GUSI 2 Reference Manual

Version 2.1
30−May−2000

Author: Matthias Neeracher

blank

Introduction

GUSI is a POSIX library for MacOS. Its name, which is an acronym for *Grand Unified Socket Interface*, hints at its original objective to provide access to all the various communications facilities in MacOS through a common, file descriptor based, interface.

The current incarnation, GUSI 2, represents a much−needed rewrite of GUSI and introduces support for POSIX threads.

The most recent version of GUSI may be obtained by anonymous ftp from ***ftp://sunsite.cnlab−switch.ch/software/platforms/macos/src/mw_c***.

There is also a mailing list devoted to discussions about GUSI. You can join the list by sending a mail to ***gusi−request@iis.ee.ethz.ch*** whose body consists of the word subscribe.

User's Manual

For ease of access, the manual has been split up into a number of sections:

```
GUSI_Install   Installing and using the GUSI headers and libraries
GUSI_Common    Routines common to all file descriptors.
GUSI_Files     Routines specific to disk based file descriptors.
GUSI_Sockets   Routines specific to network descriptors.
GUSI_Threads   Routines to manage multiple threads of execution in a program.
GUSI_Misc      Miscellaneous routines
```

GUSI User License

My primary objective in distributing GUSI is to have it used as widely as possible, while protecting my moral rights of authorship and limiting my exposure to liability.

Making Matthias Happy

While I am giving GUSI away for free, that does not mean that I don't like getting appreciation for it. If you want to do something for me beyond your obligations outlined above, you can

- Acknowledge the use of GUSI in the about box of your application and/or your documentation.
- Send me a CD as described in ***http://www.iis.ee.ethz.ch/~neeri/macintosh/donations.html***

Design Objectives

The primary objective of GUSI is to emulate as much as practical of the UNIX 98 API for the use in MacOS programs. This is in marked contrast to other approaches which at first glance might seem similar:

- GUSI is *not* designed for optimal performance of network communication (although GUSI][should be faster than GUSI 1 for many purposes). The design goal is to make the code as fast as possible without changing the POSIX API (e.g., by exposing interrupt level code to the library user).
- GUSI is *not* designed for maximal compliance with the POSIX API either. The goal is to provide as much functionality and as faithful implementation as possible while maintaining a strict library approach without writing a separate operating system.

While the original GUSI design had to appeal to nebulous "standards" eclectically drawn from POSIX and

BSD APIs, the underlying APIs have now evolved into real standards, so GUSI 2 now tries to conform to the *X/Open Single Unix Specification, Version 2* (also known as UNIX 98) as much as possible.

Changes between GUSI 1 and GUSI 2

I'm sure there must be more incompatibilities. If you find any, let me know.

- The `choose()` function has been dropped. I don't think it ever provided a benefit that outweighed the namespace intrusion.
- `inet_addr()` now returns a scalar instead of a `string in_addr`, correcting a bizzare misinterpetation of the documentation on my part.
- The BSD only `scandir()` function has been dropped.
- Many adjustments were made for UNIX 98 compatibility:

  `select()` is now in ***sys/time.h***.
  Socket length arguments are now of type `socklen_t`.
- You no longer have to call the GUSISetup routines directly, GUSI will call them for you and you just have to link a configuration file.
- The configuration file also handles most of what you needed a configuration resource for, so you normally no longer use one.

Literature

This manual is by no means a complete reference, let alone a suitable tutorial for the APIs covered. I found the following books the best texts in their fields:

- For POSIX programming in general:

  ```
  Advanced Programming in the UNIX Environment
  W. Richard Stevens, Addison−Wesley
  ```

- For socket programming:

  ```
  UNIX Network Programming, 2nd ed.
  W. Richard Stevens, Prentice Hall
  ```

- For threads programming:

  ```
  Programming with POSIX Threads
  David R. Butenhof, Addison−Wesley
  ```

- As a comprehensive UNIX 98 reference:

  ```
  CAE Specification: System Interfaces and Headers Issue 5
  The Open Group
  ```

  A free online version of this document, as well as ordering information for the (expensive) hardcopy edition, is available at ***http://www.opengroup.org/pubs/catalog/t912.htm***

Acknowledgements

GUSI has over its existence profited from numerous suggestions and code contributions. Where possible, I have tried to give credit to contributors in the README file.

Although probably no trace of it remains in today's code, GUSI grew from a socket library written by Charlie Reiman.

Many of the header files in the ***:include:*** subdirectory are adapted from BSD 4.4−lite.

Installing and using GUSI

This section discusses how you can install GUSI on your disk and use it for your programs. Experience has shown that **understanding these instructions** is an absolutely **critical step** in using GUSI and that most of the problems in using it (apart from those due to inadequacies of GUSI itself) are caused by installation and configuration problems.

Installing GUSI

To install GUSI, simply unpack the distribution archive and put the GUSI folder somewhere on your disk. If you intend to use GUSI with the MPW shell and/or the SC/SCpp or MrC/MrCpp compilers, run the GUSI_Install.MPW script from MPW.

To use GUSI in a Metrowerks CodeWarrior project

- Open your project in the CodeWarrior IDE.
- Open the XXX Settings... dialog in the Edit menu and select the Access Paths panel.
- Add the ***include*** folder in the ***GUSI*** folder at the **top** of the System Paths section of the panel (Click in System Paths and click the Add... button). It is very important that this folder appears **above** the Metrowerks Standard Library.
- Turn off recursive searching in this folder by clicking on the column to the left of the name to make the folders icon disappear. **This step is crucial**.
- Turn on the Interpret DOS and UNIX Paths checkbox in the top right of the panel.
- Add appropriate GUSI libraries to your project, paying careful attention to the **link order**.

