

LIBER GVSI

Annotated Source Code to the
Grand Unified Socket Library

Copyright ©1992–1999 Matthias Neeracher

Woven October 23, 1999
for GUSI Version 2.0

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.

Copyright ©1992–1999 Matthias Neeracher

Permission is granted to anyone to use this software for any purpose on any computer system, and to redistribute it freely, subject to the following restrictions:

- The author is not responsible for the consequences of use of this software, no matter how awful, even if they arise from defects in it.
- The origin of this software must not be misrepresented, either by explicit claim or by omission.
- Altered versions must be plainly marked as such, and must not be misrepresented as being the original software.

Permission is furthermore granted to print individual copies of the documentation and annotated source code of this software, provided that this copyright notice appears in it unchanged.

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 in your documentation.
- Send me a CD as described in my donations web page on <http://www.iis.ee.ethz.ch/~neeri/macintosh>

Contents

1	Introduction	1
1.1	Design Objectives	1
1.2	Literate Programs	2
1.3	How to Read This Book	2
1.4	Contacting the Author	2
I	Infrastructure	3
2	Common routines for GUSI	5
2.1	Definition of compiler features	7
2.2	Definition of hook handling	8
2.3	Definition of error handling	9
2.4	Definition of event handling	9
2.5	Implementation of hook handling	10
2.6	Implementation of error handling	12
2.7	Implementation of event handling	17
3	Thread and Process structures	19
3.1	Definition of completion handling	20
3.2	Implementation of completion handling	24
4	Thread Specific Variables	41
4.1	Definition of Thread Specific Variables	42
4.2	Implementation of Thread Specific Variables	43
5	The GUSI Socket Class	47
5.1	Definition of GUSISocket	48
5.2	Implementation of GUSISocket	51
5.2.1	General socket management	51
5.2.2	Context management	52
5.2.3	Operations without plausible default implementations	52
5.2.4	Operations with plausible default implementations	53
5.2.5	I/O Operations	54
6	Buffering for GUSI	59
6.1	Definition of scattering/gathering	60
6.2	Definition of ring buffering	61
6.3	Implementation of scattering/gathering	63

6.4	Implementation of ring buffering	66
7	Socket Factories	79
7.1	Definition of GUSISocketFactory	80
7.2	Definition of GUSISocketDomainRegistry	80
7.3	Definition of GUSISocketTypeRegistry	80
7.4	Implementation of GUSISocketFactory	81
7.5	Implementation of GUSISocketDomainRegistry	81
7.6	Implementation of GUSISocketTypeRegistry	83
8	Devices	87
8.1	Definition of GUSIFileToken	89
8.2	Definition of GUSIDevice	90
8.3	Definition of GUSIDeviceRegistry	90
8.4	Operations on Devices	91
8.5	Definition of GUSIDirectory	93
8.6	Implementation of GUSIFileToken	94
8.7	Implementation of GUSIDevice	95
8.8	Implementation of GUSIDeviceRegistry	96
9	GUSI Configuration settings	101
9.1	Definition of configuration settings	102
9.2	Implementation of configuration settings	104
10	Context Queues	111
10.1	Definition of context queues	112
10.2	Implementation of context queues	113
11	DCon interface	119
11.1	Definition of GUSIDConDevice	120
11.2	Definition of GUSIDConSocket	120
11.3	Implementation of GUSIDConDevice	120
11.4	Implementation of GUSIDConSocket	121
12	Timing functions	123
12.1	Definition of timing	125
12.2	Implementation of GUSITime	127
12.3	Implementation of GUSITimer	131
II	POSIX Wrappers	135
13	Mapping descriptors to sockets	137
13.1	Definition of GUSIDDescriptorTable	138
13.2	Implementation of GUSIDDescriptorTable	138
14	POSIX/Socket Wrappers	145
14.1	Implementation of POSIX wrappers	146
14.2	Implementation of Socket wrappers	155
14.3	Implementation of MPW wrappers	166

15 Pthreads Wrappers	169
15.1 Definition of Pthreads data types	170
15.2 PThread Attribute Management	171
15.3 Creation and Destruction of PThreads	173
15.4 Pthread thread specific data	175
15.5 Synchronization mechanisms of PThreads	176
15.6 Pthread varia	180
16 Signal support	183
16.1 Definition of the signal handling engine	184
16.2 Implementation of the signal handling engine	185
16.3 Definition of the signal handling POSIX functions	191
 III Domain Specific Code	 199
17 Mixin Classes for Sockets	201
17.1 Definition of GUSISocketMixins	202
17.2 Implementation of GUSISocketMixins	204
18 Null device	209
18.1 Definition of GUSINullDevice	210
18.2 Definition of GUSINullSocket	210
18.3 Implementation of GUSINullDevice	210
18.4 Implementation of GUSINullSocket	212
19 The GUSI Pipe Socket Class	215
19.1 Definition of GUSIPipeFactory	216
19.2 Implementation of GUSIPipeFactory	216
19.3 Definition of GUSIPipeSocket	217
19.4 Implementation of GUSIPipeSocket	217
20 TCP/IP shared infrastructure	223
21 Converting Between Names and IP Addresses	225
21.1 Definition of GUSINetDB	226
21.2 Definition of GUSIServiceDB	227
21.3 Definition of GUSIhostent and GUSIservent	228
21.4 Implementation of GUSINetDB	229
21.5 Implementation of GUSIServiceDB	233
21.6 Implementation of GUSIhostent and GUSIservent	239
22 Basic MacTCP code	241
22.1 Definition of GUSIMTInetSocket	242
22.2 Implementation of GUSIMTInetSocket	243
23 MacTCP TCP sockets	249
23.1 Definition of GUSIMTTcpFactory	250
23.2 Implementation of GUSIMTTcpFactory	250
23.3 Definition of GUSIMTTcpSocket	251
23.4 Implementation of GUSIMTTcpSocket	251

23.4.1	Interrupt level routines for GUSIMTTcpSocket	251
23.4.2	High level interface for GUSIMTTcpSocket	259
24	MacTCP UDP sockets	269
24.1	Definition of GUSIMTudpFactory	270
24.2	Implementation of GUSIMTudpFactory	270
24.3	Definition of GUSIMTudpSocket	271
24.4	Implementation of GUSIMTudpSocket	271
24.4.1	Interrupt level routines for GUSIMTudpSocket	271
24.4.2	High level interface for GUSIMTudpSocket	276
25	IP Name Lookup in MacTCP	283
25.1	Definition of GUSIMTNetDB	284
25.2	Implementation of GUSIMTNetDB	284
26	Open Transport socket infrastructure	289
26.1	Definition of GUSIOTFactory and descendants	290
26.2	Definition of GUSIOTStrategy	291
26.3	Definition of GUSIOT	292
26.4	Definition of GUSIOTSocket and descendants	293
26.5	Implementation of GUSIOTFactory and descendants	293
26.6	Implementation of GUSIOTStrategy	294
26.7	Implementation of GUSIOTSocket	295
26.8	Implementation of GUSIOTStreamSocket	306
26.9	Implementation of GUSIOTDatagramSocket	316
27	Open Transport TCP/IP sockets	323
27.1	Definition of Open Transport Internet hooks	324
27.2	Definition of GUSIOTTcpFactory	325
27.3	Definition of GUSIOTudpFactory	325
27.4	Definition of GUSIOTInetStrategy	325
27.5	Definition of GUSIOTTcpStrategy	326
27.6	Definition of GUSIOTMInetOptions	326
27.7	Definition of GUSIOTudpStrategy	326
27.8	Definition of GUSIOTudpSocket	326
27.9	Definition of GUSIOTTcpSocket	327
27.10	Implementation of GUSIOTTcpFactory	327
27.11	Implementation of GUSIOTudpFactory	327
27.12	Implementation of GUSIOTInetStrategy	328
27.13	Implementation of GUSIOTMInetOptions	330
27.14	Implementation of GUSIOTTcpStrategy	336
27.15	Implementation of GUSIOTTcpSocket	336
27.16	Implementation of GUSIOTudpStrategy	339
27.17	Implementation of GUSIOTudpSocket	339
27.18	Implementation of Open Transport Internet hooks	343
28	IP Name Lookup in Open Transport	345
28.1	Definition of GUSIOTNetDB	346
28.2	Implementation of GUSIOTNetDB	346
29	Pseudo-synchronous File System Calls	353

30 File specifications	365
30.1 Definition of GUSICatInfo	366
30.2 Definition of GUSIFileSpec	367
30.3 Implementation of GUSICatInfo	371
30.4 Implementation of GUSIFileSpec	371
31 Disk files	391
31.1 Definition of GUSIMacFileDevice	393
31.2 Definition of GUSIMacFileSocket	393
31.3 Definition of GUSIMacDirectory	394
31.4 Implementation of GUSIMacFileDevice	394
31.5 Implementation of GUSIMacFileSocket	411
31.5.1 High level interface for GUSIMacFileSocket	412
31.5.2 Interrupt level routines for GUSIMacFileSocket	421
31.6 Implementation of GUSIMacDirectory	426
32 The GUSI PPC Socket Class	429
32.1 Definition of GUSIPPCFactory	430
32.2 Implementation of GUSIPPCFactory	430
32.3 Definition of GUSIPPCSocket	431
32.4 Implementation of GUSIPPCSocket	431
32.4.1 Interrupt level routines for GUSIPPCSocket	431
32.4.2 High level interface for GUSIPPCSocket	436
IV Library Specific Code	447
33 The Interface to the MSL	449
33.1 Implementation of MSL override functions	450
33.2 Implementation of ANSI library specific public GUSI functions . . .	454
33.3 Implementation of ANSI library specific internal GUSI functions . . .	454
34 The Interface to the MPW Stdio library	457
34.1 Implementation of MPW ANSI library specific public GUSI functions	458
34.2 Implementation of internal GUSI functions for MPW Stdio	460
35 The Interface to the Sflo library	463
35.1 Implementation of internal GUSI functions for Sflo	464
36 MPW Support	465
36.1 Definition of GUSIMPWDevice	466
36.2 Definition of GUSIMPWSocket	467
36.3 Interfacing to MPW library routines	467
36.4 Implementation of GUSIMPWDevice	469
36.5 Implementation of GUSIMPWSocket	472
37 SIOUX Support	475
37.1 Definition of GUSISIOUXDevice	476
37.2 Definition of GUSISIOUXSocket	476
37.3 Implementation of GUSISIOUXDevice	476
37.4 Implementation of GUSISIOUXSocket	477

38 SIOW Support	481
39 Supporting threads made outside of GUSI	485
39.1 Interfacing to the Thread Manager routines	486
39.2 Redirecting thread manager calls to their GUSI equivalents	488
39.3 Installing our GUSI thread manager hooks	488
 V The Build System	 491
40 Build rules for GUSI	493
 VI Appendixes	 499
A Index	501
B Names of the Sections	513

Chapter 1

Introduction

GUSI is a POSIX library for traditional 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/pl>.

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

1.1 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 2 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).

1.2 Literate Programs

This book contains the implementation of GUSI 2. The `pnweb`¹ literate programming system generates both the source code for GUSI 2 and this book from a single source.

The source code consists of interleaved prose commentary and labelled code fragments, arranged in an order best suited for presenting to a human reader, rather than one dictated by a compiler. Fragments, referred to by their labels in angle brackets, consist of source code and references to other fragments. Several fragments may have the same name; they will be concatenated in the source code. Fragments work like macros; a source file is created by expanding one fragment. If its definition refers to other fragments, they are themselves expanded, until no fragment references remain.

1.3 How to Read This Book

This book consists of 5 parts. Part I describes the infrastructure underlying GUSI 2. Part II describes the POSIX-style APIs through which GUSI is accessed, mapping them onto the interfaces defined in part I. Part III then describes how each communication domain implements those interfaces. Part IV presents the code connecting GUSI to the vendor specific I/O libraries and console interfaces. Finally, part describes the build system used for GUSI 2.

1.4 Contacting the Author

I welcome correspondence both about the practical use of GUSI 2 and about its presentation in this book (which I will freely admit still leaves a lot to be desired).

Matthias Neeracher
20875 Valley Green Dr. #50
Cupertino, CA 95014

e-Mail: <neeri@iis.ee.ethz.ch>
Fax: +1 (408) 514-2605 ext. 0023

¹`pnweb` is a reimplementaion of Norman Ramsey's `noweb` system in `perl`

Part I

Infrastructure

Chapter 2

Common routines for GUSI

This section defines various services used by all parts of GUSI:

- Various hooks to customize GUSI.
- The propagation of **errors** to the `errno` and `h_errno` global variables.
- Waiting for completion of asynchronous calls.
- Event handling.
- Compiler features.

To protect our name space further, we maintain a strict C interface unless `GUSI_SOURCE` is defined, and may avoid defining some stuff unless `GUSI_INTERNAL` is defined. The following header is therefore included first by all GUSI source files.

```
<GUSIInternal.h 1>≡
#ifdef _GUSIInternal_
#define _GUSIInternal_

#include <ConditionalMacros.h>

#define GUSI_SOURCE
#define GUSI_INTERNAL

#if !TARGET_RT_MAC_CFM
#pragma segment GUSI
#endif

#endif /* _GUSIInternal_ */
```

```

<GUSIBasics.h 2>≡
    #ifndef _GUSIBasics_
    #define _GUSIBasics_

    #ifdef GUSI_SOURCE

    #include <errno.h>
    #include <sys/cdefs.h>

    #include <ConditionalMacros.h>

    <Definition of compiler features 4>
    <Definition of hook handling 9>
    <Definition of error handling 12>
    <Definition of event handling 14>

    #endif /* GUSI_SOURCE */

    #endif /* _GUSIBasics_ */

<GUSIBasics.cp 3>≡
    #define GUSI_MESSAGE_LEVEL 5

    #include "GUSIInternal.h"
    #include "GUSIBasics.h"
    #include "GUSIDdiag.h"

    #include <errno.h>
    #include <netdb.h>

    #include <MacTypes.h>
    #include <Events.h>
    #include <Files.h>
    #include <OSUtils.h>
    #include <EPPC.h>
    #include <LowMem.h>
    #include <AppleEvents.h>
    #include <QuickDraw.h>
    #include <MacTCP.h>
    #include <OpenTransport.h>

    <Implementation of hook handling 15>
    <Implementation of error handling 25>
    <Implementation of event handling 28>

```

2.1 Definition of compiler features

If possible, we use unnamed namespaces to wrap internal code.

(Definition of compiler features 4)≡

(2) 5▷

```
#ifdef __MWERKS__
#define GUSI_COMPILER_HAS_NAMESPACE
#endif

#ifdef GUSI_COMPILER_HAS_NAMESPACE
#define GUSI_USING_STD_NAMESPACE using namespace std; using namespace std::rel_ops;
#else
#define GUSI_USING_STD_NAMESPACE
#endif
```

Asynchronous MacOS calls need completion procedures which in classic 68K code often take parameters in address registers. The way to handle this differs a bit between compilers. Note that the pascal keyword is ignored when generating CFM code.

(Definition of compiler features 4)+≡

(2) <4 6▷

```
#if GENERATINGCFM
#define GUSI_COMPLETION_PROC_A0(proc, type) \
    void (*const proc##Entry)(type * param)    =    proc;
#define GUSI_COMPLETION_PROC_A1(proc, type) \
    void (*const proc##Entry)(type * param)    =    proc;
#elif defined(__MWERKS__)
#define GUSI_COMPLETION_PROC_A0(proc, type) \
    static pascal void proc##Entry(type * param : __A0) { proc(param); }
#define GUSI_COMPLETION_PROC_A1(proc, type) \
    static pascal void proc##Entry(type * param : __A1) { proc(param); }
#else
void * GUSIGetA0()  ONEWORDINLINE(0x2008);
void * GUSIGetA1()  ONEWORDINLINE(0x2009);
#define GUSI_COMPLETION_PROC_A0(proc, type) \
    static pascal void proc##Entry() \
    { proc(reinterpret_cast<type *>(GUSIGetA0())); }
#define GUSI_COMPLETION_PROC_A1(proc, type) \
    static pascal void proc##Entry() \
    { proc(reinterpret_cast<type *>(GUSIGetA1())); }
#endif
```

SC seems to have an issue with mutable fields.

(Definition of compiler features 4)+≡

(2) <5 7▷

```
#if defined(__SC__)
#define mutable
#define GUSI_MUTABLE(class, field) const_cast<class *>(this)->field
#else
#define GUSI_MUTABLE(class, field) field
#endif
```

SC pretends to support standard scoping rules, but is in fact broken in some cases.

(Definition of compiler features 4)+≡

(2) <6 8▷

```
#if defined(__SC__)
#define for if (0) ; else for
#endif
```

The MPW compilers don't predeclare qd.

```
(Definition of compiler features 4)+≡ (2) <7
#if defined(__SC__) || defined(__MRC__)
#define GUSI_NEEDS_QD    QDGlobals    qd;
#else
#define GUSI_NEEDS_QD
#endif
```

2.2 Definition of hook handling

GUSI supports a number of hooks. Every one of them has a different prototype, but is passed as a GUSIHook. Hooks are encoded with an OSType.

```
(Definition of hook handling 9)≡ (2) 10>
typedef unsigned long OSType;
typedef void (*GUSIHook)(void);
void GUSISetHook(OSType code, GUSIHook hook);
GUSIHook GUSIGetHook(OSType code);
```

Currently, three hooks are supported: GUSI_SpinHook defines a function to be called when GUSI waits on an event. GUSI_ExecHook defines a function that determines whether a file is to be considered “executable”. GUSI_EventHook defines a routine that is called when a certain event happens. To install an event hook, pass GUSI_EventHook plus the event code. A few events, that is mouse-down and high level events, are handled automatically by GUSI. Passing -1 for the hook disables default handling of an event.

```
(Definition of hook handling 9)+≡ (2) <9 11>
typedef bool (*GUSISpinFn)(bool wait);
#define GUSI_SpinHook 'spin'

struct FSSpec;
typedef bool (*GUSIExecFn)(const FSSpec * file);
#define GUSI_ExecHook 'exec'

struct EventRecord;
typedef void (*GUSIEventFn)(EventRecord * ev);
#define GUSI_EventHook 'evnt'
```

For the purposes of the functions who actually call the hooks, here's the direct interface.

```
(Definition of hook handling 9)+≡ (2) <10
#ifdef GUSI_INTERNAL
extern GUSISpinFn gGUSISpinHook;
extern GUSIExecFn gGUSIExecHook;
#endif /* GUSI_INTERNAL */
```


2.3 Definition of error handling

Like a good POSIX citizen, GUSI reports all errors in the `errno` global variable. This happens either through the `GUSISetPosixError` routine, which stores its argument untranslated, or through the `GUSISetMacError` routine, which translates MacOS error codes into the correct POSIX codes. The mapping of `GUSISetMacError` is not always appropriate, so some routines will have to preprocess some error codes. `GUSIMapMacError` returns the POSIX error corresponding to a MacOS error.

The domain name routines use an analogous variable `h_errno`, which is manipulated with `GUSISetHostError` and `GUSISetMacHostError`.

All routines return 0 if 0 was passed and -1 otherwise.

(Definition of error handling 12)≡ (2) 13▷
`typedef short OSErr;`

```
int GUSISetPosixError(int error);
int GUSISetMacError(OSErr error);
int GUSIMapMacError(OSErr error);
int GUSISetHostError(int error);
int GUSISetMacHostError(OSErr error);
```

POSIX routines should never set `errno` from nonzero to zero. On the other hand, it's sometimes useful to see whether some particular region of the program set the error code or not. Therefore, we have such regions allocate a `GUSIErrorSaver` statically, which guarantees that previous error codes get restored if necessary.

(Definition of error handling 12)+≡ (2) <12
`class GUSIErrorSaver {`
`public:`
 `GUSIErrorSaver() { fSavedErrno = ::errno; ::errno = 0; }`
 `~GUSIErrorSaver() { if (!::errno) ::errno = fSavedErrno; }`
`private:`
 `int fSavedErrno;`
`};`

2.4 Definition of event handling

`GUSIHandleNextEvent()` events by calling handlers installed using the `GUSI_EventHook` mechanism.

(Definition of event handling 14)≡ (2)
`void GUSIHandleNextEvent(long sleepTime);`

2.5 Implementation of hook handling

Each hook is represented by a hook variable and a piece of code in the set and get routine.

(Implementation of hook handling 15)≡ (3)

```
__BEGIN_DECLS
<Hook variables 16>
__END_DECLS
void GUSISetHook(OSType code, GUSIHook hook)
{
    switch (code) {
        <Hook setter code 17>
    default:
        GUSI_ASSERT_CLIENT(0,
            ( "Illegal code %lx ('%c%c%c%c') passed to GUSISetHook\n",
              code,
              (code >> 24) & 0xff,
              (code >> 16) & 0xff,
              (code >> 8) & 0xff,
              (code) & 0xff
            ));
        break;
    }
}
GUSIHook GUSIGetHook(OSType code)
{
    switch (code) {
        <Hook getter code 18>
    default:
        GUSI_ASSERT_CLIENT(0,
            ( "Illegal code %lx ('%c%c%c%c') passed to GUSIGetHook\n",
              code,
              (code >> 24) & 0xff,
              (code >> 16) & 0xff,
              (code >> 8) & 0xff,
              (code) & 0xff
            ));
        return (GUSIHook) nil;
    }
}
```

Every hook then has to add to the three above code sections. Here is the spin hook.

(Hook variables 16)≡ (15) 19▶

```
GUSISpinFn gGUSISpinHook;
```

(Hook setter code 17)≡ (15) 20▶

```
case GUSI_SpinHook:
    gGUSISpinHook = (GUSISpinFn) hook;
    break;
```

(Hook getter code 18)≡ (15) 21▶

```
case GUSI_SpinHook:
    return (GUSIHook) gGUSISpinHook;
```

Here is the exec hook. I guess you see the pattern now

```
⟨Hook variables 16⟩+≡ (15) ◀16 22▶  
    GUSIExecFn gGUSIExecHook;
```

```
⟨Hook setter code 17⟩+≡ (15) ◀17 23▶  
    case GUSI_ExecHook:  
        gGUSIExecHook = (GUSIExecFn) hook;  
        break;
```

```
⟨Hook getter code 18⟩+≡ (15) ◀18 24▶  
    case GUSI_ExecHook:  
        return (GUSIHook) gGUSIExecHook;
```

The event hook is somewhat different as it is a whole array of hooks.

```
⟨Hook variables 16⟩+≡ (15) ◀19  
    const short kGUSIDefaultEventMask = mDownMask+highLevelEventMask;  
    short      gGUSIEventMask        = kGUSIDefaultEventMask;  
    GUSIEventFn gGUSIEventHook[16];
```

```
⟨Hook setter code 17⟩+≡ (15) ◀20  
    case GUSI_EventHook + nullEvent:  
    case GUSI_EventHook + mouseDown:  
    case GUSI_EventHook + mouseUp:  
    case GUSI_EventHook + keyDown:  
    case GUSI_EventHook + keyUp:  
    case GUSI_EventHook + autoKey:  
    case GUSI_EventHook + updateEvt:  
    case GUSI_EventHook + diskEvt:  
    case GUSI_EventHook + activateEvt:  
    case GUSI_EventHook + osEvt:  
        code -= GUSI_EventHook;  
        if (hook == (GUSIHook) -1) {  
            gGUSIEventHook[code] = (GUSIEventFn) 0;  
            gGUSIEventMask &= ~(1 << code);  
        } else {  
            gGUSIEventHook[code] = (GUSIEventFn) hook;  
            if (hook || kGUSIDefaultEventMask & (1 << code))  
                gGUSIEventMask |= 1 << code;  
            else  
                gGUSIEventMask &= ~(1 << code);  
        }  
        break;  
    case GUSI_EventHook + kHighLevelEvent:  
        if (hook != (GUSIHook) -1)  
            gGUSIEventMask |= highLevelEventMask;  
        else  
            gGUSIEventMask &= ~highLevelEventMask;  
        break;
```

(Hook getter code 18) ≡ (15) 421

```

case GUSI_EventHook + nullEvent:
case GUSI_EventHook + mouseDown:
case GUSI_EventHook + mouseUp:
case GUSI_EventHook + keyDown:
case GUSI_EventHook + keyUp:
case GUSI_EventHook + autoKey:
case GUSI_EventHook + updateEvt:
case GUSI_EventHook + diskEvt:
case GUSI_EventHook + activateEvt:
case GUSI_EventHook + osEvt:
    return (GUSIHook) gGUSIEventHook[code - GUSI_EventHook];
case GUSI_EventHook + kHighLevelEvent:
    return (GUSIHook) ((gGUSIEventMask & highLevelEventMask) ? 0 : -1);

```

2.6 Implementation of error handling

(Implementation of error handling 25) ≡ (3) 26▶

```

int GUSISetPosixError(int error)
{
    if (error > 0) {
        errno = error;

        return -1;
    } else if (error < 0) {
        errno = EINVAL;

        return -1;
    } else
        return 0;
}

