**1. Introduction.** `GUSI` is an extension and partial replacement of the `MPW` runtime library. Its main objective is to provide a more or less simple and consistent interface across the following *communication domains*:

- ▷ **Files** Ordinary Macintosh files and `MPW` pseudo devices.
- ▷ **Unix** Memory based communication within a single machine (This name exists for historical reasons).
- ▷ **Appletalk** `ADSP` (and possibly in the future `DDP`) communication over a network.
- ▷ **PPC** Local and remote connections with the System 7 `PPC Toolbox`
- ▷ **Internet** `TCP` and `UDP` connections over `MacTCP`.
- ▷ **PAP** Connections with the `Printer Access Protocol`, typically to a networked PostScript printer.

Additionally, `GUSI` adds some `UNIX` library calls dealing with files which were missing, like *chdir*( ), *getcwd*( ), *symlink*( ), and *readlink*( ), and changes a few other library calls to behave more like their `UNIX` counterparts.
The most recent version of `GUSI` may be obtained by anonymous ftp from `nic.switch.ch` in the directory `software/mac/src/mpw_c`.
There is also a mailing list devoted to discussions about `GUSI`. You can join the list by sending email to <gusi-request@iis.ee.ethz.ch>.

**2. Copying.**



**3. Design Objectives.** `GUSI` was developed according to at least three mutually conflicting standards:

- ▷ The definition of the existing C library.
- ▷ The behavior of the corresponding UNIX calls.
- ▷ The author's judgement, prejudices, laziness, and limited resources.

In general, the behavior of the corresponding UNIX library call was implemented, since this faciliates porting UNIX utilities to the Macintosh.

**4. Acknowledgements.** I would like to thank all who have agreed to beta test this code and who have provided feedback.

The TCP/IP code in `GUSIINET.cp`, `GUSITCP.cp`, and `GUSIUDP.cp` is derived from a socket library written by Charlie Reiman <creiman@ncsa.uiuc.edu>, which in turn is based on code written by Tom Milligan <milligan@madhaus.utcs.utoronto.ca>.

The PAP code in `GUSIPAP.cp` is derived from code written by Sak Wathanasin <sw@nan.co.uk>.

Many of the header files in the `:include:` subdirectory are borrowed from BSD Unix, therefore: This product includes software developed by the University of California, Berkeley and its contributors.

**5. Installing and using GUSI.** This section discusses how you can install `GUSI` on your disk and use it for your programs.

**6.** To install `GUSI`, change in the MPW Shell to its directory and type:

`BuildProgram Install <Enter>`

This will install all necessary files in `{CIncludes}`, `{CLibraries}`, and `{RIncludes}`, respectively. It will also install `/etc/services` in your preferences folder, prompting you if you have an older version there.

**7.** To use `GUSI`, include one or more of the following header files in your program:

| | |
|---|---|
| `GUSI.h` | The main file. This includes almost everything else. |
| `TFileSpec.h` | *FSSpec* manipulation routines. |
| `dirent.h` | Routines to access all entries in a directory. |
| `netdb.h` | Looking up TCP/IP host names. |
| `netinet/in.h` | The address format for TCP/IP sockets. |
| `sys/errno.h` | The errors codes returned by GUSI routines. |
| `sys/ioctl.h` | Codes to pass to *ioctl*( ). |
| `sys/socket.h` | Data types for socket calls. |
| `sys/stat.h` | Getting information about files. |
| `sys/types.h` | More data types. |
| `sys/uio.h` | Data types for scatter/gather calls. |
| `sys/un.h` | The address format for Unix domain sockets. |
| `unistd.h` | Prototypes for most routines defined in GUSI. |

**8.** GUSI expects the Macintosh Toolbox to be initialized. This will happen automatically if you're writing an `MPW` tool or if you are linking with `SIOW` and are forcing a write to standard output or standard error before you are using any non-file `GUSI` routines. Otherwise, you should init the Toolbox in the following way:

```
InitGraf((Ptr)&qd.thePort);
InitFonts();
InitWindows();
InitMenus();
TEInit();
InitDialogs(nil);
InitCursor();
```

**9.** You have to link your program with the `GUSI` library, `{CLibraries}GUSI.o`, and optionally one or several *configuration files*. Currently, the following configuration files exist:

| | |
|---|---|
| `GUSI_Everything.cfg` | Include code for everything defined in `GUSI`. |
| `GUSI_Appletalk.cfg` | Include code for AppleTalk sockets. |
| `GUSI_Internet.cfg` | Include code for MacTCP sockets. |
| `GUSI_PAP.cfg` | Include code for PAP sockets. |
| `GUSI_PPC.cfg` | Include code for PPC sockets. |
| `GUSI_Unix.cfg` | Include code for Unix domain sockets. |

If you don't specify any configuration files, only the file related routines will be included. It's important that these files appear *before* all other libraries. Linking with `GUSI` doesn't free you from linking in the standard libraries, typically:

```
{Libraries}Runtime.o
{Libraries}Interface.o
{CLibraries}StdCLib.o
{Libraries}ToolLibs.o
```

You will get lots of warning messages about duplicate definitions, but that's ok (Which means I can't do anything about it).

**10.** You should also rez your program with `GUSI.r`. The section "Resources" below discusses when and how to add your own configuration resource to customize `GUSI` defaults.

**11. Overview.** This section discusses the routines common to all, or almost all communication domains. These routines return −1 if an error occurred, and set the variable *errno* to an error code. On success, the routines return 0 or some positive value.

Some common error codes are:

| | |
|---|---|
| *EBADF* | The descriptor number you passed doesn't refer to a valid file or socket. |
| *ENOMEM* | Some memory error occurred. |
| *EINTR* | The user interrupted a lengthy operation by pressing Command-Period. |
| *ENOTCONN* | The socket is not connected and must be connected for this operation. |

**12. Creating and destroying sockets.** A socket is created with *socket*( ) and destroyed with *close*( ).

**13.** **int** *socket*(**int** *af*, **int** *type*, **int** *protocol*) creates an endpoint for communication and returns a descriptor. *af* specifies the communication domain to be used. Valid values are:

| | |
|---|---|
| *AF_UNIX* | Communication internal to a single Mac. |
| *AF_INET* | TCP/IP, using `MacTCP`. |
| *AF_APPLETALK* | Appletalk, using ADSP. |
| *AF_PPC* | The Program-to-Program Communication Toolbox. |

*type* specifies the semantics of the communication. The following two types are available:

| | |
|---|---|
| *SOCK_STREAM* | A two way, reliable, connection based byte stream. |
| *SOCK_DGRAM* | Connectionless, unreliable messages of a fixed maximum length. |

*protocol* would be used to specify an alternate protocol to be used with a socket. In `GUSI`, however, this parameter is always ignored.

Error codes:

| | |
|---|---|
| *EINVAL* | The *af* you specified doesn't exist. |
| *EMFILE* | The descriptor table is full. |

**14.** **void** *close*(**int** *fd*) removes the access path associated with the descriptor, and closes the file or socket if the last access path referring to it was removed.

**15. Prompting the user for an address.** To give the user the opportunity of entering an address for a socket to be bound or connected to, the *choose*( ) routine was introduced in `GUSI`. This routine has no counterpart in UNIX implementations.