To use GUSI with the Metrowerks compilers for MPW

- Add the ***include*** folder to the commandline with

```
MWCxxx ... -i- -i "{GUSI}include:"
```

  The -i- option should occur **after** other includes but **before** other standard include directories listed on the command line.

- Add the appropriate GUSI libraries to the link.

To use GUSI with the standard MPW compilers (SC/SCpp and MrC/MrCpp

- Add the ***include*** folder to the commandline with

```
(SCxx|MrCxx) ... -includes unix -i "{GUSI}include:"
```

  These options should occur **before** other standard include directories listed on the command line. The -includes unix option tells your compiler that headers in subdirectories will be named by UNIX path name rules.

- Add the appropriate GUSI libraries to the link.

GUSI Header Files

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

arpa/inet.h

Converting between internet addresses and their numeric string representations.

dirent.h

Routines to access all entries in a directory.

errno.h

The error codes returned by GUSI routines.

fcntl.h

Operations on files and flag constants for them.

inttypes.h

Integer types with guaranteed sizes.

netdb.h

Looking up TCP/IP host names.

netinet/in.h

The address format for TCP/IP sockets.

pthread.h

Operations on threads.

sched.h

Scheduling operations.

sys/ioctl.h

Codes to pass to `ioctl()`.

sys/socket.h

Data types for socket calls.

sys/stat.h

Getting information about files.

sys/time.h

Operations with time and timers.

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.

utime.h

Getting the modification time of a file.

GUSI Libraries

At link time, you will have to link with the appropriate `GUSI` libraries. All of the libraries contain a suffix to identify the compilers for which they are suitable:

XXX.68K.Lib

Libraries suitable for the Metrowerks 68K C/C++ compilers.

XXX.PPC.Lib

Libraries suitable for the Metrowerks PPC C/C++ compilers.

XXX.SC.Lib

Libraries suitable for the MPW SC/SCpp compilers.

XXX.MrC.Lib

Libraries suitable for the MPW MrC/MrCpp compilers.

Typically, you should link with three component libraries, in this order:

- A library specifying the output console, such as:

  GUSI_MPW.XXX.Lib

  For MPW tools.

  GUSI_SIOUX.XXX.Lib

  For programs writing to the SIOUX console.

- A library specifying the high level stdio library, such as:

  GUSI_MSL.XXX.Lib

  For programs using the Metrowerks Standard Library stdio.

  GUSI_Stdio.SC.Lib

  For programs using the MPW stdio. This option is currently only available for `SC/SCpp`, for `MrC/MrCpp`, you have to use the `sfio` library for `stdio` support. Alternatively, you can choose to exclusively use the POSIX layer functions `(read/write/close)` for sockets and the stdio functions `(fscanf/fwrite/fclose)` for files. In this case, be careful **never** to use `fileno` or `fdopen`.

  GUSI_Sfio.XXX.Lib

  For programs using the `sfio` library. `sfio` is a new I/O library developed by `AT&T` with a source compatibility option with `stdio`. As it covers only the `stdio` part of the ANSI library, you will also have to link with standard library for your compiler. Make sure to specify `sfio` first in your link order, though.

- The `GUSI` library itself, i.e.:

  GUSI_Core.XXX.Lib

These libraries should appear **in this order, before any other libraries**. This may sometimes not be practicable, especially for 68K MPW tools. For this case, it is possible to substitute ***GUSI_Forward.68K*** in the place of ***GUSI_Core.68K*** and use ***GUSI_Core.68K*** later in the link order (usually last).

Sometimes, you want to use `GUSI` with threads created by another library, such as `PowerPlant`. For this purpose, you can additionally specify the ***GUSI_ForeignThreads.XXX.Lib*** library before any other `GUSI` library and before ***ThreadsLib***. This is sufficient for CFM applications; for non−CFM 68K applications, however, you also have to recompile the third party code while including ***GUSIForeignThreads.h*** (e.g., using a precompiled header). While this state of things is not entirely satisfactory, I don‘t see a better technique at the moment.

In addition, you will need to link with a considerable list of standard compiler libraries. Since `GUSI` is written in C++, you will also need C++ support libraries. As an example, the Open Transport MPW test tools are linked with the following libraries:

Metrowerks 68K

```
"{MW68KLibraries}MSL MPWRuntime.68K.Lib"
"{MW68KLibraries}MSL Runtime68K.Lib"
"{MW68KLibraries}MacOS.Lib"
"{MW68KLibraries}MSL C.68K MPW(NL_4i_8d).Lib"
"{MW68KLibraries}MSL C++.68K (4i_8d).Lib"
"{MW68KLibraries}MathLib68K (4i_8d).Lib"
"{MW68KLibraries}ToolLibs.o"
"{MW68KLibraries}PLStringFuncs.glue.lib"
```

```
"{MW68KLibraries}OpenTransportApp.o"
"{MW68KLibraries}OpenTransport.o"
"{MW68KLibraries}OpenTptInet.o"
```

Metrowerks PPC

```
"{MWPPCLibraries}MSL MPWCRuntime.Lib"
"{MWPPCLibraries}MSL RuntimePPC.Lib"
"{SharedLibraries}InterfaceLib"
"{MWPPCLibraries}MSL C.PPC MPW(NL).Lib"
"{MWPPCLibraries}MSL C++.PPC (NL).Lib"
"{SharedLibraries}MathLib"
"{SharedLibraries}ThreadsLib"
"{MWPPCLibraries}PPCToolLibs.o"
"{MWPPCLibraries}PLStringFuncsPPC.lib"
"{SharedLibraries}OpenTransportLib"
"{SharedLibraries}OpenTptInternetLib"
"{MWPPCLibraries}OpenTransportAppPPC.o"
"{MWPPCLibraries}OpenTptInetPPC.o"
```