```

```
inline int GUSIErrorMapping(OSErr error, int posixError)
{
    GUSI_MESSAGE(("Error MacOS %d -> POSIX %d\n", error, posixError));
    return posixError;
}

int GUSIMapMacError(OSErr error)
{
    switch (error) {
    case noErr:
        return 0;
    case opWrErr:
    case wrPermErr:
        return GUSIErrorMapping(error, EPERM);
    case bdNamErr:
        return GUSIErrorMapping(error, ENAMETOOLONG);
    case afpObjectTypeErr:
        return GUSIErrorMapping(error, ENOTDIR);
    case fnfErr:
    case dirNFErr:
        return GUSIErrorMapping(error, ENOENT);
    case dupFNErr:
        return GUSIErrorMapping(error, EEXIST);
    case dirFulErr:
    case dskFulErr:
        return GUSIErrorMapping(error, ENOSPC);
    case fBsyErr:
        return GUSIErrorMapping(error, EBUSY);
    case tmfoErr:
        return GUSIErrorMapping(error, ENFILE);
    case fLckdErr:
    case permErr:
    case afpAccessDenied:
    case kOTAccessErr: /* -3152 Missing access permission */
        return GUSIErrorMapping(error, EACCES);
    case wPrErr:
    case vLckdErr:
        return GUSIErrorMapping(error, EROFS);
    case badMovErr:
        return GUSIErrorMapping(error, EINVAL);
    case diffVolErr:
        return GUSIErrorMapping(error, EXDEV);
    case openFailed:
        return GUSIErrorMapping(error, ECONNREFUSED);
    case duplicateSocket:
    case kOTAddressBusyErr: /* -3172 Address requested is already in use */
        return GUSIErrorMapping(error, EADDRINUSE);
    case connectionTerminated:
        return GUSIErrorMapping(error, ECONNREFUSED);
    case commandTimeout:
        return GUSIErrorMapping(error, ETIMEDOUT);
    case ipBadLapErr:
    case ipBadCnfgErr:
```

```

case ipNoCnfgErr:
case ipLoadErr:
    return GUSIErrorMapping(error, ENETDOWN);
case memFullErr:
    return GUSIErrorMapping(error, ENOMEM);
case kOTNoAddressErr: /* -3154 No address was specified */
case kOTOutStateErr: /* -3155 Call issued in wrong state */
    return GUSIErrorMapping(error, ENOTCONN);
case kOTFlowErr: /* -3161 Provider is flow-controlled */
    return GUSIErrorMapping(error, EWOULDBLOCK);
case kOTNotSupportedErr: /* -3167 Command is not supported */
    return GUSIErrorMapping(error, EOPNOTSUPP);
case kOTCanceledErr: /* -3180 The command was cancelled */
    return GUSIErrorMapping(error, ECANCELED);
case kEPERMErr: /* Permission denied */
    return GUSIErrorMapping(error, EPERM);
case kENOENTErr: /* No such file or directory */
    return GUSIErrorMapping(error, ENOENT);
case kEINTRErr: /* Interrupted system service */
    return GUSIErrorMapping(error, EINTR);
case kEIOErr: /* I/O error */
    return GUSIErrorMapping(error, EIO);
case kENXIOErr: /* No such device or address */
    return GUSIErrorMapping(error, ENXIO);
case kEBADFErr: /* Bad file number */
    return GUSIErrorMapping(error, EBADF);
case kEAGAINErr: /* Seems to be returned for refused connections */
    return GUSIErrorMapping(error, ECONNREFUSED);
case kENOMEMErr: /* Not enough space */
    return GUSIErrorMapping(error, ENOMEM);
case kEACCSErr: /* Permission denied */
    return GUSIErrorMapping(error, EACCES);
case kEFAULTErr: /* Bad address */
    return GUSIErrorMapping(error, EFAULT);
case kEBUSYErr: /* Device or resource busy */
    return GUSIErrorMapping(error, EBUSY);
case kEEXISTErr: /* File exists */
    return GUSIErrorMapping(error, EEXIST);
case kENODEVErr: /* No such device */
    return GUSIErrorMapping(error, ENODEV);
case kEINVALErr: /* Invalid argument */
    return GUSIErrorMapping(error, EINVAL);
case kENOTTYErr: /* Not a character device */
    return GUSIErrorMapping(error, ENOTTY);
case kEPIPEErr: /* Broken pipe */
    return GUSIErrorMapping(error, EPIPE);
case kERANGEErr: /* Message size too large for STREAM */
    return GUSIErrorMapping(error, ERANGE);
case kEWOULDBLOCKErr:
    return GUSIErrorMapping(error, EWOULDBLOCK);
case kEALREADYErr: /*
    return GUSIErrorMapping(error, EALREADY);
case kENOTSOCKErr: /* Socket operation on non-socket */
    return GUSIErrorMapping(error, ENOTSOCK);

```

```

case kEDESTADDRREQErr:      /* Destination address required */
    return GUSIErrorMapping(error, EDESTADDRREQ);
case kEMSGSIZEErr:          /* Message too long */
    return GUSIErrorMapping(error, EMSGSIZE);
case kEPROTOTYPEErr:        /* Protocol wrong type for socket */
    return GUSIErrorMapping(error, EPROTOTYPE);
case kENOPROTOOPTErr:        /* Protocol not available */
    return GUSIErrorMapping(error, ENOPROTOOPT);
case kEPROTONOSUPPORTErr:    /* Protocol not supported */
    return GUSIErrorMapping(error, EPROTONOSUPPORT);
case kESOCKTNOSUPPORTErr:    /* Socket type not supported */
    return GUSIErrorMapping(error, ESOCKTNOSUPPORT);
case kEOPNOTSUPPErr:         /* Operation not supported on socket */
    return GUSIErrorMapping(error, EOPNOTSUPP);
case kEADDRINUSEErr:         /* Address already in use */
    return GUSIErrorMapping(error, EADDRINUSE);
case kEADDRNOTAVAILErr:      /* Can't assign requested address */
    return GUSIErrorMapping(error, EADDRNOTAVAIL);
case kENETDOWNErr:           /* Network is down */
    return GUSIErrorMapping(error, ENETDOWN);
case kENETUNREACHErr:        /* Network is unreachable */
    return GUSIErrorMapping(error, ENETUNREACH);
case kENETRESETErr:          /* Network dropped connection on reset */
    return GUSIErrorMapping(error, ENETRESET);
case kECONNABORTEErr:        /* Software caused connection abort */
    return GUSIErrorMapping(error, ECONNABORTED);
case kECONNRESETErr:         /* Connection reset by peer */
    return GUSIErrorMapping(error, ECONNRESET);
case kENOBUFSErr:            /* No buffer space available */
    return GUSIErrorMapping(error, ENOBUFS);
case kEISCONNErr:            /* Socket is already connected */
    return GUSIErrorMapping(error, EISCONN);
case kENOTCONNErr:           /* Socket is not connected */
    return GUSIErrorMapping(error, ENOTCONN);
case kESHUTDOWNErr:          /* Can't send after socket shutdown */
    return GUSIErrorMapping(error, ESHUTDOWN);
case kETOOMANYREFSErr:        /* Too many references: can't splice */
    return GUSIErrorMapping(error, ETOOMANYREFS);
case kETIMEDOUTErr:          /* Connection timed out */
    return GUSIErrorMapping(error, ETIMEDOUT);
case kECONNREFUSEErr:         /* Connection refused */
    return GUSIErrorMapping(error, ECONNREFUSED);
case kEHOSTDOWNErr:          /* Host is down */
    return GUSIErrorMapping(error, EHOSTDOWN);
case kEHOSTUNREACHErr:        /* No route to host */
    return GUSIErrorMapping(error, EHOSTUNREACH);
case kEINPROGRESSErr:         /* */
    return GUSIErrorMapping(error, EINPROGRESS);
case kESRCHErr:              /* */
    return GUSIErrorMapping(error, ESRCH);
default:
    return GUSIErrorMapping(error, EINVAL);
}
}

```

```

int GUSISetMacError(OSErr error)
{
    return GUSISetPosixError(GUSIMapMacError(error));
}

```

h_errno is defined here.

(Implementation of error handling 25)+≡

(3) 426

```

int h_errno;

```

```

int GUSISetHostError(int error)
{
    if (error) {
        h_errno = error;

        return -1;
    } else
        return 0;
}

```

```

int GUSISetMacHostError(OSErr error)
{
    switch (error) {
        case noErr:
            return 0;
        case nameSyntaxErr:
        case noNameServer:
        case authNameErr:
            h_errno = HOST_NOT_FOUND;
            break;
        case noResultProc:
        case dnrErr:
        default:
            h_errno = NO_RECOVERY;
            break;
        case noAnsErr:
        case outOfMemory:
            h_errno = TRY_AGAIN;
            break;
    }
    return -1;
}

```


2.7 Implementation of event handling

(Implementation of event handling 28)≡

(3)

```
void GUSIHandleNextEvent(long sleepTime)
{
    EventRecord event;

    if (WaitNextEvent(gGUSIEventMask, &event, sleepTime, nil))
        switch (event.what) {
        case mouseDown:
            if (!gGUSIEventHook[mouseDown]) {
                WindowPtr win;
                if (FindWindow(event.where, &win) == inSysWindow)
                    SystemClick(&event, win);
                return;
            }
            break;
        case kHighLevelEvent:
            AEPProcessAppleEvent(&event);    // Ignore errors

            return;
        }

    if (gGUSIEventHook[event.what])
        gGUSIEventHook[event.what](&event);
}
```


Chapter 3

Thread and Process structures

This section defines the process and thread switching engine of GUSI.

In some execution environments, completion routines execute at interrupt level. GUSI therefore is designed so all information needed to operate from interrupt level is accessible from a GUSISocket. This information is separated into per-process data, collected in GUSIProcess, and per-thread data, collected in GUSIContext. GUSIProcess is always a singleton, while GUSIContext is a singleton if threading is disabled, and has multiple instances if threading is enabled. By delegating the GUSIContext creation process to an instance of a GUSIContextFactory, we gain some extra flexibility.

As soon as GUSI has started an asynchronous call, it calls the `Wait()` member function of its context. `msec` will set a time limit after which the call will return in any case. Exceptional events may also cause GUSIWait to return, so it is not safe to assume that the call will have completed upon return.

```
<GUSIContext.h 29>≡
#ifndef _GUSIContext_
#define _GUSIContext_

#include <errno.h>
#include <sys/cdefs.h>
#include <sys/signal.h>

#include <MacTypes.h>
#include <Threads.h>

__BEGIN_DECLS
<Definition of thread manager hooks 36>
__END_DECLS

#ifndef GUSI_SOURCE

typedef struct GUSIContext GUSIContext;

#else

#include "GUSISpecific.h"
#include "GUSIBasics.h"
#include "GUSIContextQueue.h"
```

```

#include <Files.h>
#include <Processes.h>
#include <OSUtils.h>

<Name dropping for file GUSIContext 31>

<Definition of class GUSIThreadManagerProxy 38>
<Definition of class GUSIProcess 32>
<Definition of class GUSIContext 34>
<Definition of class GUSIContextFactory 35>
<Definition of IO wrappers 37>

<Inline member functions for file GUSIContext 41>

#endif /* GUSI_SOURCE */

#endif /* _GUSIContext_ */

<GUSIContext.cp 30>≡
#include "GUSIInternal.h"
#include "GUSIContext.h"
#include "GUSIDiag.h"
#include "GUSISignal.h"
#include "GUSIConfig.h"

#include <errno.h>
#include <signal.h>

#include <EPPC.h>
#include <LowMem.h>
#include <AppleEvents.h>
#include <QuickDraw.h>
#include <Devices.h>

#include <utility>
#include <memory>

GUSI_USING_STD_NAMESPACE

<Implementation of completion handling 40>

```

3.1 Definition of completion handling

GUSIContext is heavily circular both with classes declared herein and in other files. Therefore, we start by declaring a few class names.

```

<Name dropping for file GUSIContext 31>≡ (29)
class GUSISocket;
class GUSIContext;
class GUSIProcess;
class GUSISigProcess;
class GUSISigContext;

```

A GUSIPProcess contains all the data needed to wake up a process:

- The ProcessSerialNumber of the process.
- The ThreadTaskRef if threads are enabled.
- The contents of the A5 register.

The sole instance of GUSIPProcess is obtained by calling GUSIPProcess::Instance().

Interrupt level procedures may access the application's A5 register either manually by calling GetA5 or simply by declaring a GUSIPProcess::A5Saver in a scope.

```
(Definition of class GUSIPProcess 32)≡ (29)
class GUSIPProcess {
public:
    static GUSIPProcess *    Instance();
    void                    GetPSN(ProcessSerialNumber * psn);
    void                    AcquireTaskRef();
    ThreadTaskRef           GetTaskRef();
    long                    GetA5();
    bool                    Threading();
    void                    Yield(bool wait);
    void                    Wakeup();
    GUSISigProcess *        SigProcess() { return fSigProcess; }
    (Definition of class GUSIPProcess::A5Saver 33)
protected:
    friend class GUSIContext;

    GUSIPProcess(bool threading);

    int                    fReadyThreads;
    GUSISigProcess *        fSigProcess;
private:
    (Privatissima of GUSIPProcess 39)
};
```

A GUSIPProcess::A5Saver is a class designed to restore the process A5 register for the scope of its declaration. It is typically declared with a GUSISocket * argument, as that is what's most likely to be available in a completion procedure.

```
(Definition of class GUSIPProcess::A5Saver 33)≡ (32)
class A5Saver {
public:
    A5Saver(long processA5);
    A5Saver(GUSIContext * context);
    A5Saver(GUSIPProcess * process);
    ~A5Saver();
private:
    long    fSavedA5;
};
```

A GUSIContext gathers thread related data. The central operation on a GUSIContext is Wakeup(). If the process is not asleep when Wakeup() is called, it is marked for deferred wakeup.

A GUSIContext can either be created from an existing thread manager ThreadID or by specifying the parameters for a NewThread call.

Current() returns the current GUSIContext. Setup() initializes the default context for either the threading or the non-threading model.

Yield() suspends the current process or thread until something interesting happens if wait is true]. If [[wait is false, Yield() switches, but does not suspend. For an ordinary thread context, Yield() simply yields the thread. For the context in a non-threading application, Yield() does a WaitNextEvent(). For the main thread context, Yield() does both.

Done() tests whether the thread has terminated yet. If join is set, the caller is willing to wait. Result() returns the default location to store the thread result if no other is specified.

By default, a context is joinable. Calling Detach() will cause the context to be destroyed automatically upon thread termination, and joins are no longer allowed. A joinable context will not be destroyed automatically before the end of the program, so you will have to call Liquidate() to do that.

SetSwitchIn, SetSwitchOut, and SetTerminator set per-thread user switch and termination procedures. SwitchIn, SwitchOut, and Terminate call the user defined procedures then perform their own actions.

(Definition of class GUSIContext 34)≡ (29)

```
class GUSIContext : public GUSISpecificTable {
public:
    friend class GUSIProcess;
    friend class GUSIContextFactory;

    ThreadID          ID()          { return fThreadID;          }
    virtual void      Wakeup();
    void              ClearWakeup() { fWakeup = false;          }
    GUSIProcess *     Process()      { return fProcess;          }
    void              Detach()       { fFlags |= detached;        }
    void              Liquidate();
    OSErr             Error()        { return fError;            }
    bool              Done(bool join);
    void *            Result()        { return fResult;          }
    GUSISigContext *  SigContext() { return fSigContext;          }

    static GUSIContext * Current() { return sCurrentContext;      }
    static GUSIContext * Lookup(ThreadID id);
    static void          Setup(bool threading);
    static bool          Yield(bool wait);
    static void          SigWait(sigset_t sigs);
    static void          SigSuspend();
    static bool          Raise(bool allSigs = false);
    static sigset_t      Pending();
    static sigset_t      Blocked();

    void SetSwitchIn(ThreadSwitchProcPtr switcher, void *switchParam);
    void SetSwitchOut(ThreadSwitchProcPtr switcher, void *switchParam);
    void SetTerminator(ThreadTerminationProcPtr terminator, void *terminationParam);
```

```

        static GUSIContextQueue::iterator    begin() {    return sContexts.begin();    }
        static GUSIContextQueue::iterator    end()    {    return sContexts.end();    }
        static void LiquidateAll()            {    sContexts.LiquidateAll();    }
protected:
    ⟨Friends of GUSIContext 62⟩

    GUSIContext(ThreadID id);
    GUSIContext(
        ThreadEntryProcPtr threadEntry, void *threadParam,
        Size stackSize, ThreadOptions options = kCreateIfNeeded,
        void **threadResult = nil, ThreadID *threadMade = nil);

    virtual void SwitchIn();
    virtual void SwitchOut();
    virtual void Terminate();

    ⟨Privatissima of GUSIContext 48⟩
};

```

GUSIContext instances are created by instances of GUSIContextFactory.

⟨Definition of class GUSIContextFactory 35⟩≡ (29)

```

class GUSIContextFactory {
public:
    static GUSIContextFactory * Instance();
    static void                SetInstance(GUSIContextFactory * instance);

    virtual GUSIContext * CreateContext(ThreadID id);
    virtual GUSIContext * CreateContext(
        ThreadEntryProcPtr threadEntry, void *threadParam,
        Size stackSize, ThreadOptions options = kCreateIfNeeded,
        void **threadResult = nil, ThreadID *threadMade = nil);

    virtual ~GUSIContextFactory();
protected:
    GUSIContextFactory();
};

```

To maintain correct state, we have to remain informed which thread is active, so we install all sorts of hooks. Clients have to use the C++ interface or call GUSINewThread, GUSISetThreadSwitcher, and GUSISetThreadTerminator. instead of the thread manager routines.

⟨Definition of thread manager hooks 36⟩≡ (29)

```

OSErr GUSINewThread(ThreadStyle threadStyle, ThreadEntryProcPtr threadEntry, void *threadParam, Size stackSize,
OSErr GUSISetThreadSwitcher(ThreadID thread, ThreadSwitchProcPtr threadSwitcher, void *switchProcParam, Boolean
OSErr GUSISetThreadTerminator(ThreadID thread, ThreadTerminationProcPtr threadTerminator, void *terminationProcParam, Boolean

```

Many asynchronous calls take the same style of I/O parameter block and thus can be handled by the same completion procedure. StartIO prepares a parameter block for asynchronous I/O; FinishIO waits for the I/O to complete. The parameter block has to be wrapped in a GUSIIOPBWrapper.

(Definition of IO wrappers 37)≡ (29)

```

void GUSIStartIO(IOPParam * pb);
OSErr GUSIFinishIO(IOPParam * pb);
OSErr GUSIControl(IOPParam * pb);
template <class PB> struct GUSIIOPBWrapper {
    GUSIContext *    fContext;
    PB              fPB;

    GUSIIOPBWrapper() {}
    GUSIIOPBWrapper(const PB & pb) { memcpy(&fPB, &pb, sizeof(PB)); }

    PB * operator->(){ return &fPB; }
    void StartIO() { GUSIStartIO(reinterpret_cast<IOPParam *>(&fPB)); }
    OSErr FinishIO() { return GUSIFinishIO(reinterpret_cast<IOPParam *>(&fPB)); }
    OSErr Control() { return GUSIControl(reinterpret_cast<IOPParam *>(&fPB)); }
};

```

Ultimately, we will call through to the thread manager, but if an application uses foreign sources of threads, we might have to go through indirections.

(Definition of class GUSIThreadManagerProxy 38)≡ (29)

```

class GUSIThreadManagerProxy {
public:
    virtual OSErr NewThread(ThreadStyle threadStyle, ThreadEntryProcPtr threadEntry, void *threadParam);
    virtual OSErr SetThreadSwitcher(ThreadID thread, ThreadSwitchProcPtr threadSwitcher, void *switchParam);
    virtual OSErr SetThreadTerminator(ThreadID thread, ThreadTerminationProcPtr threadTerminator, void *terminationParam);

    virtual ~GUSIThreadManagerProxy() {}

    static GUSIThreadManagerProxy * Instance();
protected:
    GUSIThreadManagerProxy() {}

    static GUSIThreadManagerProxy * MakeInstance();
};

```

3.2 Implementation of completion handling

Instance() returns the sole instance of GUSIProcess, creating it if necessary.

(Privatissima of GUSIProcess 39)≡ (32) 42▷

```

static GUSIProcess *    sInstance;

```

(Implementation of completion handling 40)≡ (30) 44▷

```

GUSIProcess * GUSIProcess::sInstance;

```


<Inline member functions for file GUSIContext 41>+≡ (29) 43▷

```
inline GUSIPProcess * GUSIPProcess::Instance()
{
    if (!sInstance)
        sInstance = new GUSIPProcess(GUSIContext::sHasThreading);
    return sInstance;
}
```

All of the information stored in a GUSIPProcess is static and read-only.

<Privatissima of GUSIPProcess 39>+≡ (32) 439

```
ProcessSerialNumber fProcess;
ThreadTaskRef      fTaskRef;
long               fA5;
bool               fWillSleep;
bool               fDontSleep;
```

<Inline member functions for file GUSIContext 41>+≡ (29) 441 45▷

```
inline void GUSIPProcess::GetPSN(ProcessSerialNumber * psn)
    { *psn = fProcess; }

inline void GUSIPProcess::AcquireTaskRef()
    { GetThreadCurrentTaskRef(&fTaskRef); }

inline ThreadTaskRef GUSIPProcess::GetTaskRef()
    { return fTaskRef; }

inline long GUSIPProcess::GetA5()
    { return fA5; }

inline bool GUSIPProcess::Threading()
    { return fTaskRef!=0; }
```

<Implementation of completion handling 40>+≡ (30) 440 46▷

GUSI_NEEDS_QD

```
GUSIPProcess::GUSIPProcess(bool threading)
{
    GetCurrentProcess(&fProcess);
    fA5 = (long) LMGetCurrentA5();
    if (threading)
        AcquireTaskRef();
    else
        fTaskRef = 0;
    if (*(GrafPtr **) fA5 != &qd.thePort)
        InitGraf(&qd.thePort);
    fReadyThreads = 0;
    fWillSleep = false;
    fDontSleep = false;
    fSigProcess= GUSISigFactory::Instance()->CreateSigProcess();
}
```

An ASaver is trivially implemented but it simplifies bookkeeping.

(Inline member functions for file GUSIContext 41) +≡ (29) <43

```
inline GUSIProcess::ASaver::ASaver(long processA5)
{   fSavedA5 = SetA5(processA5);           }
inline GUSIProcess::ASaver::ASaver(GUSIProcess * process)
{   fSavedA5 = SetA5(process->GetA5());     }
inline GUSIProcess::ASaver::ASaver(GUSIContext * context)
{   fSavedA5 = SetA5(context->Process()->GetA5()); }
inline GUSIProcess::ASaver::~ASaver()
{   SetA5(fSavedA5);                       }
```

GUSIContext::Setup() initializes the default context.

(Implementation of completion handling 40) +≡ (30) <44 49>

```
void GUSIContext::Setup(bool threading)
{
    bool    wasThreading = sHasThreading;
    if (threading)
        sHasThreading = true;
    if (!sCurrentContext)
        sCurrentContext =
            new GUSIContext(kApplicationThreadID);
    else if (!wasThreading && threading) {
        (Upgrade application context to threading 47)
    }
}
```

Sometimes we only recognize that we need threading after the application context has already been created. Assuming a disciplined use of threads, we can assume that the current context is still the application context, so all we have to do is to set the thread switcher for it.

(Upgrade application context to threading 47) ≡ (46)

```
GUSIThreadManagerProxy::Instance()->SetThreadSwitcher(
    kApplicationThreadID, (ThreadSwitchProcPtr)GUSIThreadSwitchIn, sCurrentContext, true);
GUSIThreadManagerProxy::Instance()->SetThreadSwitcher(
    kApplicationThreadID, (ThreadSwitchProcPtr)GUSIThreadSwitchOut, sCurrentContext, false);
GUSIProcess::Instance()->AcquireTaskRef();
```

At this point, we need to introduce all the private data of a GUSIContext.

- fThreadID stores the thread manager thread ID.
- fProcess keeps a pointer to the process structure, so completion routines can get at it.
- sCurrentContext always points at the current context.
- sContexts contains a queue of all contexts.
- sHasThreads reminds us whether we are threading or not.
- Since we define our own switch-in and termination procedures, we have to keep user specified procedures in fSwitchInProc, fSwitchOutProc, and fTerminateProc and their parameters in fSwitchInParam, fSwitchOutParam, and fTerminateParam so we can call through to them from our procedures.
- fJoin contains the context waiting for us to die;
- done reminds us if the thread is still alive. detached guarantees that we will never wait for that thread anymore.
- Last of all, we keep the global error variables errno and h_errno for each context in the fErrno and fHostErrno fields.

```

(Privatissima of GUSIContext 48)≡ (34) 50▸
ThreadID          fThreadID;
GUSIProcess *     fProcess;
GUSIContext *     fNext;
GUSISigContext *  fSigContext;
ThreadSwitchProcPtr fSwitchInProc;
ThreadSwitchProcPtr fSwitchOutProc;
ThreadTerminationProcPtr fTerminateProc;
void *            fSwitchInParam;
void *            fSwitchOutParam;
void *            fTerminateParam;
void *            fResult;
GUSIContext *     fJoin;
enum {
    done      = 1 << 0,
    detached= 1 << 1,
    asleep    = 1 << 2
};
char          fFlags;
bool          fWakeup;
OSErr         fError;
int           fErrno;
int           fHostErrno;
class Queue : public GUSIContextQueue {
public:
    void LiquidateAll();

    ~Queue()          { LiquidateAll(); }
};

```


The preexisting thread constructor is now simple to define.

```
(Implementation of completion handling 40)+≡ (30) <51 53>
GUSIContext::GUSIContext(ThreadID id)
{
    StartSetup();
    fThreadID = id;
    FinishSetup();
}
```

And so is the creation constructor, come to think of it.

```
(Implementation of completion handling 40)+≡ (30) <52 54>
GUSIContext::GUSIContext(ThreadEntryProcPtr threadEntry, void *threadParam,
    Size stackSize, ThreadOptions options, void ** result, ThreadID * thread)
{
    StartSetup();
    if (!result)
        result = &fResult;
    fError = GUSIThreadManagerProxy::Instance()->NewThread(
        kCooperativeThread, threadEntry, threadParam,
        stackSize, options, result, &fThreadID);
    if (thread)
        *thread = fThreadID;
    if (fError)
        return;
    FinishSetup();
}
```

GUSIContext::Lookup() does a linear search.

```
(Implementation of completion handling 40)+≡ (30) <53 56>
GUSIContext * GUSIContext::Lookup(ThreadID id)
{
    for (GUSIContextQueue::iterator context = begin(); context != end(); ++context)
        if (context->fThreadID == id)
            return *context;
    return nil;
}
```

Destruction of a GUSIContext requires some cleanup.

```
(Privatissima of GUSIContext 48)+≡ (34) <50
~GUSIContext();

(Implementation of completion handling 40)+≡ (30) <54 57>
GUSIContext::~GUSIContext()
{
    sContexts.remove(this);
}
```

Furthermore, at the end of an application, we need some global cleanup. This is especially true for MPW tools.

```

<Implementation of completion handling 40>+≡                                     (30) <56 58>
void GUSIContext::Queue::LiquidateAll()
{
    while (!empty())
        front()->Liquidate();
}

void GUSIContext::Liquidate()
{
    GUSI_MESSAGE4(("GUSIContext::Queue::Liquidate %08x\n", fThreadID));
    switch (fThreadID) {
    case kApplicationThreadID: // Main thread, restore switchers
        if (sHasThreading) {
            SetThreadSwitcher(fThreadID, fSwitchInProc, fSwitchInParam, true);
            SetThreadSwitcher(fThreadID, fSwitchOutProc, fSwitchOutParam, true);
        }
        break;
    default: // Other thread, terminate
        if (!(fFlags & done)) {
            ThreadState state;

            GetThreadState(fThreadID, &state);
            if (state != kStoppedThreadState)
                --fProcess->fReadyThreads;

            DisposeThread(fThreadID, nil, false);
        }
        break;
    }
    delete this;
}

```

GUSIContext::Wakeup awakens a context. In every asynchronous call, there is a risk of race conditions, i.e., the call completes before the thread/process making it goes to sleep. For threads, we can handle this case by checking for a wakeup in the switch out procedure (an idea due to Quinn).

```

<Implementation of completion handling 40>+≡                                     (30) <57 59>
void GUSIContext::Wakeup()
{
    GUSI_MESSAGE(("Wakeup #%d\n", fThreadID));
    if (fThreadID && fThreadID != kApplicationThreadID) {
        fWakeup = true;
        SetThreadReadyGivenTaskRef(Process()->GetTaskRef(), fThreadID);
    }
    Process()->Wakeup();
}

```

The code for processes is quite complex. There are three cases for a wakeup:

- If the process to be woken up is not the current process, we can simply call `WakeUpProcess`.
- Otherwise, if `fWillSleep` has not yet been set, we can prevent the process from going to sleep by setting `fDontSleep`.
- In the rare case where that does not work, we post a null event. (An idea due to George Warner).