**16.** **int** *choose*(**int** *dom*, **int** *type*, **char** ∗*prompt*, **void** ∗*constraint*, **int** *flags*, **void** ∗*name*, **int** ∗*nlen*) puts up a modal dialog prompting the user to choose an address. *dom* specifies the communication domain, like in *socket*. *type* may be used by future communication domains to further differentiate within a domain, but is ignored by current domains. *prompt* is a message that will appear in the dialog. *constraint* may be used to restrict the types of acceptable addresses (For more information, consult the section of the communication domain). The following two *flags* are defined for most socket types:

| | |
|---|---|
| *CHOOSE_DEFAULT* | Offer the contents passed in *name* as the default choice. |
| *CHOOSE_NEW* | Prompt for a new address, suitable for passing to *bind*( ). Default is prompting for an existing address, to be used by *connect*( ). |

*name* on input contains a default address if *CHOOSE_DEFAULT* is set. On output, it is set to the address chosen.

Error codes:

| | |
|---|---|
| *EINVAL* | One of the *flags* is not (yet) supported by this communications domain. This error is never reported for *CHOOSE_DEFAULT*, which might get silently ignored. |
| *EINTR* | The user chose "Cancel" in the dialog. |

**17. Establishing connections between sockets.** Before you can transmit data on a stream socket, it must be connected to a peer socket. Connection establishment is asymmetrical: The server socket registers its address with *bind*( ), calls *listen*( ) to indicate its willingness to accept connections and accepts them by calling *accept*( ). The client socket, after possibly having registered its address with *bind*( ) (This is not necessary for all socket families as some will automatically assign an address) calls *connect*( ) to establish a connection with a server.
It is possible, but not required, to call *connect*( ) for datagram sockets.

**18.** **int** *bind*(**int** *s*, **const struct** *sockaddr* ∗*name*, **int** *namelen*) binds a socket to its address. The format of the address is different for every socket family. For some families, you may ask the user for an address by calling *choose*( ).

Error codes:

| | |
|---|---|
| *EAFNOSUPPORT* | *name* specifies an illegal address family for this socket. |
| *EADDRINUSE* | There is already another socket with this address. |

**19.** **int** *listen*(**int** *s*, **int** *qlen*) turns a socket into a listener. *qlen* determines how many sockets can concurrently wait for a connection, but is ignored for almost all socket families.

**20.** **int** *accept*(**int** *s*, **struct** *sockaddr* ∗*addr*, **int** ∗*addrlen*) accepts a connection for a socket *on a new socket* and returns the descriptor of the new socket. If *addr* is not *NULL*, the address of the connecting socket will be assigned to it.
You can find out if a connection is pending by calling *select*( ) to find out if the socket is ready for *reading*.

Error codes:

| | |
|---|---|
| *ENOTCONN* | You did not call *listen*( ) for this socket. |
| *EWOULDBLOCK* | The socket is nonblocking and no socket is trying to connect. |

**21.** **int** *connect*(**int** *s*, **const struct** *sockaddr* **addr*, **int** *addrlen*) tries to connect to the socket whose address is in *addr*. If the socket is nonblocking and the connection cannot be made immediately, *connect*( ) returns *EINPROGRESS*. You can find out if the connection has been established by calling *select*( ) to find out if the socket is ready for *writing*.

Error codes:

| | |
|---|---|
| *EAFNOSUPPORT* | *name* specifies an illegal address family for this socket. |
| *EISCONN* | The socket is already connected. |
| *EADDRNOAVAIL* | There is no socket with the given address. |
| *ECONNREFUSED* | The socket refused the connection. |
| *EINPROGRESS* | The socket is nonblocking and the connection is being established. |

**22. Transmitting data between sockets.** You can write data to a socket using *write*( ), *writev*( ), *send*( ), *sendto*( ), or *sendmsg*( ). You can read data from a socket using *read*( ), *readv*( ), *recv*( ), *recvfrom*( ), or *recvmsg*( ).

**23.** **int** *read*(**int** *s*, **char** **buffer*, **unsigned** *buflen*) reads up to *buflen* bytes from the socket. *read*( ) for sockets differs from *read*( ) for files mainly in that it may read fewer than the requested number of bytes without waiting for the rest to arrive.

Error codes:

| | |
|---|---|
| *EWOULDBLOCK* | The socket is nonblocking and there is no data immediately available. |

**24.** **int** *readv*(**int** *s*, **const struct** *iovec* **iov*, **int** *count*) performs the same action, but scatters the input data into the *count* buffers of the *iov*narray, always filling one buffer completely before proceeding to the next. *iovec* is defined as follows:

```
struct iovec {
  caddr_t iov_base;   /* Address of this buffer */
  int iov_len;   /* Length of the buffer */
};
```

**25.** **int** *recv*(**int** *s*, **void** **buffer*, **int** *buflen*, **int** *flags*) is identical to *read*( ), except for the *flags* parameter. If the *MSG_OOB* flag is set for a stream socket that supports out-of-band data, *recv*( ) reads out-of-band data.

**26.** **int** *recvfrom*(**int** *s*, **void** **buffer*, **int** *buflen*, **int** *flags*, **void** **from*, **int** **fromlen*) is the equivalent of *recv*( ) for unconnected datagram sockets. If *from* is not *NULL*, it will be set to the address of the sender of the message.

**27.** **int** *recvmsg*(**int** *s*, **struct** *msghdr* **msg*, **int** *flags*) is the most general routine, combining the possibilities of *readv*( ) and *recvfrom*( ). *msghdr* is defined as follows:

```
struct msghdr {
  caddr_t msg_name;   /* Like from in recvfrom( ) */
  int msg_namelen;   /* Like fromlen in recvfrom( ) */
  struct iovec *msg_iov;   /* Scatter/gather array */
  int msg_iovlen;   /* Number of elements in msg_iov */
  caddr_t msg_accrights;   /* Access rights sent/received. Not used in GUSI */
  int msg_accrightslen;
};
```

**28.** **int** *write*(**int** *s*, **char** **buffer*, **unsigned** *buflen*) writes up to *buflen* bytes to the socket. As opposed to *read*( ), *write*( ) for nonblocking sockets always blocks until all bytes are written or an error occurs.

Error codes:

*EWOULDBLOCK* The socket is nonblocking and data can't be immediately written.

**29.** **int** *writev*(**int** *s*, **const struct** *iovec* **iov*, **int** *count*) performs the same action, but gathers the output data from the *count* buffers of the *iov*narray, always sending one buffer completely before proceeding to the next.

**30.** **int** *send*(**int** *s*, **void** **buffer*, **int** *buflen*, **int** *flags*) is identical to *write*( ), except for the *flags* parameter. If the *MSG_OOB* flag is set for a stream socket that supports out-of-band data, *send*( ) sends an out-of-band message.

**31.** **int** *sendto*(**int** *s*, **void** **buffer*, **int** *buflen*, **int** *flags*, **void** **to*, **int** **tolen*) is the equivalent of *send*( ) for unconnected datagram sockets. The message will be sent to the socket whose address is given in *to*.

**32.** **int** *sendmsg*(**int** *s*, **const struct** *msghdr* **msg*, **int** *flags*) combines the possibilities of *writev*( ) and *sendto*( ).

## 33. I/O multiplexing.

**34.** **int** *select*(**int** *width*, *fd_set* **readfds*, *fd_set* **writefds*, *fd_set* **exceptfds*, **struct** *timeval* **timeout*) examines the I/O descriptors specified by the bit masks *readfs*, *writefs*, and *exceptfs* to see if they are ready for reading, writing, or have an exception pending. *width* is the number of significant bits in the bit mask. *select*( ) replaces the bit masks with masks of those descriptors which are ready and returns the total number of ready descriptors. *timeout*, if not *NULL*, specifies the maximum time to wait for a descriptor to become ready. If *timeout* is *NULL*, *select*( ) waits indefinitely. To do a poll, pass a pointer to a zero *timeval* value in *timeout*. Any of *readfds*, *writefds*, or *exceptfds* may be given as *NULL* if no descriptors are of interest.