SC

```
"{CLibraries}CPlusLib.o"
"{CLibraries}StdCLib.o"
"{Libraries}MacRuntime.o"
"{Libraries}Interface.o"
"{Libraries}IntEnv.o"
"{Libraries}MathLib.o"
"{Libraries}ToolLibs.o"
"{CLibraries}IOStreams.far.o"
"{Libraries}OpenTransport.o"
"{Libraries}OpenTransportApp.o"
"{Libraries}OpenTptInet.o"
```

MrC

```
"$(SFIO)lib:Sfio.MrC.Lib"
"{PPCLibraries}MrCPlusLib.o"
"{PPCLibraries}PPCStdCLib.o"
"{PPCLibraries}StdCRuntime.o"
"{PPCLibraries}PPCCRuntime.o"
"{PPCLibraries}PowerMathLib"
"{PPCLibraries}PPCToolLibs.o"
"{SharedLibraries}InterfaceLib"
"{SharedLibraries}ThreadsLib"
"{PPCLibraries}MrCIOStreams.o"
"{SharedLibraries}StdCLib"
"{SharedLibraries}OpenTransportLib"
"{SharedLibraries}OpenTptInternetLib"
"{PPCLibraries}OpenTransportAppPPC.o"
"{PPCLibraries}OpenTptInetPPC.o"
```

Configuration

You will need to specify what `GUSI` facilities you want to use in your application. This is done with three functions calling configuration hooks.

`void GUSISetupFactories()`

Sets up communications facilities accessible via sockets.

`void GUSISetupDevices()`

Sets up facilities accessible via special file names.

`void GUSISetupConfig()`

Sets up various configuration flags. Use this if you don‘t want to use a configuration resource (See *Resources*).

These hooks can conveniently be created and edited via the ***GUSIConfig*** application. ***GUSIConfig*** saves a C++ file which you should then compile and link to your application. If you want to write a configuration file manually, work from the templates in ***:test:GUSIConfig_MTInet.cp*** and ***:test:GUSIConfig_OTInet.cp***.

Because the configuration file has to include internal GUSI headers, it should **not** be compiled when a precompiled header including any internal GUSI headers or ***pthread.h*** is in effect. If necessary, compile your configuration file in a separate target.

Initializing the Macintosh Toolbox

GUSI expects the Macintosh Toolbox to be initialized. You should initialize the Toolbox in the following way:

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

However, `GUSI` will initialize QuickDraw automatically, which obviates the need to initialize the Toolbox if all you want to do is a basic MPW tool.

Resources

Under some (rare) circumstances, you might also want to rez your program with `GUSI.r`. The section *Resources* discusses when and how to add your own configuration resource to customize `GUSI` defaults.

Warning messages

You will get lots of warning messages about duplicate definitions, but that‘s ok (Which means I can‘t do anything about it).

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 or some positive value.

Here‘s a list of all error codes and their typical explanations. The most important of them are repeated for the individual calls.

`EACCES`

Permission denied: An attempt was made to access a file in a way forbidden by its file access permissions, e.g., to `open()` a locked file for writing.

`EADDRINUSE`

Address already in use: `bind()` was called with an address already in use by another socket.

`EADDRNOTAVAIL`

Can‘t assign requested address: `bind()` was called with an address which this socket can‘t assume, e.g., a TCP/IP address whose `in_addr` specifies a different host.

`EAFNOSUPPORT`

Address family not supported: You haven‘t linked with this socket family or have specified a nonexisting family, e.g., AF_CHAOS.

`EALREADY`

Operation already in progress, e.g., `connect()` was called twice in a row for a nonblocking socket.

`EBADF`

Bad file descriptor: The file descriptor you specified is not open.

`EBUSY`

Request for a system resource already in incompatible use, e.g., attempt to delete an open file.

`ECONNREFUSED`

Connection refused, e.g. you specified an unused port for a `connect()`

`EEXIST`

File exists, and you tried to open it with `O_EXCL`.

`EHOSTDOWN`

Remote host is down.

`EHOSTUNREACH`

No route to host.

`EINPROGRESS`

Operation now in progress. This is *not* an error, but returned from nonblocking operations, e.g., nonblocking `connect()`.

`EINTR`

Interrupted system call: The user pressed Command−. or `alarm()` timed out.

`EINVAL`

Invalid argument or various other error conditions.

`EIO` Input/output error.

`EISCONN`

Socket is already connected.

`EISDIR`
: Is a directory, e.g. you tried to `open()` a directory.

`EMFILE`
: Too many open files.

`EMSGSIZE`
: Message too long, e.g. for an UDP `send()`.

`ENAMETOOLONG`
: File name too long.

`ENETDOWN`
: Network is down, e.g., Appletalk is turned off in the chooser.

`ENFILE`
: Too many open files in system.

`ENOBUFS`
: No buffer space available.

`ENOENT`
: No such file or directory.

`ENOEXEC`
: Severe error with the PowerPC standard library.

`ENOMEM`
: Cannot allocate memory.

`ENOSPC`
: No space left on device.

`ENOTCONN`
: Socket is not connected, e.g., neither `connect()` nor `accept()` has been called successfully for it.

`ENOTDIR`
: Not a directory.

`ENOTEMPTY`
: Directory not empty, e.g., attempt to delete nonempty directory.

`ENXIO`
: Device not configured, e.g., MacTCP control panel misconfigured.