(Implementation of completion handling 40) +≡ (30) <58 60>

```
void GUSIPProcess::Wakeup()
{
    ProcessSerialNumber we;
    Boolean same;

    we.highLongOfPSN = 0;
    we.lowLongOfPSN = kCurrentProcess;
    if (!SameProcess(&fProcess, &we, &same) && !same) {
        GUSI_SMESSAGE("WakeUpProcess\n");
        GUSI_CASSERT_EXTERNAL(!WakeUpProcess(&fProcess));
    } else if (!fWillSleep) {
        GUSI_SMESSAGE("DontSleep\n");
        fDontSleep = true;
    } else {
        GUSI_SMESSAGE("PostEvent\n");
        PostEvent(nullEvent, 0);
    }
}
```

The thread wrappers are fairly trivial.

(Implementation of completion handling 40) +≡ (30) <59 61>

```
OSErr GUSINewThread(ThreadStyle, ThreadEntryProcPtr threadEntry, void *threadParam, Size stackSize, ThreadOptions options)
{
    GUSIContext * context = GUSIContextFactory::Instance()->CreateContext(
        threadEntry, threadParam, stackSize, options,
        threadResult, threadMade);
    OSErr err = context->Error();
    if (err)
        context->Liquidate();
    return err;
}
```

(Implementation of completion handling 40)+≡ (30) 460 63▷

```
static auto_ptr<GUSIContextFactory> sGUSIContextFactory;

GUSIContextFactory * GUSIContextFactory::Instance()
{
    if (!sGUSIContextFactory.get())
        SetInstance(new GUSIContextFactory());

    return sGUSIContextFactory.get();
}

void GUSIContextFactory::SetInstance(GUSIContextFactory * instance)
{
    sGUSIContextFactory = auto_ptr<GUSIContextFactory>(instance);
}

GUSIContextFactory::GUSIContextFactory()
{
}

GUSIContextFactory::~GUSIContextFactory()
{
}
```

(Friends of GUSIContext 62)+≡ (34) 67▷

```
friend class GUSIContextFactory;
```

(Implementation of completion handling 40)+≡ (30) 461 64▷

```
GUSIContext * GUSIContextFactory::CreateContext(
    ThreadEntryProcPtr threadEntry, void *threadParam,
    Size stackSize, ThreadOptions options, void **threadResult, ThreadID *threadMade
)
{
    GUSIContext::Setup(true);

    return new GUSIContext(threadEntry, threadParam, stackSize, options,
        threadResult, threadMade);
}

GUSIContext * GUSIContextFactory::CreateContext(ThreadID threadMade)
{
    GUSIContext::Setup(true);

    return new GUSIContext(threadMade);
}
```


The thread switcher dispatches to the appropriate member function.

(Implementation of completion handling 40) +≡

(30) <63 65>

```
OSErr GUSISetThreadSwitcher(ThreadID thread, ThreadSwitchProcPtr threadSwitcher, void *switchProcParam, bool i
{
    GUSIContext * context;
    if (!(context = GUSIContext::Lookup(thread)))
        return GUSIThreadManagerProxy::Instance()->SetThreadSwitcher(thread, threadSwitcher, switchProcParam,
    if (inOrOut)
        context->SetSwitchIn(threadSwitcher, switchProcParam);
    else
        context->SetSwitchOut(threadSwitcher, switchProcParam);
    return noErr;
}
```

```
void GUSIContext::SetSwitchIn(ThreadSwitchProcPtr switcher, void *switchParam)
{
    fSwitchInProc = switcher;
    fSwitchInParam= switchParam;
}
```

```
void GUSIContext::SetSwitchOut(ThreadSwitchProcPtr switcher, void *switchParam)
{
    fSwitchOutProc = switcher;
    fSwitchOutParam= switchParam;
}
```

Similar for the thread terminator.

(Implementation of completion handling 40) +≡

(30) <64 66>

```
OSErr GUSISetThreadTerminator(ThreadID thread, ThreadTerminationProcPtr threadTerminator, void *terminationPro
{
    GUSIContext * context;
    if (!(context = GUSIContext::Lookup(thread)))
        return GUSIThreadManagerProxy::Instance()->SetThreadTerminator(thread, threadTerminator, terminationPr
    context->SetTerminator(threadTerminator, terminationProcParam);

    return noErr;
}
```

```
void GUSIContext::SetTerminator(ThreadTerminationProcPtr terminator, void *terminationParam)
{
    fTerminateProc = terminator;
    fTerminateParam= terminationParam;
}
```

The hooks, after having performed their task, call through to user defined hooks.

```
<Implementation of completion handling 40>+≡ (30) <65 68>
#if GENERATING68K && GENERATINGCFM
#define CallThreadSwitchProc(userRoutine, thread, context) \
    CallUniversalProc((userRoutine), uppThreadSwitchProcInfo, (thread), (context))
#define CallThreadTerminationProc(userRoutine, thread, context) \
    CallUniversalProc((userRoutine), uppThreadTerminationProcInfo, (thread), (context))
#else
#define CallThreadSwitchProc(userRoutine, thread, context) \
    (*userRoutine)((thread), (context))
#define CallThreadTerminationProc(userRoutine, thread, context) \
    (*userRoutine)((thread), (context))
#endif
```

The thread switcher updates the pointer to the current context and switches the global error variables.

```
<Friends of GUSIContext 62>+≡ (34) <62 70>
friend pascal void GUSIThreadSwitchIn(ThreadID thread, GUSIContext * context);
friend pascal void GUSIThreadSwitchOut(ThreadID thread, GUSIContext * context);

<Implementation of completion handling 40>+≡ (30) <66 69>
pascal void GUSIThreadSwitchIn(ThreadID, GUSIContext * context)
{
    context->SwitchIn();
}

void GUSIContext::SwitchIn()
{
    if (sCurrentContext != this)
        GUSI_MESSAGE(("Yield #%d -> #%d\n", sCurrentContext->fThreadID, fThreadID));
    sCurrentContext = this;
    errno = fErrno;
    h_errno = fHostErrno;

    if (fFlags & asleep) {
        fFlags &= ~asleep;
        ++fProcess->fReadyThreads;
    }

    if (fSwitchInProc)
        CallThreadSwitchProc(fSwitchInProc, fThreadID, fSwitchInParam);
}
```

When we are switched out, we check whether we should wake up again right away.

(Implementation of completion handling 40)+≡ (30) <68 71>

```
pascal void GUSIThreadSwitchOut(ThreadID, GUSIContext * context)
{
    context->SwitchOut();
}

void GUSIContext::SwitchOut()
{
    if (fSwitchOutProc)
        CallThreadSwitchProc(fSwitchOutProc, fThreadID, fSwitchOutParam);
    fErrno      =   errno;
    fHostErrno  =   h_errno;

    ThreadTaskRef taskRef = Process()->GetTaskRef();
    ThreadState state;
    if (!GetThreadStateGivenTaskRef(taskRef, fThreadID, &state))
        if (state == kStoppedThreadState)
            if (fWakeup) {
                SetThreadReadyGivenTaskRef(taskRef, fThreadID);
            } else {
                GUSI_MESSAGE(("Sleep #%d\n", fThreadID));
                fFlags |= asleep;
                --fProcess->fReadyThreads;
                sCurrentContext = nil;
            }
}
```

The terminator wakes up all other threads if a join is pending. Joins are probably rather infrequent, so it's not worth setting up a great deal of context management for this.

(Friends of GUSIContext 62)+≡

(34) <67

```
friend pascal void GUSIThreadTerminator(ThreadID thread, GUSIContext * context);
```

(Implementation of completion handling 40)+≡

(30) <69 72>

```
pascal void GUSIThreadTerminator(ThreadID, GUSIContext * context)
{
    context->Terminate();
}

void GUSIContext::Terminate()
{
    if (fTerminateProc)
        CallThreadTerminationProc(fTerminateProc, fThreadID, fTerminateParam);
    if (fFlags & detached)
        delete this;
    else {
        fFlags |= done;
        if (fJoin)
            fJoin->Wakeup();
    }
    GUSI_MESSAGE(("Terminate #%d\n", fThreadID));
}
```

Done is simple unless join is set. Otherwise, we set fJoin and wait. If some other process is already joining, we bail out.

```

<Implementation of completion handling 40>+≡ (30) 471 73»
bool GUSIContext::Done(bool join)
{
    if ((fFlags & done) || !join || fJoin)
        return (fFlags & done);
    fJoin = GUSIContext::sCurrentContext;
    while (!(fFlags & done))
        Yield(true);
    return (fFlags & done);
}

```

Yield() tries to do the smart thing in all contexts. The basic idea is that both in threading and non-threading contexts, everyone gets their fair turn.

```

<Implementation of completion handling 40>+≡ (30) 472 77»
bool GUSIContext::Yield(bool wait)
{
    bool interrupt = false;
    GUSIProcess * process = GUSIProcess::Instance();

    <Suspend the current process if possible 74>
    <Check for eligible signals 75>
    <Suspend the current thread if possible 76>
done:
    sCurrentContext->fWakeup = false;

    return interrupt;
}

```

```

<Suspend the current process if possible 74>≡ (73 77)
if (sCurrentContext->fThreadID == kApplicationThreadID) {
    process->Yield(wait);
    wait = false;    // Main thread never blocks
}

```

```

<Check for eligible signals 75>≡ (73)
if (interrupt = Raise())
    goto done;

```

```

<Suspend the current thread if possible 76>≡ (73 77)
if (sHasThreading) {
    if (wait)
        SetThreadState(kCurrentThreadID, kStoppedThreadState, kNoThreadID);
    else
        YieldToAnyThread();
}

```

SigWait and SigSuspend are similar to Yield. The former waits for one of the specified signals to become pending, while the latter waits for any signal to become executed.

⟨Implementation of completion handling 40⟩+≡

(30) ◀73 78▶

```
void GUSIContext::SigWait(sigset_t sigs)
{
    GUSIProcess *   process = GUSIProcess::Instance();
    bool            wait    = true;

    for (;;) {
        ⟨Suspend the current process if possible 74⟩
        if (Pending() & sigs)
            break;
        Raise();
        ⟨Suspend the current thread if possible 76⟩
    }
    sCurrentContext->fWakeup = false;
}

void GUSIContext::SigSuspend()
{
    GUSIProcess *   process = GUSIProcess::Instance();
    bool            wait    = true;

    for (;;) {
        ⟨Suspend the current process if possible 74⟩
        if (Raise(true))
            break;
        ⟨Suspend the current thread if possible 76⟩
    }
    sCurrentContext->fWakeup = false;
}
```

GUSIProcess::Yield tries to narrow down the critical time for race conditions, so we rarely have to force a wakeup.

(Implementation of completion handling 40)+≡ (30) 477 79»

```
void GUSIProcess::Yield(bool wait)
{
    GUSIConfiguration::Instance()->CheckInterrupt();
    if (fReadyThreads > 1)
        wait = false;
    if (wait) {
        fWillSleep = true;
        if (fDontSleep)
            wait = false;
    }
    if (gGUSISpinHook)
        gGUSISpinHook(wait);
    else if (wait) {
        GUSI_SMESSAGE("Suspend\n");
        GUSIHandleNextEvent(600);
        GUSI_SMESSAGE("Resume\n");
    } else
        GUSIHandleNextEvent(0);

    fWillSleep = false;
    fDontSleep = false;
}
```

Raise raises all eligible signals. Pending returns the pending signals. Blocked() returns the blocked signals.

(Implementation of completion handling 40)+≡ (30) 478 80»

```
bool GUSIContext::Raise(bool allSigs)
{
    return sCurrentContext->SigContext()->Raise(
        GUSIProcess::Instance()->SigProcess(), allSigs);
}

sigset_t GUSIContext::Pending()
{
    return sCurrentContext->SigContext()->Pending(
        GUSIProcess::Instance()->SigProcess());
}

sigset_t GUSIContext::Blocked()
{
    return sCurrentContext->SigContext()->GetBlocked();
}
```

Many different asynchronous calls can be handled by GUSIIDone.

(Implementation of completion handling 40) +≡ (30) <79 81>

```
inline GUSIContext *& Context(IOParam * pb)
{
    return reinterpret_cast<GUSIContext **>(pb)[-1];
}
static void GUSIIDone(IOParam * pb)
{
    if (Context(pb))
        Context(pb)->Wakeup();
}

GUSI_COMPLETION_PROC_A0(GUSIIDone, IOParam)
```

Since we (or at least I) never quite know when a call is executed synchronously, we set the context field to nil until after the call. This avoids having to wake up a running context, which is a fairly costly operation.

(Implementation of completion handling 40) +≡ (30) <80 82>

```
void GUSIStartIO(IOParam * pb)
{
    static IOCompletionUPP sIODone = 0;

    if (!sIODone)
        sIODone = NewIOCompletionProc(GUSIIDoneEntry);
    Context(pb) = nil;
    pb->ioCompletion = sIODone;
}

OSErr GUSIFinishIO(IOParam * pb)
{
    Context(pb) = GUSIContext::Current();
    while (pb->ioResult > 0)
        GUSIContext::Yield(true);
    return pb->ioResult;
}

OSErr GUSIControl(IOParam * pb)
{
    GUSIStartIO(pb);
    PBControlAsync(reinterpret_cast<ParmBlkPtr>(pb));
    return GUSIFinishIO(pb);
}
```

The default implementation of GUSIThreadManagerProxy is trivial.

<Implementation of completion handling 40>+≡

(30) <81

```
static auto_ptr<GUSIThreadManagerProxy> sGUSIThreadManagerProxy;
```

```
OSErr GUSIThreadManagerProxy::NewThread(ThreadStyle threadStyle, ThreadEntryProcPtr threadEntry, void* threadParam,
{
    return ::NewThread(threadStyle, threadEntry, threadParam, stackSize, options, threadResult, threadParam);
}
```

```
OSErr GUSIThreadManagerProxy::SetThreadSwitcher(ThreadID thread, ThreadSwitchProcPtr threadSwitcher, void* threadParam,
{
    return ::SetThreadSwitcher(thread, threadSwitcher, switchProcParam, inOrOut);
}
```

```
OSErr GUSIThreadManagerProxy::SetThreadTerminator(ThreadID thread, ThreadTerminationProcPtr threadTerminator, void* threadParam,
{
    return ::SetThreadTerminator(thread, threadTerminator, terminationProcParam);
}
```

```
GUSIThreadManagerProxy * GUSIThreadManagerProxy::Instance()
{
    if (!sGUSIThreadManagerProxy.get())
        sGUSIThreadManagerProxy = auto_ptr<GUSIThreadManagerProxy>(MakeInstance());
    return sGUSIThreadManagerProxy.get();
}
```

```
GUSIThreadManagerProxy * GUSIThreadManagerProxy::MakeInstance()
{
    return new GUSIThreadManagerProxy;
}
```


Chapter 4

Thread Specific Variables

It is often useful to have variables which maintain a different value for each process. The GUSISpecific class implements such a mechanism in a way that is easily mappable to pthreads.

```
<GUSISpecific.h 83>≡
    #ifndef _GUSISpecific_
    #define _GUSISpecific_

    #ifndef GUSI_SOURCE

    typedef struct GUSISpecific GUSISpecific;

    #else

    #include <Types.h>

    <Definition of class GUSISpecific 85>
    <Definition of class GUSISpecificTable 86>
    <Definition of template GUSISpecificData 87>

    <Inline member functions for class GUSISpecific 89>
    <Inline member functions for class GUSISpecificTable 91>

    #endif /* GUSI_SOURCE */

    #endif /* _GUSISpecific_ */
```

```

<GUSISpecific.cp 84>≡
#include "GUSIInternal.h"
#include "GUSISpecific.h"
#include "GUSIContext.h"
#include "GUSIDdiag.h"

#include <utility>

GUSI_USING_STD_NAMESPACE

#include <TextUtils.h>

<Member functions for class GUSISpecific 88>
<Member functions for class GUSISpecificTable 93>

```

4.1 Definition of Thread Specific Variables

A GUSISpecific instance contains a variable ID and a per-process destructor.

```

<Definition of class GUSISpecific 85>≡ (83)
extern "C" {
    typedef void (*GUSISpecificDestructor)(void *);
}

class GUSISpecific {
    friend class GUSISpecificTable;
public:
    GUSISpecific(GUSISpecificDestructor destructor = nil);
    ~GUSISpecific();

    void Destruct(void * data);
private:
    GUSISpecificDestructor fDestructor;
    unsigned fID;
    static unsigned sNextID;
};

```

A GUSIContext contains a GUSISpecificTable storing the values of all thread specific variables defined for this thread.

```

<Definition of class GUSISpecificTable 86>≡ (83)
class GUSISpecificTable {
    friend class GUSISpecific;
public:
    GUSISpecificTable();
    ~GUSISpecificTable();
    void * GetSpecific(const GUSISpecific * key) const;
    void SetSpecific(const GUSISpecific * key, void * value);
    void DeleteSpecific(const GUSISpecific * key);
private:
    static void Register(GUSISpecific * key);
    static void Destruct(GUSISpecific * key);

    <Privatissima of GUSISpecificTable 90>
};

```

To simplify having a particular variable assume a different instance in every thread, we define the GUSISpecificData template.

```

(Definition of template GUSISpecificData 87)≡ (83)
template <class T, GUSISpecificDestructor D>
class GUSISpecificData {
public:
    GUSISpecificData() : fKey(D) { }
    T & operator*() { return *get(); }
    T * operator->() { return get(); }

    const GUSISpecific * Key() const { return &fKey; }
    T * get(GUSISpecificTable * table);
    T * get() { return get(GUSIContext::Current()); }
protected:
    GUSISpecific fKey;
};

template <class T, GUSISpecificDestructor D>
T * GUSISpecificData<T,D>::get(GUSISpecificTable * table)
{
    void * data = table->GetSpecific(&fKey);

    if (!data)
        table->SetSpecific(&fKey, data = new T);

    return static_cast<T *>(data);
}

```

4.2 Implementation of Thread Specific Variables

GUSISpecific is trivial to implement, so we keep all of the members inline.

```

(Member functions for class GUSISpecific 88)≡ (84)
    unsigned GUSISpecific::sNextID = 0;

(Inline member functions for class GUSISpecific 89)≡ (83)
    inline GUSISpecific::GUSISpecific(GUSISpecificDestructor destructor)
        : fDestructor(destructor), fID(sNextID++)
    {
        GUSISpecificTable::Register(this);
    }

    inline GUSISpecific::~~GUSISpecific()
    {
        GUSISpecificTable::Destruct(this);
    }

    inline void GUSISpecific::Destruct(void * data)
    {
        if (fDestructor)
            fDestructor(data);
    }

```

We store a GUSISpecificTable as a contiguous range of IDs.

```
(Privatissima of GUSISpecificTable 90)≡ (86) 97▷
void ***          fValues;
unsigned          fAlloc;

bool              Valid(const GUSISpecific * key) const;

(Inline member functions for class GUSISpecificTable 91)≡ (83) 92▷
inline bool GUSISpecificTable::Valid(const GUSISpecific * key) const
{
    return key && key->fID < fAlloc;
}

(Inline member functions for class GUSISpecificTable 91)+≡ (83) 94▷
inline GUSISpecificTable::GUSISpecificTable()
    : fValues(nil), fAlloc(0)
{
}

(Member functions for class GUSISpecificTable 93)≡ (84) 95▷
GUSISpecificTable::~~GUSISpecificTable()
{
    void * data;

    if (fValues)
        for (unsigned id = 0; id < fAlloc; ++id)
            while ((data = fValues[0][id]) && sKeys[0][id]) {
                fValues[0][id] = 0;
                sKeys[0][id]->Destruct(data);
            }
}

(Inline member functions for class GUSISpecificTable 91)+≡ (83) 92
inline void * GUSISpecificTable::GetSpecific(const GUSISpecific * key) const
{
    if (Valid(key))
        return fValues[0][key->fID];
    else
        return nil;
}
```


<Member functions for class GUSISpecificTable 93>+≡ (84) <98 100>

```
void GUSISpecificTable::Register(GUSISpecific * key)
{
    if (key->fID >= sKeyAlloc) {
        unsigned newAlloc = (key->fID & ~7) + 8;
        size_t allocSize = newAlloc*sizeof(GUSISpecific *);

        if (!sKeys)
            sKeys = (GUSISpecific ***) NewHandle(allocSize);
        else
            SetHandleSize((Handle)sKeys, allocSize);

        while (sKeyAlloc < newAlloc)
            sKeys[0][sKeyAlloc++] = nil;
    }
    sKeys[0][key->fID] = key;
}
```

Note that this doesn't call any destructors.

<Member functions for class GUSISpecificTable 93>+≡ (84) <99

```
void GUSISpecificTable::Destruct(GUSISpecific * key)
{
    sKeys[0][key->fID] = nil;
}
```

Chapter 5

The GUSI Socket Class

GUSI is constructed around the `GUSISocket` class. This class is mostly an abstract superclass, but all virtual procedures are implemented to return sensible error codes.

```
<GUSISocket.h 101>≡
    #ifndef _GUSISocket_
    #define _GUSISocket_

    #ifdef GUSI_SOURCE

        #include "GUSIBasics.h"
        #include "GUSIContext.h"
        #include "GUSIContextQueue.h"
        #include "GUSIBuffer.h"

        #include <sys/types.h>
        #include <sys/socket.h>
        #include <stdarg.h>

        <Definition of class GUSISocket 103>
        <Inline member functions for class GUSISocket 115>

    #endif /* GUSI_SOURCE */

    #endif /* _GUSISocket_ */

<GUSISocket.cp 102>≡
    #include "GUSIInternal.h"
    #include "GUSISocket.h"
    #include "GUSIDdiag.h"
    #include "GUSIBuffer.h"

    #include <errno.h>
    #include <sys/stat.h>

    <Auxiliary data structures for class GUSISocket 127>
    <Member functions for class GUSISocket 116>
```

5.1 Definition of GUSISocket

GUSISocket consists of a few maintenance functions and the socket operations. Each operation consists to a POSIX/BSD function with the file descriptor operand left out.

(Definition of class GUSISocket 103)≡ (101)

```
class GUSISocket {
    <Reference counting for GUSISocket 104>
    <Context links for GUSISocket 105>
    <Configuration options for GUSISocket 106>
public:
    <Socket name management for GUSISocket 107>
    <Connection establishment for GUSISocket 108>
    <Sending and receiving data for GUSISocket 109>
    <Maintaining properties for GUSISocket 110>
    <File oriented operations for GUSISocket 111>
    <Multiplexing for GUSISocket 112>
    <Miscellaneous operations for GUSISocket 113>
};
```

Since a single GUSISocket may (through `dup()`) be installed multiply in a descriptor table or even in multiple descriptor tables, GUSISockets are not destroyed directly, but by manipulating a reference count. As soon as the reference count hits zero, the destructor (which, of course, should probably be overridden) is called.

Since destructors cannot call virtual functions, we call `close()` which then calls the destructor.

(Reference counting for GUSISocket 104)≡ (103)

```
public:
    void    AddReference();
    void    RemoveReference();

    virtual void close();
    virtual ~GUSISocket();
protected:
    GUSISocket();
private:
    u_long  fRefCount;
```

GUSIContexts are defined in GUSIBasics. A context references all information you need in a completion procedure: The contents of A5, the process ID, and thread information. `Wakeup()` wakes up the threads and/or processes associated with the socket and is guaranteed to work even at interrupt level. `AddContext()` adds another context. `RemoveContext()` indicates that this context no longer should be woken up when something happens.

(Context links for GUSISocket 105)≡ (103)

```
public:
    void                Wakeup();
    void                AddContext(GUSIContext * context = nil);
    void                RemoveContext(GUSIContext * context = nil);
private:
    GUSIContextQueue    fContexts;
```


Both read and write calls on sockets come in five different variants:

1. `read()` and `write()`
2. `recv()` and `send()`
3. `readv()` and `writenv()`
4. `recvfrom()` and `sendto()`
5. `recvmsg()` and `sendmsg()`

GUSI initially maps variants 3 and 5 of these calls to the `recvmsg()` and `sendmsg()` member functions, variants 2 and 4 to the `recvfrom()` and `sendto()` member functions, and variant 1 to the `read()` and `write()` member functions.