Error codes:

*EBADF* One of the bit masks specified an invalid descriptor.

**35.** The descriptor bit masks can be manipulated with the following macros:

```
FD_ZERO(fds);      /* Clear all bits in *fds */
FD_SET(n, fds);    /* Set bit n in *fds */
FD_CLR(n, fds);    /* Clear bit n in *fds */
FD_ISSET(n, fds);  /* Return 1 if bit n in *fds is set, else 0 */
```

**36. Getting and changing properties of sockets.** You can obtain the address of a socket and the socket it is connected to by calling *getsockname*( ) and *getpeername*( ) respectively. You can query and manipulate other properties of a socket by calling *ioctl*( ), *fcntl*( ), *getsockopt*( ), and *setsockopt*( ). You can create additional descriptors for a socket by calling *dup*( ) or *dup2*( ).

**37.** **int** *getsockname*(**int** *s*, **struct** *sockaddr* **name*, **int** **namelen*) returns in **name* the address the socket is bound to. **namelen* should be set to the maximum length of *name* and will be set by *getsockname*( ) to the actual length of the name.

**38.** **int** *getpeername*(**int** *s*, **struct** *sockaddr* **name*, **int** **namelen*) returns in **name* the address of the socket that this socket is connected to. **namelen* should be set to the maximum length of *name* and will be set by *getpeername*( ) to the actual length of the name.

**39.** **int** *ioctl*(**int** *d*, **unsigned int** *request*, **long** **argp*) performs various operations on the socket, depending on the *request*. The following codes are valid for all socket families:

| | |
|---|---|
| *FIONBIO* | Make the socket blocking if the **long** pointed to by argp is 0, else make it nonblocking. |
| *FIONREAD* | Set **argp* to the number of bytes waiting to be read. |

Error codes:

| | |
|---|---|
| *EOPNOTSUPP* | The operation you requested with *request* is not supported by this socket family. |

**40.** **int** *fcntl*(**int** *s*, **unsigned int** *cmd*, **int** *arg*) provides additional control over a socket. The following values of *cmd* are defined for all socket families:

| | |
|---|---|
| *F_DUPFD* | Return a new descriptor $\geq$ *arg* which refers to the same socket. |
| *F_GETFL* | Return descriptor status flags. |
| *F_SETFL* | Set descriptor status flags to *arg*. |

The only status flag implemented is *FNDELAY* which is true if the socket is nonblocking.

Error codes:

| | |
|---|---|
| *EOPNOTSUPP* | The operation you requested with *cmd* is not supported by this socket family. |

**41.** **int** *getsockopt*(**int** *s*, **int** *level*, **int** *optname*, **void** **optval*, **int** **optlen*) is used to request information about sockets. It is not implemented in `GUSI`.

**42.** **int** *setsockopt*(**int** *s*, **int** *level*, **int** *optname*, **void** **optval*, **int** *optlen*) is used to set options associated with a socket. It is not implemented in `GUSI`.

**43.** **int** *dup*(**int** *fd*) returns a new descriptor referring to the same socket as *fd*. The old and new descriptors are indistinguishible. The new descriptor will always be the smallest free descriptor.

**44.** **int** *dup2*(**int** *oldfd*, **int** *newfd*) closes *newfd* if it was open and makes it a duplicate of *oldfd*. The old and new descriptors are indistinguishible.

**45. Detail Description.** This section discusses for each socket domain the routines that behave different from their description in the previous section and a few calls specific to one domain.

**46. File system calls.** Files are unlike sockets in many respects: Their length is never changed by other processes, they can be rewound. There are also many calls which are specific to files.

**47. Differences to generic behavior.**

**48.** The following calls make no sense for files and return an error of *EOPNOTSUPP*:

*socket*( )
*bind*( )
*listen*( )
*accept*( )
*connect*( )
*getsockname*( )
*getpeername*( )
*getsockopt*( )
*setsockopt*( )

**49.** The following calls *will* work, but might be frowned upon by your friends (besides, UNIX systems generally wouldn't like them):

*recv*( )
*recvfrom*( )
*recvmsg*( )
*send*( )
*sendto*( )
*sendmsg*( )

**50.** *choose*( ) returns zero terminated C strings in *name*. It accepts an additional flag *CHOOSE_DIR*. If this is set, *choose*( ) will select directories instead of files.

You may restrict the files presented for choosing by passing a pointer to the following structure for the *constraint* argument:

```
typedef struct {
  short numTypes;     /* Number of legitimate file types */
  SFTypeList types;   /* The types, like 'TEXT' */
} sa_constr_file;
```

**51.** *select*( ) will give boring results. File descriptors are *always* considered ready to read or write, and *never* give exceptions.

**52.** *ioctl*( ) and *fcntl*( ) don't support manipulating the blocking state of a file descriptor or reading the number of bytes available for reading, but will accept lots of other requests—Check with your trusty MPW C documentation.

**53. Routines specific to the file system.** In this section, you'll meet lots of good old friends.

**54.** **int** *stat*(**const char** **path*, **struct** *stat* **buf*) returns information about a file. **struct** *stat* is defined as follows:

```
struct stat {
    dev_t st_dev;      /* Volume reference number of file */
    ino_t st_ino;      /* File or directory ID */
    u_short st_mode;   /* Type and permission of file */
    short st_nlink;    /* Always 1 */
    short st_uid;      /* Set to 0 */
    short st_gid;      /* Set to 0 */
    dev_t st_rdev;     /* Set to 0 */
    off_t st_size;
    time_t st_atime;   /* Set to st_mtime */
    time_t st_mtime;
    time_t st_ctime;
    long st_blksize;
    long st_blocks;
};
```

**55.** *st_mode* is composed of a file type and of file permissions. The file type may be one of the following:

| | |
|---|---|
| *S_IFREG* | A regular file. |
| *S_IFDIR* | A directory. |
| *S_IFLNK* | A finder alias file. |
| *S_IFCHR* | A console file under MPW or SIOW. |
| *S_IFSOCK* | A file representing a UNIX domain socket. |

Permissions consist of an octal digit repeated three times. The three bits in the digit have the following meaning:

| | |
|---|---|
| 4 | File can be read. |
| 2 | File can be written. |
| 1 | File can be executed, i.e., its type is 'APPL', 'MPST' or 'TEXT' |

**56.** **int** *lstat*(**const char** **path*, **struct** *stat* **buf*) works just like *stat*( ), but if *path* is a symbolic link, *lstat*( ) will return information about the link and not about the file it points to.

**57.** **int** *fstat*(**int** *fd*, **struct** *stat* **buf*) is the equivalent of *stat*( ) for descriptors representing open files. While it is legal to call *fstat*( ) for sockets, the information returned is not really interesting. The file type in *st_mode* will be *S_IFSOCK* for sockets.

**58.** **int** *utime*(**const char** **file*, **const struct** *utimbuf* **tim*) changes the modification time of a file. **struct** *utimbuf* is defined as:

```
struct utimbuf {
    time_t actime;     /* Access time */
    time_t modtime;    /* Modification time */
};
```

**59.** *tim* → *actime* is ignored, as the Macintosh doesn't store access times. The modification of *file* is set to *tim* → *modtime*.

**60.** **int** *isatty*(**int** *fd*) returns 1 if *fd* represents a terminal (i.e. is connected to `"Dev:StdIn"` and the like), 0 otherwise.

**61. long** *lseek*( **int** , **long** , **int** ) works the same as the `MPW` routine, and will return *ESPIPE* if called for a socket.

**62. int** *remove*(**const char** **filename*) removes the named file. If *filename* is a symbolic link, the link will be removed and not the file.

