
1
Gamma> gensym("tag");
tag4
Gamma>
```

modules

`modules` — is obsolete, and returns nothing of value.

Syntax

```
modules ()
```

Arguments

none

nil

Returns

A string.

Description

This function is obsolete, and returns nothing of value.

stack

`stack` — lists all functions called so far.

Syntax

```
stack ()
```

Arguments

none

Returns

A list of all of the functions called up to this point in the execution of the Gamma program.

Description

A function that calls `stack` is presented in order, with the most recently called function at the end of the function list. `stack` can be useful for debugging programs by requesting a stack trace when an error occurs.

Example

The following program:

```
#!/usr/cogent/bin/gamma

function hms_to_sec(hms)
{
    hms = list_to_array(string_split(hms, ":", -1));
    (number(hms[0]) * 60 + number(hms[1])) * 60 + number(hms[2]);
    stk = stack();
}

tocheck = list(12,5,13);
hms_to_sec("tocheck");
princ(stk,"\n");
```

Yields these results:

```
((hms_to_sec tocheck) (progn (setq hms (list_to_array (string_split hms
 : (neg 1)))) (+ (* (+ (* (number (aref hms 0)) 60) (number (aref hms 1)
)) 60) (number (aref hms 2))) (setq stk #0=(stack))) #0#)
```

See Also

`print_stack`

VII. IPC

Table of Contents

add_hook

`add_hook` — hooks a function to an event.

Syntax

```
add_hook (hook_sym, function_sym)
```

Arguments

`hook_sym`

One of several symbols used to identify the hook, as listed below.

`function_sym`

The function that is to run when the event occurs.

Returns

The hooked function (*`function_sym`*) that was added.

Description

This function sets up a hook, which is a function that is called when a particular event takes place. The arguments to the function identified by the *`function_sym`* are determined by the particular event. A hook function must be defined with the correct number of arguments, or else with optional or variable length arguments. The currently available hooks and the respective events that trigger their functions are as follows:

- `taskstarted_hook`: triggered whenever a task starts.
- `taskdied_hook`: triggered whenever a task dies.
- `exception_hook`: triggered whenever an exception for any point is emitted by the Cogent DataHub.
- `echo_hook`: triggered whenever an echo for any point is emitted by the Cogent DataHub.
- `gc_hook`: triggered whenever the garbage collector runs.
- The following are related to tracing code executions, but haven't been fully documented.
  ```
  trace_symbol_hook
  trace_entry_hook
  trace_exit_hook
  breakpoint_hook
  ```

One common use of this function is to add the internal `taskstarted` or `taskdied` functions, with the `taskstarted_hook` or `taskdied_hook`. Whenever a task that is registered with **nserve** starts or dies, **nserve** sends a message to all Gamma applications running IPC. Any of these applications that has added the `taskstarted_hook` or `taskdied_hook` then runs their corresponding *`function_sym`* function.

Example

This example program requires **qserve** and **nserve** to be running. It gives the output shown below:

```
#!/usr/cogent/bin/gamma

//Program: ex_addrunhooks.g
```

```
function main ()
{
  init_ipc ("x","x");

  add_hook (#taskstarted_hook, #hook_started);
  add_hook (#taskdied_hook, #hook_died);

  run_hooks (#taskstarted_hook, "testing start");
  run_hooks (#taskdied_hook, "testing died");

  while(t)
    next_event();
}

function hook_started (!a?...=nil)
{
  princ ("Hooked task started: ", a, "\n");
}

function hook_died (!a?...=nil)
{
  princ ("Hooked task died: ", a, "\n");
}
```

Output from `ex_addrunhooks.g` at startup:

```
Hooked task started: (testing start)
Hooked task died: (testing died)
```

Starting a new Gamma task named `mytask`...

```
Gamma> init_ipc("mytask", "myqueue");
t
Gamma>
```

...elicits this output from `ex_addrunhooks.g`:

```
Hooked task started: (mytask default myqueue 0 0 1874 0)
```

Checking process status with **nsnames**:

```
[home/robert]$ nsnames
Name   Domain  Queue    NID PID
mytask default myqueue  0   1874
x      default x        0   1873
```

Terminating `mytask` elicits this output from `ex_addrunhooks.g`:

```
Hooked task died: (mytask default myqueue 0 0 1874 0)
```

See Also

`run_hooks`, `remove_hook`, `init_ipc`

close_task

close_task — closes a task opened by locate_task.

Syntax

close_task (*task*)

Arguments

task

A task descriptor as assigned to a locate_task call.

Returns

t if the task could be closed, else nil.

Description

When a task is opened (located) for interprocess communication, a communication link may be established. This link must be cleaned up if it is to be re-used. There is no hard limit to the number of tasks which may be open with QNX 4 message passing, but TCP/IP exerts an operating system-dependent limit on the number of simultaneously open tasks. Tasks will automatically be closed by the garbage collector when they are no longer referenced.

Example

```
Gamma> task = locate_task("Task 1",nil);
#<Task:9684>
Gamma> close_task(task);
t
Gamma>
```

See Also

locate_task

`_destroy_task`

`_destroy_task` — should never be used.

This function should not be used under any circumstances.

init_async_ipc

`init_async_ipc` — requests queue information from a task.

Syntax

`init_async_ipc (`*`other_task`*`)`

Arguments

`other_task`

A task descriptor as assigned to a `locate_task` call.

Returns

Non-`nil` on success, or `nil` on failure.

Description

This function initializes the interprocess communication system to allow this task to make calls to `register_point`, `register_existing_point`, `send_async` and `send_string_async`. It requests queue information from the given task. A deadlock situation could occur if two tasks attempt to initialize asynchronous communication with one another at the same time. The queue server task, **qserve**, must be running for this call to succeed.

Example

```
Gamma> init_ipc("mytask","mytask_q");
t
Gamma> task = locate_task("server", t);
#<Task: 32271>
Gamma> init_async_ipc(task);
t
```

See Also

`init_ipc`, `locate_task`, `register_point`, `send_async`, `send_string_async`

init_ipc

`init_ipc` — sets up necessary data structures for IPC.

Syntax

```
init_ipc (my_name, my_queue_name?, domain?)
```

Arguments

my_name

A name for this task, as a string. It is only used internally.

my_queue_name

Optional queue name for this task, as a string. This is necessary for asynchronous communication, and it must be unique on the system.

domain

Optional domain name for this task.

Returns

`t` on success, otherwise `nil`.

Description

Sets up all of the data structures needed prior to attempting any interprocess communication from this task. Messages can be neither sent nor received before this call is made. All Cogent DataHub functions use IPC. If the value of *my_queue_name* is `nil`, no queue name is assigned and no asynchronous IPC is possible.

Example

```
Gamma> init_ipc("myname","myqueue");
t
Gamma>
```

See Also

`isend`, `next_event`, `next_event_nb`, `read_point`, `read_existing_point`, `register_point`, `send`, `send_async`, `send_string`, `send_string_async`, `write_point`, `write_existing_point`

isend

`isend` — sends a synchronous message and doesn't wait for the result.

Syntax

```
isend (task, s_exp)
```

Arguments

`task`

A task descriptor as assigned to a `locate_task` call.

`s_exp`

Any Gamma or Lisp expression.

Returns

`t` if the message was sent successfully, otherwise `nil`.

Description

This function sends a message via synchronous interprocess communication, but does not wait for the result. The receiving task must respond immediately, prior to actually evaluating the message that was sent. The result code can only show whether the message was sent successfully. This is a compromise between synchronous and asynchronous messaging techniques.

Example

```
Gamma> init_ipc("mytask","myqueue");
t
Gamma> task = locate_task("other_task",nil);
<task id>
Gamma> isend(task,#list(do_something));
t
```

See Also

`locate_task`, `send`, `send_async`

locate_task

`locate_task` — finds and connects to tasks by name.

Syntax

`locate_task (task_name, async_reqd)`

Arguments

`task_name`

The name of the task to locate. The other task must have declared this name through `init_ipc` or `name_attach`.

`async_reqd`

`t` if `locate_task` should automatically call `init_async_ipc` for this task.

Returns

A task if successful, otherwise `nil`.

Description

This function makes a call to the name locator task for the current operating system. If it finds the named task it makes an IPC connection (in TCP/IP) or creates a virtual circuit (in QNX 4) to that task. If *`async_reqd`* is `t`, then `init_async_ipc` is also called.

When Gamma locates a task, it returns a printed representation of it, which looks like this: #<Task:10120>. This representation cannot be read back into Gamma, so a symbol is usually assigned when calling `locate_task` to facilitate refering to or working with a task. We refer to this symbol as the task descriptor. For instance, in the example below, the symbol `tsk` is the task descriptor.

Example

The two tasks are initiated with `init_ipc` before calling `locate_task`.

Task 1:

```
Gamma> init_ipc("first","Q1");
t
Gamma> getpid();
9231
Gamma> tsk = locate_task("second",nil);
#<Task:9092>
Gamma> send (tsk, #princ("Are you there?\n"));
t
Gamma>
```

Task 2:

```
Gamma> init_ipc("second","Q2");
t
Gamma> getpid();
9092
Gamma> next_event
Are you there?
t
```

```
Gamma>
```

See Also

locate_task_id

locate_task_id

`locate_task_id` — finds and connects to tasks by task ID and network node.

Syntax

`locate_task_id (task_id, node_id, channel_id, async_reqd)`

Arguments

`task_id`

The task ID for this task (as a number).

`node_id`

The network node number for this task.

`channel_id`

The task ID for this task. This is required for QNX 6, ignored in QNX 4 and Linux.

`async_reqd`

`t` if `locate_task` should automatically call `init_async_ipc` for this task.

Returns

A task if successful, otherwise `nil`.

Description

This function makes a TCP/IP connection or QNX 4 virtual circuit to the named task based on the *`task_id`* and the *`node`* number on which the task is running. If *`async_reqd`* is `t`, then `init_async_ipc` is also called. If the *`node`* number is zero, the current node is used.

Example

Task 1:

```
Gamma> init_ipc("Task 1","14");
t
Gamma> getpid();
9271
Gamma>
```

Task 2:

```
Gamma> init_ipc("Task 2","25");
t
Gamma> locate_task_id(9271,1,nil);
#<Task:9271>
Gamma>
```

See Also

`locate_task`

name_attach

`name_attach` — attaches a name to a task.

Syntax