The simpler member functions can always be translated into the complex member functions, and under some circumstances, the opposite is also possible. To avoid translation loops, the translation routines (i.e., the default implementation of `GUSISocket::read()` and `GUSISocket::recvmsg()`) check for the availability of the other function by calling `Supports()`. This member function must be overridden for any reasonable socket class.

```
<Configuration options for GUSISocket 106>≡ (103)
protected:
    enum ConfigOption {
        kSimpleCalls,      // [[read()]], [[write()]]
        kSocketCalls,      // [[recvfrom()]], [[sendto()]]
        kMsgCalls           // [[recvmsg()]], [[sendmsg()]]
    };
    virtual bool    Supports(ConfigOption config);
```

Most sockets have names, which to `GUSISocket` are just opaque blocks of memory. A name for a socket is established (before the socket is actually used, of course) through `bind()`. The name may be queried with `getsockname()` and once the socket is connected, the name of the peer endpoint may be queried with `getpeername()`.

```
<Socket name management for GUSISocket 107>≡ (103)
virtual int bind(void * name, socklen_t namelen);
virtual int getsockname(void * name, socklen_t * namelen);
virtual int getpeername(void * name, socklen_t * namelen);
```

Sockets follow either a virtual circuit model where all data is exchanged with the same peer throughout the lifetime of the connection, or a datagram model where potentially every message is exchanged with a different peer.

The vast majority of protocols follow the virtual circuit model. The server end, typically after calling `bind()` to attach the socket to a well known address, calls `listen()` to establish its willingness to accept connections. `listen()` takes a queue length parameter, which however is ignored for many types of sockets.

Incoming connections are then accepted by calling `accept()`. When `accept()` is successful, it always returns a new `GUSISocket`, while the original socket remains available for further connections. To avoid blocking on `accept()`, you may poll for connections with an `accept()` call in nonblocking mode or query the result of `[[select()` whether the socket is ready for reading.

The client end in the virtual circuit model connects itself to the well known address by calling `connect()`. To avoid blocking on `connect()`, you may call it in nonblocking mode and then query the result of `select()` whether the socket is ready for writing.

In the datagram model, you don't need to establish connections. You may call `connect()` anyway to temporarily establish a virtual circuit.

(Connection establishment for GUSISocket 108)≡ (103)

```
virtual int listen(int qlen);
virtual GUSISocket * accept(void * address, socklen_t * addrlen);
virtual int connect(void * address, socklen_t addrlen);
```

As mentioned before, there are three variants each for reading and writing. The socket variants provide a means to pass a peer address for the datagram model, while the msg variants also provides fields for passing access rights, which is, however not currently supported in GUSI. As syntactic sugar, the more traditional flavors with buffer/length buffers are also supported.

(Sending and receiving data for GUSISocket 109)≡ (103)

```
virtual ssize_t read(const GUSIScatterer & buffer);
virtual ssize_t write(const GUSIGatherer & buffer);
virtual ssize_t recvfrom(const GUSIScatterer & buffer, int flags, void * from, socklen_t * fromlen);
virtual ssize_t sendto(const GUSIGatherer & buffer, int flags, const void * to, socklen_t tolen);
virtual ssize_t recvmsg(msghdr * msg, int flags);
virtual ssize_t sendmsg(const msghdr * msg, int flags);

ssize_t read(void * buffer, size_t length);
ssize_t write(const void * buffer, size_t length);
ssize_t recvfrom(void * buffer, size_t length, int flags, void * from, socklen_t * fromlen);
ssize_t sendto(const void * buffer, size_t length, int flags, const void * to, socklen_t tolen);
```

A multitude of parameters can be manipulated for a `GUSISocket` through the socket oriented calls `getsockopt()`, `setsockopt()`, the file oriented call `fcntl()`, and the device oriented call `ioctl()`.

`isatty()` returns whether the socket should be considered an interactive console.

(Maintaining properties for GUSISocket 110)≡ (103)

```
virtual int getsockopt(int level, int optname, void *optval, socklen_t * optlen);
virtual int setsockopt(int level, int optname, void *optval, socklen_t optlen);
virtual int fcntl(int cmd, va_list arg);
virtual int ioctl(unsigned int request, va_list arg);
virtual int isatty();
```

Three of the operations make sense primarily for files, and most other socket types accept the default implementations. `fstat()` returns information about an open file, `lseek()` repositions the read/write pointer, and `ftruncate` cuts off an open file at a certain point.

```
<File oriented operations for GUSISocket 111>≡ (103)
virtual int fstat(struct stat * buf);
virtual off_t lseek(off_t offset, int whence);
virtual int ftruncate(off_t offset);
```

`select()` polls or waits for one of a group of `GUSISocket` to become ready for reading, writing, or for an exceptional condition to occur. First, `pre_select()` is called once for all `GUSISockets` in the group. It returns true is the socket will wake up as soon as one of the events occurs and false if GUSI needs to poll. Next, `select()` is called for all `GUSISockets` once or multiple times, until a condition becomes true or the call times out. Finally, `post_select` is called for all members of the group.

```
<Multiplexing for GUSISocket 112>≡ (103)
virtual bool pre_select(bool wantRead, bool wantWrite, bool wantExcept);
virtual bool select(bool * canRead, bool * canWrite, bool * exception);
virtual void post_select(bool wantRead, bool wantWrite, bool wantExcept);
```

A socket connection is usually full duplex. By calling `shutdown(1)`, you indicate that you won't write any more data on this socket. The values 0 (no more reads) and 2 (no more read/write) are used less frequently.

```
<Miscellaneous operations for GUSISocket 113>≡ (103) 114▷
virtual int shutdown(int how);
```

Some socket types do not write out data immediately. Calling `fsync()` guarantees that all data is written.

```
<Miscellaneous operations for GUSISocket 113>+≡ (103) ◀113
virtual int fsync();
```

5.2 Implementation of GUSISocket

5.2.1 General socket management

```
<Inline member functions for class GUSISocket 115>≡ (101) 117▷
inline void GUSISocket::AddReference()
{
    ++fRefCount;
}

inline void GUSISocket::RemoveReference()
{
    if (!--fRefCount)
        close();
}
```

```

(102) 118▶ Member functions for class GUSISocket 116 ≡
GUSISocket::GUSISocket()
: fRefCount(0)
{
    GUSIContext::Setup(false);
}

void GUSISocket::close()
{
    delete this;
}

GUSISocket::~~GUSISocket()
{
    Wakeup();
}

bool GUSISocket::Supports(ConfigOption)
{
    return false;
}

```

5.2.2 Context management

```

    {
        fContexts.Wakeup();
    }
}

```

```

{Member functions for class GUSISocket 116}+≡ (102) ◀116 120▶
void GUSISocket::AddContext(GUSIContext * context)
{
    fContexts.push_front(context ? context : GUSIContext::Current());
}

void GUSISocket::RemoveContext(GUSIContext * context)
{
    fContexts.remove(context ? context : GUSIContext::Current());
}

```

5.2.3 Operations without plausible default implementations

Many socket operations by default simply return EOPNOTSUPP.

$$\langle \text{No default implementation, return EOPNOTSUPP 119} \rangle \equiv \begin{cases} \text{return GUSISetPosixError(EOPNOTSUPP);} \\ \end{cases} \quad (120)$$

```

<Member functions for class GUSISocket 116>+≡ (102) <118 121>
int GUSISocket::bind(void *, socklen_t)
    <No default implementation, return EOPNOTSUPP 119>
int GUSISocket::getsockname(void *, socklen_t *)
    <No default implementation, return EOPNOTSUPP 119>
int GUSISocket::getpeername(void *, socklen_t *)
    <No default implementation, return EOPNOTSUPP 119>
int GUSISocket::listen(int)
    <No default implementation, return EOPNOTSUPP 119>
int GUSISocket::connect(void *, socklen_t)
    <No default implementation, return EOPNOTSUPP 119>
int GUSISocket::getsockopt(int, int, void *, socklen_t *)
    <No default implementation, return EOPNOTSUPP 119>
int GUSISocket::setsockopt(int, int, void *, socklen_t )
    <No default implementation, return EOPNOTSUPP 119>
int GUSISocket::fcntl(int, va_list)
    <No default implementation, return EOPNOTSUPP 119>
int GUSISocket::ioctl(unsigned int, va_list)
    <No default implementation, return EOPNOTSUPP 119>
int GUSISocket::ftruncate(off_t)
    <No default implementation, return EOPNOTSUPP 119>
int GUSISocket::shutdown(int)
    <No default implementation, return EOPNOTSUPP 119>

```

`accept()` behaves similarly to the above, but returns a `NULL` pointer, not `-1`.

```

<Member functions for class GUSISocket 116>+≡ (102) <120 122>
GUSISocket * GUSISocket::accept(void *, socklen_t *)
{
    return GUSISetPosixError(EOPNOTSUPP), static_cast<GUSISocket *>(nil);
}

```

`lseek()` should, according to the POSIX standard, return `ESPIPE` rather than `EOPNOTSUPP`.

```

<Member functions for class GUSISocket 116>+≡ (102) <121 123>
off_t GUSISocket::lseek(off_t, int)
{
    return GUSISetPosixError(ESPIPE), -1;
}

```

`fsync()` returns `EINVAL` as in BSD.

```

<Member functions for class GUSISocket 116>+≡ (102) <122 124>
int GUSISocket::fsync()
{
    return GUSISetPosixError(EINVAL);
}

```

5.2.4 Operations with plausible default implementations

By default, we assume that a socket is not a console.

```

<Member functions for class GUSISocket 116>+≡ (102) <123 125>
int GUSISocket::isatty()
{
    return 0;
}

```

`pre_select()` and `post_select()` often don't have to do anything. By default, we assume that a socket can sleep.

```

<Member functions for class GUSISocket 116>+≡ (102) <124 126>
    bool GUSISocket::pre_select(bool, bool, bool)
    {
        AddContext();

        return true;
    }

    void GUSISocket::post_select(bool, bool, bool)
    {
        RemoveContext();
    }

```

`select()` by default reports that nothing happens (although a socket like this is useless for inclusion in a `select()` statement).

```

<Member functions for class GUSISocket 116>+≡ (102) <125 129>
    bool GUSISocket::select(bool *, bool *, bool *)
    {
        return false;
    }

```

5.2.5 I/O Operations

As mentioned above, there are three variants of each call. If one of them is not implemented, the default implementation tries to emulate it with first another call.

To simplify filling in a `msghdr`, we provide an extension class with a convenient constructor.

```

<Auxiliary data structures for class GUSISocket 127>≡ (102) 128>
    struct GUSImsghdr : public msghdr {
        GUSImsghdr(const GUSIScattGath & buffer, const void * addr = nil, socklen_t addrlen = 0);
    };

```

The constructor of `GUSImsghdr` translates a simple buffer into an `iovec`.

```

<Auxiliary data structures for class GUSISocket 127>+≡ (102) <127
    GUSImsghdr::GUSImsghdr(const GUSIScattGath & buffer, const void * addr, socklen_t addrlen)
    {
        msg_name      = static_cast<char *>(const_cast<void *>(addr));
        msg_namelen   = addrlen;
        msg_iov       = const_cast<iovec *>(buffer.IOVec());
        msg_iovlen    = buffer.Count();
        msg_control    = nil;
        msg_controllen = 0;
    }

```

`read()` tries the more complex calls `recvfrom()` and `recvmsg()`. A socket family which supports none of the three calls must be very strange indeed. To keep implementors honest, an `assert()` verifies that they don't claim support of a call variant and then dispatch to the default implementation anyway.

⟨Member functions for class GUSISocket 116⟩+≡ (102) ◀126 130▶

```
ssize_t GUSISocket::read(const GUSIScatterer & buffer)
{
    GUSI_CASSERT_INTERNAL(!Supports(kSimpleCalls));

    if (Supports(kSocketCalls)) {
        socklen_t fromlen = 0;

        return recvfrom(buffer, 0, nil, &fromlen);
    } else if (Supports(kMsgCalls)) {
        GUSImsghdr msg(buffer);

        return recvmsg(&msg, 0);
    } else
        return GUSISetPosixError(EOPNOTSUPP);
}
```

`recvfrom()` can always be translated to `recvmsg()` and sometimes to `read()`.

⟨Member functions for class GUSISocket 116⟩+≡ (102) ◀129 131▶

```
ssize_t GUSISocket::recvfrom(const GUSIScatterer & buffer, int flags, void * from, socklen_t * fromlen)
{
    GUSI_CASSERT_INTERNAL(!Supports(kSocketCalls));

    if (!flags && !from && Supports(kSimpleCalls))
        return read(buffer);
    else if (Supports(kMsgCalls)) {
        GUSImsghdr msg(buffer, from, *fromlen);

        int result = recvmsg(&msg, flags);
        *fromlen = msg.msg_namelen;

        return result;
    } else
        return GUSISetPosixError(EOPNOTSUPP);
}
```

recvmsg() can always (in GUSI) be translated to recvfrom() and sometimes to read().

(Member functions for class GUSISocket 116) +≡ (102) <130 132>

```
ssize_t GUSISocket::recvmsg(msghdr * msg, int flags)
{
    GUSI_CASSERT_INTERNAL(!Supports(kMsgCalls));

    if (!flags && !msg->msg_name && Supports(kSimpleCalls)) {
        GUSIScatterer scatter(msg->msg_iov, msg->msg_iovlen);

        return scatter.SetLength(read(scatter));
    } else if (Supports(kSocketCalls)) {
        GUSIScatterer scatter(msg->msg_iov, msg->msg_iovlen);

        return scatter.SetLength(
            recvfrom(scatter, flags, msg->msg_name, &msg->msg_namelen));
    } else
        return GUSISetPosixError(EOPNOTSUPP);
}
```

(Member functions for class GUSISocket 116) +≡ (102) <131 133>

```
ssize_t GUSISocket::write(const GUSIGatherer & buffer)
{
    GUSI_CASSERT_INTERNAL(!Supports(kSimpleCalls));

    if (Supports(kSocketCalls))
        return sendto(buffer, 0, nil, 0);
    else if (Supports(kMsgCalls)) {
        GUSImsghdr msg(buffer);

        return sendmsg(&msg, 0);
    } else
        return GUSISetPosixError(EOPNOTSUPP);
}
```

(Member functions for class GUSISocket 116) +≡ (102) <132 134>

```
ssize_t GUSISocket::sendto(const GUSIGatherer & buffer, int flags, const void * to, socklen_t tolen)
{
    GUSI_CASSERT_INTERNAL(!Supports(kSocketCalls));

    if (!flags && !to && Supports(kSimpleCalls))
        return write(buffer);
    else if (Supports(kMsgCalls)) {
        GUSImsghdr msg(buffer, to, tolen);

        return sendmsg(&msg, flags);
    } else
        return GUSISetPosixError(EOPNOTSUPP);
}
```



```

<Member functions for class GUSISocket 116>+≡ (102) <133 136>
ssize_t GUSISocket::sendmsg(const msghdr * msg, int flags)
{
    GUSI_CASSERT_INTERNAL(!Supports(kMsgCalls));

    if (!flags && !msg->msg_name && Supports(kSimpleCalls)) {
        GUSIGatherer gather(msg->msg_iov, msg->msg_iovlen);

        return write(gather);
    } else if (Supports(kSocketCalls)) {
        GUSIGatherer gather(msg->msg_iov, msg->msg_iovlen);

        return sendto(gather, flags, msg->msg_name, msg->msg_namelen);
    } else
        return GUSISetPosixError(EOPNOTSUPP);
}

```

The traditional flavors of the I/O calls are translated to the scatterer/gatherer variants.

```

<Inline member functions for class GUSISocket 115>+≡ (101) <117
inline ssize_t GUSISocket::read(void * buffer, size_t length)
{
    return read(GUSIScatterer(buffer, length));
}

inline ssize_t GUSISocket::write(const void * buffer, size_t length)
{
    return write(GUSIGatherer(buffer, length));
}

inline ssize_t GUSISocket::recvfrom(void * buffer, size_t length, int flags, void * from, socklen_t * fromlen)
{
    return recvfrom(GUSIScatterer(buffer, length), flags, from, fromlen);
}

inline ssize_t GUSISocket::sendto(const void * buffer, size_t length, int flags, const void * to, socklen_t tolen)
{
    return sendto(GUSIGatherer(buffer, length), flags, to, tolen);
}

```

fstat() actually has a quite reasonable default.

(Member functions for class GUSISocket 116) +≡

(102) 4134

```
int GUSISocket::fstat(struct stat * buf)
{
    buf->st_dev      = 0;
    buf->st_ino       = 0;
    buf->st_mode      = S_IFSOCK | 0666 ;
    buf->st_nlink     = 1;
    buf->st_uid       = 0;
    buf->st_gid       = 0;
    buf->st_rdev      = 0;
    buf->st_size      = 1;
    buf->st_atime     = time(NULL);
    buf->st_mtime     = time(NULL);
    buf->st_ctime     = time(NULL);
    buf->st_blksize   = 1;
    buf->st_blocks    = 1;

    return 0;
}
```

Chapter 6

Buffering for GUSI

This section defines four classes that handle buffering for GUSI: GUSIScatterer, GUSIGatherer, and their common ancestor GUSIScattGath convert between iovecs and simple buffers in the absence of specialized communications routines. A GUSIRingBuffer mediates between a producer and a consumer, one of which is typically normal code and the other interrupt level code.

```
<GUSIBuffer.h 137>≡
#ifndef _GUSIBuffer_
#define _GUSIBuffer_

#ifdef GUSI_SOURCE

#include <sys/types.h>
#include <sys/uio.h>

#include <MacTypes.h>

#include "GUSIDdiag.h"

<Definition of class GUSIScattGath 139>
<Definition of class GUSIScatterer 142>
<Definition of class GUSIGatherer 143>

<Definition of class GUSIRingBuffer 144>

<Inline member functions for class GUSIScattGath 152>
<Inline member functions for class GUSIRingBuffer 193>
#endif /* GUSI_SOURCE */

#endif /* _GUSIBuffer_ */
```

```

<GUSIBuffer.cp 138>≡
#include "GUSIInternal.h"
#include "GUSIBuffer.h"

#include <algorithm>
#include <stdlib.h>
#include <Memory.h>

<Member functions for class GUSIScattGath 153>
<Member functions for class GUSIRingBuffer 169>
<Member functions for class GUSIRingBuffer::Peeker 199>

```

6.1 Definition of scattering/gathering

A GUSIScattGath translates between an array of iovecs and a simple buffer, allocating scratch space if necessary.

```

<Definition of class GUSIScattGath 139>≡ (137)
class GUSIScattGath {
protected:
    <Constructor and destructor for GUSIScattGath 140>
public:
    <Public interface to GUSIScattGath 141>
private:
    <Privatissima of GUSIScattGath 151>
};

```

On constructing a GUSIScattGath, we pass an array of iovecs. For the simpler functions, a variant with a single buffer and length is also available.

```

<Constructor and destructor for GUSIScattGath 140>≡ (139)
GUSIScattGath(const iovec *iov, int count, bool gather);
GUSIScattGath(void * buffer, size_t length, bool gather);
virtual ~GUSIScattGath();

```

The iovec, the buffer and its length are then available for public scrutiny. Copy constructor and assignment both are a bit nontrivial.

```

<Public interface to GUSIScattGath 141>≡ (139)
const iovec *   IOVec() const;
int             Count() const;
void *          Buffer() const;
operator        void *() const;
int             Length() const;
int             SetLength(int len) const;
void            operator=(const GUSIScattGath & other);
GUSIScattGath(const GUSIScattGath & other);

```

A GUSIScatterer distributes the contents of a buffer over an array of iovecs.

(*Definition of class GUSIScatterer 142*)≡ (137)

```
class GUSIScatterer : public GUSIScattGath {
public:
    GUSIScatterer(const iovec *iov, int count)
        : GUSIScattGath(iov, count, false) {}
    GUSIScatterer(void * buffer, size_t length)
        : GUSIScattGath(buffer, length, false) {}

    GUSIScatterer & operator=(const GUSIScatterer & other)
        { *static_cast<GUSIScattGath *>(this) = other; return *this; }
};
```

A GUSIGatherer collects the contents of an array of iovecs into a single buffer.

(*Definition of class GUSIGatherer 143*)≡ (137)

```
class GUSIGatherer : public GUSIScattGath {
public:
    GUSIGatherer(const struct iovec *iov, int count)
        : GUSIScattGath(iov, count, true) {}
    GUSIGatherer(const void * buffer, size_t length)
        : GUSIScattGath(const_cast<void *>(buffer), length, true) {}

    GUSIGatherer & operator=(const GUSIGatherer & other)
        { *static_cast<GUSIScattGath *>(this) = other; return *this; }
};
```

6.2 Definition of ring buffering

A GUSIRingBuffer typically has on one side a non-preemptive piece of code and on the other side a piece of interrupt code. To transfer data from and to the buffer, two interfaces are available: A direct interface that transfers memory, and an indirect interface that allocates memory regions and then has OS routines transfer data from or to them

(*Definition of class GUSIRingBuffer 144*)≡ (137)

```
class GUSIRingBuffer {
public:
    (Constructor and destructor for GUSIRingBuffer 145)
    (Direct interface for GUSIRingBuffer 146)
    (Indirect interface for GUSIRingBuffer 147)
    (Synchronization support for GUSIRingBuffer 148)
    (Buffer switching for GUSIRingBuffer 149)
    (Definition of class GUSIRingBuffer::Peeker 150)
private:
    (Privatissima of GUSIRingBuffer 165)
};
```

On construction of a GUSIRingBuffer, a buffer of the specified size is allocated and not released until destruction. `operator void*` may be used to determine whether construction was successful.

(*Constructor and destructor for GUSIRingBuffer 145*)≡ (144)

```
GUSIRingBuffer(size_t bufsiz);
~GUSIRingBuffer();
operator void*();
```

The direct interface to GUSIRingBuffer is straightforward: Produce() copies memory into the buffer, Consume() copies memory from the buffer, Free() determines how much space there is for Produce() and Valid() determines how much space there is for Consume().

```
(Direct interface for GUSIRingBuffer 146)≡ (144)
void    Produce(void * from, size_t & len);
void    Produce(const GUSIGatherer & gather, size_t & len, size_t & offset);
void    Produce(const GUSIGatherer & gather, size_t & len);
void    Consume(void * to, size_t & len);
void    Consume(const GUSIScatterer & scatter, size_t & len, size_t & offset);
void    Consume(const GUSIScatterer & scatter, size_t & len);
size_t  Free();
size_t  Valid();
```

ProduceBuffer() tries to find in the ring buffer a contiguous free block of memory of the specified size len or otherwise the biggest available free block, returns a pointer to it and sets len to its length. ValidBuffer() specifies that the next len bytes of the ring buffer now contain valid data.

ConsumeBuffer returns a pointer to the next valid byte and sets len to the minimum of the number of contiguous valid bytes and the value of len on entry. FreeBuffer() specifies that the next len bytes of the ring buffer were consumed and are no longer needed.

```
(Indirect interface for GUSIRingBuffer 147)≡ (144)
void *  ProduceBuffer(size_t & len);
void *  ConsumeBuffer(size_t & len);
void    ValidBuffer(void * buffer, size_t len);
void    FreeBuffer(void * buffer, size_t len);
```

Before the nonpreemptive partner changes any of the buffer's data structures, the GUSIRingBuffer member functions call Lock(), and after the change is complete, they call Release(). An interrupt level piece of code before changing any data structures has to determine whether the buffer is locked by calling Locked(). If the buffer is locked or otherwise in an unsuitable state, the code can specify a procedure to be called during the next Release() by calling Defer(). A deferred procedure should call ClearDefer() to avoid getting activated again at the next opportunity.

```
(Synchronization support for GUSIRingBuffer 148)≡ (144)
void    Lock();
void    Release();
bool    Locked();
typedef void (*Deferred)(void *);
void    Defer(Deferred def, void * ar);
void    ClearDefer();
```

It is possible to switch buffer sizes during the existence of a buffer, but we have to be somewhat careful, since some asynchronous call may still be writing into the old buffer. PurgeBuffers(), called at safe times, cleans up old buffers.

```
(Buffer switching for GUSIRingBuffer 149)≡ (144)
void    SwitchBuffer(size_t bufsiz);
size_t  Size();
void    PurgeBuffers();
```

Sometimes, it's necessary to do nondestructive reads, a task complex enough to warrant its own class.

(Definition of class GUSIRingBuffer::Peeker 150)≡ (144)

```

class Peeker {
public:
    Peeker(GUSIRingBuffer & buffer);
    ~Peeker();

    void Peek(void * to, size_t & len);
    void Peek(const GUSIScatterer & scatter, size_t & len);
private:
    (Privatissima of GUSIRingBuffer::Peeker 198)
};
friend class Peeker;

void Peek(void * to, size_t & len);
void Peek(const GUSIScatterer & scatter, size_t & len);

```

6.3 Implementation of scattering/gathering

A GUSIScattGath always consists of fIo, an array of iovecs, fCount, the number of sections in the array, and fLen, the total size of the data area. If fCount is 1, fBuf will be a copy of the pointer to the single section. If fCount is greater than 1, fScratch will contain a Handle to a scratch area of size len and fBuf will contain *scratch. If the object was constructed without providing an iovec array, fTrivialIo will be set up to hold one.

(Privatissima of GUSIScattGath 151)≡ (139)

```

const iovec * fIo;
iovec fTrivialIo;
mutable int fCount;
mutable Handle fScratch;
mutable void * fBuf;
mutable int fLen;
bool fGather;

```

Clients need readonly access to the buffer address and read/write access to the length. operator void* server to check whether the GUSIScattGath was constructed successfully.

