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`.

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.

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

Every capitalized word in this documentation might be a trademark of somebody. Everything relating to Macs is probably a trademark of Apple. UNIX is a trademark of AT&T. AT&T is a modem initialization command.

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 header files in `{CIncludes}`, and the library file in `{CLibraries}`.

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.
`sys/stat.h`	Getting information about files.
`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/types.h`	More data types.
`sys/uio.h`	Data types for scatter/gather calls.

8. Additionally, you have to link your program with either `''{CLibraries}GUSI.o''` , if you are using sockets, or `''{CLibraries}GUSI_F.o''`, if you are using only the file system component. It's important that these files appear *before* all other libraries. You will get lots of warning messages about duplicate definitions, but that's ok (Which means I can't do anything about it).

9. 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.

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

11. 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.

12. 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.

13. 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.

14. **int** *choose*(**int** *domain*, **int** *type*, **char** **prompt*, **void** **constraint*, **int** *flags*, **void** **name*, **int** **nlen*) puts up a modal dialog prompting the user to choose an address. *domain* 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.

15. 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.

16. **int** *bind*(**int** *s*, **void** **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.

17. **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.

18. **int** *accept*(**int** *s*, **void** **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.

19. **int** *connect*(**int** *s*, **void** **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.

20. 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*( ).

21. **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.

22. **int** *readv*(**int** *s*, **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 */
};
```

23. **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.

24. **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.

25. **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;
};
```

26. **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.

27. **int** *writev*(**int** *s*, **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.

28. **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.

29. **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*.

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

31. I/O multiplexing.

32. **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.

33. 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 */

34. 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*( ).

35. **int** *getsockname*(**int** *s*, **void** **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.

36. **int** *getpeername*(**int** *s*, **void** ∗*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.

37. **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.

38. **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.

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

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

41. **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.

42. **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.

43. 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.

44. 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.

45. Differences to generic behavior.

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

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

47. 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( )

48. *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;
```

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

50. *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.

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

52. **int** *stat*(**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;
};
```

53. *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_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'

54. **int** *lstat*(**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.

55. **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.

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

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

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

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

60. **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 MPW routine, they may specify different folders.

61. **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).

62. **int** *creat*(**char** *∗filename*) is identical to *open*(*filename*, *O_WRONLY* + *O_TRUNC* + *O_CREAT*).

63. **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.

64. **int** *symlink*(**char** *∗linkto*, **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.

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

66. **int** *mkdir*(**char** *∗path*) creates a new directory.

67. **int** *rmdir*(**char** *∗path*) deletes an empty directory.

68. **int** *chdir*(**char** *∗path*) makes all future partial pathnames relative from this directory.

69. **char** *∗getcwd* (**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.

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

71. **Differences to generic behavior.**

72. 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 */
};
```

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

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

75. Differences to generic behavior.

76. 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 */
};
```

77. 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 */
};
```

78. *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;
```

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

80. Differences to generic behavior.

81. PPC socket addresses have the following format:

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

82. *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 {
  Str32 type;
} sa_constr_ppc;
```

83. *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.

84. 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.

85. Differences to generic behavior.

86. 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];
};
```

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

88. 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 */
};
```

89. **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.

90. **struct** *hostent* **gethostbyaddr* (**struct** *in_addr* **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. The last two parameters are ignored.

91. **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")

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

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

94. Services are represented by the following data structure:

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

95. **struct** *servent* ∗*getservbyname* (**char** ∗*name*, **char** ∗*proto*) finds a named service. The current implementation relies on a hardcoded table of services, which is not very nice.

96. 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 */
};
```

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

98. Miscellaneous.

99. BSD memory routines.

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

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

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

103. **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.

104. 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.

105. 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 */
  } spin_msg;
  typedef int (*GUSISpinFn) ( spin_msg msg , long param ) ;
```

106. You can modify the spin routine with the following calls:

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

107. 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);
```

108. 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[];
```

109. Advanced techniques. This section discusses building specialized subset libraries, adding your own communication families, and some routines for manipulating *FSSpecs*.

110. Building subset libraries. Unfortunately, even if you use only one socket family, the MPW Linker has to include code for all of them, as it is impossible to determine which of them will be used beforehand. To remedy this situation, you might want to build a library conating only a subset of all files. The `GUSI` makefile contains already a few examples of such subsets.

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

- ▷ Pick an unused number between 6 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`.

112. 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.

113. Known problems.

▷ *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*.

114. References. The following books might provide you with more information about various aspects.

[**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

[**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

115. Index.

GUSI — Grand Unified Socket Interface