```
name_attach (task_name)
```

Arguments

`task_name`

The name to attach to this task.

Returns

`t` if the name was successfully attached, otherwise `nil`.

Description

This function sends a message to the QNX 4 name locator task (`nameloc`) to attach a name on this node. If the name locator is not running or the name has already been attached by another task, the call will fail.

Example

```
// attach my name
Gamma> name_attach("firstname");
t
// attach an alternate name
Gamma> name_attach("pseudonym");
t
// attempt to attach my name again
Gamma> name_attach("firstname");
nil
```

nserve_query

`nserve_query` — puts information from **nserve** into an array.

Syntax

```
nserve_query ()
```

Arguments

none

Returns

An array of instances of the class `TaskInfo`, or `nil` on failure.

Description

This function retrieves all the information available in the Cascade NameServer (**nserve**), and puts it into an array. Each item in the array is an instance of the `TaskInfo` class, as returned from the function `task_info`. Please refer to the documentation of that function for more details.

This function requires that `init_ipc` be called first.

Example

```
Gamma> init_ipc("a", "aq");
t
Gamma> pretty_princ(nserve_query(), "\n");
[{TaskInfo (channel_id . 0) (domain . toolsdemo) (name . /dh/toolsdemo)
    (node_id . 0) (node_name . 0) (pid . 5394)
    (queue_name . /dh/toolsde) (queue_size . 0)}
 {TaskInfo (channel_id . 0) (domain . toolsdemo) (name . control)
    (node_id . 0) (node_name . 0) (pid . 16995)
    (queue_name . controlq) (queue_size . 0)}
 {TaskInfo (channel_id . 0) (domain . toolsdemo) (name . emul) (node_id . 0)
    (node_name . 0) (pid . 16998) (queue_name . emulq)
    (queue_size . 0)}
 {TaskInfo (channel_id . 0) (domain . default) (name . a) (node_id . 0)
    (node_name . 0) (pid . 16999) (queue_name . aq) (queue_size . 0)}]
t
Gamma>
```

See Also

`task_info`

remove_hook

remove_hook — removes a hooked function.

Syntax

```
remove_hook (hook_sym, function_sym)
```

Arguments

hook_sym

One of several symbols used to identify a hook, as listed below.

function_sym

The function that is to be removed.

Returns

The hooked function (*function_sym*) that was removed.

Description

This function removes a hook that was previously set up with add_hook. The currently available hooks are:

```
taskstarted_hook
taskdied_hook
exception_hook
echo_hook
gc_hook
trace_symbol_hook
trace_entry_hook
trace_exit_hook
breakpoint_hook
```

Example

Modifying the example in add_hook by adding one line:

```
...

add_hook (#taskstarted_hook, #hook_started);
add_hook (#taskdied_hook, #hook_died);

run_hooks (#taskstarted_hook, "testing start");
run_hooks (#taskdied_hook, "testing died");

/* Remove the hook */
remove_hook (#taskstarted_hook, #hook_started);

while(t)
...
```

would remove the taskstarted_hook.

See Also

add_hook, run_hooks, init_ipc

run_hooks

`run_hooks` — runs a hooked function.

Syntax

```
run_hooks (hook_sym, args...?)
```

Arguments

`hook_sym`

One of several symbols used to identify a hook, as listed below.

`args`

The arguments of the function that is to run when the event occurs.

Returns

`t` on success or `nil` on failure.

Description

This function runs a hook that was previously set up with `add_hook`. The currently available hooks are:

```
taskstarted_hook
taskdied_hook
exception_hook
echo_hook
gc_hook
trace_symbol_hook
trace_entry_hook
trace_exit_hook
breakpoint_hook
```

Example

Please refer to the example in `add_hook`.

See Also

`remove_hook`, `init_ipc`

send

`send` — transmits expressions for evaluation.

Syntax

`send (task, s_exp)`

Arguments

`task`

A task descriptor as assigned to a `locate_task` call.

`s_exp`

Any Gamma or Lisp expression.

Returns

A result depending on the receiving task, which could include:

- `t` if the message was delivered successfully.
- `nil` if the message could not be delivered.
- An expression in the form: (error "error message") if there was an error. See `error`.

Description

This function constructs an ASCII string representing the *`s_exp`* and transmits it via synchronous interprocess communication to the receiving *`task`*. The *`task`* processes the message and returns a result based on that processing. If the *`task`* is another Gamma process, the message will be interpreted as a Gamma expression and evaluated. The return value will be the result of that evaluation.

Example

Task 1:

```
Gamma> init_ipc ("a","a");
t
Gamma> tsk = locate_task("b",nil);
#<Task:9751>
Gamma> send(tsk, #princ("hello\n"));
hello
t
Gamma> send_async(tsk, #princ(cos(5), "\n"));
t
Gamma> send(tsk, #princ("goodbye\n"));
t
Gamma>
```

Task 2:

```
Gamma> init_ipc ("b","b");
t
Gamma> while(t) next_event();
hello
0.28366218546322624627
goodbye
```

See Also

`isend`, `locate_task`, `send_async`, `send_string`, `send_string_async`

send_async

send_async — transmits expressions asynchronously.

Syntax

send_async (*task*, *s_exp*)

Arguments

task

A task descriptor as assigned to a locate_task call.

s_exp

Any Gamma or Lisp expression.

Returns

t if the message was successfully delivered, otherwise nil.

Description

This function constructs a string representation of the given expression and delivers it via asynchronous interprocess communication to the receiving task. If the message could not be delivered, send_async returns nil. There is no indication of the status of the receiving task as a result of processing the message.

Example

Task 1:

```
Gamma> init_ipc ("a","a");
t
Gamma> tsk = locate_task("b",t);
#<Task:9751>
Gamma> send_async(tsk, #princ("hello, b\n"));
t
Gamma> send_async(tsk, #princ(cos(5), "\n"));
t
Gamma>
```

Task 2:

```
Gamma> init_ipc ("b","b");
t
Gamma> while(t) next_event();
hello, b
0.28366218546322624627
```

See Also

isend, locate_task, send, send_string, send_string_async

send_string

send_string — transmits strings for evaluation.

Syntax

```
send_string (task, string)
```

Arguments

task

A task descriptor as assigned to a locate_task call.

string

Any string.

Returns

A result depending on the receiving *task*.

Description

This function transmits the *string* via synchronous interprocess communication to a non-Cogent DataHub receiving *task*. The *task* processes the message and returns a result based on that processing. If the *task* is a Gamma process, the message will be interpreted as a Lisp expression and evaluated. The return value will be the result of that evaluation. If an error occurs during the evaluation, an expression of the form: (error "error message") will be returned. If the message could not be delivered, nil is returned.

Example

```
Gamma> a = 5;
5
Gamma> b = 6;
5
Gamma> send_string(task,string("(+",a," ",b,")"));
11
```

See Also

isend, locate_task, send, send_async, send_string_async

send_string_async

send_string_async — transmits a string asynchronously.

Syntax

send_string_async (*task*, *string*)

Arguments

task

A task descriptor as assigned to a locate_task call.

string

A string.

Returns

t if the message was successfully delivered, otherwise nil.

Description

This function delivers the *string* via asynchronous interprocess communication to a non-Cogent DataHub receiving *task*. If the message could not be delivered, send_string_async returns nil. There is no indication of the status of the receiving *task* as a result of processing the message.

Example

Task 1:

```
Gamma> init_ipc ("a","a");
t
Gamma> tsk = locate_task("b",t);
#<Task:9751>
Gamma> send_string_async(tsk, "2 + 2");
t
Gamma> send_string_async(tsk,string(list(#a,#b,#c)));
t
Gamma>
```

Task 2:

```
Gamma> init_ipc ("b","b");
t
Gamma> while(t) next_event();
```

See Also

isend, locate_task, send, send_async, send_string

taskdied, taskstarted

`taskdied, taskstarted` — internal functions that call another function when a task starts or stops.

Syntax

```
taskdied (task_name, qname, domain, node, task_id)
taskstarted (task_name, qname, domain, node, task_id)
```

Arguments

task_name

The name of the task which started or stopped.

node

The node on which the task started or stopped.

task_id

The process ID for the task.

qname

The name of the task's queue, if any.

domain

The Cogent DataHub domain for this task.

Returns

User-defined.

Description

These functions are internal to Gamma. They call `run_hooks (#taskstarted_hook, args...)` and `run_hooks (#taskdied_hook, args...)` respectively. They are called whenever a task registered with the Cascade NameServer (**nserve**) starts or stops. You can set up hooks to use these functions through the `add_hook` function.

These functions were originally available to programmers, and have been internalized to allow for the greater flexibility of the `add_hook` and `run_hook` functions. However, if you have existing code that you don't want to change, you can define your own versions of `taskdied` and `taskstarted` that shadow the built-in functions and do what they always used to do. Your old code will not break, but it will hide the hook version of the `taskdied` and `taskstarted` functions.

On the other hand, you could get both with something like this:

```
builtin_taskdied = taskdied;
builtin_taskstarted = taskstarted;

function main ()
{
    init_ipc ("x","x");

    add_hook (#taskdied_hook, #hook_taskdied);
    add_hook (#taskstarted_hook, #hook_taskstarted);

    while(t)
        next_event();
}

function taskdied (!a?...=nil)
{
    princ ("task died: ", a, "\n");
    funcall (builtin_taskdied, a);
}

function taskstarted (!a?...=nil)
{
    princ ("task started: ", a, "\n");
    funcall (builtin_taskstarted, a);
}

function hook_taskdied (!a?...=nil)
{
    princ ("hook task died: ", a, "\n");
}

function hook_taskstarted (!a?...=nil)
{
    princ ("hook task started: ", a, "\n");
}
```

task_info

`task_info` — gets information from a task descriptor.

Syntax

`task_info (tsk)`

Arguments

`tsk`

A task descriptor, as returned by the `locate_task` function.

Returns

An instance of the class `TaskInfo`, or `nil` on failure.

Description

This function returns an instance of Gamma's `TaskInfo` class. The instance variables of this class correspond to information contained in the task descriptor, as follows:

channel_id

The channel ID number, which is used in QNX 6 but not in QNX 4 or Linux.

domain

The name of the Cogent DataHub domain for the *`tsk`*.

name

The name of the *`tsk`*, as recorded in the Cascade NameServer. This attribute is not contained in a task descriptor, and thus is always returned as `nil` from this function.

node_id

The node ID number.

node_name

The `node_id` expressed as a string.

pid

The process ID number.

queue_name

The name of the Cascade QueueServer queue, as registered with the Cascade NameServer.

queue_size

The size of the Cascade QueueServer queue.

Example