(Inline member functions for class GUSIScattGath 152)≡ (137)

```

inline const iovec * GUSIScattGath::IOVec() const
{ return fIo; }
inline int GUSIScattGath::Count() const
{ return fCount; }
inline GUSIScattGath::operator void *() const
{ return Buffer(); }
inline int GUSIScattGath::Length() const
{ return fLen; }
inline int GUSIScattGath::SetLength(int len) const
{ return GUSI_MUTABLE(GUSIScattGath, fLen) = len; }

```

The constructor of GUSIScattGath does a case distinction between an empty IO vector, a vector with only one element, and a vector with several elements.

```

<Member functions for class GUSIScattGath 153>≡ (138) 157▶
GUSIScattGath::GUSIScattGath(const iovec *iov, int cnt, bool gather)
:   fIo(iov), fCount(cnt), fScratch(nil), fGather(gather)
{
    if (fCount < 1) {
        <GUSIScattGath constructed with empty IO vector 154>
    } else if (fCount == 1) {
        <GUSIScattGath constructed with contiguous IO vector 155>
    } else {
        <GUSIScattGath constructed with sparse IO vector 156>
    }
}

```

The first two cases are obvious.

```

<GUSIScattGath constructed with empty IO vector 154>≡ (153)
fBuf    =   nil;
fLen    =    0;

```

```

<GUSIScattGath constructed with contiguous IO vector 155>≡ (153)
fBuf    =   (void *) iov->iov_base;
fLen    =   (int)   iov->iov_len;

```

Only the third case is of any interest. We construct a scratch area. Note that cnt and iov are used; we don't want to alter the member fields. Since GUSIScattGath usually persist only for short times, fScratch is allocated as a locked handle rather than a pointer.

```

<GUSIScattGath constructed with sparse IO vector 156>≡ (153)
fBuf      = nil;
for (fLen = 0; cnt--; ++iov)
    fLen += (int) iov->iov_len;

```

The backwards compatible constructor of GUSIScattGath uses the fTrivialIo field to simulate the iovec array.

```

<Member functions for class GUSIScattGath 153>+≡ (138) <153 158▶
GUSIScattGath::GUSIScattGath(void * buffer, size_t length, bool gather)
:   fIo(&fTrivialIo), fCount(1), fScratch(nil), fGather(gather)
{
    fTrivialIo.iov_base = static_cast<caddr_t>(fBuf = buffer);
    fTrivialIo.iov_len  = fLen = length;
}

```


A call to Buffer is an indication that the caller prefers for the data to be contiguous, so we allocate a scratch buffer and gather, if necessary.

```

⟨Member functions for class GUSIScattGath 153⟩+≡ (138) ◀157 159▶
void * GUSIScattGath::Buffer() const
{
    if (!fScratch && fCount > 1) {
        if (GUSI_MUTABLE(GUSIScattGath, fScratch) = NewHandle(fLen)) {
            HLock(fScratch);
            GUSI_MUTABLE(GUSIScattGath, fBuf) = (void *) *fScratch;
            if (fGather) {
                ⟨Gather fIo contents in fBuf 163⟩
            }
        } else {
            GUSI_MUTABLE(GUSIScattGath, fCount) = 0;
            GUSI_MUTABLE(GUSIScattGath, fBuf) = nil;
        }
    }

    return fBuf;
}

```

The destructor of GUSIScattGath gets rid of the scratch area if one was allocated. If it works in scatter mode, the data is first scattered.

```

⟨Member functions for class GUSIScattGath 153⟩+≡ (138) ◀158 160▶
GUSIScattGath::~GUSIScattGath()
{
    if (fScratch) {
        if (!fGather) {
            ⟨Scatter fBuf contents to fIo 164⟩
        }
        DisposeHandle(fScratch);
    }
}

```

The copy constructor and the assignment operator both have to ensure that the two objects don't end up sharing the same scratch area.

```

⟨Member functions for class GUSIScattGath 153⟩+≡ (138) ◀159
void GUSIScattGath::operator=(const GUSIScattGath & other)
{
    if (fScratch)
        DisposeHandle(fScratch);
    ⟨Copy uncontroversial fields of GUSIScattGath 161⟩
    ⟨Copy fBuf unless it points to a scratch area 162⟩
}

GUSIScattGath::GUSIScattGath(const GUSIScattGath & other)
{
    ⟨Copy uncontroversial fields of GUSIScattGath 161⟩
    ⟨Copy fBuf unless it points to a scratch area 162⟩
}

```

⟨Copy uncontroversial fields of GUSIScattGath 161⟩≡ (160)

```
fIo      = other.fIo;
fCount   = other.fCount;
fLen     = other.fLen;
fGather  = other.fGather;
```

⟨Copy fBuf unless it points to a scratch area 162⟩≡ (160)

```
fScratch = nil;
fBuf      = other.fScratch ? nil : other.fBuf;
```

⟨Gather fIo contents in fBuf 163⟩≡ (158)

```
char *      buffer = static_cast<char *>(fBuf);
const iovec * io    = fIo;
for (int count = fCount; count--; ++io) {
    memcpy(buffer, io->iov_base, io->iov_len);

    buffer += io->iov_len;
}
```

⟨Scatter fBuf contents to fIo 164⟩≡ (159)

```
char *      buffer = static_cast<char *>(fBuf);
const iovec * io    = fIo;
int         length = fLen;
for (int count = fCount; count-- && length; ++io) {
    int sect = min(length, int(io->iov_len));

    memcpy(io->iov_base, buffer, sect);

    buffer += sect;
    length -= sect;
}
```

6.4 Implementation of ring buffering

The buffer area of a ring buffer extends between `fBuffer` and `fEnd`. `fValid` contains the number of valid bytes, while `fFree` and `fSpare` (Whose purpose will be explained later) sum up to the number of free bytes. `fProduce` points at the next free byte, while `fConsume` points at the next valid byte. `fInUse` indicates that an asynchronous call might be writing into the buffer.

⟨Privatissima of GUSIRingBuffer 165⟩≡ (144) 168»

```
Ptr      fBuffer;
Ptr      fEnd;
Ptr      fConsume;
Ptr      fProduce;
size_t   fFree;
size_t   fValid;
size_t   fSpare;
bool     fInUse;
```

```

<Initialize buffer of GUSIRingBuffer 166>≡ (173 176)
fBuffer      = fConsume = fProduce = bufsiz ? NewPtr(bufsiz) : 0;
fEnd         = fBuffer+bufsiz;

fValid       = fSpare    = 0;
fFree        = bufsiz;

```

```

<Initialize fields of GUSIRingBuffer 167>≡ (173) 172▷
fInUse       = false;

```

The relationships between the various pointers are captured by `Invariant()` which uses the auxiliary function `Distance()` to determine the distance between two pointers in the presence of wrap around areas.

```

<Privatissima of GUSIRingBuffer 165>+≡ (144) <165 171▷
bool Invariant();
size_t Distance(Ptr from, Ptr to);

```

```

<Member functions for class GUSIRingBuffer 169>≡ (138) 170▷
size_t GUSIRingBuffer::Distance(Ptr from, Ptr to)
{
    if (from > to)
        return (fEnd - from) + (to - fBuffer);
    else
        return to-from;
}

```

```

<Member functions for class GUSIRingBuffer 169>+≡ (138) <169 173▷
bool GUSIRingBuffer::Invariant()
{
    Lock();

    bool invariant =
        GUSI_CASSERT_INTERNAL(fProduce >= fBuffer && fProduce < fEnd)
        && GUSI_CASSERT_INTERNAL(fConsume >= fBuffer && fConsume < fEnd)
        && GUSI_CASSERT_INTERNAL(fFree + fValid + fSpare == fEnd - fBuffer)
        && GUSI_CASSERT_INTERNAL(Distance(fConsume, fProduce) == (fValid + fSpare) % (fEnd - fBuffer))
        && GUSI_CASSERT_INTERNAL(Distance(fProduce, fConsume) == fFree % (fEnd - fBuffer));

    Release();

    return invariant;
}

```

The lock mechanism relies on `fLocked`, and the deferred procedure and its argument are stored in `fDeferred` and `fDeferredArg`.

```

<Privatissima of GUSIRingBuffer 165>+≡ (144) <168 174▷
int fLocked;
Deferred fDeferred;
void * fDeferredArg;

```

```

<Initialize fields of GUSIRingBuffer 167>+≡ (173) <167 175▷
fLocked      = 0;
fDeferred    = nil;
fDeferredArg  = nil;

```

The constructor initializes everything.

```

<Member functions for class GUSIRingBuffer 169>+≡ (138) <170 176>
GUSIRingBuffer::GUSIRingBuffer(size_t bufsiz)
{
    <Initialize buffer of GUSIRingBuffer 166>
    <Initialize fields of GUSIRingBuffer 167>

    GUSI_SASSERT_INTERNAL(
        Invariant(), "Invariant violated in GUSIRingBuffer::GUSIRingBuffer()!\n");
}

```

We only switch the next time the buffer is empty, so we are prepared to create the new buffer dynamically and forward requests to it for a while.

```

<Privatissima of GUSIRingBuffer 165>+≡ (144) <171 190>
GUSIRingBuffer *    fNewBuffer;
GUSIRingBuffer *    fOldBuffer;
void                ObsoleteBuffer();

<Initialize fields of GUSIRingBuffer 167>+≡ (173) <172
fNewBuffer = nil;
fOldBuffer = nil;

```

```

<Member functions for class GUSIRingBuffer 169>+≡ (138) <173 177>
void GUSIRingBuffer::SwitchBuffer(size_t bufsiz)
{
    PurgeBuffers();
    Lock();
    if (fNewBuffer)
        fNewBuffer->SwitchBuffer(bufsiz);
    else if (bufsiz == fEnd-fBuffer) // No change
        return;
    else if (!fInUse && !fValid) {
        if (fBuffer)
            DisposePtr(fBuffer);
        <Initialize buffer of GUSIRingBuffer 166>
    } else
        fNewBuffer = new GUSIRingBuffer(bufsiz);
    Release();
}

void GUSIRingBuffer::PurgeBuffers()
{
    if (fOldBuffer)
        delete fOldBuffer;
}

```

ObsoleteBuffer() swaps the data structures of fNewBuffer and the current buffer and finally prepends fNewBuffer to fOldBuffer.

```

<Member functions for class GUSIRingBuffer 169>+≡ (138) <176 178>
void GUSIRingBuffer::ObsoleteBuffer()
{
    Ptr oldBuffer = fBuffer;
    fBuffer      = fNewBuffer->fBuffer;
    fEnd         = fNewBuffer->fEnd;
    fConsume     = fNewBuffer->fConsume;
    fProduce     = fNewBuffer->fProduce;
    fFree        = fNewBuffer->fFree;
    fValid       = fNewBuffer->fValid;
    fSpare       = fNewBuffer->fSpare;
    fInUse       = fNewBuffer->fInUse;
    fNewBuffer->fOldBuffer = fOldBuffer;
    fOldBuffer   = fNewBuffer;
    fNewBuffer   = fOldBuffer->fNewBuffer;
    fOldBuffer->fBuffer = oldBuffer;
    fOldBuffer->fNewBuffer = nil;
}

```

The destructor cleans up deferred procedures and allocated memory.

```

<Member functions for class GUSIRingBuffer 169>+≡ (138) <177 179>
GUSIRingBuffer::~GUSIRingBuffer()
{
    Lock();
    Release();
    if (fBuffer)
        DisposePtr(fBuffer);
    PurgeBuffers();
    if (fNewBuffer)
        delete fNewBuffer;
}

```

Since the direct interface is built on the indirect interface, we deal with the latter first. ProduceBuffer() is the most complicated member function, as it has the most freedom in how to do its job.

```

<Member functions for class GUSIRingBuffer 169>+≡ (138) <178 184>
void * GUSIRingBuffer::ProduceBuffer(size_t & len)
{
    Lock();
    <Forward ProduceBuffer to fNewBuffer 180>
    size_t requested_length = len;

    <Reset pointers if GUSIRingBuffer is empty 182>
    <Calculate the size of a free block in GUSIRingBuffer and adjust len 181>
    GUSI_SASSERT_INTERNAL(
        Invariant(), "Invariant violated in GUSIRingBuffer::ProduceBuffer()!\n");
    GUSI_CASSERT_INTERNAL(len <= requested_length);
    void * result = fProduce;
    fInUse       = true;
    Release();
    return result;
}

```

If a new buffer exists, ProduceBuffer requests get directed there.

```

<Forward ProduceBuffer to fNewBuffer 180>≡ (179)
while (fNewBuffer) {
    if (!fValid)
        ObsoleteBuffer();
    else {
        void * buf = fNewBuffer->ProduceBuffer(len);
        Release();
        return buf;
    }
}

```

At the core of ProduceBuffer() is the code to calculate the best possible free block in streak and adjust len.

```

<Calculate the size of a free block in GUSIRingBuffer and adjust len 181>≡ (179)
size_t streak = fEnd - fProduce;
if (streak >= fFree)
    streak = fFree;
else
    <Avoid silly windows in GUSIRingBuffer 183>
    if (len > streak)
        len = streak;

```

To keep free space as contiguous as possible, we reset the production and consumption pointers whenever the buffer becomes empty. Since this code is only called from ProduceBuffer(), we know that there is no outstanding data producing call. This is also a good opportunity to switch buffers.

```

<Reset pointers if GUSIRingBuffer is empty 182>≡ (179)
if (!fValid) {
    fProduce = fConsume = fBuffer;
    fSpare = 0;
    fFree = fEnd - fBuffer;
}

```

Now we turn to the secret of fSpare: If a large produce buffer is asked for, the free area wraps around, and most of the free area is at the beginning of the buffer, fProduce skips the rest of the buffer and the number of skipped bytes is noted in fSpare so fConsume will skip them later, too.

```

<Avoid silly windows in GUSIRingBuffer 183>≡ (181)
if (streak < (fFree >> 1) && streak < len) {
    fSpare = streak;
    fProduce = fBuffer;
    fFree -= fSpare;
    streak = fFree;
}

```

Compared to ProduceBuffer(), ConsumeBuffer() is quite simple as we don't have the option of skipping anything.

```

<Member functions for class GUSIRingBuffer 169>+≡ (138) <179 186>
void * GUSIRingBuffer::ConsumeBuffer(size_t & len)
{
    Lock();
    size_t requested_length = len;
    <ObsoleteBuffer if possible 185>
    size_t streak = fEnd - fConsume - fSpare;
    if (streak > fValid)
        streak = fValid;
    if (len > streak)
        len = streak;
    GUSI_SASSERT_INTERNAL(
        Invariant(), "Invariant violated in GUSIRingBuffer::ConsumeBuffer()!\n");
    GUSI_CASSERT_INTERNAL(len <= requested_length);
    void * result = fConsume;
    Release();
    return result;
}

```

At this point, empty buffers can be thrown away unless they are in use

```

<ObsoleteBuffer if possible 185>≡ (184 186 187)
while (fNewBuffer && !fValid && !fInUse)
    ObsoleteBuffer();

```

ValidBuffer() concludes an action started by ProduceBuffer by advancing fProduce.

```

<Member functions for class GUSIRingBuffer 169>+≡ (138) <184 187>
void GUSIRingBuffer::ValidBuffer(void * buffer, size_t len)
{
    Lock();
    if (fNewBuffer && (buffer < fBuffer || buffer >= fEnd)) {
        fNewBuffer->ValidBuffer(buffer, len);
        Release();
        return;
    }
    GUSI_CASSERT_INTERNAL(len <= fFree);
    GUSI_CASSERT_INTERNAL(fProduce + len <= fEnd);
    fInUse = false;
    fValid += len;
    fFree -= len;
    fProduce += len;
    if (fProduce == fEnd)
        fProduce = fBuffer;
    <ObsoleteBuffer if possible 185>
    GUSI_SASSERT_INTERNAL(
        Invariant(), "Invariant violated in GUSIRingBuffer::ValidBuffer()!\n");
    Release();
}

```

FreeBuffer() concludes an action started by ConsumeBuffer by advancing fConsume.

```
(Member functions for class GUSIRingBuffer 169)+≡ (138) <186 188>
void GUSIRingBuffer::FreeBuffer(void *, size_t len)
{
    Lock();
    fFree      += len;
    fValid     -= len;
    fConsume   += len;

    if (fConsume == fEnd-fSpare) {
        fConsume = fBuffer;
        fFree    += fSpare;
        fSpare    = 0;
    }
    (ObsoleteBuffer if possible 185)
    GUSI_SASSERT_INTERNAL(
        Invariant(), "Invariant violated in GUSIRingBuffer::FreeBuffer()!\n");
    Release();
}
```

Now for the direct interface. Produce() combines ProduceBuffer with ValidBuffer.

```
(Member functions for class GUSIRingBuffer 169)+≡ (138) <187 189>
void GUSIRingBuffer::Produce(void * from, size_t & len)
{
    size_t part;
    size_t rest;
    void * buf;

    PurgeBuffers();
    for (rest = len; (part = rest) && fFree; rest -= part) {
        buf = ProduceBuffer(part);
        BlockMoveData(from, buf, part);
        ValidBuffer(buf, part);

        from = static_cast<char *>(from)+part;
    }
    len -= rest;
    GUSI_SASSERT_INTERNAL(
        Invariant(), "Invariant violated in GUSIRingBuffer::Produce()!\n");
}
```


<Member functions for class GUSIRingBuffer 169>+≡

(138) <189 194>

```
void GUSIRingBuffer::IterateIOVec(const GUSIScattGath & sg, size_t & len, size_t & offset, bool pro
{
    const iovec *   vec = sg.IOVec();
    iovec           io  = vec[0];

    if (!len)      // Surprising as it is, nobody notices this any sooner
        return;

    Lock();
    <Skip offset bytes from beginning of vec 192>
    size_t part;
    size_t rest = len;
    while (part = min(rest, io.iov_len)) {
        size_t donepart = part;
        if (produce)
            Produce(io.iov_base, donepart);
        else
            Consume(io.iov_base, donepart);
        rest -= donepart;
        if (donepart != part)
            break;
        do {
            io = *++vec;
        } while (!io.iov_len);
    }
    len -= rest;
    offset += len;
    Release();
}
```

<Skip offset bytes from beginning of vec 192>≡

(191)

```
size_t      off = offset;

while (off >= io.iov_len) {
    off -= io.iov_len;
    do {
        io = *++vec;
    } while (!io.iov_len);
}
io.iov_base = static_cast<char *>(io.iov_base)+off;
io.iov_len  = io.iov_len-off;
```

```

<Inline member functions for class GUSIRingBuffer 193>≡ (137) 196▷
inline void GUSIRingBuffer::Produce(const GUSIGatherer & gather, size_t & len, size_t & offset)
{
    IterateIOVec(gather, len, offset, true);
}

inline void GUSIRingBuffer::Consume(const GUSIScatterer & scatter, size_t & len, size_t & offset)
{
    IterateIOVec(scatter, len, offset, false);
}

inline void GUSIRingBuffer::Produce(const GUSIGatherer & gather, size_t & len)
{
    size_t offset = 0;

    IterateIOVec(gather, len, offset, true);
}

inline void GUSIRingBuffer::Consume(const GUSIScatterer & scatter, size_t & len)
{
    size_t offset = 0;

    IterateIOVec(scatter, len, offset, false);
}

```

Free returns the free buffer space of the most recent buffer.

```

<Member functions for class GUSIRingBuffer 169>+≡ (138) ◀191 195▷
size_t GUSIRingBuffer::Free()
{
    if (fNewBuffer)
        return fNewBuffer->Free();
    else
        return fFree;
}

```

Free returns the sum of the valid bytes of all buffers.

```

<Member functions for class GUSIRingBuffer 169>+≡ (138) ◀194
size_t GUSIRingBuffer::Valid()
{
    if (fNewBuffer)
        return fNewBuffer->Valid()+fValid;
    else
        return fValid;
}

```

The lock support is rather straightforward.

```

<Inline member functions for class GUSIRingBuffer 193>+≡ (137) <193 197>
inline void      GUSIRingBuffer::Lock()      { ++fLocked;      }
inline bool      GUSIRingBuffer::Locked()    { return (fLocked!=0); }
inline void      GUSIRingBuffer::ClearDefer() { fDeferred = nil;    }
inline void      GUSIRingBuffer::Release()
{
    GUSI_CASSERT_INTERNAL(fLocked > 0);
    if (--fLocked <= 0 && fDeferred)
        fDeferred(fDeferredArg);
}
inline void      GUSIRingBuffer::Defer(Deferred def, void * ar)
{
    fDeferred      = def;
    fDeferredArg    = ar;
}

```

The size is stored only implicitly.

```

<Inline member functions for class GUSIRingBuffer 193>+≡ (137) <196 205>
inline size_t GUSIRingBuffer::Size() { return fEnd - fBuffer; }

```

A GUSIRingBuffer::Peeker has to keep its associated GUSIRingBuffer locked during its entire existence.

```

<Privatissima of GUSIRingBuffer::Peeker 198>≡ (150) 200>
GUSIRingBuffer & fTopBuffer;
GUSIRingBuffer * fCurBuffer;
Ptr fPeek;

<Member functions for class GUSIRingBuffer::Peeker 199>≡ (138) 201>
GUSIRingBuffer::Peeker::Peeker(GUSIRingBuffer & buffer)
: fTopBuffer(buffer)
{
    fTopBuffer.Lock();
    for (fCurBuffer = &fTopBuffer; fCurBuffer && !fCurBuffer->fValid; )
        fCurBuffer = fCurBuffer->fNewBuffer;
    if (fCurBuffer)
        fPeek = fCurBuffer->fConsume;
}
GUSIRingBuffer::Peeker::~Peeker()
{
    fTopBuffer.Release();
}

```

The core routine for reading is PeekBuffer which automatically advances the peeker as well.

```

<Privatissima of GUSIRingBuffer::Peeker 198>+≡ (150) <198
void * PeekBuffer(size_t & len);

```

⟨Member functions for class GUSIRingBuffer::Peeker 199⟩+≡ (138) ◀199 203▶

```
void * GUSIRingBuffer::Peeker::PeekBuffer(size_t & len)
{
    size_t streak;

    if (!fCurBuffer)
        return nil;

    if (fPeek < fCurBuffer->fConsume)
        streak = fCurBuffer->fConsume - fPeek;
    else
        streak = fCurBuffer->fEnd - fPeek - fCurBuffer->fSpare;
    if (streak > fCurBuffer->fValid)
        streak = fCurBuffer->fValid;
    if (len > streak)
        len = streak;
    void * result = fPeek;
    ⟨Advance fPeek by len 202⟩

    return result;
}
```

⟨Advance fPeek by len 202⟩≡ (201)

```
fPeek += len;
if (fPeek == fCurBuffer->fEnd-fCurBuffer->fSpare)
    fPeek = fCurBuffer->fBuffer;
if (fPeek == fCurBuffer->fConsume) {
    while ((fCurBuffer = fCurBuffer->fNewBuffer) && !fCurBuffer->fValid)
        ;
    if (fCurBuffer)
        fPeek = fCurBuffer->fConsume;
}
```

The implementation of Peek itself is then fairly simple.