**63. int** *unlink*(**const char** **filename*) is identical to *remove*( ). Note that on the Mac, *unlink*( ) on open files behaves differently from `UNIX`.

**64. int** *rename*(**const char** **oldname*, **const char** **newname*) renames and/or moves a file. *oldname* and *newname* must specify the same volume, but as opposed to the standard `MPW` routine, they may specify different folders.

**65. int** *open*(**const char** *, **int** *flags*) opens a named file. The *flags* consist of one of the following modes:

| | |
|---|---|
| *O_RDONLY* | Open for reading only. |
| *WR_ONLY* | Open for writing only. |
| *O_RDWR* | Open for reading and writing. |

Optionally combined with one or more of:

| | |
|---|---|
| *O_APPEND* | The file pointer is set to the end of the file before each write. |
| *O_RSRC* | Open resource fork. |
| *O_CREAT* | If the file does not exist, it is created. |
| *O_EXCL* | In combination with *O_CREAT*, return an error if file already exists. |
| *O_TRUNC* | If the file exists, its length is truncated to 0; the mode is unchanged. |
| *O_ALIAS* | If the named file is a symbolic link, open the link, not the file it points to (This is most likely an incredibly bad idea). |

**66. int** *creat*(**const char** **name*) is identical to *open*(*name*, *O_WRONLY* + *O_TRUNC* + *O_CREAT*). If the file didn't exist before, `GUSI` determines its file type and creator of the according to rules outlined in the section "Resources" below.

**67. int** *faccess*(**const char** **filename*, **unsigned int** *cmd*, **long** **arg*) works the same as the corresponding `MPW` routine, but respects calls to *chdir*( ) for partial filenames.

**68. void** *fgetfileinfo*(**char** **filename*, **unsigned long** **newcreator*, **unsigned long** **newtype*) returns the file type and creator of a file.

**69. void** *fsetfileinfo*(**char** **filename*, **unsigned long** *newcreator*, **unsigned long** *newtype*) sets the file type and creator of a file to the given values.

**70. int** *symlink*(**const char** **linkto*, **const char** **linkname*) creates a file named *linkname* that contains an alias resource pointing to *linkto*. The created file should be indistinguishible from an alias file created by the System 7 Finder. Note that aliases bear only superficial similiarities to `UNIX` symbolic links, especially once you start renaming files.

**71. int** *readlink*(**const char** **path*, **char** **buf*, **int** *bufsiz*) returns in *buf* the name of the file that *path* points to.

**72. int** *mkdir*(**const char** **path*) creates a new directory.

**73.** **int** *rmdir*(**const char** **path*) deletes an empty directory.

**74.** **int** *chdir*(**const char** **path*) makes all future partial pathnames relative to this directory.

**75.** **char** **getcwd* ( **const char** **buf* , **int** *size* ) returns a pointer to the current directory pathname. If *buf* is *NULL*, *size* bytes will be allocated using *malloc*( ).

Error codes:

| | |
|---|---|
| *ENAMETOOLONG* | The pathname of the current directory is greater than *size*. |
| *ENOMEM* | *buf* was *NULL* and *malloc*( ) failed. |

**76.** A number of calls facilitate scanning directories. Directory entries are represented by following structure:

```
struct dirent {
   u_long d_fileno;    /* file number of entry */
   u_short d_reclen;    /* length of this record */
   u_short d_namlen;    /* length of string in d_name */
#define MAXNAMLEN 255
   char d_name[MAXNAMLEN + 1];    /* name must be no longer than this */
};
```

**77.** *DIR *opendir* ( **const char** **dirname* ) opens a directory stream and returns a pointer or *NULL* if the call failed.

**78.** **struct** *dirent *readdir*(*DIR*dirp*) returns the next entry from the directory or *NULL* if all entries have been processed.

**79.** **long** *telldir*( **const** *DIR *dirp* ) returns the position in the directory.

**80.** **void** *seekdir*(*DIR *dirp*, **long** *loc*) changes the position.

**81.** **void** *rewinddir*( *DIR *dirp* ) restarts a scan at the beginning.

**82.** **int** *closedir*( *DIR *dirp* ) closes the directory stream.

**83.** **int** *scandir*(**const char** **path*, **struct** *dirent ***entries*, **int** (**want*)(**struct** *dirent **), **int** (**sort*) ( **const void** * , **const void** * ) ) scans a whole directory at once and returns a possibly sorted list of entries. If *want* is not *NULL*, only entries for which *want* returns 1 are returned. If *sort* is not *NULL*, the list is sorted using *qsort*( ) with sort as a comparison function. If sort is *NULL*, the list will be sorted alphabetically on a Mac, but not necessarily on other machines.

**84.** **int** *truncate*(**const char** **path*, *off_t length*) causes a file to have a size equal to *length* bytes, shortening it or extending it with zero bytes as necessary.

**85.** **int** *ftruncate*(**int** *fd*, *off_t length*) does the same thing with an open file.

**86.** **Unix domain sockets.** This domain is quite regular and supports all calls that work on any domain, except for out-of-band data.

**87.** **Differences to generic behavior.**

**88.** Addresses are file system pathnames. `GUSI` complies to the Unix implementation in that it doesn't require the name to be terminated by a zero. Names that are generated by `GUSI`, however, will always be zero terminated (but the zero won't be included in the count).

```
struct sockaddr_un {
  short sun_family;   /* Always AF_UNIX */
  char sun_path[108];   /* A pathname to a file */
};
```

**89.** *choose*( ) works both for existing and new addresses, and no restriction is possible (or necessary).

**90. Appletalk sockets.** Currently, only stream sockets (including out-of-band data) are supported.

**91. Differences to generic behavior.**

**92.** Two classes of addresses are supported for AppleTalk. The main address type specifies numeric addresses.

```
struct sockaddr_atlk {
  short family;   /* Always AF_APPLETALK */
  AddrBlock addr;   /* The numeric AppleTalk socket address */
};
```

**93.** For *bind*( ) and *connect*( ), however, you are also allowed to specify symbolic addresses. *bind*( ) registers an NBP address, and *connect*( ) performs an NBP lookup. Registered NBP adresses are automatically released when the socket is closed. No call ever *returns* a symbolic address.

```
struct sockaddr_atlk_sym {
  short family;   /* Always ATALK_SYMADDR */
  EntityName name;   /* The symbolic NBP address */
};
```

**94.** *choose*( ) currently only works for existing sockets. The peer must have registered a symbolic address. To restrict the choice of addresses presented, pass a pointer to the following structure for the *constraint* argument:

```
typedef struct {
  short numTypes;   /* Number of allowed types */
  NLType types;   /* List of types */
} sa_constr_atlk;
```

**95. PPC sockets.** These provide authenticated stream sockets without out-of-band data.

**96. Differences to generic behavior.**

**97.** PPC socket addresses have the following format:

```
struct sockaddr_ppc {
  short family;   /* Always AF_PPC */
  LocationNameRec location;   /* Check your trusty Inside Macintosh */
  PPCPortRec port;
};
```

**98.** *choose*( ) currently only works for existing sockets. To restrict the choice of addresses presented, pass a pointer to the following structure for the *constraint* argument:

```
typedef struct {
    short flags;
    Str32 nbpType;
    PPCPortRec match;
} sa_constr_ppc;
```

**99.** *flags* is obtained by or'ing one or several of the following constants:

| | |
|---|---|
| *PPC_CON_NEWSTYLE* | Always required for compatibility reasons. |
| *PPC_CON_MATCH_NBP* | Only display machines that have registered an entity of type *nbpType*. |
| *PPC_CON_MATCH_NAME* | Only display ports whose name matches *match.name*. |
| *PPC_CON_MATCH_TYPE* | Only display ports whose type matches *match.u.portType*. |

*nbpType* specifies the machines to be displayed, as explained above. *match* contains the name and/or type to match against.