```
Gamma> init_ipc("a", "aq");
t
Gamma> tsk = locate_task("/dh/toolsdemo", nil);
#>Task:5394<
Gamma> task_info(tsk);
{TaskInfo (channel_id . 0) (domain . "toolsdemo") (name)
(node_id . 0) (node_name . "0") (pid . 5394)
(queue_name . "/dh/toolsde") (queue_size . 0)}
Gamma>
```

See Also

nserve_query

VIII. Events and Callbacks

Table of Contents

add_set_function

add_set_function — sets an expression to be evaluated when a given symbol changes value.

Syntax

add_set_function (*symbol*, *s_exp*)

Arguments

symbol

A symbol.

s_exp

Any Gamma or Lisp expression.

Returns

t

Description

This function binds an expression to be evaluated whenever the value of the *symbol* changes. This expression is available globally, so if the value of the *symbol* changes during a change in scope, the expression will be evaluated. All changes in that sub-scope will trigger new evaluations of the expression. This can be used to automatically maintain consistency between the program and a the Cogent DataHub, or to implement forward chaining in calculation rules. The expression will not be evaluated if the new value is eq to the previous value.

When a set expression (the *s_exp*) is being evaluated the special variables this, value and previous are all bound:

- **this** The symbol whose value has changed.
- **value** The current value of this as a result of the change.
- **previous** The value of this immediately prior to the change.

Example

```
Gamma> b = 5;
5
Gamma> add_set_function(#b,#princ("changed\n"));
(princ "changed\n")
Gamma> b = 4;
changed
4
Gamma>
```

The following code automatically sounds an alarm whenever a computed_tank_level rises above 10000 and silences the alarm whenever it drops below 10000. The Cogent DataHub is automatically updated to maintain the same value as the Gamma task.

```
function send_to_datahub (point)
{
 write_point(point, value);
}

function check_alarm (value)
```

```
{
 if (value == 1)
  sound_alarm();
 else
  silence_alarm();

 send_to_datahub(this);
}

function check_tank_level (depth)
{
 if (depth > 10000)
  high_alarm = 1;
 else
  high_alarm = 0;

 send_to_datahub(this);
}

add_set_function(#high_alarm, #check_alarm(high_alarm));
add_set_function(#computed_tank_level,#check_tank_level(computed_tank_level));
```

See Also

when_set_fns, remove_set_function

flush_events

`flush_events` — handles all pending events, then exits.

Syntax

```
flush_events ()
```

Arguments

none

Returns

The result of executing all pending events, then exits.

Description

This function ensures that an appropriate event-handling function is called to handle all pending events from: a window system (where applicable), other tasks (interprocess communication messages), timers, or signals. Upon completion, `flush_events` causes the program to exit.

Example

```
Gamma> flush_events();

(the result of any pending events)

[/user/cogent/bin]$
```

See Also

`next_event`

next_event, next_event_nb

next_event, next_event_nb — wait for an event and call the event handling function.

Syntax

```
next_event ()
next_event_nb ()
```

Arguments

none

Returns

The result of executing the next event. If no event was processed, next_event_nb will return undefined, and next_event will not return.

Description

next_event blocks, waiting for an event from: a window system (where applicable), another task (an interprocess communication message), a timer, or a signal. An event handling function is automatically called if one has been defined for the event. The result of next_event is the result returned from the event handler, or nil if no event handler had been defined.

next_event_nb behaves exactly like next_event, except that next_event_nb (nb stands for non-blocking) returns immediately with undefined if no event is waiting to be processed.

Example

1. Here is the simplest use of next_event, causing Gamma to wait for and process the next event.

```
Gamma> while(t) next_event();
```

2. This program does basically the same thing, creating a main loop for program event processing, but it features error protection as well.

```
while (t)
{
 try
 {
  next_event();
 }
 catch
 {
  princ("last error: ", _last_error_," calling stack: ",
   stack(),"\n");
 }
}
```

remove_set_function

`remove_set_function` — removes a set function from a symbol.

Syntax

`remove_set_function (`*`symbol`*`, `*`s_exp`*`)`

Arguments

`symbol`

The symbol from which to remove the expression.

`s_exp`

An expression set for the *`symbol`*, such as that added by *`add_set_function`*.

Returns

The expression, in Lisp syntax, which was removed, or `nil` if none was removed.

Description

This function removes a set expression from the *`symbol`*. The *`s_exp`* is compared to all of the current expression set for the *`symbol`* using the comparison function `eq`.

Example

```
Gamma> b = 5;
5
Gamma>  add_set_function(#b,#princ("changed\n"));
(princ "changed\n")
Gamma> b = 4;
changed
4
Gamma> remove_set_function(#b,#princ("changed\n"));
(princ "changed\n")
Gamma> b = 3;
3
Gamma>
```

See Also

`add_set_function`, `when_set_fns`

when_set_fns

when_set_fns — returns all functions set for a symbol.

Syntax

when_set_fns (*symbol*)

Arguments

symbol

A symbol.

Returns

The expressions, in Lisp syntax, that have been set to be evaluated whenever the *symbol*'s value changes.

Example

```
Gamma> b = 5;
5
Gamma> add_set_function(#b,#princ("Changed.\n"));
(princ "Changed.\n")
Gamma> add_set_function(#b,#princ("Update now.\n"));
(princ "Update now.\n")
Gamma> b = 4;
Update now.
Changed.
4
Gamma> when_set_fns(#b);
((princ "Update now.\n") (princ "Changed.\n"))
Gamma>
```

See Also

add_set_function, remove_set_function

IX. Time, Date, and Timers

Table of Contents

after

`after` — a timer that initiates an action after a period of time.

Syntax

```
after (seconds, action...)
```

Arguments

`seconds`

A number of seconds, which may be fractional.

`action`

One or more statements to be executed. This argument is evaluated, so literal statements must be quoted.

Returns

An integer timer number which may be used as the argument to `cancel`.

Description

This function specifies an action to be performed after a given period of time in seconds has elapsed. The number of *`seconds`* may be specified to arbitrary precision, but will be limited in fact by the timer resolution of the operating system. In most cases this is practically limited to 20 milliseconds (0.05 seconds).

The timer functions `after`, `every` and `at` all cause an action to occur at the specified time, regardless of what is happening at that time, except if the timer expires during garbage collection. In this case, the timer will be serviced as soon as the garbage collection finishes.

For Gamma to notice a timer, you must make a call to `next_event`.

Example

```
Gamma> after(30, #princ("Time's up!\n"));
1
Gamma> next_event();

(30 seconds pass)

Time's up!
nil
Gamma>
```

See Also

`at`, `every`, `cancel`, `_timers_` in Predefined Symbosl

at

`at` — a timer that initiates an action at a given time, or regularly.

Syntax

`at (`*`day`*`, `*`month`*`, `*`year`*`, `*`hour`*`, `*`minute`*`, `*`second`*`, `*`actions...`*`)`

Arguments

`day`

Restriction on the day of the month (`1-31`), or `nil` for none.

`month`

Restriction on the month of the year (`1-12`), or `nil` for none.

`year`

Restriction on the year (`1994-2026`), or `nil` for none.

`hour`

Restriction on the hour of the day (`0-23`), or `nil` for none.

`minute`

Restriction on the minute in the hour (`0-59`), or `nil` for none.

`second`

Restriction on the second in the minute (`0-59`), or `nil` for none.

`actions`

The actions to perform when the specified time arrives.

Returns

An integer number which may be used as the argument to `cancel`.

Description

This function specifies an action to be performed at a given time, or to occur regularly at certain times of the *`minute`*, *`hour`*, *`day`*, *`month`* or *`year`*. A restriction on a particular attribute of the time will cause `at` to fire only if that restriction is true.

A restriction may be any number in the legal range of that attribute, or a list of numbers in that range. Illegal values for the time will be normalized. For example, a time specified as July 0, 1994 00:00:00 will be treated as June 30, 1994 00:00:00. If `nil` is specified for any attribute of the time, this implies no restriction and `at` will fire cyclically at every legal value for that attribute.

For Gamma to notice a timer, you must make a call to `next_event`. To notice repeating timers, the call to `next_event` can be used with a call to `while(t)`.

Example

```
//To print "hello" at 12:00 noon on June 2, 1994:
at(2,6,1994,12,0,0,#princ("hello\n"));

//To print "hello" at 12:00 noon on the first day
//of every month in 1994:
at(1,nil,1994,12,0,0,#princ("hello\n"));
```

```
//To print "hello" every half minute at 30 seconds
//and on the minute on the 1st and 15th of every month
//except July and August, for any year:
at(list(1,15), list(1,2,3,4,5,6,9,10,11,12),list(0,30), #(princ "hello\n"));

//To print "hello" every 10 seconds during the hour
//of 3:00pm every December 21st.
at(21,12,nil,15,nil,list(0,10,20,30,40,50), #princ("hello\n"));
```

See Also

after, every, _timers_ in Predefined Symbosl

block_timers, unblock_timers

block_timers, unblock_timers — block and unblock timer firing.

Syntax

```
block_timers ()
unblock_timers ()
```

Arguments

none

Returns

t

Description

Timers are potentially handled by a different mechanism from operating system signals. It may be desirable to block all timers from firing for the duration of an operation, which may not be possible using the block_signal mechanism. If a timer fires while timers are blocked, the timer function will be called as soon as timers are unblocked. This will not delay subsequent timers.

For example, if a timer is intended to fire every 5 seconds at 5, 10, ... seconds after the minute and the 5-second timer is blocked until second 7, the next timer in the sequence will still fire at 10 seconds. If the 5-second timer were blocked until second 27, then the 5, 10, 15, 20 and 25-second timers would *all* fire at second 27 and the next timer would fire at second 30. Code which blocks timers should be surrounded by a call to unwind_protect.

Example

```
Gamma> block_timers();
t
Gamma> protected_function();
<function return>
Gamma> unblock_timers();
t
```

See Also

block_signal, unblock_signal, after, at, every, _timers_ in Predefined Symbosl

cancel

`cancel` — removes a timer from the set of pending timers.

Syntax

```
cancel (timer_number)
```

Arguments

`timer_number`

An integer number returned from a call to `after`, `at` or `every`.

Returns

The complete timer definition for the canceled timer, or `nil` if no timer was canceled.

Description

Removes a timer from the set of pending timers based on its unique timer ID as returned by the function which created the timer. If no timer could be found with the corresponding timer number, nothing happens.

Example

To set a timer to repeat every 5 seconds, then stop it:

```
Gamma> every(5, #princ("hello\n"));
1
Gamma> cancel(1);
[945884155 683256506 5 ((princ "hello\n")) 1]
Gamma>
```

See Also

`_timers_` in Predefined Symbols

clock, nanoclock

clock, nanoclock — get the OS time.

Syntax

```
clock ()
nanoclock ()
```

Arguments

none

Returns

The current clock value in seconds from the operating system as a long integer. nanoclock includes the nanoseconds as well.

Description

This function gets the operating system clock setting in seconds. The time is usually expressed as the number of seconds from midnight January 1, 1970 on UNIX systems, though it may differ across implementations.

Example

```
Gamma> clock();
999810273
Gamma> nanoclock();
999810273.66378700733
Gamma>
```

See Also

date, date_of

date

`date` — gets the OS date and time; translates seconds into dates.

Syntax

```
date (seconds?, is_utc?)
```

Arguments

`seconds`

A number of seconds, such as returned from a call to `clock`.

`is_utc`

A value of `t` puts the date in Coordinated Universal Time (formerly known as Greenwich Mean Time, GMT).

Returns

The date as a character string.

Description

This function returns a character string which represents the current date and time in human-readable form. This form depends on the operating system, but will look like "Sat Mar 21 15:58:27 2000" on most UNIX systems. The *`seconds`* parameter returns the date that corresponds to the number of seconds since Jan 1, 1970, Coordinated Universal Time.

Example

```
Gamma> date();
"Fri Mar 31 09:18:27 2000"
Gamma> date(987654321);
"Thu Apr 19 00:25:21 2001"
Gamma> date(987654321,t);
"Thu Apr 19 04:25:21 2001"
Gamma> date(0,t);
"Thu Jan  1 00:00:00 1970"
Gamma>
```

See Also

`clock`

date_of

`date_of` — is obsolete, see `date`

Syntax

```
date_of (seconds)
```

Arguments

seconds

A system time as a long integer, which may be obtained from the `clock` function.

Returns

The date as a character string.

Description

This function has been superceded by `date`. It returns a character string which represents the given date and time in human-readable form. This form depends on the operating system, but will look like "`Fri Feb 16 21:50:32 1973`" on most UNIX systems.

Example

```
Gamma> date_of(987654321);
"Thu Apr 19 00:25:21 2001"
```

See Also

`clock`, `date`

every

`every` — a timer that initiates an action every number of seconds.

Syntax

```
every (seconds, action...)
```

Arguments

`seconds`

The number of seconds. This may be fractional. Realistically the operating system will not be able to keep up with numbers below about 0.05. This will differ from machine to machine.

`action`

The actions to perform continuously every given number of seconds.

Returns

An integer number which may be used as the argument to `cancel`.

Description

This function specifies an action to be performed every time the number of *`seconds`* elapses. The return value is a unique timer number which may be used to cancel the *`action`* prior to the time expiring by calling `cancel`. The number of seconds may be specified to arbitrary precision, but will be limited in fact by the timer resolution of the operating system. In most cases this is practically limited to 20 milliseconds (0.05 seconds).

The timer functions `after`, `every` and `at` all cause the action to occur at the specified time, regardless of what is happening at that time, except if the timer expires during garbage collection. In this case, the timer will be serviced as soon as the garbage collection finishes.

For Gamma to notice a timer, you must make a call to `next_event`. To notice repeating timers, the call to `next_event` can be used with a call to `while(t)`.

Example

Print hello every 5 seconds.

```
Gamma> every(5, #princ("Hello\n"));
1
Gamma> while(t) next_event();
Hello
Hello
Hello
...
```

See Also

`after`, `at`, `_timers_` in Predefined Symbosl

gmtime

`gmtime` — transforms Unix time to UTC time and date in ASCII format.

Syntax

```
gmtime (time_t)
```

Arguments

`time_t`

The time, usually expressed as the number of seconds from midnight January 1, 1970 on UNIX systems, though it may differ across implementations.

Returns

A instance of the class `tm`, whose members are as follows:

.sec

The number of seconds after the minute (`0` - `59`).

.min

The number of minutes after the hour (`0` - `59`).

.hour

The number of hours past midnight (`0` - `23`).

.mday

The day of the month (`1` - `31`).

.mon

The number of months since January (`0` - `11`)

.year

The number of years since 1900.

.wday

The number of days since Sunday (`0` - `6`).

.yday

The number of days since January 1 (`0` - `365`)

.isdst

`1` if daylight saving time is in effect, `0` if not, and a negative number ifthe information is not available.

Example

```
    Gamma> pretty_princ ("UTC breakout:\t", gmtime (1149261975.5000002), "\n");
    UTC breakout: {tm (hour . 15) (isdst . 0) (mday . 2) (min . 26)
    (mon . 5) (sec . 15) (wday . 5) (yday . 152) (year . 106)}
t
    Gamma>
```

See Also

`gmtime`, `mktime`

localtime

`localtime` — transforms Unix time to local time and date in ASCII format.

Syntax

```
localtime (time_t)
```

Arguments

time_t

The time, usually expressed as the number of seconds from midnight January 1, 1970 on UNIX systems, though it may differ across implementations.

Returns

An instance of the class `tm`, whose members are as follows:

.sec

The number of seconds after the minute (`0` - `59`).

.min

The number of minutes after the hour (`0` - `59`).

.hour

The number of hours past midnight (`0` - `23`).

.mday

The day of the month (`1` - `31`).

.mon

The number of months since January (`0` - `11`)

.year

The number of years since 1900.

.wday

The number of days since Sunday (`0` - `6`).

.yday

The number of days since January 1 (`0` - `365`)

.isdst

`1` if daylight saving time is in effect, `0` if not, and a negative number ifthe information is not available.

Example

```
    Gamma> pretty_princ ("Local breakout:\t", localtime (1149261975.5000002), "\n");
    Local breakout: {tm (hour . 11) (isdst . 1) (mday . 2) (min . 26)
    (mon . 5) (sec . 15) (wday . 5) (yday . 152) (year . 106)}
t
    Gamma>
```

See Also

gmtime, mktime

mktime

`mktime` — converts the ASCII date and time data in a `tm` class to Unix time.

Syntax

```
mktime (time_t)
```

Arguments

`tm`

A `tm` class, as created by `localtime` or `gmtime`

Returns

The time, usually expressed as the number of seconds from midnight January 1, 1970 on UNIX systems, though it may differ across implementations.

Example

```
    Gamma> princ ("Local breakout to Unix:\t", mktime (localtime(1149261975.5000002)), "\n");
    Local breakout to Unix: 1149261975
t
    Gamma>
```

See Also

`gmtime`, `localtime`

timer_is_proxy

`timer_is_proxy` — controls timer handling in Gamma.

Syntax

```
timer_is_proxy (t_or_nil);
```

Arguments

`t_or_nil`

An expression that evaluates to `t` or `nil`.

Returns

The passed argument (`t` or `nil`).

Description

This function controls how timers are fundamentally handled within Gamma. By default, timers are handled by the processing of proxies which allows Gamma to delay the timer, if necessary, if a critical system process is occurring.

Calling `timer_is_proxy` with `nil` makes all timers operate by using signals. In the QNX 4 operating system SIGUSER1 (SIGALRM?) is used, and the attach code is run as a handled signal.

Running timers via signals has some very dramatic consequences. When running in this mode ALL TIMER CODE MUST BE SIGNAL SAFE.

Example

```
Gamma> timer_is_proxy (nil);
nil
Gamma> timer_is_proxy (t);
t
Gamma>
```

See Also

`every`, `at`, `after`, `block_timers`, `nunblock_timers`, `cancel`

X. Cogent DataHub

Table of Contents

add_exception_function, add_echo_function

add_exception_function, add_echo_function — assign functions for exceptions or echoes on a point.

Syntax

```
add_exception_function (symbol, s_exp);
add_echo_function (symbol, s_exp);
```

Arguments

symbol

A point name, as a symbol.

s_exp

Any Gamma or Lisp expression.

Returns

t

Description

When a Gamma or Lisp program is run in conjunction with the Cogent DataHub, process points may change at any time, causing point change events to occur. A point change event is referred to as an exception. It is possible to bind any Gamma or Lisp expression to a symbol to be evaluated when an exception occurs. If a program can both write a point on the DataHub and react to exceptions on that point, it is possible that the DataHub will "echo" a point written by the program itself. If this is not handled, an infinite loop between the program and the DataHub could occur. The DataHub tags point echoes so that a different function can be called in the program when that echo arrives back at its origin. Only the originating task will see a point exception as an echo. All other tasks will see a normal exception.

When an exception handler (the *s_exp* argument) is being evaluated the special variables this, value and previous are all bound:

- **this** The symbol which received the exception.
- **value** The current value of this as a result of the exception.
- **previous** The value of this immediately prior to the exception.

Example

```
Gamma> add_exception_function(#temp, #princ("temp change\n"));
(princ "temp change\n")
Gamma> add_echo_function(#temp,nil);
nil
Gamma> next_event();
temp change
(t)
Gamma> read_point(#temp);
30
Gamma> write_point(#temp,25);
t
Gamma> next_event();
```

```
(nil)
Gamma>
```

See Also

register_point, when_echo_fns, when_exception_fns, remove_echo_function, remove_exception_function

lock_point

`lock_point` — locks or unlocks points.

Syntax

```
lock_point (symbol, locked)
```

Arguments

`symbol`

A point name, as a symbol.

`locked`

`t` to set a lock, `nil` to release a lock.

Returns

`t` if the function is successful, otherwise `nil`.

Description

This function locks or unlocks a point in the Cogent DataHub. The current security level must be greater than or equal to the security level on the point.

Example

```
Gamma> init_ipc("locker","lq");
t
Gamma> write_point(#a,5);
t
Gamma> lock_point(#a,t);
t
Gamma> write_point(#a,300);
nil
Gamma> lock_point(#a,nil);
t
Gamma> write_point(#a,300);
t
Gamma>
```

See Also

`set_security`, `point_locked`

point_locked

`point_locked` — indicates if a point is locked.

Syntax

```
point_locked (symbol)
```

Arguments

`symbol`

A point name, as a symbol.

Returns

`t` if locked, or `nil` if not locked.

Example