⟨Member functions for class GUSIRingBuffer::Peeker 199⟩+≡ (138) ◀201 204▶

```
void GUSIRingBuffer::Peeker::Peek(void * to, size_t & len)
{
    size_t part;
    size_t rest;
    void * buf;

    for (rest = len; (part = rest) && (buf = PeekBuffer(part)); rest -= part) {
        BlockMoveData(buf, to, part);
        to = static_cast<char *>(to)+part;
    }

    len -= rest;
}
```

(Member functions for class GUSIRingBuffer::Peeker 199)+≡ (138) ¶203

```
void GUSIRingBuffer::Peeker::Peek(const GUSIScatterer & scatter, size_t & len)
{
    const iovec *   vec = scatter.IOVec();
    iovec           io  = vec[0];

    while (!io.iov_len)
        io = *++vec;

    size_t part;
    size_t rest = len;
    while (part = min(rest, io.iov_len)) {
        size_t donepart = part;
        Peek(io.iov_base, donepart);
        rest -= donepart;
        if (donepart != part)
            break;
        do {
            io = *++vec;
        } while (!io.iov_len);
    }
    len -= rest;
}
```

(Inline member functions for class GUSIRingBuffer 193)+≡ (137) ¶197

```
inline void GUSIRingBuffer::Peek(void * to, size_t & len)
{
    Peeker peeker(*this);

    peeker.Peek(to, len);
}

inline void GUSIRingBuffer::Peek(const GUSIScatterer & scatter, size_t & len)
{
    Peeker peeker(*this);

    peeker.Peek(scatter, len);
}
```

Chapter 7

Socket Factories

Instead of creating sockets of some specific subtype of `GUSISocket`, directly, we choose the more flexible approach of creating them in some instance of a subtype of the abstract factory class `GUSISocketFactory`. For even more flexibility and a direct mapping to BSD socket domains, `GUSISocketFactory` instances are collected in a `GUSISocketDomainRegistry`. If several types and or protocols in a domain are implemented, they are collected in a `GUSISocketTypeRegistry`.

```
<GUSIFactory.h 206>≡
    #ifndef _GUSIFactory_
    #define _GUSIFactory_

    #ifdef GUSI_SOURCE

    #include "GUSISocket.h"

    <Definition of class GUSISocketFactory 208>
    <Definition of class GUSISocketDomainRegistry 209>
    <Definition of class GUSISocketTypeRegistry 212>

    <Definition of GUSISetupFactories hook 216>

    <Inline member functions for class GUSISocketDomainRegistry 219>
    <Inline member functions for class GUSISocketTypeRegistry 227>

    #endif /* GUSI_SOURCE */

    #endif /* _GUSIFactory_ */

<GUSIFactory.cp 207>≡
    #include "GUSIInternal.h"
    #include "GUSIFactory.h"
    #include "GUSIDiag.h"
    #include "GUSIInet.h"

    <Member functions for class GUSISocketFactory 215>
    <Member functions for class GUSISocketDomainRegistry 218>
    <Member functions for class GUSISocketTypeRegistry 229>
```

7.1 Definition of GUSISocketFactory

GUSISocketFactory consists of a few maintenance functions and the socket operations.

```
(Definition of class GUSISocketFactory 208)≡ (206)
class GUSISocketFactory {
public:
    virtual int socketpair(int domain, int type, int protocol, GUSISocket * s[2]);
    virtual GUSISocket * socket(int domain, int type, int protocol) = 0;
protected:
    GUSISocketFactory() {}
    virtual ~GUSISocketFactory() {}
};
```

7.2 Definition of GUSISocketDomainRegistry

The GUSISocketDomainRegistry is a singleton class registering all socket domains.

```
(Definition of class GUSISocketDomainRegistry 209)≡ (206)
class GUSISocketDomainRegistry : public GUSISocketFactory {
public:
    {Socket creation interface of GUSISocketDomainRegistry 210}
    {Registration interface of GUSISocketDomainRegistry 211}
private:
    {Privatissima of GUSISocketDomainRegistry 217}
};
```

The only instance of GUSISocketDomainRegistry is, as usual, obtained by calling Instance(). Calling socket() on this instance will then create a socket.

```
{Socket creation interface of GUSISocketDomainRegistry 210}≡ (209)
virtual GUSISocket * socket(int domain, int type, int protocol);
virtual int socketpair(int domain, int type, int protocol, GUSISocket * s[2]);
static GUSISocketDomainRegistry * Instance();
```

AddFactory() and RemoveFactory() add and remove a GUSISocketFactory for a given domain number. Both return the previous registrant.

```
{Registration interface of GUSISocketDomainRegistry 211}≡ (209)
GUSISocketFactory * AddFactory(int domain, GUSISocketFactory * factory);
GUSISocketFactory * RemoveFactory(int domain);
```

7.3 Definition of GUSISocketTypeRegistry

A GUSISocketTypeRegistry registers factories for some domain by type and protocol.

```
(Definition of class GUSISocketTypeRegistry 212)≡ (206)
class GUSISocketTypeRegistry : public GUSISocketFactory {
public:
    {Socket creation interface of GUSISocketTypeRegistry 213}
    {Registration interface of GUSISocketTypeRegistry 214}
private:
    {Privatissima of GUSISocketTypeRegistry 226}
};
```


GUSISocketTypeRegistry is not a singleton, but each instance is somewhat singletonish in that it does some delayed initialization only when it's used and at that point registers itself with the GUSISocketDomainRegistry. Calling `socket()` on these instances will then create a socket.

```
<Socket creation interface of GUSISocketTypeRegistry 213>≡ (212)
GUSISocketTypeRegistry(int domain, int maxfactory);
virtual GUSISocket * socket(int domain, int type, int protocol);
virtual int socketpair(int domain, int type, int protocol, GUSISocket * s[2]);
```

`AddFactory()` and `RemoveFactory()` add and remove a GUSISocketFactory for a given type and protocol (both of which can be specified as 0 to match any value). Both return the previous registrant.

```
<Registration interface of GUSISocketTypeRegistry 214>≡ (212)
GUSISocketFactory * AddFactory(int type, int protocol, GUSISocketFactory * factory);
GUSISocketFactory * RemoveFactory(int type, int protocol);
```

7.4 Implementation of GUSISocketFactory

We define this so compilers know where to place the vtable. Furthermore, very few domains provide a definition for `socketpair`, while defining `socket` is pretty much mandatory.

```
<Member functions for class GUSISocketFactory 215>≡ (207)
int GUSISocketFactory::socketpair(int, int, int, GUSISocket * [2])
{
    return GUSISetPosixError(EOPNOTSUPP);
}
```

7.5 Implementation of GUSISocketDomainRegistry

By now, you should know how singletons are created. Could it be that the combination of Design Patterns and Literate Programming leads to a proliferation of clichés?

```
<Definition of GUSISetupFactories hook 216>≡ (206)
extern "C" void GUSISetupFactories();
```

```
<Privatissima of GUSISocketDomainRegistry 217>≡ (209) 220▷
static GUSISocketDomainRegistry * sInstance;
```

```
<Member functions for class GUSISocketDomainRegistry 218>≡ (207) 221▷
GUSISocketDomainRegistry * GUSISocketDomainRegistry::sInstance;
```

```
<Inline member functions for class GUSISocketDomainRegistry 219>≡ (206) 225▷
inline GUSISocketDomainRegistry * GUSISocketDomainRegistry::Instance()
{
    if (!sInstance) {
        sInstance = new GUSISocketDomainRegistry();
        GUSISetupFactories();
    }

    return sInstance;
}
```

We store domain factories in a table that is quite comfortably sized.

```
(Privatissima of GUSISocketDomainRegistry 217)+≡ (209) <217
GUSISocketFactory * factory[AF_MAX];
GUSISocketDomainRegistry();
```

```
(Member functions for class GUSISocketDomainRegistry 218)+≡ (207) <218 222>
GUSISocketDomainRegistry::GUSISocketDomainRegistry()
{
    for (int i = 0; i<AF_MAX; ++i)
        factory[i] = nil;
}
```

The socket() call is swiftly delegated to the registered domain.

```
(Member functions for class GUSISocketDomainRegistry 218)+≡ (207) <221 223>
GUSISocket * GUSISocketDomainRegistry::socket(int domain, int type, int protocol)
{
    if (!GUSI_CASSERT_CLIENT(domain >= 0 && domain < AF_MAX) || !factory[domain])
        return GUSISetPosixError(EAFNOSUPPORT), static_cast<GUSISocket *>(nil);
    return factory[domain]->socket(domain, type, protocol);
}
```

So is socketpair.

```
(Member functions for class GUSISocketDomainRegistry 218)+≡ (207) <222 224>
int GUSISocketDomainRegistry::socketpair(int domain, int type, int protocol, GUSISocket * s[2])
{
    if (!GUSI_CASSERT_CLIENT(domain >= 0 && domain < AF_MAX) || !factory[domain])
        return GUSISetPosixError(EAFNOSUPPORT);
    return factory[domain]->socketpair(domain, type, protocol, s);
}
```

AddFactory() and RemoveFactory() add and remove a GUSISocketFactory for a given domain number. Both return the previous registrant. Out of range errors are considered internal errors rather than merely client errors.

```
(Member functions for class GUSISocketDomainRegistry 218)+≡ (207) <223
GUSISocketFactory * GUSISocketDomainRegistry::AddFactory(int domain, GUSISocketFactory * f)
{
    if (!GUSI_CASSERT_INTERNAL(domain >= 0 && domain < AF_MAX))
        return nil;
    GUSISocketFactory * old = factory[domain];
    factory[domain] = f;

    return old;
}
```

RemoveFactory() can actually be implemented in terms of AddFactory() but that might confuse readers.

```
(Inline member functions for class GUSISocketDomainRegistry 219)+≡ (206) <219
inline GUSISocketFactory * GUSISocketDomainRegistry::RemoveFactory(int domain)
{
    return AddFactory(domain, nil);
}
```

7.6 Implementation of GUSISocketTypeRegistry

We store type factories in a fixed size table. This table is only initialized when any non-constructor public member is called.

(Privatissima of GUSISocketTypeRegistry 226)≡ (212) 228▷

```
struct Entry {
    int          type;
    int          protocol;
    GUSISocketFactory * factory;
    Entry() : type(0), protocol(0), factory(nil) {}
};
Entry *   factory;
int       domain;
int       maxfactory;
```

(Inline member functions for class GUSISocketTypeRegistry 227)≡ (206)

```
inline GUSISocketTypeRegistry::GUSISocketTypeRegistry(int domain, int maxfactory)
: domain(domain), maxfactory(maxfactory), factory(nil)
{
}
```

Initialize() initializes the table and registers the object with the GUSISocketDomainRegistry() the first time it's called.

(Privatissima of GUSISocketTypeRegistry 226)+≡ (212) ◀226 230▷

```
void      Initialize();
```

(Member functions for class GUSISocketTypeRegistry 229)≡ (207) 231▷

```
void GUSISocketTypeRegistry::Initialize()
{
    if (!factory) {
        factory = new Entry[maxfactory];
        GUSISocketDomainRegistry::Instance()->AddFactory(domain, this);
    }
}
```

Unlike for a GUSISocketDomainRegistry, match identification for a GUSISocketTypeRegistry takes a linear search. Find() stops when it has found either a match or an empty slot.

(Privatissima of GUSISocketTypeRegistry 226)+≡ (212) ◀228

```
bool Find(int type, int protocol, bool exact, Entry *&found);
```

```

<Member functions for class GUSISocketTypeRegistry 229>+≡ (207) <229 234>
bool GUSISocketTypeRegistry::Find(int type, int protocol, bool exact, Entry *&found)
{
    Initialize();
    for (Entry * ent = factory; ent<factory+maxfactory; ++ent)
        if (!ent->factory) {
            found = ent;
            return false;
        } else if (
            <Socket type matches ent->type 232>
            && <Socket protocol matches ent->protocol 233>
        ) {
            found = ent;
            return true;
        }
    found = nil;
    return false;
}

```

protocol may be specified as 0 in the call, but not type.

```

<Socket type matches ent->type 232>≡ (231)
(ent->type == type || (!exact && !ent->type))

```

```

<Socket protocol matches ent->protocol 233>≡ (231)
(ent->protocol == protocol || (!exact && (!ent->protocol || !protocol)))

```

The socket and socketpair calls are swiftly delegated to some registered domain.

```

<Member functions for class GUSISocketTypeRegistry 229>+≡ (207) <231 235>
GUSISocket * GUSISocketTypeRegistry::socket(int domain, int type, int protocol)
{
    Entry * ent;
    bool found = Find(type, protocol, false, ent);
    if (!GUSI_CASSERT_CLIENT(found))
        return GUSISetPosixError(EPROTONOSUPPORT), static_cast<GUSISocket *>(nil);
    return ent->factory->socket(domain, type, protocol);
}

int GUSISocketTypeRegistry::socketpair(int domain, int type, int protocol, GUSISocket * s[2])
{
    Entry * ent;
    bool found = Find(type, protocol, false, ent);
    if (!GUSI_CASSERT_CLIENT(found))
        return GUSISetPosixError(EPROTONOSUPPORT);
    return ent->factory->socketpair(domain, type, protocol, s);
}

```

AddFactory() and RemoveFactory() add and remove a GUSISocketFactory. Both return the previous registrant. Table overflow errors are considered internal errors rather than merely client errors.

```

⟨Member functions for class GUSISocketTypeRegistry 229⟩+≡ (207) 4234 236
GUSISocketFactory * GUSISocketTypeRegistry::AddFactory(int type, int protocol, GUSISocketFactory * f)
{
    Entry * ent;
    bool previous = Find(type, protocol, true, ent);
    if (!GUSI_CASSERT_INTERNAL(ent))
        return nil;
    GUSISocketFactory * old = previous ? ent->factory : nil;
    ent->type = type;
    ent->protocol = protocol;
    ent->factory = f;

    return old;
}

```

RemoveFactory() has to take care of keeping all valid entries together.

```

⟨Member functions for class GUSISocketTypeRegistry 229⟩+≡ (207) 4235
GUSISocketFactory * GUSISocketTypeRegistry::RemoveFactory(int type, int protocol)
{
    Entry * ent;
    if (!Find(type, protocol, true, ent))
        return nil;
    GUSISocketFactory * old = ent->factory;
    while (++ent - factory < maxfactory && ent->factory)
        ent[-1] = ent[0];
    ent[-1].factory = nil;

    return old;
}

```


Chapter 8

Devices

Similar to the creation of sockets, operations on files like opening or renaming them need to be dispatched to a variety of special cases (Most of them of the form "Dev:" preceded by a device name). Analogous to the GUSISocketFactory subclasses registered in a GUSISocketDomainRegistry, we therefore have subclasses of GUSIDevice registered in a GUSIDeviceRegistry, although the details of the two registries are quite different.

During resolution of a file name, the name and information about it is passed around in a GUSIFileToken.

```
<GUSIDevice.h 237>≡
    #ifndef _GUSIDevice_
    #define _GUSIDevice_

    #ifdef GUSI_SOURCE

    #include "GUSISocket.h"
    #include "GUSIFileSpec.h"

    #include <dirent.h>
    #include <utime.h>

    <Definition of class GUSIFileToken 239>
    <Definition of class GUSIDirectory 288>
    <Definition of class GUSIDeviceRegistry 241>
    <Definition of class GUSIDevice 240>

    <Definition of GUSISetupDevices hook 296>

    <Inline member functions for class GUSIDeviceRegistry 299>

    #endif /* GUSI_SOURCE */

    #endif /* _GUSIDevice_ */
```

```
<GUSIDevice.cp 238>≡
#include "GUSIInternal.h"
#include "GUSIDevice.h"
#include "GUSIMacFile.h"
#include "GUSINull.h"
#include "GUSIDiag.h"

#include <fcntl.h>
#include <utility>

GUSI_USING_STD_NAMESPACE

<Member functions for class GUSIFileToken 289>
<Member functions for class GUSIDevice 292>
<Member functions for class GUSIDeviceRegistry 298>
```


8.1 Definition of GUSIFileToken

A GUSIFileToken consists of a pointer to the name as a C string, of a pointer to the GUSIDevice the token resolves to, and, if the token refers to a file name rather than a device name, a pointer to a GUSIFileSpec. Since depending on the call, different GUSIDevice subclasses may handle it, a request code has to be passed to the constructor, too.

(*Definition of class GUSIFileToken 239*)≡ (237)

```
class GUSIDevice;

class GUSIFileToken : public GUSIFileSpec {
public:
    enum Request {
        (Requests for GUSIFileToken 246)
        kNoRequest
    };

    GUSIFileToken(const char * path, Request request, bool useAlias = false);
    GUSIFileToken(const GUSIFileSpec & spec, Request request);
    GUSIFileToken(short fRefNum, Request request);

    bool          IsFile()          const { return fIsFile;          }
    bool          IsDevice()         const { return !fIsFile;        }
    Request       WhichRequest()     const { return fRequest;        }
    GUSIDevice *  Device()           const { return fDevice;         }
    const char *  Path()             const { return fPath;           }

    static bool   StrFragEqual(const char * name, const char * frag);
    enum StdStream {
        kStdin,
        kStdout,
        kStderr,
        kConsole,
        kNoStdStream = -2
    };
    static StdStream StrStdStream(const char * name);

private:
    GUSIDevice *    fDevice;
    const char *    fPath;
    bool            fIsFile;
    Request         fRequest;
};
```

8.2 Definition of GUSIDevice

GUSIDevice consists of a few maintenance functions and the device operations. The request dispatcher first calls Want() for each candidate device and as soon as it's successful, calls the specific operation. Devices are kept in a linked list by the GUSIDeviceRegistry.

```
<Definition of class GUSIDevice 240>≡ (237)
class GUSIDevice {
public:
    virtual bool    Want(GUSIFileToken & file);

    <Operations for GUSIDevice 247>
protected:
    friend class GUSIDeviceRegistry;
    friend class GUSIDeviceRegistry::iterator;

    GUSIDevice() : fNextDevice(nil)    {}
    virtual ~GUSIDevice()              {}

    GUSIDevice *    fNextDevice;
};
```

8.3 Definition of GUSIDeviceRegistry

The GUSIDeviceRegistry is a singleton class registering all socket domains.

```
<Definition of class GUSIDeviceRegistry 241>≡ (237)
class GUSIDeviceRegistry {
public:
    <Creation of GUSIDeviceRegistry 242>
    <Operations for GUSIDeviceRegistry 248>
    <Registration interface of GUSIDeviceRegistry 243>
    <Iterator on GUSIDeviceRegistry 245>
protected:
    <Looking up a device in the GUSIDeviceRegistry 244>
private:
    <Privatissima of GUSIDeviceRegistry 297>
};
```

The only instance of GUSIDeviceRegistry is, as usual, obtained by calling Instance().

```
<Creation of GUSIDeviceRegistry 242>≡ (241)
static GUSIDeviceRegistry *    Instance();
```

AddDevice() and RemoveDevice() add and remove a GUSIDevice.

```
<Registration interface of GUSIDeviceRegistry 243>≡ (241)
void AddDevice(GUSIDevice * device);
void RemoveDevice(GUSIDevice * device);
```

On construction, a GUSIFileToken looks up the appropriate device in the GUSIDeviceRegistry.

```
<Looking up a device in the GUSIDeviceRegistry 244>≡ (241)
friend class GUSIFileToken;

GUSIDevice *    Lookup(GUSIFileToken & file);
```

It is convenient to define iterators to iterate across all devices.

```
<Iterator on GUSIDeviceRegistry 245>≡ (241)
class iterator;

iterator &      begin();
iterator &      end();
```

8.4 Operations on Devices

The open operation creates a new socket for the specified path or file specification.

```
<Requests for GUSIFileToken 246>≡ (239) 249▷
    kWillOpen,

<Operations for GUSIDevice 247>≡ (240) 250▷
    virtual GUSISocket * open(GUSIFileToken & file, int flags);

<Operations for GUSIDeviceRegistry 248>≡ (241) 251▷
    GUSISocket * open(const char * path, int flags);

    remove deletes a path or file specification.

<Requests for GUSIFileToken 246>+≡ (239) ◀246 252▷
    kWillRemove,

<Operations for GUSIDevice 247>+≡ (240) ◀247 253▷
    virtual int remove(GUSIFileToken & file);

<Operations for GUSIDeviceRegistry 248>+≡ (241) ◀248 254▷
    int remove(const char * path);

    rename renames a path or file specification.

<Requests for GUSIFileToken 246>+≡ (239) ◀249 255▷
    kWillRename,

<Operations for GUSIDevice 247>+≡ (240) ◀250 256▷
    virtual int rename(GUSIFileToken & from, const char * newname);

<Operations for GUSIDeviceRegistry 248>+≡ (241) ◀251 257▷
    int rename(const char * oldname, const char * newname);

    stat gathers statistical data about a file or directory.

<Requests for GUSIFileToken 246>+≡ (239) ◀252 258▷
    kWillStat,

<Operations for GUSIDevice 247>+≡ (240) ◀253 259▷
    virtual int stat(GUSIFileToken & file, struct stat * buf);

<Operations for GUSIDeviceRegistry 248>+≡ (241) ◀254 260▷
    int stat(const char * path, struct stat * buf, bool useAlias);

    chmod changes file modes, to the extent that this is meaningful on MacOS.

<Requests for GUSIFileToken 246>+≡ (239) ◀255 261▷
    kWillChmod,

<Operations for GUSIDevice 247>+≡ (240) ◀256 262▷
    virtual int chmod(GUSIFileToken & file, mode_t mode);
```

<i><Operations for GUSIDeviceRegistry 248>+≡</i>	(241) <257 263>
int chmod(const char * path, mode_t mode);	
utime bumps a file's modification time.	
<i><Requests for GUSIFileToken 246>+≡</i>	(239) <258 264>
kWillUtime,	
<i><Operations for GUSIDevice 247>+≡</i>	(240) <259 265>
virtual int utime(GUSIFileToken & file, const utimbuf * times);	
<i><Operations for GUSIDeviceRegistry 248>+≡</i>	(241) <260 266>
int utime(const char * path, const utimbuf * times);	
access checks access permissions for a file.	
<i><Requests for GUSIFileToken 246>+≡</i>	(239) <261 267>
kWillAccess,	
<i><Operations for GUSIDevice 247>+≡</i>	(240) <262 268>
virtual int access(GUSIFileToken & file, int mode);	
<i><Operations for GUSIDeviceRegistry 248>+≡</i>	(241) <263 269>
int access(const char * path, int mode);	
mkdir creates a directory.	
<i><Requests for GUSIFileToken 246>+≡</i>	(239) <264 270>
kWillMkdir,	
<i><Operations for GUSIDevice 247>+≡</i>	(240) <265 271>
virtual int mkdir(GUSIFileToken & file);	
<i><Operations for GUSIDeviceRegistry 248>+≡</i>	(241) <266 272>
int mkdir(const char * path);	
rmdir deletes a directory.	
<i><Requests for GUSIFileToken 246>+≡</i>	(239) <267 273>
kWillRmdir,	
<i><Operations for GUSIDevice 247>+≡</i>	(240) <268 274>
virtual int rmdir(GUSIFileToken & file);	
<i><Operations for GUSIDeviceRegistry 248>+≡</i>	(241) <269 275>
int rmdir(const char * path);	
opendir opens a directory handle on the given directory.	
<i><Requests for GUSIFileToken 246>+≡</i>	(239) <270 276>
kWillOpendir,	
<i><Operations for GUSIDevice 247>+≡</i>	(240) <271 277>
virtual GUSIDirectory * opendir(GUSIFileToken & file);	
<i><Operations for GUSIDeviceRegistry 248>+≡</i>	(241) <272 278>
GUSIDirectory * opendir(const char * path);	
symlink creates a symbolic link to a file.	
<i><Requests for GUSIFileToken 246>+≡</i>	(239) <273 279>
kWillSymlink,	

(Operations for GUSIDevice 247)+≡ (240) ◀274 280▶
 virtual int symlink(GUSIFileToken & to, const char * newlink);

(Operations for GUSIDeviceRegistry 248)+≡ (241) ◀275 281▶
 int symlink(const char * target, const char * newlink);
 readlink reads the contents of a symbolic link.

(Requests for GUSIFileToken 246)+≡ (239) ◀276 282▶
 kWillReadlink,

(Operations for GUSIDevice 247)+≡ (240) ◀277 283▶
 virtual int readlink(GUSIFileToken & link, char * buf, int bufsize);

(Operations for GUSIDeviceRegistry 248)+≡ (241) ◀278 284▶
 int readlink(const char * path, char * buf, int bufsize);
 fgetfileinfo and fsetfileinfo reads and set the type and creator code of a file.

(Requests for GUSIFileToken 246)+≡ (239) ◀279 285▶
 kWillGetfileinfo,
 kWillSetfileinfo,

(Operations for GUSIDevice 247)+≡ (240) ◀280 286▶
 virtual int fgetfileinfo(GUSIFileToken & file, OSType * creator, OSType * type);
 virtual int fsetfileinfo(GUSIFileToken & file, OSType creator, OSType type);

(Operations for GUSIDeviceRegistry 248)+≡ (241) ◀281 287▶
 int fgetfileinfo(const char * path, OSType * creator, OSType * type);
 int fsetfileinfo(const char * path, OSType creator, OSType type);
 faccess manipulates MPW properties of files.

(Requests for GUSIFileToken 246)+≡ (239) ◀282
 kWillFaccess,

(Operations for GUSIDevice 247)+≡ (240) ◀283
 virtual int faccess(GUSIFileToken & file, unsigned * cmd, void * arg);

(Operations for GUSIDeviceRegistry 248)+≡ (241) ◀284
 int faccess(const char * path, unsigned * cmd, void * arg);

8.5 Definition of GUSIDirectory

GUSIDirectory is a directory handle to iterate over all entries in a directory.