`EOPNOTSUPP`
: Operation not supported on socket, e.g., `sendto()` on a stream socket.

`EPFNOSUPPORT`
: Protocol family not supported, i.e., attempted use of ADSP on a machine that has AppleTalk but not ADSP.

`EPROTONOSUPPORT`
: Protocol not supported, e.g., you called `getprotobyname()` with neither "tcp" nor "udp" specified.

`ERANGE`
: Result too large, e.g., `getcwd()` called with insufficient buffer.

EROFS

Read−only file system.

ESHUTDOWN

Can‘t send after socket shutdown.

ESOCKTNOSUPPORT

Socket type not supported, e.g., datagram PPC toolbox sockets.

ESPIPE

Illegal seek, e.g., `lseek()` called for a TCP socket.

EWOULDBLOCK

Nonblocking operation would block.

EXDEV

Cross−device link, e.g. `FSpSmartMove()` attempted to move file to a different volume.

Creating and destroying sockets

A socket is created with `socket()` and destroyed with `close()`. In some situations, it‘s useful to create a pair of connected sockets with `socketpair()` or `pipe()`. You can gradually shut down data transfer with `shutdown()`.

`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
AF_LOCAL

Communication internal to a single Mac.

AF_INET

TCP/IP, using `MacTCP` or `Open Transport` depending on your configuration.

AF_APPLETALK

Appletalk, using the ADSP and DDP protocols (not implemented yet in GUSI 2).

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.

`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.

`shutdown(int how)`

if `how` is `SHUT_RD`(0), shut down the socket for reading, for `SHUT_WR`(1), shut down for writing, and for `SHUT_RDWR`, shut down for both reading and writing.

`int socketpair(int domain, int type, int protocol, int fds[2])`

creates, in `fds[0]` and `fds[1]`, an unnamed pair if indistinguishable sockets in the indicated domain (currently only `AF_LOCAL` is accepted).

`int pipe(int fds[2])`

is a shorthand notion for

```
socketpair(AF_LOCAL, SOCK_STREAM, 0, fds)
```

but fds[0] will be read−only and fds[1] will be write only.

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.

`int bind(int s, const struct sockaddr *name, socklen_t namelen)`

binds a socket to its address. The format of the address is different for every socket family.

Error codes:

`EAFNOSUPPORT`

`name` specifies an illegal address family for this socket.

`EADDRINUSE`

There is already another socket with this address.

`int listen(int s, int qlen)`

turns a socket into a listener. `qlen` determines how many clients can concurrently wait for a connection.

`int accept(int s, struct sockaddr *addr, socklen_t *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.

`int connect(int s, const struct sockaddr *addr, socklen_t 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.

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()`.

`int read(int s, void *buffer, size_t 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.

`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`Êarray, always filling one buffer completely before proceeding to the next. `iovec` is defined as follows:

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

`int recv(int s, void *buffer, size_t 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.

`int recvfrom(int s, void *buffer, size_t buflen, int flags, struct sockaddr *from, socklen_t *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.

`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 i
    int      msg_accrightslen;
};
```

`int write(int s, void *buffer, size_t 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.

`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`Êarray, always sending one buffer completely before proceeding to the next.

`int send(int s, void *buffer, size_t 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.

`int sendto(int s, void *buffer, size_t buflen, int flags, struct sockaddr *to, socklen_t tolen)`

is the equivalent of `send()` for unconnected datagram sockets. The message will be sent to the socket whose address is given in `to`.

`int sendmsg(int s, const struct msghdr *msg, int flags)`

combines the possibilities of `writev()` and `sendto()`.

I/O multiplexing

`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.

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 */
```

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()`.

`int getsockname(int s, struct sockaddr *name, socklen_t *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.

`int getpeername(int s, struct sockaddr *name, socklen_t *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.

`int ioctl(int d, unsigned int request, ...)`

performs various operations on the socket, depending on the `request`. The following codes are valid for all socket families:

`ioctl(d, FIONBIO, int * argp)`

Make the socket blocking if the `int` pointed to by argp is , else make it nonblocking.

`ioctl(d, FIONREAD, int * argp)`

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.

`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 greater than or equal to `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 `O_NONBLOCK` (Also known under its older name `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.

`int getsockopt(int s, int level, int optname, void *optval, int * optlen)`

`int setsockopt(int s, int level, int optname, void *optval, int optlen)`

are used to get and set options associated with a socket. The following options are implemented (many of them only for OpenTransport sockets, though):

Level SOL_SOCKET:

SO_BROADCAST
: permit sending of broadcast datagrams.

SO_DONTROUTE
: bypass routing table lookup.

SO_ERROR
: get pending asynchronous error.

SO_KEEPALIVE
: periodically test if connection is still alive.

SO_LINGER
: linger on `close()` if there is data to send.

SO_RCVBUF
: manipulates the size of the buffer used for reading data.

SO_SNDBUF
: manipulates the size of the buffer used for writing data.

SO_RCVLOWAT
: receive low−water mark.

SO_SNDLOWAT
: send low−water mark.

SO_REUSEADDR
SO_REUSEPORT
: allow local address reuse.

Level IPPROTO_IP:

IP_TOS
: type−of−service and precedence.

IP_TTL
: time−to−live.

IP_MULTICAST_IF
: specify outgoing interface.

IP_MULTICAST_TTL
: specify outgoing time−to−live.

IP_MULTICAST_LOOP
: specify loopback.

IP_ADD_MEMBERSHIP
: join a multicast group.

IP_DROP_MEMBERSHIP
: leave a multicast group.

Level IPPROTO_TCP:

TCP_KEEPALIVE
: seconds between keepalive probes.

`TCP_MAXSEG`

TCP maximum segment size.