```
Gamma> lock_point(#f,t);
t
Gamma> next_event();
nil
Gamma> point_locked(#f);
t
Gamma> lock_point(#f,nil);
t
Gamma> next_event();
nil
Gamma> point_locked(#f);
nil
Gamma>
```

See Also

`lock_point`

point_nanoseconds

`point_nanoseconds` — gives the nanoseconds from `point_seconds` that a point value changed.

Syntax

```
point_nanoseconds (symbol)
```

Arguments

`symbol`

A point name, as a symbol.

Returns

A number of nanoseconds.

Description

This function returns the number of nanoseconds after `point_seconds` that a given point's value changed.

Example

```
Gamma> clock();
938631678
Gamma> write_point(#l,44);
t
Gamma> next_event();
nil
Gamma> point_seconds(#l);
938631693
Gamma> point_nanoseconds(#l);
735100000
Gamma>
```

See Also

`point_seconds`

point_seconds

`point_seconds` — gives the time the point value changed.

Syntax

`point_seconds (`*`symbol`*`)`

Arguments

`symbol`

A point name, as a symbol.

Returns

A time in seconds.

Description

This function returns the time in seconds when a given point's value changed.

Example

```
Gamma> clock();
938631678
Gamma> write_point(#l,44);
t
Gamma> next_event();
nil
Gamma> point_seconds(#l);
938631693
Gamma>
```

See Also

`point_nanoseconds`

point_security

`point_security` — gives the security level of a point.

Syntax

`point_security (`*`symbol`*`)`

Arguments

`symbol`

A point name, as a symbol.

Returns

The security level.

Example

```
Gamma> set_security(5);
0
Gamma> secure_point(#f,3);
t
Gamma> point_security(#f);
3
Gamma>
```

See Also

`secure_point`, `set_security`,

read_existing_point, read_point

read_existing_point, read_point — retrieve points.

Syntax

```
read_existing_point (symbol)
read_point (symbol)
```

Arguments

symbol

A point name, as a symbol.

Returns

The value of a point in the Cogent DataHub. If the point is unavailable then nil is returned. For read_existing_point, if the point does not exist than nil is returned. For read_point, if the point does not exist then the point is created and a default value is returned.

Description

These functions makes a call to the DataHub to retrieve the point whose name is the same as the *symbol*. If the point does not exist in the DataHub, read_existing_point returns nil and does not create the point. read_point will create a point in the DataHub if necessary, whose value and confidence are both zero. If the point name is pre-qualified with a domain name and a colon (:), this function will search that domain's data rather than the DataHub for the default domain.

Example

This example uses data points entered in the write_point reference entry example.

```
Gamma> init_ipc("reader","rq");
t
Gamma> read_point(#my);
600
Gamma> read_point(#dog);
130
Gamma> read_point(#has);
140
Gamma> read_point(#fleas);
150
Gamma> read_existing_point(#cat);
nil
Gamma> read_point(#cat);
0
Gamma>
```

See Also

register_point, write_point

register_all_points

`register_all_points` — registers an application to receive exceptions for all points.

Syntax

```
register_all_points (domain?, newflag?)
```

Arguments

`domain`

The Cogent DataHub domain in which to register.

`newflag`

A flag determining whether to automatically register all future points from the DataHub.

Returns

`t` on success, or `nil` on failure.

Description

This function registers the current application to receive exceptions from the Cogent DataHub for all points in the given domain. Once this function has been called, any changes to the value of any point in the DataHub will be transmitted to the input queue of the application. These changes are events, and as such must be processed by calling `next_event` or `next_event_nb` before the application will recognize the new value of the point.

If the domain is `nil`, then the current default domain (set by `set_domain`) will be used. If the domain is named, even if it is the default domain, then the DataHub will transmit all points as fully qualified names, in the domain:name format. If the *`newflag`* is given and is non-`nil`, then any points which are created on the DataHub after this call is made will be automatically registered. If *`newflag`* is `nil` or not provided, then the DataHub will not automatically register points which were created since this call.

Example

```
Gamma> register_all_points(nil,t);
t
Gamma> write_point(#b,22);
t
Gamma> next_event();
nil
Gamma> b;
22
Gamma> register_all_points("plant",t);
t
Gamma>
```

See Also

`register_point`

register_exception

`register_exception` — not yet documented.

Syntax

```
register_exception (symbol, s_exp, execute_p?)
```

Arguments

symbol

s_exp

execute_p

Returns

Description

Example

See Also

register_point, register_existing_point

register_point, register_existing_point — register an application to receive exceptions for a single point.

Syntax

```
register_point (symbol)
register_existing_point (symbol)
```

Arguments

symbol

A point name, as a symbol.

Returns

The current value of the point in the Cogent DataHub. If the point does not exist, register_point will create the point in the DataHub and return a default value. However register_existing_point will return nil if the point does not exist.

Description

These functions register an application to receive changes in the value of a point whenever they occur. The current value of the point is returned as a result of the registration. Once this function has been called, any changes to the value of the point in the DataHub will be transmitted to the input queue of the application. These changes are events, and as such must be processed by calling next_event or next_event_nb before the application will recognize the new value of the point.

A function may be attached to the value change event using the when_exception and when_echo functions. Regardless of whether an event is attached to the point, the interpreter will update the value of the symbol whose name is the same as the point name. This means that once a point has been registered its value will always be current in the global scope of the interpreter. If the point name is pre-qualified with a domain name and a colon (:), this function will search that domain's data rather than the DataHub for the default domain.

Example

```
Gamma> register_point(#f);
26
Gamma> write_point(#f,85);
t
Gamma> f;
26
Gamma> next_event();
nil
Gamma> f;
85
Gamma> register_existing_point(#newpoint);
nil
Gamma> register_point("newpoint");
0
Gamma> register_point("acme:newpoint");
0
```

See Also

`init_ipc`, `next_event`, `next_event_nb`, `when_echo`, `when_exception`

remove_echo_function

`remove_echo_function` — removes an echo function from a symbol.

Syntax

`remove_echo_function (`*`symbol`*`, `*`echo_fn`*`)`

Arguments

`symbol`

The point name, as a symbol, from which to remove the echo function.

`echo_fn`

The echo function body.

Returns

The echo function which was removed, or `nil` if no function was removed.

Description

This function removes an echo function (Cogent DataHub echo handler) from the *`symbol`*. The *`echo_fn`* is compared to all of the current echo functions for the *`symbol`* using the comparison function `eq`.

Example

```
Gamma> add_echo_function(#temp,#princ("echo\n"));
(princ "echo\n")
Gamma> write_point(#temp,28);
t
Gamma> next_event();
echo
(t t)
Gamma> remove_echo_function(#temp,#princ("echo\n"));
(princ "echo\n")
Gamma> write_point(#temp,32);
t
Gamma> next_event();
(t)
Gamma>
```

See Also

`add_echo_function`

remove_exception_function

`remove_exception_function` — removes an exception function from a symbol.

Syntax

```
remove_exception_function (symbol, exc_fn)
```

Arguments

`symbol`

The point name, as a symbol, from which to remove the exception function.

`exc_fn`

The exception function body.

Returns

The exception function which was removed, or `nil` if no function was removed.

Description

This function removes an exception function (Cogent DataHub exception handler) from the *`symbol`*. The *`exc_fn`* is compared to all of the current exception functions for the *`symbol`* using the comparison function `eq`.

Example

```
Gamma> add_exception_function(#temp, #princ("temp change\n"));
(princ "temp change\n")
Gamma> next_event();
temp change
(t)
Gamma> temp;
40
Gamma> remove_exception_function(#temp, #princ("temp change\n"));
(princ "temp change\n")
Gamma>  next_event();
nil
Gamma> temp;
35
Gamma>
```

See Also

`add_exception_function`

secure_point

`secure_point` — alters the security level on a point.

Syntax

```
secure_point (symbol, security)
```

Arguments

`symbol`

The point to alter, as a symbol.

`security`

The new security level for this point.

Returns

`t` on success, or `nil` if an error occurred.

Description

This function alters the security level on a point in the Cogent DataHub. If the current process security level is lower than the named point (*`symbol`*), then the function returns `nil`, otherwise it returns `t`. The initial security level for a process is `0`.

Example

```
Gamma> init_ipc("spt","spq");
t
Gamma> secure_point(#d,5);
nil
Gamma> set_security(9);
0
Gamma> secure_point(#d,5);
t
Gamma> secure_point(#d,12);
nil
Gamma> set_security(15);
9
Gamma> secure_point(#d,12);
t
Gamma>
```

See Also

`point_security`, `set_security`

set_domain

`set_domain` — sets the default domain for future calls.

Syntax

```
set_domain (domain_name)
```

Arguments

`domain_name`

A string.

Returns

The *`domain_name`* argument.

Description

This function sets the default Cogent DataHub domain for all future calls to `read_point`, `read_existing_point`, `register_point`, `register_existing_point`, `write_point` and `write_existing_point`. The default domain can be overridden by explicitly placing the domain name at the beginning of the point name, separated by a colon (:). For example, a variable named `tank_level` in the default domain would have to be named `acme:tank_level` in the "acme" domain.

There is a possibility of aliasing points. If the default domain is "acme" then the point `tank_level` and the point `acme:tank_level` refer to the same DataHub point. If both of these names are used in a Gamma program then one of them will not behave correctly. It is the responsibility of the programmer to ensure that there is no aliasing in the assigned names, either by always explicitly naming a point's domain or by programming carefully.

Example

```
Gamma> set_domain("acme");
t
Gamma> read_point(#tank_level);
12.5
Gamma> set_domain("steamplant");
t
Gamma> read_point("tank_level");
4.35
Gamma> read_point("acme:tank_level");
12.5
```

See Also

`read_point`, `read_existing_point`, `register_point`, `write_point`, `write_existing_point`

set_security

`set_security` — changes the security level for the current process.

Syntax

`set_security (`*`security_level`*`)`

Arguments

`security_level`

The new security level for this process.

Returns

The previous security level for this process.

Description

This function changes the security level for the current process to the given value. There is no restriction on the security level argument. A low-security process can alter its own security level to be higher.

If it is necessary to have a process's security level to be unalterable, then the `set_security` function can be re-bound after the security level is originally set (see second example). The only use of security level is in conjunction with the Cogent DataHub.

Example

```
Gamma> init_ipc("spt","spq");
t
Gamma> secure_point(#d,5);
nil
Gamma> set_security(9);
0
Gamma> secure_point(#d,5);
t
Gamma> secure_point(#d,12);
nil
Gamma> set_security(15);
9
Gamma> secure_point(#d,12);
t
Gamma>
```

The example below sets the current process's security to 5, and then re-binds `set_security` so that the program can no longer alter its security. The `list` function is used in the re-binding, as it will accept any number of arguments without error, and will have no side-effects.