(Definition of class GUSIDirectory 288)≡ (237)
 class GUSIDirectory {
 public:
 virtual ~GUSIDirectory() {}
 virtual dirent * readdir() = 0;
 virtual long telldir() = 0;
 virtual void seekdir(long pos) = 0;
 virtual void rewinddir() = 0;
 protected:
 friend class GUSIDevice;
 GUSIDirectory() {}
 };

8.6 Implementation of GUSIFileToken

Identifying a name starting with "Dev:" as a file or a device is a dangerous job. Currently, there is no possibility to entirely disable device interpretation, so we're using the following heuristics in order to minimize conflicts with real file names:

- Any name corresponding to an existing file is a file
- Any name not recognized by any device domain is a file

We need many case insensitive comparisons. Since we always compare to fixed strings withing the ASCII set, we can do a cheap implementation.

```
(Member functions for class GUSIFileToken 289)≡ (238) 290»
bool GUSIFileToken::StrFragEqual(const char * name, const char * frag)
{
    do {
        if ((*name++ | 0x20) != *frag++)
            return false;
    } while (*frag);

    return true;
}

GUSIFileToken::GUSIFileToken(const char * path, Request request, bool useAlias)
    : fPath(path), GUSIFileSpec(path, useAlias), fDevice(nil), fRequest(request), fIsFile(false)
{
    if (!StrFragEqual(path, "dev:") || (!Error() && Exists()))
        goto treatAsFile;

    if (fDevice = GUSIDeviceRegistry::Instance()->Lookup(*this))
        return;

treatAsFile:
    fIsFile = true;
    fDevice = GUSIDeviceRegistry::Instance()->Lookup(*this);
}
```

Some names need to be detected frequently. We've already established that the name starts with "dev:".

```
(Member functions for class GUSIFileToken 289)+≡ (238) «289 291»
GUSIFileToken::StdStream GUSIFileToken::StrStdStream(const char * name)
{
    if (StrFragEqual(name+4, "console") && !name[11])
        return kConsole;
    if (StrFragEqual(name+4, "std"))
        if (StrFragEqual(name+7, "in") && !name[9])
            return kStdin;
        else if (StrFragEqual(name+7, "out") && !name[10])
            return kStdout;
        else if (StrFragEqual(name+7, "err") && !name[10])
            return kStderr;
    return kNoStdStream;
}
```

The other constructors of GUSIFileToken are always called for real files and thus present no ambiguities.

```

⟨Member functions for class GUSIFileToken 289⟩+≡ (238) 4290
GUSIFileToken::GUSIFileToken(const GUSIFileSpec & spec, Request request)
    : fPath(nil), GUSIFileSpec(spec), fDevice(nil), fRequest(request), fIsFile(true)
{
    fDevice = GUSIDeviceRegistry::Instance()->Lookup(*this);
}

GUSIFileToken::GUSIFileToken(short fRefNum, Request request)
    : fPath(nil), GUSIFileSpec(fRefNum), fDevice(nil), fRequest(request), fIsFile(true)
{
    fDevice = GUSIDeviceRegistry::Instance()->Lookup(*this);
}

```

8.7 Implementation of GUSIDevice

By default, a device wants no files.

```

⟨Member functions for class GUSIDevice 292⟩≡ (238) 295▷
bool    GUSIDevice::Want(GUSIFileToken &)
{
    return false;
}

```

Since individual devices are free to opt out of every operation, we have to implement them all as empty functions here. Calling any of these functions is an internal error.

```

⟨This should not happen, fail assertion and return EOPNOTSUPP 293⟩≡ (295)
{
    GUSI_SASSERT_INTERNAL(false, "Undefined device operation");

    return GUSISetPosixError(EOPNOTSUPP);
}

```

```

⟨This should not happen, fail assertion and return nil 294⟩≡ (295)
{
    GUSI_SASSERT_INTERNAL(false, "Undefined device operation");

    GUSISetPosixError(EOPNOTSUPP);

    return nil;
}

```

(*Member functions for class GUSIDevice* 292) +≡ (238) 4292

```

GUSISocket * GUSIDevice::open(GUSIFileToken &, int)
⟨This should not happen, fail assertion and return nil 294⟩

int GUSIDevice::remove(GUSIFileToken &)
⟨This should not happen, fail assertion and return EOPNOTSUPP 293⟩

int GUSIDevice::rename(GUSIFileToken &, const char *)
⟨This should not happen, fail assertion and return EOPNOTSUPP 293⟩

int GUSIDevice::stat(GUSIFileToken &, struct stat *)
⟨This should not happen, fail assertion and return EOPNOTSUPP 293⟩

int GUSIDevice::chmod(GUSIFileToken &, mode_t)
⟨This should not happen, fail assertion and return EOPNOTSUPP 293⟩

int GUSIDevice::utime(GUSIFileToken &, const utimbuf *)
⟨This should not happen, fail assertion and return EOPNOTSUPP 293⟩

int GUSIDevice::access(GUSIFileToken &, int)
⟨This should not happen, fail assertion and return EOPNOTSUPP 293⟩

int GUSIDevice::mkdir(GUSIFileToken &)
⟨This should not happen, fail assertion and return nil 294⟩

int GUSIDevice::rmdir(GUSIFileToken &)
⟨This should not happen, fail assertion and return nil 294⟩

GUSIDirectory * GUSIDevice::opendir(GUSIFileToken &)
⟨This should not happen, fail assertion and return nil 294⟩

int GUSIDevice::symlink(GUSIFileToken &, const char *)
⟨This should not happen, fail assertion and return EOPNOTSUPP 293⟩

int GUSIDevice::readlink(GUSIFileToken &, char *, int)
⟨This should not happen, fail assertion and return EOPNOTSUPP 293⟩

int GUSIDevice::fgetfileinfo(GUSIFileToken &, OSType *, OSType *)
⟨This should not happen, fail assertion and return EOPNOTSUPP 293⟩

int GUSIDevice::fsetfileinfo(GUSIFileToken &, OSType, OSType)
⟨This should not happen, fail assertion and return EOPNOTSUPP 293⟩

int GUSIDevice::faccess(GUSIFileToken &, unsigned *, void *)
⟨This should not happen, fail assertion and return EOPNOTSUPP 293⟩

```

8.8 Implementation of GUSIDeviceRegistry

(*Definition of GUSISetupDevices hook* 296) ≡ (237)

```

extern "C" void GUSISetupDevices();

```

(*Privatissima of GUSIDeviceRegistry* 297) ≡ (241) 300»

```

static GUSIDeviceRegistry * sInstance;

```



```

<Member functions for class GUSIDeviceRegistry 298>≡ (238) 301▷
    GUSIDeviceRegistry *    GUSIDeviceRegistry::sInstance;

```

```

<Inline member functions for class GUSIDeviceRegistry 299>≡ (237) 319▷
    inline GUSIDeviceRegistry * GUSIDeviceRegistry::Instance()
    {
        if (!sInstance) {
            sInstance = new GUSIDeviceRegistry();
            GUSISetupDevices();
        }

        return sInstance;
    }

```

Devices are stored in a linked list. On creation of the registry, it immediately registers the instance for plain Macintosh file sockets, to which pretty much all operations default. This device will never refuse any request.

```

<Privatissima of GUSIDeviceRegistry 297>+≡ (241) ◀297
    GUSIDevice *    fFirstDevice;
    GUSIDeviceRegistry();

```

```

<Member functions for class GUSIDeviceRegistry 298>+≡ (238) ◀298 302▷
    GUSIDeviceRegistry::GUSIDeviceRegistry()
        : fFirstDevice(nil)
    {
        AddDevice(GUSIMacFileDevice::Instance());
    }

```

AddDevice() and RemoveDevice() add and remove a GUSIDevice.

```

<Member functions for class GUSIDeviceRegistry 298>+≡ (238) ◀301 303▷
    void GUSIDeviceRegistry::AddDevice(GUSIDevice * device)
    {
        device->fNextDevice = fFirstDevice;
        fFirstDevice        = device;
    }

    void GUSIDeviceRegistry::RemoveDevice(GUSIDevice * device)
    {
        if (fFirstDevice == device)
            fFirstDevice = device->fNextDevice;
        else
            for (iterator dev = begin(); dev != end(); ++dev)
                if (dev->fNextDevice == device) {
                    dev->fNextDevice = device->fNextDevice;

                    break;
                }
    }

```

To look up, we iterate through all devices.

```
(Member functions for class GUSIDeviceRegistry 298)+≡ (238) <302 304>
GUSIDevice * GUSIDeviceRegistry::Lookup(GUSIFileToken & file)
{
    for (iterator dev = begin(); dev != end(); ++dev)
        if (dev->Want(file))
            return &(*dev);

    return static_cast<GUSIDevice *>(nil);
}

(Member functions for class GUSIDeviceRegistry 298)+≡ (238) <303 305>
GUSISocket * GUSIDeviceRegistry::open(const char * path, int flags)
{
    GUSIFileToken file(path, GUSIFileToken::kWillOpen, (flags & O_ALIAS) != 0);

    return file.Device()->open(file, flags);
}

(Member functions for class GUSIDeviceRegistry 298)+≡ (238) <304 306>
int GUSIDeviceRegistry::remove(const char * path)
{
    GUSIFileToken file(path, GUSIFileToken::kWillRemove, true);

    return file.Device()->remove(file);
}

(Member functions for class GUSIDeviceRegistry 298)+≡ (238) <305 307>
int GUSIDeviceRegistry::rename(const char * oldname, const char * newname)
{
    GUSIFileToken oldfile(oldname, GUSIFileToken::kWillRename, true);

    return oldfile.Device()->rename(oldfile, newname);
}

(Member functions for class GUSIDeviceRegistry 298)+≡ (238) <306 308>
int GUSIDeviceRegistry::stat(const char * path, struct stat * buf, bool useAlias)
{
    GUSIFileToken file(path, GUSIFileToken::kWillStat, useAlias);

    return file.Device()->stat(file, buf);
}

(Member functions for class GUSIDeviceRegistry 298)+≡ (238) <307 309>
int GUSIDeviceRegistry::chmod(const char * path, mode_t mode)
{
    GUSIFileToken file(path, GUSIFileToken::kWillChmod, false);

    return file.Device()->chmod(file, mode);
}
```

```

<Member functions for class GUSIDeviceRegistry 298>+≡ (238) <308 310>
int GUSIDeviceRegistry::utime(const char * path, const utimbuf * times)
{
    GUSIFileToken    file(path, GUSIFileToken::kWillUtime, false);

    return file.Device()->utime(file, times);
}

<Member functions for class GUSIDeviceRegistry 298>+≡ (238) <309 311>
int GUSIDeviceRegistry::access(const char * path, int mode)
{
    GUSIFileToken    file(path, GUSIFileToken::kWillAccess, false);

    return file.Device()->access(file, mode);
}

<Member functions for class GUSIDeviceRegistry 298>+≡ (238) <310 312>
int GUSIDeviceRegistry::mkdir(const char * path)
{
    GUSIFileToken    file(path, GUSIFileToken::kWillMkdir);

    return file.Device()->mkdir(file);
}

<Member functions for class GUSIDeviceRegistry 298>+≡ (238) <311 313>
int GUSIDeviceRegistry::rmdir(const char * path)
{
    GUSIFileToken    file(path, GUSIFileToken::kWillRmdir);

    return file.Device()->rmdir(file);
}

<Member functions for class GUSIDeviceRegistry 298>+≡ (238) <312 314>
GUSIDirectory * GUSIDeviceRegistry::opendir(const char * path)
{
    GUSIFileToken    file(path, GUSIFileToken::kWillOpendir);

    return file.Device()->opendir(file);
}

<Member functions for class GUSIDeviceRegistry 298>+≡ (238) <313 315>
int GUSIDeviceRegistry::symlink(const char * target, const char * newlink)
{
    GUSIFileToken    file(target, GUSIFileToken::kWillSymlink, true);

    return file.Device()->symlink(file, newlink);
}

<Member functions for class GUSIDeviceRegistry 298>+≡ (238) <314 316>
int GUSIDeviceRegistry::readlink(const char * path, char * buf, int bufsize)
{
    GUSIFileToken    file(path, GUSIFileToken::kWillReadlink, true);

    return file.Device()->readlink(file, buf, bufsize);
}

```

```

<Member functions for class GUSIDeviceRegistry 298>+≡ (238) <315 317>
int GUSIDeviceRegistry::fgetfileinfo(const char * path, OSType * creator, OSType * type)
{
    GUSIFileToken file(path, GUSIFileToken::kWillGetfileinfo, true);

    return file.Device()->fgetfileinfo(file, creator, type);
}

```

```

<Member functions for class GUSIDeviceRegistry 298>+≡ (238) <316 318>
int GUSIDeviceRegistry::fsetfileinfo(const char * path, OSType creator, OSType type)
{
    GUSIFileToken file(path, GUSIFileToken::kWillSetfileinfo, true);

    return file.Device()->fsetfileinfo(file, creator, type);
}

```

```

<Member functions for class GUSIDeviceRegistry 298>+≡ (238) <317
int GUSIDeviceRegistry::faccess(const char * path, unsigned * cmd, void * arg)
{
    GUSIFileToken file(path, GUSIFileToken::kWillFaccess, true);

    return file.Device()->faccess(file, cmd, arg);
}

```

The GUSIDeviceRegistry forward iterator is simple.

```

<Inline member functions for class GUSIDeviceRegistry 299>+≡ (237) <299
class GUSIDeviceRegistry::iterator {
public:
    iterator(GUSIDevice * device = 0) : fDevice(device) {}
    GUSIDeviceRegistry::iterator & operator++()
        { fDevice = fDevice->fNextDevice; return *this; }
    GUSIDeviceRegistry::iterator operator++(int)
        { GUSIDeviceRegistry::iterator old(*this); fDevice = fDevice->fNextDevice; return old; }
    bool operator==(const GUSIDeviceRegistry::iterator other) const
        { return fDevice==other.fDevice; }
    GUSIDevice & operator*() { return *fDevice; }
    GUSIDevice * operator->() { return fDevice; }
private:
    GUSIDevice * fDevice;
};

inline GUSIDeviceRegistry::iterator & GUSIDeviceRegistry::begin()
{
    return GUSIDeviceRegistry::iterator(fFirstDevice);
}

inline GUSIDeviceRegistry::iterator & GUSIDeviceRegistry::end()
{
    return GUSIDeviceRegistry::iterator();
}

```

Chapter 9

GUSI Configuration settings

GUSI stores its global configuration settings in the GUSIConfiguration singleton class. To create the instance, GUSI calls the GUSISetupConfig hook.

```
<GUSIConfig.h 320>≡
    #ifndef _GUSIConfig_
    #define _GUSIConfig_

    #ifdef GUSI_SOURCE

    #include "GUSIFileSpec.h"

    <Definition of class GUSIConfiguration 322>
    <Definition of GUSISetupConfig hook 332>
    <Inline member functions for class GUSIConfiguration 335>
    #endif /* GUSI_SOURCE */

    #endif /* _GUSIConfig_ */

<GUSIConfig.cp 321>≡
    #include "GUSIInternal.h"
    #include "GUSIConfig.h"
    #include "GUSIDiag.h"
    #include "GUSIContext.h"
    #include "GUSIBasics.h"
    #include "GUSIFSWrappers.h"

    #include <signal.h>
    #include <sys/signal.h>

    #include <PLStringFuncs.h>
    #include <Resources.h>
    #include <QuickDraw.h>
    #include <LowMem.h>
    #include <Script.h>

    <Definition of struct GUSIConfigRsrc 337>
    <Member functions for class GUSIConfiguration 334>
```

9.1 Definition of configuration settings

The GUSIConfiguration has a single instance with read only access, accessible with the static Instance() member function.

```
(Definition of class GUSIConfiguration 322)≡ (320)
class GUSIConfiguration {
public:
    enum { kNoResource = -1, kDefaultResourceID = 10240 };

    static GUSIConfiguration * Instance();
    static GUSIConfiguration * CreateInstance(short resourceID = kDefaultResourceID);

    <Type and creator rules for newly created files 323>
    <Automatic cursor spin 325>
    <Automatic initialization of QuickDraw 326>
    <Various flags 327>
protected:
    GUSIConfiguration(short resourceID = kDefaultResourceID);
private:
    <Privatissima of GUSIConfiguration 333>
};
```

To determine the file type and creator of a newly created file, we first try to match one of the FileSuffix suffices.

```
(Type and creator rules for newly created files 323)≡ (322) 324▶
struct FileSuffix {
    char    suffix[4];
    OSType  suffType;
    OSType  suffCreator;
};
short      fNumSuffices;
FileSuffix * fSuffices;

void ConfigureSuffices(short numSuffices, FileSuffix * suffices);
```

If none of the suffices matches, we apply the default type and creator. These rules are applied with SetDefaultFType().

```
(Type and creator rules for newly created files 323)+≡ (322) 4323
OSType      fDefaultType;
OSType      fDefaultCreator;
```

```
void ConfigureDefaultTypeCreator(OSType defaultType, OSType defaultCreator);
void SetDefaultFType(const GUSIFileSpec & name) const;
```

To simplify Macintosh friendly ports of simple, I/O bound programs it is possible to specify automatic yielding on read() and write() calls. AutoSpin() will spin a cursor and/or yield the CPU if desired.

```
(Automatic cursor spin 325)≡ (322)
bool fAutoSpin;

void ConfigureAutoSpin(bool autoSpin);
void AutoSpin() const;
```

GUSI applications can crash hard if QuickDraw is not initialized. Therefore, we offer to initialize it automatically with the `fAutoInitGraf` feature.

```

⟨Automatic initialization of QuickDraw 326⟩≡ (322)
    bool fAutoInitGraf;

    void ConfigureAutoInitGraf(bool autoInitGraf);
    void AutoInitGraf();

```

Due to the organization of a UNIX filesystem, it is fairly easy to find out how many subdirectories a given directory has, since the `nlink` field of its inode will automatically contain the number of subdirectories+2. Therefore, some UNIX derived code depends on this behaviour. When `fAccurateStat` is set, GUSI emulates this behaviour, but be warned that this makes `stat()` on directories a much more expensive operation. If `fAccurateStat` is not set, `stat()` gives the total number of entries in the directory+2 as a conservative estimate.

```

⟨Various flags 327⟩≡ (322) 328▷
    bool fAccurateStat;

    void ConfigureAccurateStat(bool accurateState);

```

The `fSigPipe` feature causes a signal `SIGPIPE` to be raised if an attempt is made to write to a broken pipe.

```

⟨Various flags 327⟩+≡ (322) ◀327 329▷
    bool fSigPipe;

    void ConfigureSigPipe(bool sigPipe);
    void BrokenPipe();

```

The `fSigInt` feature causes a signal `SIGINT` to be raised if the user presses command-period.

```

⟨Various flags 327⟩+≡ (322) ◀328 330▷
    bool fSigInt;

    void ConfigureSigInt(bool sigInt);
    void CheckInterrupt();

```

If `fSharedOpen` is set, `open()` opens files with shared read/write permission.

```

⟨Various flags 327⟩+≡ (322) ◀329 331▷
    bool fSharedOpen;

    void ConfigureSharedOpen(bool sharedOpen);

```

If `fHandleAppleEvents` is set, GUSI automatically handles `AppleEvents` in its event handling routine.

```

⟨Various flags 327⟩+≡ (322) ◀330
    bool fHandleAppleEvents;

    void ConfigureHandleAppleEvents(bool handleAppleEvents);

```

To create the sole instance of `GUSIConfiguration`, we call `GUSISetupConfig` which has to call `GUSIConfiguration::CreateInstance()`.

```

⟨Definition of GUSISetupConfig hook 332⟩≡ (320)
    extern "C" void GUSISetupConfig();

```

9.2 Implementation of configuration settings

The sole instance of GUSIConfiguration is created on demand.

```
(Privatissima of GUSIConfiguration 333)≡ (322) 342▷
    static GUSIConfiguration * sInstance;

(Member functions for class GUSIConfiguration 334)≡ (321) 336▷
    GUSIConfiguration * GUSIConfiguration::sInstance = nil;

(Inline member functions for class GUSIConfiguration 335)≡ (320) 341▷
    inline GUSIConfiguration * GUSIConfiguration::Instance()
    {
        if (!sInstance)
            GUSISetupConfig();
        if (!sInstance)
            sInstance = new GUSIConfiguration();

        return sInstance;
    }

    inline GUSIConfiguration * GUSIConfiguration::CreateInstance(short resourceID)
    {
        if (!sInstance)
            sInstance = new GUSIConfiguration(resourceID);

        return sInstance;
    }
```

The default implementation of GUSISetupConfig simply creates the instance. Feel free to override it.

```
(Member functions for class GUSIConfiguration 334)+≡ (321) ◀334 338▷
    void GUSISetupConfig()
    {
        GUSIConfiguration::CreateInstance();
    }
```

The configuration resource format of GUSI is quite old by now and has grown over the years. We have to be careful to keep it compatible across versions.

```
(Definition of struct GUSIConfigRsrc 337)≡ (321)
    struct GUSIConfigRsrc {
        OSType          fDefaultType;
        OSType          fDefaultCreator;

        bool            fAutoSpin;
        unsigned char    fFlags;

        OSType          fVersion;

        short            fNumSuffices;
        GUSIConfiguration::FileSuffix fSuffices[1];
    };
```


The GUSIConfiguration constructor initializes all fields to reasonable defaults and then looks for a configuration resources, if one is specified.

```

⟨Member functions for class GUSIConfiguration 334⟩+≡ (321) ◀336 345▶
GUSIConfiguration::GUSIConfiguration(short resourceID)
{
    ⟨Set defaults for GUSIConfiguration members 339⟩

    if (resourceID == kNoResource)
        return;

    GUSIConfigRsrc ** rsrc =
        reinterpret_cast<GUSIConfigRsrc **>(Get1Resource('GUI', resourceID));
    long size = GetHandleSize(Handle(rsrc));

    if (!rsrc || !size)
        return;

    OSType version = '0102';
    if (size >= 14)
        version = rsrc[0]->fVersion;

    ⟨Read configuration resource into GUSIConfiguration 340⟩
}

```

We show all fields with their three initialization methods tracks:

- How they are initialized.
- How they are read from the configuration resource.
- How they are set via a member function.

After that, we show how they are used.

ConfigureDefaultTypeCreator() sets the default type and creator.

```

⟨Set defaults for GUSIConfiguration members 339⟩≡ (338) 343▶
fDefaultType = 'TEXT';
fDefaultCreator = 'MPS ';

⟨Read configuration resource into GUSIConfiguration 340⟩≡ (338) 344▶
if (size >= 4 && rsrc[0]->fDefaultType)
    fDefaultType = rsrc[0]->fDefaultType;
if (size >= 8 && rsrc[0]->fDefaultCreator)
    fDefaultCreator = rsrc[0]->fDefaultCreator;

⟨Inline member functions for class GUSIConfiguration 335⟩+≡ (320) ◀335 350▶
inline void GUSIConfiguration::ConfigureDefaultTypeCreator(OSType defaultType, OSType defaultCreator)
{
    fDefaultType = defaultType;
    fDefaultCreator = defaultCreator;
}

ConfigureSuffices() sets up the suffix table.

⟨Privatissima of GUSIConfiguration 333⟩+≡ (322) ◀333 351▶
bool fWeOwnSuffices;

```

```

<Set defaults for GUSIConfiguration members 339>+≡ (338) <339 348>
    fNumSuffices    =    0;
    fSuffices       =    nil;
    fWeOwnSuffices  =    false;

<Read configuration resource into GUSIConfiguration 340>+≡ (338) <340 349>
    if (version >= '0120' && size >= 16)
        fNumSuffices = rsrc[0]->fNumSuffices;
    if (fNumSuffices && (fSuffices = new FileSuffix[fNumSuffices])) {
        fWeOwnSuffices = true;
        memcpy(fSuffices, rsrc[0]->fSuffices, fNumSuffices*sizeof(FileSuffix));
        for (int i=0; i<fNumSuffices; i++)
            for (int j=0; j<4; j++)
                if (fSuffices[i].suffix[j] == ' ')
                    fSuffices[i].suffix[j] = 0;
    }

<Member functions for class GUSIConfiguration 334>+≡ (321) <338 346>
void GUSIConfiguration::ConfigureSuffices(short numSuffices, FileSuffix * suffices)
{
    if (fWeOwnSuffices) {
        delete fSuffices;
        fWeOwnSuffices = false;
    }
    fNumSuffices = numSuffices;
    fSuffices = suffices;
}

SetDefaultFType() assigns the preferred file type to a file.

<Member functions for class GUSIConfiguration 334>+≡ (321) <345 353>
void GUSIConfiguration::SetDefaultFType(const GUSIFileSpec & name) const
{
    FInfo    info;

    if (GUSIFSGetFInfo(&name, &info))
        return;
    info.fdType    =    fDefaultType;
    info.fdCreator =    fDefaultCreator;

    <Try matching file suffix rules 347>
determined:
    GUSIFSSetFInfo(&name, &info);
}

```

⟨Try matching file suffix rules 347⟩≡ (346)

```
Ptr dot = PLstrrchr(name->name, '.');
if (dot && (name->name[0] - (dot-Ptr(name->name))) <= 4) {
    char searchsuffix[5];
    strncpy(searchsuffix, dot+1, name->name[0] - (dot-Ptr(name->name)));

    for (int i = 0; i<fNumSuffices; i++)
        if (!strcmp(fSuffices[i].suffix, searchsuffix, 4)) {
            info.fdType      = fSuffices[i].suffType;
            info.fdCreator   = fSuffices[i].suffCreator;

            goto determined;
        }
}
```

ConfigureAutoSpin() sets the autospin flag, which is enabled by default.

⟨Set defaults for GUSIConfiguration members 339⟩+≡ (338) ◀343 354▶

```
fAutoSpin = true;
```

⟨Read configuration resource into GUSIConfiguration 340⟩+≡ (338) ◀344 355▶

```
if (size >= 9)
    fAutoSpin = (rsrc[0]->fAutoSpin != 0);
```

⟨Inline member functions for class GUSIConfiguration 335⟩+≡ (320) ◀341 352▶

```
inline void GUSIConfiguration::ConfigureAutoSpin(bool autoSpin)
{
    fAutoSpin = autoSpin;
}
```

AutoSpin() tests the flag inline, but performs the actual spinning out of line.

⟨Privatissima of GUSIConfiguration 333⟩+≡ (322) ◀342 357▶

```
void DoAutoSpin() const;
```

⟨Inline member functions for class GUSIConfiguration 335⟩+≡ (320) ◀350 356▶

```
inline void GUSIConfiguration::AutoSpin() const
{
    if (fAutoSpin)
        DoAutoSpin();
}
```

⟨Member functions for class GUSIConfiguration 334⟩+≡ (321) ◀346 359▶

```
void GUSIConfiguration::DoAutoSpin() const
{
    GUSIContext::Yield(false);
}
```

ConfigureAutoInitGraf() controls automatic initialization of QuickDraw.

⟨Set defaults for GUSIConfiguration members 339⟩+≡ (338) ◀348 360▶

```
fAutoInitGraf = true;
```

⟨Read configuration resource into GUSIConfiguration 340⟩+≡ (338) ◀349 361▶

```
fAutoInitGraf = version < '0174' || (rsrc[0]->fFlags & 0x04) == 0;
```

⟨Inline member functions for class GUSIConfiguration 335⟩+≡ (320) ◀352 358▶

```
inline void GUSIConfiguration::ConfigureAutoInitGraf(bool autoInitGraf)
{
    fAutoInitGraf = autoInitGraf;
}
```

AutoInitGraf() works rather similarly to AutoSpin.

```
<Privatissima of GUSIConfiguration 333>+≡ (322) <351 367>
void DoAutoInitGraf();
```

```
<Inline member functions for class GUSIConfiguration 335>+≡ (320) <356 362>
inline void GUSIConfiguration::AutoInitGraf()
{
    if (fAutoInitGraf)
        DoAutoInitGraf();
}
```

To make sure that A5 doesn't point into an overly wild location, we perform some sanity checks before getting to the point.

```
<Member functions for class GUSIConfiguration 334>+≡ (321) <353 363>
void GUSIConfiguration::DoAutoInitGraf()
{
    Ptr curA5 = LMGetCurrentA5();

    if (!(reinterpret_cast<long>(curA5) & 3) && curA5 > reinterpret_cast<Ptr>(ApplicationZone())) &&
        if (*reinterpret_cast<GrafPtr **>(curA5) != &qd.thePort)
            InitGraf(&qd.thePort);
    fAutoInitGraf = false;
}
```

ConfigureSigPipe controls whether SIGPIPE signals are generated.

```
<Set defaults for GUSIConfiguration members 339>+≡ (338) <354 364>
fSigPipe = false;
```

```
<Read configuration resource into GUSIConfiguration 340>+≡ (338) <355 365>
fSigPipe = version >= '0174' && (rsrc[0]->fFlags & 0x01) != 0;
```

```
<Inline member functions for class GUSIConfiguration 335>+≡ (320) <358 366>
inline void GUSIConfiguration::ConfigureSigPipe(bool sigPipe)
{
    fSigPipe = sigPipe;
}
```

BrokenPipe() raises a SIGPIPE signal if desired.

```
<Member functions for class GUSIConfiguration 334>+≡ (321) <359 368>
void GUSIConfiguration::BrokenPipe()
{
    if (fSigPipe)
        raise(SIGPIPE);
}
```

ConfigureSigInt controls whether [[SIGINT signals are generated.

```
<Set defaults for GUSIConfiguration members 339>+≡ (338) <360 372>
fSigInt = true;
```

```
<Read configuration resource into GUSIConfiguration 340>+≡ (338) <361 373>
fSigInt = true;
```

```
<Inline member functions for class GUSIConfiguration 335>+≡ (320) <362 374>
inline void GUSIConfiguration::ConfigureSigInt(bool sigInt)
{
    fSigInt = sigInt;
}
```

CheckInterrupt() raises a SIGINT signal if desired.