`TCP_NODELAY`

disable Nagle algorithm.

`optval` is a pointer to an unsigned integer in both cases.

`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.

`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.

Socket Family Specific Interfaces

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. Internet sockets are also suited for interapplication communication on a single machine, provided it runs MacTCP or Open Transport.

Internet socket addresses have the following format:

```
struct in_addr {
  u_long s_addr;
};

struct sockaddr_in {
   u_short  sin_family;       /* Always C<AF_INET> */
   u_short  sin_port;         /* Port number */
   struct   in_addr sin_addr; /* Host ID */
   char     sin_zero[8];
};
```

There are many routines to convert between numeric and symbolic addresses.

- 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 t
  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
};
```

`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.

`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.

`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"`)

`in_addr_t inet_addr(char *address)`
`int inet_aton(const char * addr, struct in_addr * ina)`

convert a numeric string into an internet address (If `x` is a valid address, `inet_addr(inet_ntoa(x)) == x`).

`int gethostname(char *machname, long buflen)`

gets our name into `buffer`.

- 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") */
};
```

`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()`.

`void endservent()`

closes the file of services.

`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`.

`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`.

`struct servent * getservbyport(int port, const char * proto)`

finds a service by calling `getservent()` until the protocol matches `proto` and the port matches `port`.

- 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 */
};
```

`struct protoent * getprotobyname(char * name)`

finds a named protocol. This call is rather unexciting.

`struct protoent * getprotobynumber(int number)`

does the reverse lookup. This call is even less exciting.

For OpenTransport TCP/IP sockets, there are a number of `ioctl` calls to obtain information about the available interfaces.

`SIOCGIFCONF`

stores the list of interfaces in the `struct ifconf` pointed to by the third parameter. Note that an entry is created for each alias address.

`SIOCGIFADDR`

Return the address of the interface named by the `struct ifreq` pointed to by the third parameter in that structure.

`SIOCGIFFLAGS`

Return the flags for the interface named by the `struct ifreq` pointed to by the third parameter in that structure.

`SIOCGIFBRDADDR`

Return the broadcast address of the interface named by the `struct ifreq` pointed to by the third parameter in that structure.

`SIOCGIFNETMASK`

Return the subnet mask of the interface named by the `struct ifreq` pointed to by the third parameter in that structure.

PPC sockets

These provide authenticated stream sockets without out−of−band data. PPC sockets should work between all networked Macintoshes running System 7 or later, and between applications on a single Macintosh running System 7 or later.

PPC socket addresses have the following format:

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

In addition, the following behavior in PPC sockets differs from the standard:

- `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.

File system calls

Files are unlike sockets in many respects: They can be rewound and re−read several times. `write()` calls can directly influence the results of subsequent `read()` calls. There are also many calls which are specific to files.

Differences to generic behavior

- The following calls make no sense for files and return an error of `EOPNOTSUPP`:

```
socket()
bind()
listen()
accept()
connect()
getsockname()
getpeername()
```

- 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()
```

Routines specific to the file system

In this section, you‘ll meet lots of good old friends. Some of these routines also exist in the standard compiler libraries, but the `GUSI` versions have a few differences:

- File names are relative to the directory specified by `chdir()`.
- You can define special treatment for some file names (See below under "Adding your own file families").

`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;
};
```

`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.

`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' or 'appe'. The definition of executability can be customized with the `GUSI_ExecHook` discussed in the advanced section.

`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.

`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.

`int chmod(const char * filename, mode_t mode)`

changes the mode returned by `stat()`. Currently, the only thing you can do with `chmod()` is to turn the write permission off and on. This is translated to setting and clearing the file lock bit.

=item `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 */
};
```

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

`int isatty(int fd)`

returns `1` if `fd` represents a terminal (i.e. is connected to `"Dev:Stdin"` and the like), otherwise.

`long lseek(int, long, int)`

changes the read/write position in an open file, and will return `ESPIPE` if called for a socket. If `lseek()` sets the position beyond EOF, the gap will be filled with 0 bytes if a `write()` is subsequently called at the position.

`int remove(const char *filename)`

removes the named file. If `filename` is a symbolic link, the link will be removed and not the file.

`int unlink(const char *filename)`

is identical to `remove()`.

`int rename(const char *oldname, const char *newname)`

renames and/or moves a file. `oldname` and `newname` must specify the same volume, but they may specify different folders.

`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.

`O_WRONLY`

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).

`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.

`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.

`void fgetfileinfo(char *filename, unsigned long *newcreator, unsigned long *newtype)`

returns the file type and creator of a file.

`void fsetfileinfo(char *filename, unsigned long newcreator, unsigned long newtype)`

sets the file type and creator of a file to the given values.

`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 resemblances to `UNIX` symbolic links, especially once you start renaming files.

`int readlink(const char* path, char* buf, int bufsiz)`

returns in `buf` the name of the file that `path` points to.

`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.

`int ftruncate(int fd, off_t length)`

does the same thing with an open file.

`int access(const char * path, int mode)`

tests if you have the specified access rights to a file. `mode` may be either `F_OK`, in which case the file is tested for existence, or a combination of the following:

`R_OK`

Tests if file is readable.

`W_OK`

Tests if file is writeable.

`X_OK`

Tests if file is executable. As with `stat()`, the definition of executability may be customized.

`access()` returns 0 if the specified access rights exist, otherwise it sets `errno` and returns −1.

`int mkdir(const char * path, ...)`

creates a new directory.

`int rmdir(const char * path)`

deletes an empty directory.

`int chdir(const char * path)`

makes all future partial pathnames relative to this directory.

`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.