**100.** *connect*( ) will block even if the socket is nonblocking. In practice, however, delays are likely to be quite short, as it never has to block on a higher level protocol and the PPC ToolBox will automatically establish the connection.

**101. Internet sockets.** These are the real thing for real programmers. Out-of-band data only works for sending. Both stream (TCP) and datagram (UDP) sockets are supported.

**102. Differences to generic behavior.**

**103.** Internet socket addresses have the following format:

```
struct in_addr {
    u_long s_addr;
};
struct sockaddr_in {
    u_char sin_len;   /* Ignored */
    u_char sin_family;   /* Always AF_INET */
    u_short sin_port;   /* Port number */
    struct in_addr sin_addr;   /* Host ID */
    char sin_zero[8];
};
```

**104. Routines specific to TCP/IP sockets.** There are several routines to convert between numeric and symbolic addresses.

**105.** Hosts are represented by the following structure:

```
struct hostent {
    char *h_name;   /* Official name of the host */
    char **h_aliases;   /* A zero terminated array of alternate names for the host */
    int h_addrtype;   /* Always AF_INET */
    int h_length;   /* The length, in bytes, of the address */
    char **h_addr_list;   /* A zero terminated array of network addresses for the host */
};
```

**106.** **struct** *hostent* $*$*gethostbyname* ( **char** $*$*name* ) returns an entry for the host with the given *name* or *NULL* if a host with this name can't be found.

**107.** **struct** *hostent* $*$*gethostbyaddr* ( **const char** $*$*addrP*, **int** , **int** ) returns an entry for the host with the given address or *NULL* if a host with this name can't be found. *addrP* in fact has to be a **struct** *in_addr* $*$. The last two parameters are ignored.

**108.** **char** $*$*inet_ntoa* ( **struct** *in_addr inaddr* ) converts an internet address into the usual numeric string representation (e.g., 0x8184023C is converted to "129.132.2.60")

**109.** **struct** *in_addr inet_addr*(**char** $*$*address*) converts a numeric string into an internet address (If $x$ is a valid address, $inet\_addr(inet\_ntoa(x)) \equiv x$).

**110.** **int** *gethostname*(**char** $*$*machname*, **long** *buflen*) gets our name into *buffer*.

**111.** Services are represented by the following data structure:

```
struct servent {
  char *s_name;    /* official service name */
  char **s_aliases;    /* alias list */
  int s_port;    /* port number */
  char *s_proto;    /* protocol to use ("tcp" or "udp") */
};
```

**112.** **void** *setservent*(**int** *stayopen*) rewinds the file of services. If *stayopen* is set, the file will remain open until *endservent*( ) is called, else it will be closed after the next call to *getservbyname*( ) or *getservbyport*( ).

**113.** **void** *endservent*() closes the file of services.

**114.** **struct** *servent* $*$*getservent*( ) returns the next service from the file of services, opening the file first if necessary. If the file is not found (`/etc/services` in the preferences folder), a small built-in list is consulted. If there are no more services, *getservent*( ) returns *NULL*.

**115.** **struct** *servent* $*$*getservbyname* ( **const char** $*$*name*, **const char** $*$*proto* ) finds a named service by calling *getservent*( ) until the protocol matches *proto* and either the name or one of the aliases matches *name*.

**116.** **struct** *servent* $*$*getservbyport* ( **int** *port*, **const char** $*$*proto* ) finds a service by calling *getservent*( ) until the protocol matches *proto* and the port matches *port*.

**117.** Protocols are represented by the following data structure:

```
struct protoent {
  char *p_name;    /* official protocol name */
  char **p_aliases;    /* alias list (always NULL for GUSI) */
  int p_proto;    /* protocol number */
};
```

**118.** **struct** *protoent* $*$*getprotobyname* ( **char** $*$*name* ) finds a named protocol. This call is rather unexciting.

**119. PAP sockets.** PAP, the AppleTalk Printer Access Protocol is a protocol which is almost exclusively used to access networked printers. The current implementation of PAP in `GUSI` is quite narrow in that it only implements the workstation side of PAP and only in communication to the currently selected LaserWriter. It is also doomed, as it depends on Apple system resources that probably won't be supported anymore in Apple's forthcoming new printing architecture, but if there is enough interest, the current implementation is probably going to be replaced until then.

**120. Routines specific to PAP sockets.** While PAP sockets behave in most respects like other sockets, they can currently not be created with the *socket*( ) call, but are opened with *open*( ).

**121.** **int** *open*( `"Dev:Printer"` , **int** *flags* ) opens a connection to the last selected LaserWriter. *flags* is currently ignored.

**122.** Communication with LaserWriters is somewhat strange. The three main uses of PAP sockets are probably interactive sessions, queries, and downloads, which will be discussed in the following sections. As in all other socket families, `GUSI` does no filtering of the transmitted data, which means that lines sent by the LaserWriter will be separated by linefeeds (ASCII 10) rather than carriage returns (ASCII 13), which are used for this purpose in most other Mac contexts. For data you *send*, it doesn't matter which one you use.

**123.** You start an *interactive session* by sending a line `"executive"` after opening the socket. This will put lots of LaserWriters (certainly all manufactured by Apple, but probably not a Linotronic) into interactive mode. If you want to, you can now play terminal emulator and use your LaserWriter as an expensive desk calculator.

**124.** A *query* is some PostScript code you send to a LaserWriter that you expect to be answered. This is quite straightforward, except that LaserWriters don't seem to answer until you have indicated to them that no more data from you will be coming. Therefore, you have to call *shutdown*(*s*, 1) to shut the socket down for writing after you have written your query and before you try to read the answer. The following code demonstrates how to send a query to the printer:

```
int s ← open("Dev:Printer", O_RDWR);
write(s, "FontDirectory␣/Gorilla-SemiBold␣exch␣known...", len);
  /* We won't write any more */
shutdown(s, 1);
while (read(s, buf, len) > 0)
  do_something();
close(s);
```

**125.** If you want to simply *download* a file, you can also ignore the LaserWriter's response and simply close the socket after downloading.

**126. Miscellaneous.**

**127. BSD memory routines.**

**128.** **void** *bzero*(**void** **from*, **int** *len*) zeroes *len* bytes, starting at *from*.

**129.** *bfill*( **void** **from*, **int** *len*, **int** *x* ) fills *len* bytes, starting at *from* with *x*.

**130.** **void** *bcopy*(**void** **from*, **void** **to*, **int** *len*) copies *len* bytes from *from* to *to*.

**131.** **int** *bcmp*(**void** **s1*, **void** **s2*, **int** *len*) compares *len* bytes at *s1* against *len* bytes at *s2*, returning zero if the two areas are equal, nonzero otherwise.

**132. Blocking calls.** Since the Macintosh doesn't have preemptive task switching, it is important that other applications get a chance to run during blocking calls. This section discusses the mechanism GUSI uses for that purpose.