```
Gamma> set_security(5);
0
Gamma> set_security = list;
(defun list (&optional &rest s_exp...) ...)
Gamma> set_security(9);
(9)
Gamma> secure_point(#g,10);
nil
Gamma> secure_point(#g,6);
nil
Gamma> secure_point(#g,4);
t
Gamma>
```

See Also

`point_security`, `secure_point`

unregister_point

`unregister_point` — stops echo and exception message sending.

Syntax

```
unregister_point (symbol)
```

Arguments

`symbol`

The point to unregister.

Returns

`t` on success, or `nil` on failure.

Description

This function causes the Cogent DataHub to immediately stop sending echo and exception messages for the named point. It is possible that exceptions and echos which are queued to the task will arrive after this function is called, but the DataHub will not generate any new messages.

Example

```
Gamma> register_all_points(nil,t);
t
Gamma> b;
55
Gamma> unregister_point(#b);
t
Gamma> write_point(#b,33);
t
Gamma> write_point(#c,77);
t
Gamma> next_event();
nil
Gamma> b;
55
Gamma> c;
77
Gamma>
```

See Also

`register_point`, `register_all_points`

when_echo_fns, when_exception_fns

when_echo_fns, when_exception_fns — indicate the functions for echos or exceptions on a point.

Syntax

```
when_echo_fns (symbol)
when_exception_fns (symbol)
```

Arguments

symbol

A point name, as a symbol.

Returns

A list of expressions to be evaluated when an echo or exception occurs on the *symbol*.

Example

```
    Gamma> add_echo_function(#temp,#princ("echo\n"));
    (princ "echo\n")
    Gamma> add_echo_function(#temp,#temp/2);
    (/ temp 2)
    Gamma> when_echo_fns(#temp);
    ((/ temp 2) (princ "echo\n") t)
    Gamma> write_point(#temp,22);
    t
    Gamma> next_event();
    echo
(11 t t)
    Gamma>
```

See Also

add_echo_function, add_exception_function, remove_echo_function, remove_exception_function

write_existing_point, write_point

write_existing_point, write_point — write point values.

Syntax

```
write_existing_point (symbol, value, seconds?, nanoseconds?)
write_point (symbol, value, seconds?, nanoseconds?)
```

Arguments

symbol

A point name, as a symbol.

value

Any numeric or string expression.

seconds

Number of seconds since Jan 1st, 1970.

nanoseconds

Number of nanoseconds within the second.

Returns

t on success or nil on failure.

Description

These functions write a point value to the Cogent DataHub. If the point does not exist in the DataHub, write_point will create the point and set its value. write_existing_point will return nil if the point does not exist.

Example

```
Gamma> init_ipc("writer","wq");
t
Gamma> write_existing_point("my",150);
nil
Gamma> write_point(#my,120);
t
Gamma> write_point(#dog,130,450000000);
t
Gamma> write_point("has",140);
t
Gamma> write_point("fleas",150,1210947,2134444);
t
Gamma> write_existing_point("my",600);
t
Gamma>
```

These points can be viewed in the DataHub (sorted in alphabetical order) using the **dhview** command at the shell prompt:

```
#     Point Name           Conf Value
1     dog                  100  130
2     fleas                100  150
3     has                  100  140
4     my                   100  600
```

See Also

read_point, read_existing_point, register_point

XI. QNX 4

Table of Contents

dev_read

`dev_read` — is a modification of QNX 4 `dev_read`.

Syntax

```
dev_read (devno, nchars, min_chars, time, timeout)
```

Arguments

devno

A device id number returned from dev_open.

nchars

The number of characters to read.

min_chars

The minimum number of characters to read.

time

The inter-character time limit.

timeout

A timeout value for the entire read.

Returns

A buffer containing the characters which were successfully returned from this call, or `nil` if an error occurred.

Description

This function is a modification of the QNX 4 `dev_read` function. It is currently only available in QNX 4.

This function does not support the proxy and armed arguments to the C `dev_read` function. The characters read from the device are returned in a buffer rather than filled in to a buf argument as with the C function. Otherwise, the function of this call is identical to the QNX 4 `dev_read` function.

The min, time, and timeout values are used as follows:

Table 1. `dev_read` min, time, and timeout values

min	time	timeout	Description
0	0	0	Returns immediately with as many bytes as are currently available (up to nchars bytes).
M	0	0	Return with up to nchars bytes only when at least M bytes are available.
0	T	0	Return with up to nchars bytes when at least one byte is available, or T * 0.1 sec has expired.
M	T	0	Return with up to nchars bytes when either M bytes are available or at least one bytes has been received and the inter-byte time between any subsequently received characters exceeds T * 0.1 sec.
0	0	t	Reserved.

min	time	timeout	Description
M	0	t	Return with up to nchars bytes when t * 0.1 sec has expired, or M bytes are available.
0	T	t	Reserved.
M	T	t	Return with up to nchars when M bytes are available, or t * 0.1 sec has expired and no characters are received, or at least one byte has been received and the inter-byte time between any subsequently received characters exceeds T * 0.1 sec.
0	T	t	Reserved.
M	T	t	Return with up to nchars when M bytes are available, or t * 0.1 sec has expired and no characters are received, or at least one byte has been received and the inter-byte time between any subsequently received characters exceeds T * 0.1 sec.

If an error occurs the errno is set. The following error constants are relevant:

- **EAGAIN** The O_NONBLOCK flag is set on the devno.
- **EBADF** The devno argument is invalid or not open for read
- **EINTR** The function was interrupted by a signal
- **EIO** The process cannot read data from the devno
- **ENOSYS** This function is not supported for the given devno

Example

```
Gamma> id = dev_open("/dev/ser1",O_RDWR);
4
Gamma> x = dev_read(id,5,1,0,100);
#[104 101 108 108 111]
Gamma> buffer_to_string(x);
"hello"
```

See Also

fd_open, fd_close, ser_setup

dev_setup

dev_setup — is obsolete, see ser_setup.

Syntax

```
dev_setup (devno, baud, bits/char, parity, stopbits, min, time)
```

Description

This function has been renamed ser_setup, and is documented in detail on that page.

`inp`, `inpw`

`inp, inpw` — query hardware ports.

Syntax

```
inp (port)
inpw (port)
```

Arguments

`port`

A hardware port address.

Returns

The byte (`inp`) or word (`inpw`) located at the hardware port address.

Description

These functions query a hardware port. In order for these functions to succeed, the interpreter must be running with root permissions and a privity of 1, otherwise these functions may cause a segmentation fault.

The `inp` function reads a single byte from the specified port location. The `inpw` function reads a word (two bytes) from the specified port location.

Example

Examples of this function are beyond the scope of this documentation.

See Also

`outp`, `outpw`

mmap

`mmap` — implements the C `mmap` function call.

Syntax

```
mmap (size, prot_flags, share_flags, shm_file, offset)
```

Arguments

`size`

The size (in bytes) of the shared memory segment to map.

`prot_flags`

Access capability flags.

`share_flags`

Sharing flags.

`shm_file`

A file descriptor as returned by shm_open.

`offset`

An offset from the beginning of the shared memory area.

Returns

A buffer which is mapped to the shared memory region, or `nil` on failure with `errno` set.

Description

This function is currently only available in QNX 4. It implements the C `mmap` function call, returning a buffer which maps to the shared memory region. The `shm_open` function needs to be called before using this function.

The *`prot_flags`* specifies the access capability. Valid *`prot_flags`* are bitwise OR-ed combinations of:

- **PROT_EXEC** The region can be executed.
- **PROT_NOCACHE** Disable caching of the region (use for dual port RAM).
- **PROT_NONE** The region cannot be accessed.
- **PROT_READ** The region can be read.
- **PROT_WRITE** The region can be written.

The *`share_flags`* specify the handling of the memory region. Valid *`share_flags`* are bitwise OR-ed combinations of:

- **MAP_PRIVATE** Changes are private.
- **MAP_SHARED** Share changes.

Remember to require the "const/mman.lsp" file before using these constants.

Possible errors when using this function are:

- **EACCES** The shm_file is not open for the correct mode
- **EAGAIN** The mapping could not be locked in memory due to a lack of resources
- **EBADF** The passed shm_file is bad.
- **EINVAL** The *`prot_flags`* of *`share_flags`* argument is invalid
- **ENODEV** The shm_file arg refers to an argument for which mmap is meaningless
- **ENOMEM** The mapping could not be locked because it would require more space than the system is able to supply
- **ENOSYS** The function mmap is not supported by this implementation
- **ENOTSUP** MAP_PRIVATE was specified but the implementation does not support this functionality
- **ENXIO** The offset or size arguments are invalid.

Example

```
//This code maps the first 1000 bytes from video
//memory (0xA0000) into a buffer named buf.

require_lisp("const/filesys");
require_lisp("const/mman");
fd = shm_open("Physical",O_RDONLY, 0o777);
buf = mmap(1000, PROT_READ , MAP_SHARED, fd, 0xA0000);
```

See Also

`shm_open`

`outp`, `outpw`

`outp, outpw` — write values to hardware ports.

Syntax

```
outp (port, value)
outpw (port, value)
```

Arguments

`port`

A hardware port address.

`value`

A value to write to the port.

Returns

The value.

Description

These functions write a value to a hardware port. In order for these functions to succeed, the interpreter must be running with root permissions and a privity of 1, otherwise these functions may cause a segmentation fault.

The `outp` function writes a single byte to the specified *`port`* location. The `outpw` function writes a word (two bytes) to the specified *`port`* location.

Example

Examples of this function are beyond the scope of this documentation.

See Also

`inp`, `inpw`

qnx_name_attach

`qnx_name_attach` — registers a local or global name.

Syntax

```
qnx_name_attach (node, name)
```

Arguments

node

The node (0 is local node).

name

The name to attach.

Returns

A name id, useful when calling the `qnx_name_detach` function. On failure this function returns -1 and sets errno.

Description

This function registers a local or global name. Names can be up to 32 characters long. If a name starts with a slash '/' then it is considered to be a global name. Global names are registered with optionally run global process name servers (nameloc).

Errors are:

EAGAIN Name space used up on this node EBUSY Specified name already exists on that node

Example

Examples of this function are beyond the scope of this documentation.

See Also

`qnx_name_detach`, `qnx_name_locate`, ProfilingandDebugging

qnx_name_detach

qnx_name_detach — detaches a name.

Syntax