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

```
struct dirent {
  ino_t    d_ino;                          /* file number of entry */
#define MAXNAMLEN   255
  char  d_name[MAXNAMLEN + 1];  /* name must be no longer than this */
};
```

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

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

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

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

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

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

Threading support

One of the major features new to GUSI 2 is a fairly complete implementation of the POSIX threads API on top of the MacOS thread manager. This section discusses the thread API, but is not intended to be a comprehensive reference on POSIX threads in general. Refer to the literature list for good books.

Principles of thread support

GUSI threads are based on cooperative MacOS threads. This means that threads will never get preempted executing a compute−bound loop. The only conditions under which they get preempted are

- When they explicitly request a thread switch by calling `sched_yield()`.
- When they call any GUSI library routine that does not complete immediately.

In practice, the second condition makes GUSI threading appear quite natural, such that, especially in code like network servers, explicit yields are rarely necessary.

Each thread gets some independent context, including its own stack and its own copies of the `errno` and `h_errno` variables. However, all threads share other resources, including memory, MacOS toolbox elements, and file descriptors.

Thread Data Types

All thread related data types are opaque. They have no structure known to the public and are only manipulated through procedure calls.

`pthread_t`

A thread identifier.

`pthread_attr_t`

An object collecting attributes specified at the creation of a thread.

`pthread_key_t`

An identifier for a piece of thread specific data.

`pthread_once_t`

A flag registering whether a *once routine* has already executed or not.

`pthread_mutex_t`

A mutual exclusion variable.

`pthread_mutexattr_t`

Creation attributes for a `pthread_mutex_t`.

`pthread_cond_t`

A condition variable.

`pthread_condattr_t`

Creation attributes for a `pthread_cond_t`.

Manipulating Threads

```
int pthread_create(pthread_t *th, const pthread_attr_t *attr, void
    *(*proc)(void *), void *arg)
```

Create a new thread and make `th` a reference to it. `attr` (which can be `NULL`) specifies creation attributes, `proc` specifies the code to execute in the thread, and `arg` is an initial argument to be passed to the code.

```
pthread_t pthread_self()
```

Returns the currently executing thread.

`int pthread_equal(pthread_t t1, pthread_t t2)`

Compares two threads for identity.

`int sched_yield()`

Yields the CPU to the next eligible thread.

`int pthread_join(pthread_t th, void **value)`

Wait for the thread to die and return its result if `value` is not `NULL`.

`int pthread_detach(pthread_t th)`

Declare that we will never call `pthread_join()` for this thread and that it simply should go away when done.

`int pthread_exit(void *value)`

Terminate the current thread, giving the specified return value.

`int pthread_attr_init(pthread_attr_t * attr)`

Initialize a thread attribute object with the default settings.

`int pthread_attr_destroy(pthread_attr_t * attr)`

Delete a thread attribute object.

```
int pthread_attr_setdetachstate(pthread_attr_t * attr, int state)
int pthread_attr_getdetachstate(pthread_attr_t * attr, int * state)
```

If `state` is `PTHREADS_CREATE_JOINABLE` (the default), `pthread_join()` should eventually called on the thread. If `state` is `PTHREADS_CREATE_DETACHED`, the thread is created detached.

```
int pthread_attr_setstacksize(pthread_attr_t * attr, size_t size)
int pthread_attr_getstacksize(pthread_attr_t * attr, size_t * size)
```

Manipulates the size of the stack allocated for the thread (20K default). Be sure to choose this size carefully and generously, as stack overflows will lead to nasty crashes.

Manipulating Thread Specific Data

Thread specific data makes it possible to have variables whose value differs from thread to thread. Each piece of thread specific data is identified by a key which has to be allocated once at the beginning of the program.

`int pthread_key_create(pthread_key_t * key, void (*destructor)(void *))`

Creates a new key for thread specific data. All existing and new threads initially have a `NULL` value for this key until `pthread_setspecific()` is called. When a thread with a non−`NULL` value for the key ends, `destructor` is called with that value as its argument.

`int pthread_key_delete(pthread_key_t key)`

Deletes a key, but does *not* call any destructors for it.

```
int pthread_setspecific(pthread_key_t key, void * value)
void * pthread_getspecific(pthread_key_t key)
```

Manipulates the value associated for `key` in the current thread.

Synchronizing Threads

Two mechanisms are available to coordinate threads: Mutual exclusion and the more complex condition variables. Furthermore, the *once* mechanism is available for initialization.

`int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutexattr_t * attr)`

Initialize a mutex variable dynamically. Alternatively, you can initialize it statically with the declaration:

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

`int pthread_mutex_destroy(pthread_mutex_t * mutex)`

Destroy a mutex.

`int pthread_mutex_lock(pthread_mutex_t * mutex)`

Lock the mutex. Until this thread calls `pthread_mutex_unlock`, no other thread will be able to lock this mutex. If the mutex was already locked, block until it becomes available.

`int pthread_mutex_trylock(pthread_mutex_t * mutex)`

If the mutex is unlocked, lock it. If it is locked, return `EBUSY`.

`int pthread_mutex_unlock(pthread_mutex_t * mutex)`

Unlock the mutex and if any other threads were blocking for it, lock it for the first of them.

`int pthread_mutexattr_init(pthread_mutexattr_t * attr)`

Create a default mutex attribute object. Currently, none of the attributes may be changed.

`int pthread_mutexattr_destroy(pthread_mutexattr_t * attr)`

Destroy a mutex attribute object.

`int pthread_cond_init(pthread_cond_t * cond, const pthread_condattr_t * attr)`

Initialize a condition variable. Static initialization is available as

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

`int pthread_cond_destroy(pthread_cond_t * cond)`

Destroy a condition variable.