**133.** While a routine is waiting for a blocking call to terminate, it repeatedly calls a spin routine with the following parameters:

```
typedef enum spin_msg { SP_MISC ,
    /* some weird thing, usually just return immediately if you get this */
  SP_SELECT ,    /* in a select call */
  SP_NAME ,    /* getting a host by name */
  SP_ADDR ,    /* getting a host by address */
  SP_STREAM_READ ,    /* Stream read call */
  SP_STREAM_WRITE ,    /* Stream write call */
  SP_DGRAM_READ ,    /* Datagram read call */
  SP_DGRAM_WRITE ,    /* Datagram write call */
  SP_SLEEP ,    /* sleeping, passes ticks left to sleep */
  SP_AUTO_SPIN    /* Automatically spinning, passes spin count */
  } spin_msg;
  typedef int (*GUSISpinFn) ( spin_msg msg , long param ) ;
```

**134.** You can modify the spin routine with the following calls:

```
int GUSISetSpin(GUSISpinFn routine);
GUSISpinFn GUSIGetSpin( void ) ;
```

**135.** Usually, however, the default spin routine will do what you want: It spins a cursor and occasionally calls *GetNextEvent*( ) or *WaitNextEvent*( ). By default, only mouse down and suspend/resume events are handled, but you can change that by passing your own *GUSIEvtTable* to *GUSISetEvents*( ).

```
int GUSISetEvents(GUSIEvtTable table);
GUSIEvtHandler *GUSIGetEvents(void);
```

**136.** A *GUSIEvtTable* is a table of *GUSIEvtHandlers*, indexed by event code. Presence of a non-nil entry in the table will cause that event class to be allowed for *GetNextEvent*( ) or *WaitNextEvent*( ). GUSI includes one event table to be used with the SIOW library.

```
typedef void (*GUSIEvtHandler)(EventRecord*ev);
typedef GUSIEvtHandler GUSIEvtTable[24];
extern GUSIEvtHandler GUSISIOWEvents[];
```

**137. Resources.** A few `GUSI` routines (currently primarily choose()) need resources to work correctly. These are added if you Rez your program with `GUSI.r`. On startup, `GUSI` also looks for a *preference* resource with type `'GUΣI'` and ID *GUSIRsrcID*, which is currently defined as follows:

```
#ifndef GUSI_PREF_VERSION
#define GUSI_PREF_VERSION '0102'
#endif
  type 'GUΣI' {
      literal longint text ←'TEXT';    /* Type for creat'ed files */
      literal longint mpw ←'MPS';    /* Creator for creat'ed files */
      byte noAutoSpin, autoSpin;    /* Automatically spin cursor ? */
#if GUSI_PREF_VERSION ≥'0110'
      boolean useChdir, dontUseChdir;    /* Use chdir() ? */
      boolean approxStat, accurateStat;    /* statbuf.st_nlink = # of subdirectories ? */
      boolean noTCPDaemon, isTCPDaemon;    /* Inetd client ? */
      boolean noUDPDaemon, isUDPDaemon;
      fill bit[4];
      literal longint ← GUSI_PREF_VERSION;
#if GUSI_PREF_VERSION ≥'0120'
      integer ← $$Countof(SuffixArray);
      wide array SuffixArray { literal longint;    /* Suffix of file */
        literal longint;    /* Type for file */
        literal longint;    /* Creator for file */
        };
#endif
#endif
        };
```

**138.** To keep backwards compatible, the preference version is included, and you are free to use whatever version of the preferences you want by defining *GUSI_PREF_VERSION*.

**139.** The first two fields define the file type and creator, respectively, to be used for files created by `GUSI`. The type and creator of existing files will never be changed unless explicitely requested with fsetfileinfo(). The default is to create text files (type 'TEXT') owned by the `MPW Shell` (creator 'MPS '). If you request a preference version of 1.2.0 and higher, you are also allowed to specify a list of suffixes that are given different types. An example of such a list would be:

{' *SYM* ' , ' *MPSY* ' , ' *sade* ' }

**140.** The *autoSpin* value, if nonzero, makes `GUSI` call the spin routine for every call to *read*( ), *write*( ), *send*( ), or *recv*( ). This is useful for making an I/O bound program MultiFinder friendly without having to insert explicit calls to *SpinCursor*( ). If you don't specify a preference resource, *autoSpin* is assumed to be 1. You may specify arbitrary values greater than one to make your program even friendlier; note, however, that this will hurt performance.

**141.** The *useChdir* flag tells `GUSI` whether you change directories with the toolbox calls *PBSetVol*( ) or *PBHSetVol*( ) or with the `GUSI` call *chdir*( ). The current directory will start with the directory your application resides in or the current `MPW` directory, if you're running an `MPW` tool. If *useChdir* is specified, the current directory will only change with *chdir*( ) calls. If *dontUseChdir* is specified, the current directory will change with toolbox calls, until you call *chdir*( ) the first time. This behaviour is more consistent with the standard `MPW` library, but has IMHO no other redeeming value. If you don't specify a preference resource, *useChdir* is assumed.

**142.** If *approxStat* is specified, *stat*( ) and *lstat*( ) for directories return in *st_nlink* the number of *items* in the directory +2. If *accurateStat* is specified, they return the number of *subdirectories* in the directory. The latter has probably the best chances of being compatible with some Unix software, but the former is often a sufficient upper bound, is much faster, and most programs don't care about this value anyway. If you don't specify a preference resource, *approxStat* is assumed.

**143.** The *isTCPDaemon* and *isUDPDaemon* flags turn `GUSI` programs into clients for David Petersons `inetd`, as discussed below. If you don't specify a preference resource, *noTCPDaemon* and *noUDPDaemon* are assumed.

**144. Advanced techniques.** This section discusses a few techniques that probably not every user of `GUSI` needs.

**145. Writing daemons.** David Peterson <david@stealth.usc.edu> has written an Internet daemon for the Macintosh. This is a program that listens on various ports for a connection, and if one arrives, launches a server program to handle it. `GUSI` contains code that makes it extremely easy to interface with inetd: All you have to do is to set the *tcpDaemon* and/or the *udpDaemon* flag in the configuration resource. `GUSI` and inetd will then conspire to have file descriptors 0, 1, and 2 connected to the incoming socket. Therefore, all you have to do is to read and write from and to 0, 1, 2 (or stdin, stdout, and stderr, if you're using stdio).
inetd is available by anonymous ftp from the following locations:

```
usc.edu        pub/davidp
nic.switch.ch  software/mac/src/mpw_c
```

**146. FSSpec routines.** If you need to do complicated things with the Mac file system, the normal `GUSI` routines are probably not sufficient, but you still might want to use the internal mechanism `GUSI` uses. This mechanism is provided in the header file `TFileSpec.h`, which defines both *C* and *C++* interfaces.

**147. Adding your own communication families.** It is rather easy to add your own socket types to `GUSI`:

- ▷ Pick an unused number between 17 and *GUSI_MAX_DOMAINS* to use for your address family.
- ▷ Include `GUSI_P.h`.
- ▷ Write a subclass of *SocketDomain* and override *choose*( ) and either *open*( ) or *socket*( ). If you override *open*( ), you have to write your own routine to create sockets of this type.
- ▷ Write a subclass of *Socket* and override whatever you want. If you override *recvfrom*( ) and *sendto*( ), *read*( ) and *write*( ) are automatically defined.
- ▷ For more information, study the code in `GUSIDispatch.cp` and `GUSISocket.cp`, which implement the generic socket code. The easiest actual socket implementation to study is probably `GUSIUnix.cp`.

### 148. Known problems, words of warning.

▷ *O_NRESOLVE*, as introduced in the E.T.O #8 Prerelease libraries, is interpreted the same way as *O_ALIAS*, i.e. intermediate aliases are silently resolved. On the other hand, I can't think of a good reason why anybody would use *O_NRESOLVE*.

▷ *getsockname*( ) for TCP/IP sockets for which neither *bind*( ) with a nonzero port number nor *connect*( ) or *listen*( ) have been called returns zero.

▷ *bind*( ) for TCP/IP sockets doesn't check for duplicate port numbers.