```
qnx_name_detach (node, name_id)
```

Arguments

node

The node.

name_id

The name id returned by qnx_name_attach.

Returns

true if able to detach, else nil, and errno is set.

Description

Detaches a name attached by the qnx_name_attach function.

Errno can be set to :

- **EINVAL** The name does not exist, or if it does, you do not own it

Example

Examples of this function are beyond the scope of this documentation.

See Also

qnx_name_attach, qnx_name_locate

qnx_name_locate

`qnx_name_locate` — is an implementation of the C function `qnx_name_locate`.

Syntax

```
qnx_name_locate (node, name, size)
```

Arguments

node

The node on which to locate the name, or 0 for global name service.

name

The name to locate.

size

The initial size of a virtual circuit buffer, if necessary.

Returns

A list of the form `(taskid . number_of_instances)`, or `nil` if no name was located.

Description

This is an implementation of the `qnx_name_locate` C function. The only difference is that the return value contains both the located task ID and the number of copies of the name found. The task ID is the car of the return value, and the number of copies is the cdr of the return value.

Example

```
Gamma> qnx_name_locate(0,"qnx/pipe",0);
(35 . 1)
```

See Also

`qnx_name_attach`, `qnx_name_detach`

qnx_osinfo

`qnx_osinfo` — returns a class very similar to QNX 4 `struct_osinfo`.

Syntax

`qnx_osinfo (`*`node`*`)`

Arguments

node

The node to query.

Returns

An instance of the class `Osinfo`, or `nil` on failure.

Description

This function returns a class very similar to the QNX 4 `struct _osinfo`. It is currently only available in QNX 4. The instance variables of this class are:

- **bootsrc** a single integer value that is the ASCII value for the character 'F' for floppy, 'H' for hard, or 'N' for network. Use char() function to convert.
- **cpu** processor type, typically given as the last 3 digits (386,486,586,686,etc.)
- **cpu_speed** a measure of the CPU speed using a 16-bit algorithm. A classic PC/XT is 96. A 133MHz Pentium is approx. 18000-19000.
- **fpu** the floating point processor type given as the last 3 digits of the chip (387,487,587,687,etc.)
- **freememk** free memory in the system
- **machine** machine name
- **max_nodes** the number of nodes you are licensed for
- **nodename** logical node numberof this cpu
- **num_handlers** maximum number of interrupt handlers
- **num_names** maximum number of names
- **num_procs** mamimum number of processes (programs + virtual curcuits + proxies)
- **num_sessions** maximum number of sessions
- **num_timers** maximum number of timers
- **pidmask** process ID bit mask
- **release** a single number that is the ASCII equivalent of the release letter. Use char() function to convert.
- **reserve64k** Relocation offset
- **sflags** System flags, see below
- **tick_size** tick size, in microseconds
- **timesel** segment in which the time is kept

- **totmemk** total memory (kB) in the system
- **version** the QNX 4 version number * 100

- **The following system flags are available:** nil
- **_PSF_PROTECTED** The system is running in protected mode
- **_PSF_NDP_INSTALLED** An 80x87 numeric processor is installed
- **_PSF_EMULATOR_INSTALLED** An 80x87 emulator is running
- **_PSF_EMU16_INSTALLED** The 16-bit 80x87 emulator is running
- **_PSF_EMU32_INSTALLED** The 32-bit 80x87 emulator is running
- **_PSF_APM_INSTALLED** Advanced Power Management is running in the BIOS of this system
- **_PSF_32BIT_KERNEL** This system is running with a 32 bit kernel
- **_PSF_PCI_BIOS** This system has a PCI BIOS
- **_PSF_32BIT** Proc32 is running
- **_PSF_RESERVE_DOS** The lower 640k is reserved for DOS
- **_PSF_RESERVE_DOS32** ???

Example

```
Gamma> sin = qnx_osinfo(0);
<instance of Osinfo class>
Gamma> char(sin.bootsrc);
"H"
Gamma> sin.cpu;
586
Gamma> sin.cpu_speed;
18883
Gamma> sin.freememk;
18260
Gamma> sin.machine;
"PCI"
Gamma> sin.max_nodes;
34
Gamma> sin.nodename;
2
Gamma> sin.num_handlers;
64
Gamma> sin.num_names;
100
Gamma> sin.num_procs;
500
Gamma> sin.num_sessions;
64
Gamma> sin.num_timers;
125
Gamma> sin.pidmask;
511
Gamma> char(sin.release);
"G"
Gamma> sin.reserve64k;
0
Gamma> sin.ticksize;
9999
Gamma> sin.totmemk;
32384
Gamma> sin.version / 100;
4.23
//These examples use loaded constants.  Any return value greater
```

```
//than zero indicates a positive result.
require_lisp("const/QNXOS");
Gamma> sin.sflags & _PSF_PROTECTED;
1
Gamma> sin.sflags & _PSF_NDP_INSTALLED;
2
Gamma> sin.sflags & _PSF_EMULATOR_INSTALLED
0
Gamma> sin.sflags & _PSF_RESERVE_DOS
0
```

See Also

qnx_osstat

qnx_osstat

`qnx_osstat` — lists processor loads and number of READY processes at each priority level.

Syntax

`qnx_osstat (`*`node`*`)`

Arguments

node

The node for which to collect statistics. Zero indicates the current node.

Returns

A list of two arrays of 32 elements.

Description

This function is currently only available in QNX 4. It returns a list of two arrays of 32 elements. The first array contains the average processor load at each priority level. The number are relative to one another and have a sum of 1. The second array contains the number of processes in the READY state at each priority level.

Example

```
info = qnx_osstat(0);
load = car(info);
for(i=0;i<32;i++)
{
   princ(format("%2d. %2f\n",i,aload[i] * 100));
}
```

See Also

`qnx_osinfo`

qnx_proxy_attach

`qnx_proxy_attach` — creates a proxy message for a process.

Syntax

`qnx_proxy_attach (`*`pid`*`, `*`command`*`, `*`priority`*`? )`

Arguments

`pid`

The process ID of the task to receive the message.

`command`

The message code.

`priority`

The priority of the message (optional).

Returns

The process ID of the proxy, else -1.

Description

This function creates a message proxy that will deliver a standard message to a task whenever it is called. The proxy will accept any message from any other task, but will always send it's one message to the task identified with *`pid`*. The sending process will not block, and its message data will be discarded by the proxy.

Proxies are used to wake receive-blocked processes that are waiting for messages. They can also be used to send non-blocking messages between processes, but `send_async` is probably more useful for this. Proxy messages are queued, so if a proxy is triggered 95 times, its receiving task will get 95 identical messages. A proxy may have at most 65535 messages pending.

In case of error, the following errnos are possible:

- **EAGAIN** No process entries free to make a proxy.
- **EINVAL** The proxy message exceeds the maximum.
- **ENOMEM** The process manager doesn't have enough memory to hold the message.
- **ESRCH** The process ID (*`pid`*) does not exist.

Example

Examples of this function are beyond the scope of this documentation.

See Also

`qnx_proxy_detach`

qnx_proxy_detach

qnx_proxy_detach — removes a proxy.

Syntax

```
qnx_proxy_detach (proxyid)
```

Arguments

proxyid

The proxy ID number as returned by qnx_proxy_attach.

Returns

0 on success, else -1.

Description

This function releases a proxy from its associated task. When a task dies all proxies attached to it are automatically removed.

In case of error, the following errnos are possible:

- **EPERM** This task is not the owner of the proxy.
- **ESRCH** The proxy does not exist.

Example

Examples of this function are beyond the scope of this documentation.

See Also

qnx_proxy_attach

qnx_proxy_rem_attach

`qnx_proxy_rem_attach` — creates a remote proxy message for a task.

Syntax

`qnx_proxy_rem_attach (`*`node`*`, `*`proxyid`*`)`

Arguments

`node`

The node ID of the remote node.

`proxyid`

The proxy ID of the associated proxy.

Returns

The remote proxy ID number, else -1.

Description

This function creates and attaches a remote message proxy to task. The task must already have a local proxy, which the remote proxy is associated with.

The remote proxy is activated by a call to `qnx_trigger` by any task on its *`node`*. A call to `qnx_trigger` on the remote proxy causes a subsequent call to `qnx_trigger` on its associated local proxy. The receiving task sees no difference between a message from a local proxy or a remote proxy.

If the task's own node number or zero is specified for *`node`*, a local proxy is created instead of a remote proxy.

In case of error, the following errnos are possible:

- **EINVAL** Invalid proxy.
- **EPERM** Proxy does not belong to task specified.
- **ESRCH** Proxy does not exist.

Example

Examples of this function are beyond the scope of this documentation.

See Also

`qnx_trigger` `qnx_rem_proxy_detach`

qnx_proxy_rem_detach

`qnx_proxy_rem_detach` — removes a remote proxy.

Syntax

`qnx_proxy_rem_detach (`*`node`*`, `*`proxyid`*`)`

Arguments

`node`

The node of the remote proxy.

`proxyid`

The proxy ID number of the remote proxy.

Returns

0 upon success, else -1.

Description

This function removes a remote proxy. When a task dies, all remote proxies attached to it will automatically be detached.

In case of error, the following errnos are possible:

- **EINVAL** Invalid proxy.
- **EPERM** Proxy does not belong to task specified.
- **ESRCH** Proxy does not exist.

Example

Examples of this function are beyond the scope of this documentation.

See Also

`qnx_rem_proxy_attach`

qnx_receive

`qnx_receive` — performs a QNX 4 `Receive`.

Syntax

`qnx_receive (`*`taskid`*`)`

Arguments

`taskid`

Task id of the acceptable sender, or 0 to receive from any task.

Returns

A list whose car is the process ID of the sending task and whose cdr is a character string containing the message which was sent, or -1 on error.

Description

This function performs a QNX 4 Receive. Once a message has been received the sending task is blocked until the recipient issues a reply message using the `qnx_reply` function.

This function should only be used to receive a specific message from a specific task. There is no need to make your own event loop in Gamma since there is a choice of `next_event`, `next_event_nb`, and PtMainloop.

Possible errno values are:

- **EFAULT** invalid buffer size
- **EINTR** the function call was interrupted by a signal
- **ESRCH** the process ID does not exist

Example

Examples of this function are beyond the scope of this documentation.

See Also

`qnx_send`, `qnx_reply`, `send`, `send_async`, `next_event`, `next_event_nb`, `PtMainLoop`

qnx_reply

qnx_reply — replies to qnx_receive messages.

Syntax

qnx_reply (*taskid*, *message*)

Arguments

taskid

The task which will get the reply.

message

A string containing the reply.

Returns

Returns zero on success and -1 on failure. Errno is set on error.

Description

The qnx_reply function is a wrap of the C function Reply. Any message that you receive using the qnx_receive function should be replied to. This function should only be used if you are also using qnx_receive. The event handling functions next_event, next_event_nb, and PtMainLoop are the preferred choice for Gamma programs since that automatically handle receiving, replying, and signal, timer, Cogent DataHub events, and Photon messages.

The errno possile values for this function are:

- **EAGAIN** Out of queue packets to network manager
- **EFAULT** Invalid message
- **EINTR** Function call interrupted by signal
- **EINVAL** invalud virtual circuit buffer
- **ENOMEM** not enough memory for operation
- **ESRCH** the process ID does not exist

Example

Examples of this function are beyond the scope of this documentation.

See Also

qnx_send, qnx_receive, send, send_async, next_event, next_event_nb, PtMainLoop

qnx_send

`qnx_send` — implements QNX 4 `Send`.

Syntax