`int pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * mutex)`

Temporarily unlocks `mutex` (which must have been locked), wait for an event on the condition variable, and lock `mutex` again.

`int pthread_cond_timedwait(pthread_cond_t * cond, pthread_mutex_t * mutex, const struct timespec * abstime)`

Like `pthread_cond_wait()`, but only waits for a condition until the absolute time specified by `abstime`.

`int pthread_cond_signal(pthread_cond_t * cond)`

Sends an event to the first thread waiting on the condition variable.

`int pthread_cond_broadcast(pthread_cond_t * cond)`

Sends an event to all threads waiting on the condition variable.

`int pthread_condattr_init(pthread_condattr_t * attr)`

Create a default condition attribute object. Currently, none of the attributes may be changed.

`int pthread_condattr_destroy(pthread_condattr_t * attr)`

Destroy a condition attribute object.

`int pthread_once(pthread_once_t * once_block, void (*proc)(void))`

If the specified `once_block` hasn‘t executed yet, execute it. `once_block` must have been statially initialized as

```
pthread_once_t once = PTHREAD_ONCE_INIT;
```

Miscellaneous APIs

Timing routines

Since the Metrowerks Standard Library defines `time_t` from an epoch of 1970, GUSI 2 reimplements `time()`, `mktime()`, `gmtime()`, and `localtime()` to ensure the traditional MacOS behavior.

GUSI also implements the following timing−related UNIX 98 APIs:

`u_int sleep(u_int seconds)`

Tries to sleep for the specified number of seconds and returns the remaining number of seconds if interrupted for any reason.

`void usleep(u_int usecs)`

Tries to sleep for the specified number of microseconds.

`gettimeofday(struct timeval * tv, struct timezone * tz)`

Returns time in the same base as `time()`, but with a higher resolution (theoretically microseconds). If `tz` is not `NULL`, also returns the current time zone and DST flag.

Signal manipulation routines

GUSI makes some attempt to provide a reasonable emulation of UNIX 98 signal handling behavior, specifically:

- Most of the signal handling API is supported.
- GUSI will generate `SIGALRM` when the `alarm()` timer runs out, `SIGPIPE` when writing to a closed socket, and `SIGINT` when the interrupt key (`Cmd−.`) is pressed.
- In a significant departure from UNIX 98 behavior, signals are not delivered asynchronously, but are checked only when a thread is about to yield control by calling a blocking system call or `sched_yield()`.

All signal functionality is defined in the ***signal.h*** header. The central data structures for signal handling are the signal handling function type and the `sigaction` structure:

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

struct      sigaction {
   __sig_handler sa_handler;  /* signal handler */
   sigset_t      sa_mask;     /* signal mask to apply */
   int           sa_flags;    /* see signal options below */
};
```

When a signal is raised, the following happens:

- The signals specified in `sa_mask` are blocked. Furthermore, the siganl currently raised is also blocked unless `SA_RESETHAND` or `SA_NODEFER` are set in `sa_flags`.
- The signal handler specified in `sa_handler` is executed, and if `SA_RESETHAND` is set in `sa_flags`, the signal handler is reset to `SIG_DFL` just before executing the handler.
- If a "slow" system call, i.e., a call that can take an indefinite time to complete, such as a read call on a socket, is executing, that call is interrupted unless `SA_RESTART` is set in `sa_flags`.

The following functions are supported:

`int sigaddset(sigset_t * set, int signo)`

Adds a signal to a signal set.

`int sigdelset(sigset_t * set, int signo)`

Deletes a signal from a signal set.

`int sigemptyset(sigset_t * set)`

Sets a signal set to the empty set.

`int sigfillset(sigset_t * set)`

Sets a signal set to the set containing all signals.

`int sigismember(const sigset_t * set, int signo)`

Tests if a signal is a member of the set.

`int sigaction(int signo, const struct sigaction * act, struct sigaction * oact)`

Gets and/or sets handling behavior for a signal. If `act` is not `NULL`, sets the new behavior. If `oact` is not `NULL`, returns the previous behavior.

`__sig_handler signal(int signo, __sig_handler handler)`

The historical interface to signal handling, equivalent to `sigaction` with an empty `sa_mask` and `SA_RESETHAND` set.

`int raise(int signo)`

Sends a signal to a process. It will be delivered to the first thread that hasn‘t blocked it.

`int sigpending(sigset_t * set)`

Returns the set of signals pending in the process or the calling thread that are blocked from delivery.

`int sigprocmask(int how, const sigset_t * set, sigset_t * oset)`

Manipulates the mask of signals to be blocked from delivery in the process. If `set` is not `NULL`, the mask is changed depending on the `how` parameter:

`SIG_BLOCK`

The mask is set to the union of its current value and `set`.

`SIG_SETMASK`

The mask is set to `set`.

`SIG_UNBLOCK`

The mask is set to the intersection of its current value and the complement of `set`.

If `oset` is not null, it is set to the previous value of the process mask. In a multithreaded program, the behavior of `sigprocmask` is undefined in UNIX 98. GUSI defines this case to do the same as `pthread_sigmask`.

`int sigsuspend(const sigset_t * set)`

Temporarily replace the signal mask by `set` and then suspend execution until a signal is delivered.

`int sigwait(const sigset_t * set, int * signo)`

Waits for a signal in `set` to become pending, then clears it and returns its number in `signo`. All signals in `set` have to be blocked in the calling thread.

`int pthread_kill(pthread_t thread, int signo)`

Sends a signal to a specific thread (waking it up if it was asleep).

`int pthread_sigmask(int how, const sigset_t * set, sigset_t * oset)`

Manipulates the signal mask for a thread similar to the way that `sigprocset` manipulates signal masks in the singlethreaded case.

`void abort()`

Raises `SIGABRT` and quits the process.

`unsigned int alarm(unsigned int delay)`

If `delay` is not , arranges to have `SIGALRM` generated after the number of seconds specified. Returns the number of seconds that would have remained in the previous call to `alarm`.

`useconds_t ualarm(useconds_t delay, useconds_t interval)`

Similar to `alarm`, but manipulates times specified in microseconds. If `interval` is not , generates regular instances of `SIGALRM` spaced at `interval` microseconds.

BSD memory routines

If you