**149. References.** The following books might provide you with more information about various aspects of `GUSI`.


[**Appl85**] Apple Computer, Inc., *Inside Macintosh Volume I–VI*, Addison Wesley, 1985–91

[**Appl88**] Apple Computer, Inc., *Macintosh Programmer's Workshop C*, 1988

[**Crow13**] Aleister Crowley, *The Book of Lies*, 1913

[**Holz91**] David A. Holzgang, *Programming the LaserWriter*, Addison-Wesley 1991

[**LMKQ89**] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, John S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison Wesley, 1989

[**Stev90**] W. Richard Stevens, *UNIX Network Programming*, Prentice Hall, 1990

[**Stev92**] W. Richard Stevens, *Advanced Programming in the UNIX Environment*, Prentice Hall, 1992

[**Sun88**] Sun Microsystems, Inc., *SunOS Reference Manual*, 1988

**150. Index.**


*accept*: 17, 20, 48.
*accurateStat*: 137, 142.
*actime*: 58, 59.
*addr*: 20, 21, 92.
*AddrBlock*: 92.
*address*: 109.
*addrlen*: 20, 21.
*addrP*: 107.
*af*: 13.
*AF_APPLETALK*: 13, 92.
*AF_INET*: 13, 103, 105.
*AF_PPC*: 13, 97.
*AF_UNIX*: 13, 88.
*approxStat*: 137, 142.
*arg*: 40, 67.
*argp*: 39.
*array*: 137.
*ATALK_SYMADDR*: 93.
*autoSpin*: 137, 140.
*bcmp*: 131.
*bcopy*: 130.
*bfill*: 129.
*bind*: 16, 17, 18, 48, 93, 148.
*bit*: 137.
*boolean*: 137.
*buf*: 54, 56, 57, 71, 75, 124.
*buffer*: 23, 25, 26, 28, 30, 31, 110.
*buflen*: 23, 25, 26, 28, 30, 31, 110.
*bufsiz*: 71.
*byte*: 137.
*bzero*: 128.
*caddr_t*: 24, 27.
*chdir*: 1, 67, 74, 141.
*choose*: 15, 16, 18, 50, 89, 94, 98, 147.
*CHOOSE_DEFAULT*: 16.
*CHOOSE_DIR*: 50.
*CHOOSE_NEW*: 16.
*close*: 12, 14, 124.
*closedir*: 82.
*cmd*: 40, 67.
*connect*: 16, 17, 21, 48, 93, 100, 148.
*constraint*: 16, 50, 94, 98.
*count*: 24, 29.
*Countof*: 137.
*creat*: 66.
*d_fileno*: 76.
*d_name*: 76.
*d_namlen*: 76.
*d_reclen*: 76.
*define*: 76, 137.
*dev_t*: 54.
*DIR*: 77, 78, 79, 80, 81, 82.
*dirent*: 76, 78, 83.
*dirname*: 77.
*dirp*: 78, 79, 80, 81, 82.
*do_something*: 124.
*dom*: 16.
*dontUseChdir*: 137, 141.
*dup*: 36, 43.
*dup2*: 36, 44.
*EADDRINUSE*: 18.
*EADDRNOAVAIL*: 21.
*EAFNOSUPPORT*: 18, 21.
*EBADF*: 11, 34.
*ECONNREFUSED*: 21.
*EINPROGRESS*: 21.
*EINTR*: 11, 16.
*EINVAL*: 13, 16.
*EISCONN*: 21.
*EMFILE*: 13.
*ENAMETOOLONG*: 75.
*endif*: 137.
*endservent*: 112, 113.
*ENOMEM*: 11, 75.
*ENOTCONN*: 11, 20.
*EntityName*: 93.
*entries*: 83.
*EOPNOTSUPP*: 39, 40, 48.
*errno*: 11.
*ESPIPE*: 61.
*ev*: 136.
*EventRecord*: 136.
*EWOULDBLOCK*: 20, 23, 28.
*exceptfds*: 34.
*exceptfs*: 34.
*F_DUPFD*: 40.
*F_GETFL*: 40.
*F_SETFL*: 40.
*faccess*: 67.
*family*: 92, 93, 97.
*fcntl*: 36, 40, 52.
*fd*: 14, 43, 57, 60, 85.
*FD_CLR*: 35.
*FD_ISSET*: 35.
*FD_SET*: 35.
*fd_set*: 34.
*FD_ZERO*: 35.
*fds*: 35.

fgetfileinfo: 68.
file: 58, 59.
filename: 62, 63, 67, 68, 69.
fill: 137.
FIONBIO: 39.
FIONREAD: 39.
flags: 16, 25, 26, 27, 30, 31, 32, 65, 98, 99, 121.
FNDELAY: 40.
from: 26, 27, 128, 129, 130.
fromlen: 26, 27.
fsetfileinfo: 69.
FSSpec: 7.
fstat: 57.
ftruncate: 85.
getcwd: 1, 75.
gethostbyaddr: 107.
gethostbyname: 106.
gethostname: 110.
GetNextEvent: 135, 136.
getpeername: 36, 38, 48.
getprotobyname: 118.
getservbyname: 112, 115.
getservbyport: 112, 116.
getservent: 114, 115, 116.
getsockname: 36, 37, 48, 148.
getsockopt: 36, 41, 48.
GUSI_MAX_DOMAINS: 147.
GUSI_PREF_VERSION: 137, 138.
GUSIEvtHandler: 135, 136.
GUSIEvtHandlers: 136.
GUSIEvtTable: 135, 136.
GUSIGetEvents: 135.
GUSIGetSpin: 134.
GUSIRsrcID: 137.
GUSISetEvents: 135.
GUSISetSpin: 134.
GUSISIOWEvents: 136.
GUSISpinFn: 133, 134.
h_addr_list: 105.
h_addrtype: 105.
h_aliases: 105.
h_length: 105.
h_name: 105.
hostent: 105, 106, 107.
ifndef: 137.
in_addr: 103, 107, 108, 109.
inaddr: 108.
inet_addr: 109.
inet_ntoa: 108, 109.
InitCursor: 8.
InitDialogs: 8.
InitFonts: 8.
InitGraf: 8.
InitMenus: 8.
InitWindows: 8.
ino_t: 54.
integer: 137.
ioctl: 7, 36, 39, 52.
iov: 24, 29.
iov_base: 24.
iov_len: 24.
iovec: 24, 27, 29.
isatty: 60.
isTCPDaemon: 137, 143.
isUDPDaemon: 137, 143.
len: 124, 128, 129, 130, 131.
length: 84, 85.
level: 41, 42.
linkname: 70.
linkto: 70.
listen: 17, 19, 20, 48, 148.
literal: 137.
loc: 80.
location: 97.
LocationNameRec: 97.
longint: 137.
lseek: 61.
lstat: 56, 142.
machname: 110.
malloc: 75.
match: 98, 99.
MAXNAMLEN: 76.
mkdir: 72.
modtime: 58, 59.
MPS: 137.
MPSY: 139.
mpw: 137.
msg: 27, 32, 133.
msg_accrights: 27.
msg_accrightslen: 27.
msg_iov: 27.
msg_iovlen: 27.
msg_name: 27.
msg_namelen: 27.
MSG_OOB: 25, 30.
msghdr: 27, 32.
name: 16, 18, 21, 37, 38, 50, 66, 93, 99, 106, 115, 118.
namelen: 18, 37, 38.
nbpType: 98, 99.