```
qnx_send (taskid, message)
```

Arguments

`taskid`

The task to whom the message is sent.

`message`

The message, as a string, to send.

Returns

Returns a reply message character string from the recipient task. On an error this function returns `nil` and sets errno.

Description

This function implements the QNX 4 `Send` function. The reply is returned as a character string.

Possible values of errno are:

- **EAGAIN** No queue packets available for network manager
- **EFAULT** invalid message
- **EHOSTUNREACH** Destination node not in the netmap, or physical I/O error has occurred
- **EINTR** The function was interrupted by a signal
- **EINVAL** invalid message length
- **ENOMEM** not enough memory available for operation
- **ESRCH** the process ID does not exist

Example

Examples of this function are beyond the scope of this documentation.

See Also

`qnx_reply`, `qnx_receive`, `send`, `send_async`, `next_event`, `next_event_nb`, `PtMainLoop`

qnx_spawn_process

qnx_spawn_process — is an implementation of the C function qnx_spawn.

Syntax

```
qnx_spawn_process (exec, node, priority, scheduler,
flags, program, arg_list, file_list, ctfile)
```

Arguments

exec

If non-nil, then execute the process, otherwise spawn a separate task.

node

The node on which to spawn, or zero for the current node.

priority

The scheduler priority of the new task.

scheduler

the scheduler algorithm for the new task.

flags

The spawn flags.

program

The program name.

arg_list

Arguments to the program, as a list, excluding the program name.

file_list

A list of up to 10 files to be associated with the first 10 file descriptors of the new process. If any element in the list is a non-file, then use the corresponding file descriptor in the current process. A nil file-list indicates that all 10 file descriptors will be inherited from the calling task.

ctfile

A file associated with the controlling terminal for the new process.

Returns

The task ID of the new task, or -1 if an error occurs, and the errno is set.

Description

This is an implementation of the qnx_spawn C function. It is currently only available in QNX 4. This is the lowest level function for creating a new process.

Passing -1 to the priority option will cause the new task to inherit its priority from the parent, otherwise a value from 1-31 is acceptable.

Passing -1 to the scheduler option will cause the new task to inherit its scheduler activities from the parent, otherwise the following flags are defined for the scheduler option:

- **SCHED_FIFO** First-in, First-Out scheduling algorithm.

- **SCHED_RR** Round-robin scheduling algorithm.
- **SCHED_OTHER** Adaptive scheduling.

The following spawn flags are defined for the flags option:

- **_SPAWN_BGROUND** The process will be started with SIGINT and SIGQUIT ignored.
- **_SPAWN_DEBUG** The process will be started with the single step flag set. Rarely used.
- **_SPAWN_HOLD** The process will be started in a STOPPED state.
- **_SPAWN_NEWPGRP** The new process will start a new process group.
- **_SPAWN_NOHUP** The process will be started with SIGHUP ignored
- **_SPAWN_NOZOMBIE** When the new process terminates it will not become a zombie waiting for its father to do a wait on its death. The parent process will not see the child process die and a SIGCHILD will not be set.
- **_SPAWN_SETSID** The new process will start a new session.
- **_SPAWN_SIGCLR** The new process will not inherit ignored signals from its parent.
- **_SPAWN_TCSETPGRP** The new process will start a new terminal group. ALL keyboard breaks will be directed a it.
- **_SPAWN_XCACHE** Instruct the file system to place the executable in cache in hopes that it will be loaded again soon.

The library "const/QNXOS" should be required to use these constants

Errors that can happen when using this function:

- **E2BIG** The sum of the bytes used by the new process image’s argument list and environment is too big.
- **EACCES** No permissions to execute program.
- **EAGAIN** No free process entries or local memory.
- **EINVAL** The priority or the scheduling policy is invalid.
- **ENAMETOOLONG** The length of the program name, expanded to it’s full path, is too long.
- **ENOENT** The program does not exist.
- **ENOEXEC** The program is not the correct format (not an executable).
- **ENOLIC** Insufficient licenses to use this function.
- **ENOMEM** Not enough system memory.
- **ENONDP** The program needs an 80x87. A co-processor is not installed and the emulator (emu87) is not running.
- **ETXTBUSY** The program to launch is open for write (busy).

The ctfile is a file descriptor associated with the new process. This parameter is only meaningful if the _SPAWN_SETID flag is set. If you wish to start a new session without a controlling terminal then pass -1.

Example

Examples of this function are beyond the scope of this documentation.

See Also

exec, fork, wait

qnx_trigger

`qnx_trigger` — tells a proxy to send its message.

Syntax

```
qnx_trigger (pid, command, priority? )
```

Arguments

`proxyid`

The process ID of the proxy.

`command`

The message code.

`priority`

The priority of the message.

Returns

The process ID of the proxy triggered, else -1.

Description

This function is a mapping of the QNX 4 kernel function `Trigger`. It triggers a proxy to send its message to the receiving task. Its calling process does not block, and if more than one `qnx_trigger` call is sent while the task is busy, up to 65535 proxy messages will be queued for later delivery.

In case of error, the errno ESRCH means the proxy does not exist.

Example

Examples of this function are beyond the scope of this documentation.

qnx_vc_attach

qnx_vc_attach — establishes a virtual circuit between two processes on two computers.

Syntax

```
qnx_vc_attach (node, taskid, max_msg_length, flags)
```

Arguments

node

The node to which to attach.

taskid

The task id to which to attach.

max_msg_length

The maximum message size which will be passed between tasks.

flags

Virtual circuit flags.

Returns

A task id of a virtual circuit, or -1 on error, with errno set.

Description

This function establishes a network link between two processes on two computers. Once this link is established the two processes can communicate using the IPC function qnx_send/qnx_receive/qnx_reply.

Legal virtual circuit flags that can be OR-ed together are:

- **0 (zero)** A new virtual process and a new buffer will be allocated on both ends.
- **_VC_AT_SHARE** Use an existing virtual circuit, if it exists.
- **_VC_AT_REM_ZOMBIE** The remote virtual circuit id (vid) will become a zombie process when the remote process ID terminates.

Possible values for errno are:

- **EAGAIN** Proc to Net enqueuing failed
- **EHOSTUNREACH** Destination node not in netmap or physical I/O error
- **EINVAL** buffer too big
- **ENOLIC** no license to communicate with this node
- **ENOMEM** not enough memory to complete operation
- **ENOVPE** not enough proc entries to new vc
- **ENOSYS** no Net manager found
- **ESRCH** process ID not valid

Example

Examples of this function are beyond the scope of this documentation.

See Also

qnx_vc_name_attach, qnx_send, send, send_async

qnx_vc_detach

qnx_vc_detach — detaches a virtual circuit.

Syntax

```
qnx_vc_detach (taskid)
```

Arguments

taskid

The task id returned by qnx_vc_attach.

Returns

t on success, nil on failure, with errno set.

Description

This function detaches a virtual circuit previously attached with qnx_vc_attach or qnx_vc_name_attach.

Possible values for errno are:

- **EAGAIN** Proc to Net enqueuing failed
- **ESRCH** pid not valid
- **EPERM the vid is not your to detach** nil

Example

Examples of this function are beyond the scope of this documentation.

See Also

qnx_vc_attach, qnx_vc_name_attach

qnx_vc_name_attach

qnx_vc_name_attach — attaches a virtual circuit with a name instead of a process ID number.

Syntax

```
qnx_vc_name_attach (node, max_msg_length, name)
```

Arguments

node

The node for the attachment.

max_msg_length

The maximum message size which will be passed between tasks.

name

The name of the task to attach.

Returns

Virtual circuit ID on success, -1 on failure, with errno set.

Description

This function performs the same operation as qnx_vc_attach except that a name attached with qnx_name_attach can be specified instead of a process ID number. The function is currently only available in QNX 4.

Possible errno values are:

- **EAGAIN** Proc to Net enqueuing failed
- **EHOSTUNREACH** Destination node not in netmap or physical I/O error
- **EINVAL** buffer too big
- **ENOLIC** no license to communicate with this node
- **ENOMEM** not enough memory to complete operation
- **ENOVPE** not enough proc entries to new vc
- **ENOSYS** no Net manager found
- **ESRCH** pid not valid

Example

Examples of this function are beyond the scope of this documentation.

See Also

qnx_vc_attach, qnx_vc_detach

Index

Symbols

A

B

C

D

E

F

G

S

T

U

W

Colophon

This book was produced by Cogent Real-Time Systems, Inc. from a single-source group of SGML files. Gnu Emacs was used to edit the SGML files. The DocBook DTD and related DSSSL stylesheets were used to transform the SGML source into HTML, PDF, and QNX Helpviewer output formats. This processing was accomplished with the help of OpenJade, JadeTeX, Tex, and various scripts and makefiles. Details of the process are described in our book: Preparing Cogent Documentation, which is published on-line at
`http://developers.cogentrts.com/cogent/prepdoc/book1.html`.

Text written by Andrew Thomas, Mark Oliver, Bob McIlvride, and Elena Devdariani.