```
#include <compat.h>
```

the following routines will be available as macros:

`void bzero(void * from, int len)`

zeroes `len` bytes, starting at `from`.

`void bfill(void * from, int len, int x)`

fills `len` bytes, starting at `from`, with `x`.

`void bcopy(void * from, void * to, int len)`

copies `len` bytes from `from` to `to`.

`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.

Hooks

You can override some of GUSI's behaviour by providing hooks to GUSI. Note that these often get called from deep within GUSI, so be sure you understand what is required of a hook before overriding it.

GUSI hooks can be accessed with the following routines:

```
typedef void (*GUSIHook)(void);
void GUSISetHook(GUSIHookCode code, GUSIHook hook);
GUSIHook GUSIGetHook(GUSIHookCode code);
```

Currently, three types of hooks are defined.

`GUSI_SpinHook`

This hook is called when the main thread in the GUSI application wants to yield control. To provide your own hook, call

```
GUSISetHook(GUSI_SpinHook, (GUSIHook) my_spin_hook);
```

where `my_spin_hook` is defined as

```
void my_spin_hook(bool wait)
```

where `wait` is `false` if the thread has more work to do immediately and just wants to yield control as a courtesy, and `true` if the thread is blocked for an indefinite time. Specifying a `GUSI_SpinHook` disables the `GUSI_EventHook` handling described below unless you call `GUSIHandleNextEvent(bool wait)`.

`GUSI_EventHook`

If no `GUSI_SpinHook` is specified, `GUSI` calls `WaitNextEvent()` according to rules established by calling `GUSIEventHook+eventCode` as follows:

AppleEvents are always enabled and processed, unless you call

```
GUSISetHook(GUSI_EventHook+kHighLevelEvent, (GUSIHook) −1);
```

Mouse down events are enabled unless you call

```
GUSISetHook(GUSI_EventHook+mouseDown, (GUSIHook) −1);
```

only system clicks are processed unless you call

```
GUSISetHook(GUSI_EventHook+mouseDown, (GUSIHook) my_mousedown_hand
```

where `my_mousedown_handler` is declared as

```
void my_mousedown_handler(EventRecord * ev)
```

and should handle both system and application clicks.

All other events are disabled unless you specify handlers for them.

`GUSI_ExecHook`

This hook is used by GUSI to decide whether a file or folder is to be considered "executable" or not. The default hook considers all folders and all applications (i.e., files of type `'APPL'` and `'appe'` to be executable. To provide your own hook, call

```
GUSISetHook(GUSI_ExecHook, (GUSIHook) my_exec_hook);
```

where `my_exec_hook` is defined as

```
Boolean my_exec_hook(const GUSIFileRef & ref);
```

Resources

The information in this section is likely to change in the near future.

On startup, `GUSI` looks for a *preference* resource with type `'GUZI'` (the `'Z'` actually must be a capital Sigma) and ID `GUSIRsrcID`, which is currently defined as follows:

```
#ifndef GUSI_PREF_VERSION
#define GUSI_PREF_VERSION '0102'
#endif

type 'GUZI' {
   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 subdirectorie
   boolean  noTCPDaemon, isTCPDaemon;  /* Inetd client ?                  */
   boolean  noUDPDaemon, isUDPDaemon;
#if GUSI_PREF_VERSION >= '0150'
   boolean  noConsole, hasConsole;     /* Are we providing our own dev:console
#if GUSI_PREF_VERSION >= '0180'
   boolean   autoInitGraf, noAutoInitGraf;   /* Automatically do InitGraf ? */
   boolean   exclusiveOpen, sharedOpen;  /* Shared open() ?
   boolean   noSigPipe, sigPipe;         /* raise SIGPIPE on write to closed PI
#else
   fill      bit[3];
#endif
#else
   fill      bit[4];
#endif
   literal longint = GUSI_PREF_VERSION;
```

```
#if GUSI_PREF_VERSION >= '0120'
   integer = @t$$@>Countof(SuffixArray);

   wide array SuffixArray {
         literal longint;              /* Suffix of file */
         literal longint;              /* Type for file */
         literal longint;              /* Creator for file */
   };
#endif
#endif
};
```

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`.

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'  }
```

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.

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.

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.

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.

The `hasConsole` flag should be set if you are overriding the default "dev:console", as discussed below.

`GUSI` by default spins the cursor to indicate progress, and this will crash unless QuickDraw is initialized. While previous versions of GUSI required explicit Toolbox initialization, versions 1.8.0 and later will detect that QuickDraw is uninitialized and call InitGraf before spinning. To disable that behavior, set the `noAutoInitGraf` flag.

By default, `GUSI` opens files with *exclusive* read/write permissions. If you are sure you can deal with the consequences, you can request *shared* permissions by specifying the `sharedOpen` flag.

If a `GUSI` client attempts to read from a socket that was closed from the other side, an error code will be returned. As of version 1.8.0, you can specify the `sigPipe` flag to request that a `SIGPIPE` signal be raised additionally.