*newcreator*: 68, 69.
*newfd*: 44.
*newname*: 64.
*newtype*: 68, 69.
*nil*: 8.
*nlen*: 16.
*NLType*: 94.
*noAutoSpin*: 137.
*noTCPDaemon*: 137, 143.
*noUDPDaemon*: 137, 143.
*NULL*: 20, 26, 34, 75, 77, 78, 83, 106, 107, 114, 117.
*numTypes*: 50, 94.
*O_ALIAS*: 65, 148.
*O_APPEND*: 65.
*O_CREAT*: 65, 66.
*O_EXCL*: 65.
*O_NRESOLVE*: 148.
*O_RDONLY*: 65.
*O_RDWR*: 65, 124.
*O_RSRC*: 65.
*O_TRUNC*: 65, 66.
*O_WRONLY*: 66.
*off_t*: 54, 84, 85.
*oldfd*: 44.
*oldname*: 64.
*open*: 65, 66, 120, 121, 124, 147.
*opendir*: 77.
*optlen*: 41, 42.
*optname*: 41, 42.
*optval*: 41, 42.
*p_aliases*: 117.
*p_name*: 117.
*p_proto*: 117.
*param*: 133.
*path*: 54, 56, 71, 72, 73, 74, 83, 84.
*PBHSetVol*: 141.
*PBSetVol*: 141.
*port*: 97, 116.
*portType*: 99.
*PPC_CON_MATCH_NAME*: 99.
*PPC_CON_MATCH_NBP*: 99.
*PPC_CON_MATCH_TYPE*: 99.
*PPC_CON_NEWSTYLE*: 99.
*PPCPortRec*: 97, 98.
*prompt*: 16.
*proto*: 115, 116.
*protocol*: 13.
*protoent*: 117, 118.
*Ptr*: 8.
*qd*: 8.
*qlen*: 19.
*qsort*: 83.
*read*: 22, 23, 25, 28, 124, 140, 147.
*readdir*: 78.
*readfds*: 34.
*readfs*: 34.
*readlink*: 1, 71.
*readv*: 22, 24, 27.
*recv*: 22, 25, 26, 49, 140.
*recvfrom*: 22, 26, 27, 49, 147.
*recvmsg*: 22, 27, 49.
*remove*: 62, 63.
*rename*: 64.
*request*: 39.
*rewinddir*: 81.
*rmdir*: 73.
*routine*: 134.
*s_addr*: 103.
*s_aliases*: 111.
*S_IFCHR*: 55.
*S_IFDIR*: 55.
*S_IFLNK*: 55.
*S_IFREG*: 55.
*S_IFSOCK*: 55, 57.
*s_name*: 111.
*s_port*: 111.
*s_proto*: 111.
*sa_constr_atlk*: 94.
*sa_constr_file*: 50.
*sa_constr_ppc*: 98.
*sade*: 139.
*scandir*: 83.
*seekdir*: 80.
*select*: 20, 21, 34, 51.
*send*: 22, 30, 31, 49, 140.
*sendmsg*: 22, 32, 49.
*sendto*: 22, 31, 32, 49, 147.
*servent*: 111, 114, 115, 116.
*setservent*: 112.
*setsockopt*: 36, 42, 48.
*SFTypeList*: 50.
*shutdown*: 124.
*sin_addr*: 103.
*sin_family*: 103.
*sin_len*: 103.
*sin_port*: 103.
*sin_zero*: 103.
*size*: 75.
*SOCK_DGRAM*: 13.

*SOCK_STREAM*: 13.
*sockaddr*: 18, 20, 21, 37, 38.
*sockaddr_atlk*: 92.
*sockaddr_atlk_sym*: 93.
*sockaddr_in*: 103.
*sockaddr_ppc*: 97.
*sockaddr_un*: 88.
*Socket*: 147.
*socket*: 12, 13, 16, 48, 120, 147.
*SocketDomain*: 147.
*sort*: 83.
*SP_ADDR*: 133.
*SP_AUTO_SPIN*: 133.
*SP_DGRAM_READ*: 133.
*SP_DGRAM_WRITE*: 133.
*SP_MISC*: 133.
*SP_NAME*: 133.
*SP_SELECT*: 133.
*SP_SLEEP*: 133.
*SP_STREAM_READ*: 133.
*SP_STREAM_WRITE*: 133.
*spin_msg*: 133.
*SpinCursor*: 140.
*st_atime*: 54.
*st_blksize*: 54.
*st_blocks*: 54.
*st_ctime*: 54.
*st_dev*: 54.
*st_gid*: 54.
*st_ino*: 54.
*st_mode*: 54, 55, 57.
*st_mtime*: 54.
*st_nlink*: 54, 142.
*st_rdev*: 54.
*st_size*: 54.
*st_uid*: 54.
*stat*: 54, 56, 57, 142.
*stayopen*: 112.
*Str32*: 98.
*SuffixArray*: 137.
*sun_family*: 88.
*sun_path*: 88.
*SYM*: 139.
*symlink*: 1, 70.
*s1*: 131.
*s2*: 131.
*table*: 135.
*tcpDaemon*: 145.
*TEInit*: 8.
*telldir*: 79.
*text*: 137.
*TEXT*: 137.
*thePort*: 8.
*tim*: 58, 59.
*time_t*: 54, 58.
*timeout*: 34.
*timeval*: 34.
*to*: 31, 130.
*tolen*: 31.
*truncate*: 84.
*type*: 13, 16.
*types*: 50, 94.
*u_char*: 103.
*u_long*: 76, 103.
*u_short*: 54, 76, 103.
*udpDaemon*: 145.
*unlink*: 63.
*useChdir*: 137, 141.
*utimbuf*: 58.
*utime*: 58.
*WaitNextEvent*: 135, 136.
*want*: 83.
*wide*: 137.
*width*: 34.
*WR_ONLY*: 65.
*write*: 22, 28, 30, 124, 140, 147.
*writefds*: 34.
*writefs*: 34.
*writev*: 22, 29, 32.

# GUSI — Grand Unified Socket Interface


| | Section | Page |
|---|---|---|
| **Introduction** | 1 | 1 |
| Copying | 2 | 1 |
| Design Objectives | 3 | 1 |
| Acknowledgements | 4 | 1 |
| **Installing and using GUSI** | 5 | 2 |
| **Overview** | 11 | 4 |
| Creating and destroying sockets | 12 | 4 |
| Prompting the user for an address | 15 | 4 |
| Establishing connections between sockets | 17 | 5 |
| Transmitting data between sockets | 22 | 6 |
| I/O multiplexing | 33 | 7 |
| Getting and changing properties of sockets | 36 | 7 |
| **Detail Description** | 45 | 9 |
| File system calls | 46 | 9 |
| Differences to generic behavior | 47 | 9 |
| Routines specific to the file system | 53 | 9 |
| Unix domain sockets | 86 | 12 |
| Differences to generic behavior | 87 | 12 |
| Appletalk sockets | 90 | 13 |
| Differences to generic behavior | 91 | 13 |
| PPC sockets | 95 | 13 |
| Differences to generic behavior | 96 | 13 |
| Internet sockets | 101 | 14 |
| Differences to generic behavior | 102 | 14 |
| Routines specific to TCP/IP sockets | 104 | 14 |
| PAP sockets | 119 | 16 |
| Routines specific to PAP sockets | 120 | 16 |
| Miscellaneous | 126 | 16 |
| BSD memory routines | 127 | 16 |
| Blocking calls | 132 | 17 |
| Resources | 137 | 18 |
| **Advanced techniques** | 144 | 20 |
| Writing daemons | 145 | 20 |
| FSSpec routines | 146 | 20 |
| Adding your own communication families | 147 | 20 |
| **Known problems, words of warning** | 148 | 21 |
| **References** | 149 | 22 |
| **Index** | 150 | 23 |