<Privatissima of GUSIConfiguration 333>+≡ (322) <357
 bool CmdPeriod(const EventRecord * event);

<Member functions for class GUSIConfiguration 334>+≡ (321) <363 369>
 void GUSIConfiguration::CheckInterrupt()
 {
 if (fSigInt) {
 EvQElPtr eventQ;

 for (eventQ = (EvQElPtr) LMGetEventQueue()->qHead; eventQ;)
 if (CmdPeriod(reinterpret_cast<EventRecord *>(&eventQ->evtQWhat))) {
 raise(SIGINT);
 FlushEvents(-1, 0);
 break;
 } else
 eventQ = (EvQElPtr)eventQ->qLink;
 }
 }

Checking for the Command-Period key combination is rather complex. Our technique is copied straight from tech note 263.

<Member functions for class GUSIConfiguration 334>+≡ (321) <368 377>
 bool GUSIConfiguration::CmdPeriod(const EventRecord * event)
 {
 <Constants for CmdPeriod 370>

 if ((event->what == keyDown) || (event->what == autoKey)) {
 // see if the command key is down. If it is, find out the ASCII
 // equivalent for the accompanying key.

 if (event->modifiers & cmdKey) {
 <Find ASCII equivalent of virtual key 371>

 if ((keyInfo & kMaskASCII2) == '.' || (keyInfo & kMaskASCII1) == ('.' << 16))
 return true;
 } // end the command key is down
 } // end key down event

 return false;
 }

<Constants for CmdPeriod 370>≡ (369)
 const long kMaskModifiers = 0xFE00; // we need the modifiers without the
 // command key for KeyTrans
 const long kMaskVirtualKey = 0x0000FF00; // get virtual key from event message
 // for KeyTrans
 const long kUpKeyMask = 0x0080;
 const long kShiftWord = 8; // we shift the virtual key to mask it
 // into the keyCode for KeyTrans
 const long kMaskASCII1 = 0x00FF0000; // get the key out of the ASCII1 byte
 const long kMaskASCII2 = 0x000000FF; // get the key out of the ASCII2 byte

```

<Find ASCII equivalent of virtual key 371>≡ (369)
short   virtualKey = (event->message & kMaskVirtualKey) >> kShiftWord;
short   keyCode    = (event->modifiers & kMaskModifiers) | virtualKey;
UInt32  state      = 0;
Handle  hKCHR      = nil;
Ptr     KCHRPtr    = reinterpret_cast<Ptr>(GetScriptManagerVariable(smKCHRCache));
if (!KCHRPtr) {
    short script    = GetScriptManagerVariable(smKeyScript);
    short kcID      = GetScriptVariable(script, smScriptKeys);
    hKCHR          = GetResource('KCHR', kcID);
    KCHRPtr        = *hKCHR;
}
short   keyInfo;
if (KCHRPtr) {
    keyInfo = KeyTrans(KCHRPtr, keyCode, &state);
    if (hKCHR)
        ReleaseResource(hKCHR);
} else
    keyInfo = event->message;

```

fAccurateStat and fSharedOpen are tested from client code.

```

<Set defaults for GUSIConfiguration members 339>+≡ (338) <364 375>
fAccurateStat    = false;
fSharedOpen      = false;

```

```

<Read configuration resource into GUSIConfiguration 340>+≡ (338) <365 376>
fAccurateStat    = (rsrc[0]->fFlags & 0x80) != 0;
fSharedOpen      = version >= '0174' && (rsrc[0]->fFlags & 0x02) != 0;

```

```

<Inline member functions for class GUSIConfiguration 335>+≡ (320) <366
inline void GUSIConfiguration::ConfigureAccurateStat(bool accurateStat)
{
    fAccurateStat = accurateStat;
}
inline void GUSIConfiguration::ConfigureSharedOpen(bool sharedOpen)
{
    fSharedOpen = sharedOpen;
}

```

ConfigureHandleAppleEvents is somewhat unusual in that it calls out into client code instead of the other way around.

```

<Set defaults for GUSIConfiguration members 339>+≡ (338) <372
fHandleAppleEvents = true;

```

```

<Read configuration resource into GUSIConfiguration 340>+≡ (338) <373
ConfigureHandleAppleEvents(version < '0181' || (rsrc[0]->fFlags & 0x08) == 0);

```

```

<Member functions for class GUSIConfiguration 334>+≡ (321) <369
void GUSIConfiguration::ConfigureHandleAppleEvents(bool handleAppleEvents)
{
    if (fHandleAppleEvents != handleAppleEvents) {
        fHandleAppleEvents = handleAppleEvents;
        GUSISetHook(GUSI_EventHook+kHighLevelEvent, (GUSIHook) (fHandleAppleEvents ? 0 : -1));
    }
}

```

Chapter 10

Context Queues

At all times through its existence, a `GUSIContext` will exist in various queues: A queue of all contexts, queues of contexts waiting on a socket event, a mutex, or a condition variable, and so on. Since a context is often in several queues simultaneously, it's better to define queues non-intrusively.

```
<GUSIContextQueue.h 378>≡
    #ifndef _GUSIContextQueue_
    #define _GUSIContextQueue_

    #ifndef GUSI_SOURCE

    typedef struct GUSIContextQueue GUSIContextQueue;
    #else

    #include <stdlib.h>

    <Name dropping for file GUSIContextQueue 380>

    <Definition of class GUSIContextQueue 381>

    <Inline member functions for class GUSIContextQueue 390>
    #endif /* GUSI_SOURCE */

    #endif /* _GUSIContextQueue_ */
<GUSIContextQueue.cp 379>≡
    #include "GUSIInternal.h"

    #define GUSI_DIAG GUSI_DIAG_CAUTIOUS
    #include "GUSIDiag.h"

    #include "GUSIContextQueue.h"
    #include "GUSIContext.h"

    #include <string.h>

    #include <Memory.h>

    <Implementation of context queues 385>
```

10.1 Definition of context queues

We'd like to avoid having to include `GUSIContext` here, for reasons that should be rather obvious.

(Name dropping for file GUSIContextQueue 380)≡ (378)
`class GUSIContext;`

The class `GUSIContextQueue` tries to present an interface that is a subset of what C++ standard library list template classes offer.

(Definition of class GUSIContextQueue 381)≡ (378)
`class GUSIContextQueue {`
`public:`
`GUSIContextQueue();`
`~GUSIContextQueue();`

`bool empty();`
`GUSIContext * front() const;`
`GUSIContext * back() const;`
`void push_front(GUSIContext * context);`
`void push_back(GUSIContext * context);`
`void push(GUSIContext * context) { push_back(context); }`
`void pop_front();`
`void pop() { pop_front(); }`
`void remove(GUSIContext * context);`

`void Wakeup();`

(Define iterator for GUSIContextQueue 382)
`private:`
(Privatissima of GUSIContextQueue 383)
`};`

We define a forward iterator, but no reverse iterator.

(Define iterator for GUSIContextQueue 382)≡ (381)
`struct element;`
`class iterator {`
`friend class GUSIContextQueue;`
`public:`
`iterator & operator++();`
`iterator operator++(int);`
`bool operator==(const iterator other) const;`
`GUSIContext * operator*();`
`GUSIContext * operator->();`
`private:`
(Privatissima of GUSIContextQueue::iterator 396)
`};`

`iterator begin();`
`iterator end();`

10.2 Implementation of context queues

Efficiency of context queues is quite important, so we provide a custom allocator for queue elements.

```
<Privatissima of GUSIContextQueue 383>≡ (381) 389▷  
    struct element {  
        GUSIContext *   fContext;  
        element *       fNext;  
  
        element(GUSIContext * context, element * next = 0)  
            : fContext(context), fNext(next) {}  
        void * operator new(size_t);  
        void operator delete(void *, size_t);  
private:  
    <Privatissima of GUSIContextQueue::element 384>  
};
```

Elements are allocated in blocks of increasing size.

```
<Privatissima of GUSIContextQueue::element 384>≡ (383)  
    struct header {  
        short   fFree;  
        short   fMax;  
        header *fNext;  
    };  
    static header * sBlocks;  
  
<Implementation of context queues 385>≡ (379) 386▷  
    GUSIContextQueue::element::header * GUSIContextQueue::element::sBlocks = 0;
```

operator new must be prepared to allocate a new block.

(Implementation of context queues 385)+≡ (379) <385 387>

```
void * GUSIContextQueue::element::operator new(size_t)
{
    header * b;

    for (b = sBlocks; b; b = b->fNext)
        if (b->fFree)
            break;

    if (!b) {
        short max = sBlocks ? (sBlocks->fMax << 1) : 64;
        b = reinterpret_cast<header *>(NewPtr(max << 3));
        memset(b, 0, max << 3);
        b->fMax = max;
        b->fFree = max-1;
        b->fNext = sBlocks;
        sBlocks = b;
    }

    element * e = reinterpret_cast<element *>(b);
    while ((++e)->fContext)
        ;
    --b->fFree;

    return e;
}
```

To avoid borderline cases, operator delete only deletes a block if there are enough other free elements.

(Implementation of context queues 385)+≡ (379) <386 391>

```
void GUSIContextQueue::element::operator delete(void * elem, size_t)
{
    header * h = static_cast<header *>(elem);
    header * b;
    header * p = 0;

    for (b = sBlocks; b; b = b->fNext)
        if (h > b && h < b+b->fMax) {
            memset(h, 0, sizeof(header));
            if (++b->fFree == b->fMax-1) {
                (Delete b if enough free elements remain 388)
            }
            return;
        }
    // Can't reach
}
```

<Delete b if enough free elements remain 388>≡ (387)

```

int sum = 0;
for (header * s = sBlocks; s; s = s->fNext)
    if (s != b)
        if ((sum += s->fFree) > 32) {
            if (p)
                p->fNext = b->fNext;
            else
                sBlocks = b->fNext;
            DisposePtr(reinterpret_cast<Ptr>(b));

            break;
        }

```

A GUSIContextQueue is a single linked list with a separate back pointer.

<Privatissima of GUSIContextQueue 383>+≡ (381) ◀383

```

element *   fFirst;
element *   fLast;

```

<Inline member functions for class GUSIContextQueue 390>≡ (378) 392▷

```

inline GUSIContextQueue::GUSIContextQueue()
    : fFirst(0), fLast(0)
{
}

```

<Implementation of context queues 385>+≡ (379) ◀387 393▷

```

GUSIContextQueue::~GUSIContextQueue()
{
    while (!empty())
        pop_front();
}

```

None of the member functions are very large, so we'll inline them.

```

<Inline member functions for class GUSIContextQueue 390>+≡ (378) <390 397>
inline bool GUSIContextQueue::empty()
{
    return !fFirst;
}

inline GUSIContext * GUSIContextQueue::front() const
{
    return fFirst ? fFirst->fContext : reinterpret_cast<GUSIContext *>(0);
}

inline GUSIContext * GUSIContextQueue::back() const
{
    return fLast ? fLast->fContext : reinterpret_cast<GUSIContext *>(0);
}

inline void GUSIContextQueue::push_front(GUSIContext * context)
{
    fFirst = new element(context, fFirst);
    if (!fLast)
        fLast = fFirst;
}

inline void GUSIContextQueue::pop_front()
{
    if (element * e = fFirst) {
        if (!(fFirst = fFirst->fNext))
            fLast = 0;
        delete e;
    }
}

```

Making remove() inline as well would probably push things too far, though.

```

<Implementation of context queues 385>+≡ (379) <391 394>
void GUSIContextQueue::remove(GUSIContext * context)
{
    if (fFirst)
        if (fFirst->fContext == context)
            pop_front();
        else {
            element * prev = fFirst;
            for (element * cur = prev->fNext; cur; cur = cur->fNext)
                if (cur->fContext == context) {
                    if (!(prev->fNext = cur->fNext)) // Delete last element
                        fLast = prev;
                    delete cur;

                    return;
                } else
                    prev = cur;
        }
}

```

I'd love to have `push_back` inline, but it doesn't seem to work.

```

<Implementation of context queues 385>+≡ (379) <393 395>
void GUSIContextQueue::push_back(GUSIContext * context)
{
    element * e = new element(context);
    if (fLast)
        fLast->fNext = e;
    else
        fFirst = e;
    fLast = e;
}

```

`Wakeup()` is the only context specific operation.

```

<Implementation of context queues 385>+≡ (379) <394
void GUSIContextQueue::Wakeup()
{
    GUSI_MESSAGE(("Wakeup #%h\n", this));
    for (element * cur = fFirst; cur; cur = cur->fNext)
        cur->fContext->Wakeup();
}

```

A `GUSIContextQueue::iterator` is just a wrapper for a `GUSIContextQueue::element`.

```

<Privatissima of GUSIContextQueue::iterator 396>≡ (382)
element * fCurrent;

iterator(element * elt) : fCurrent(elt) {}
iterator() : fCurrent(0) {}

```

The constructors are not public, so only `begin()` and `end()` call them.

```

<Inline member functions for class GUSIContextQueue 390>+≡ (378) <392 398>
inline GUSIContextQueue::iterator GUSIContextQueue::begin()
{
    return iterator(fFirst);
}

inline GUSIContextQueue::iterator GUSIContextQueue::end()
{
    return iterator();
}

```

```

(Inline member functions for class GUSIContextQueue 390)+≡ (378) 4397
inline GUSIContextQueue::iterator & GUSIContextQueue::iterator::operator++()
{
    fCurrent = fCurrent->fNext;

    return *this;
}

inline GUSIContextQueue::iterator GUSIContextQueue::iterator::operator++(int)
{
    GUSIContextQueue::iterator it(*this);
    fCurrent = fCurrent->fNext;

    return it;
}

inline bool GUSIContextQueue::iterator::operator==(const iterator other) const
{
    return fCurrent == other.fCurrent;
}

inline GUSIContext * GUSIContextQueue::iterator::operator*()
{
    return fCurrent->fContext;
}

inline GUSIContext * GUSIContextQueue::iterator::operator->()
{
    return fCurrent->fContext;
}

```

Chapter 11

DCon interface

A GUSIDConSocket implements an interface to DCon, Cache Computing's debugging console. For more information about DCon, see Cache Computing's site at <http://www.cache-computing.com/products/dcon>

All instances of GUSIDConSocket are created by the GUSIDConDevice singleton, so there is no point in exporting the class itself.

```
<GUSIDCon.h 399>≡
    #ifndef _GUSIDCon_
    #define _GUSIDCon_

    #ifdef GUSI_INTERNAL

    #include "GUSIDevice.h"

    <Definition of class GUSIDConDevice 401>

    <Inline member functions for class GUSIDConDevice 405>

    #endif /* GUSI_INTERNAL */

    #endif /* _GUSIDCon_ */

<GUSIDCon.cp 400>≡
    #include "GUSIInternal.h"
    #include "GUSIDCon.h"
    #include "GUSIBasics.h"
    #include "GUSIDdiag.h"

    #include <fcntl.h>
    #include <stddef.h>

    #include <DCon.h>

    <Definition of class GUSIDConSocket 402>
    <Member functions for class GUSIDConDevice 403>
    <Member functions for class GUSIDConSocket 408>
```

11.1 Definition of GUSIDConDevice

GUSIDConDevice is a singleton subclass of GUSIDDevice.

(Definition of class GUSIDConDevice 401)≡ (399)

```
class GUSIDConDevice : public GUSIDDevice {
public:
    static GUSIDConDevice * Instance();
    virtual bool    Want(GUSIFileToken & file);
    virtual GUSISocket * open(GUSIFileToken & file, int flags);
protected:
    GUSIDConDevice()          {}
    static GUSIDConDevice * sInstance;
};
```

11.2 Definition of GUSIDConSocket

A GUSIDConSocket writes to the console or to a file.

(Definition of class GUSIDConSocket 402)≡ (400)

```
class GUSIDConSocket :
    public GUSISocket
{
    char * fLog;
public:
    GUSIDConSocket(const char * log);
    ~GUSIDConSocket();
    (Overridden member functions for GUSIDConSocket 409)
};
```

11.3 Implementation of GUSIDConDevice

You can use GUSIDConSockets directly from C++, but the usual way to use them is to call GUSIwithDConSockets to have "Dev:DCon" and "Dev:DCon:xxx" mapped to them.

(Member functions for class GUSIDConDevice 403)≡ (400) 404▷

```
extern "C" void GUSIwithDConSockets()
{
    GUSIDDeviceRegistry::Instance()->AddDevice(GUSIDConDevice::Instance());
}
```

(Member functions for class GUSIDConDevice 403)+≡ (400) ◀403 406▷

```
GUSIDConDevice * GUSIDConDevice::sInstance = nil;
```

(Inline member functions for class GUSIDConDevice 405)≡ (399)

```
inline GUSIDConDevice * GUSIDConDevice::Instance()
{
    if (!sInstance)
        sInstance = new GUSIDConDevice;
    return sInstance;
}
```


GUSIDConDevice will handle only the open request.

```
(Member functions for class GUSIDConDevice 403)+≡ (400) <404 407>
bool GUSIDConDevice::Want(GUSIFileToken & file)
{
    const char * path = file.Path();

    return file.WhichRequest() == GUSIFileToken::kWillOpen
        && file.StrFragEqual(path+4, "dcon")
        && (!path[8] || (path[8] == ':' && path[9]));
}
```

Open will never fail except for lack of memory.

```
(Member functions for class GUSIDConDevice 403)+≡ (400) <406
GUSISocket * GUSIDConDevice::open(GUSIFileToken & file, int)
{
    const char * path = file.Path();

    GUSISocket * sock =
        path[8] ? new GUSIDConSocket(path+9) : new GUSIDConSocket(nil);
    if (!sock)
        GUSISetPosixError(ENOMEM);
    return sock;
}
```

11.4 Implementation of GUSIDConSocket

The implementation of GUSIDConSocket is trivial.

```
(Member functions for class GUSIDConSocket 408)≡ (400) 410>
GUSIDConSocket::GUSIDConSocket(const char * log)
    : fLog(nil)
{
    if (log)
        fLog = strcpy(new char[strlen(log)+1], log);
}
GUSIDConSocket::~GUSIDConSocket()
{
    delete fLog;
}
```

Read always returns EOF.

```
(Overridden member functions for GUSIDConSocket 409)≡ (402) 411>
virtual ssize_t read(const GUSIScatterer & buffer);

(Member functions for class GUSIDConSocket 408)+≡ (400) <408 412>
ssize_t GUSIDConSocket::read(const GUSIScatterer &)
{
    return 0;
}
```

Writes get translated into their DCon equivalents.

```
(Overridden member functions for GUSIDConSocket 409)+≡ (402) <409 413>
virtual ssize_t write(const GUSIGatherer & buffer);
```

<Member functions for class GUSIDConSocket 408>+≡ (400) <410 414>
 ssize_t GUSIDConSocket::write(const GUSIGatherer & buffer)
 {
 dfprintf(fLog, "%.s", buffer.Length(), buffer.Buffer());
 return buffer.Length();
 }

DCon sockets implement simple calls.

<Overridden member functions for GUSIDConSocket 409>+≡ (402) <411 415>
 virtual bool Supports(ConfigOption config);

<Member functions for class GUSIDConSocket 408>+≡ (400) <412 416>
 bool GUSIDConSocket::Supports(ConfigOption config)
 {
 return config == kSimpleCalls;
 }

As the name says, DCon sockets are consolish.

<Overridden member functions for GUSIDConSocket 409>+≡ (402) <413
 virtual int isatty();

<Member functions for class GUSIDConSocket 408>+≡ (400) <414
 int GUSIDConSocket::isatty()
 {
 return 1;
 }

Chapter 12

Timing functions

This section defines mechanisms to measure time. The basic mechanism is GUSITimer which can wake up a GUSIContext at some later time.

```
<GUSITimer.h 417>≡
#ifndef _GUSITimer_
#define _GUSITimer_

#ifndef GUSI_SOURCE

typedef struct GUSITimer GUSITimer;

#else
#include <errno.h>
#include <sys/cdefs.h>
#include <sys/types.h>
#include <time.h>
#include <sys/time.h>
#include <inttypes.h>

#include <MacTypes.h>
#include <Timer.h>
#include <Math64.h>

<Definition of class GUSITime 419>
<Definition of class GUSITimer 420>

#endif /* GUSI_SOURCE */

#endif /* _GUSITimer_ */
```

```
<GUSITimer.cp 418>≡
#include "GUSInternal.h"
#include "GUSITimer.h"
#include "GUSIContext.h"
#include "GUSIDdiag.h"

#include <errno.h>

<Member functions for class GUSITime 421>
<Member functions for class GUSITimer 429>
```

12.1 Definition of timing

GUSITime is an universal (if somewhat costly) format for the large variety of timing formats used in MacOS and POSIX.

```
(Definition of class GUSITime 419)≡ (417)
class GUSITime {
public:
    enum Format {seconds, ticks, msecs, usecs, nsecs};

#if !TYPE_LONGLONG
    GUSITime(int32_t val, Format format);
#endif
    GUSITime(int64_t val, Format format=nsecs) { Construct(val, format); }
    GUSITime(const timeval & tv);
    GUSITime(const timespec & ts);
    GUSITime(const tm & t);
    GUSITime() {}

    int32_t      Get(Format format)      { return S32Set(Get64(format)); }
    uint32_t     UGet(Format format)     { return U32SetU(SInt64ToUInt64(Get64(format))); }
    int64_t      Get64(Format format);

    operator int64_t()      { return fTime; }
    operator timeval();
    operator timespec();
    operator tm();

    GUSITime GMTTime();
    GUSITime LocalTime();

    GUSITime & operator +=(const GUSITime & other)
        { fTime = S64Add(fTime, other.fTime); return *this; }
    GUSITime & operator -=(const GUSITime & other)
        { fTime = S64Subtract(fTime, other.fTime); return *this; }

    static GUSITime      Now();
    static timezone &    Zone();
private:
    void      Construct(int64_t val, Format format);
    time_t    Deconstruct(int64_t & remainder);

    int64_t fTime;

    static int64_t      sTimeOffset;
    static timezone     sTimeZone;
};

inline GUSITime operator+(const GUSITime & a, const GUSITime & b)
    { GUSITime t(a); return t+=b; }
inline GUSITime operator-(const GUSITime & a, const GUSITime & b)
    { GUSITime t(a); return t-=b; }
```

A GUSITimer is a time manager task that wakes up a GUSIContext.

```
(Definition of class GUSITimer 420)≡ (417)
#if PRAGMA_ALIGN_SUPPORTED
#pragma options align=mac68k
#endif
class GUSIContext;

extern "C" void GUSIKillTimers(void * timers);

class GUSITimer : public TMTask {
public:
    GUSITimer(bool wakeup = true, GUSIContext * context = 0);
    virtual ~GUSITimer();

    void Sleep(long ms, bool driftFree = false);
    void MicroSleep(long us, bool driftFree = false)
        { Sleep(-us, driftFree); }
    GUSIContext * Context() { return fQueue->fContext; }
    GUSITimer * Next() { return fNext; }
    bool Primed() { return (qType&kTMTTaskActive) != 0; }
    virtual void Wakeup();
    void Kill();

    struct Queue {
        GUSITimer * fTimer;
        GUSIContext * fContext;

        Queue() : fTimer(0) {}
    };

    QElem * Elem() { return reinterpret_cast<QElem *>(&this->qLink); }
protected:
    Queue * fQueue;
    GUSITimer * fNext;

    class TimerQueue : public GUSISpecificData<Queue,GUSIKillTimers> {
    public:
        ~TimerQueue();
    };

    static TimerQueue sTimerQueue;
    static TimerUPP sTimerProc;
};
#if PRAGMA_ALIGN_SUPPORTED
#pragma options align=reset
#endif
```

12.2 Implementation of GUSITime

The constructors need various multipliers.

```
(Member functions for class GUSITime 421)≡ (418) 422▷
  #if !TYPE_LONGLONG
  GUSITime::GUSITime(int32_t val, Format format)
  {
    Construct(S64Set(val), format);
  }
  #endif

void GUSITime::Construct(int64_t val, Format format)
{
  switch (format) {
  case seconds:
    fTime = S64Multiply(val, S64Set(1000000000));
    break;
  case ticks:
    fTime = S64Multiply(val, S64Set(16666667));
    break;
  case msecs:
    fTime = S64Multiply(val, S64Set(1000000));
    break;
  case usecs:
    fTime = S64Multiply(val, S64Set(1000));
    break;
  case nsecs:
    fTime = val;
    break;
  }
}
```

The struct constructors are simple arithmetic.

```
(Member functions for class GUSITime 421)+≡ (418) 4421 423▷
  GUSITime::GUSITime(const timeval & tv)
  {
    *this = GUSITime(tv.tv_sec, seconds) + GUSITime(tv.tv_usec, usecs);
  }

  GUSITime::GUSITime(const timespec & ts)
  {
    *this = GUSITime(ts.tv_sec, seconds) + GUSITime(ts.tv_nsec, nsecs);
  }
```

The basic get operation is the one returning an int64_t.

(Member functions for class GUSITime 421) +≡ (418) <422 424>

```
int64_t GUSITime::Get64(Format format)
{
    switch (format) {
    case seconds:
        return S64Div(fTime, S64Set(1000000000));
    case ticks:
        return S64Div(fTime, S64Set(16666667));
    case msecs:
        return S64Div(fTime, S64Set(1000000));
    case usecs:
        return S64Div(fTime, S64Set(1000));
    default:
    case nsecs:
        return fTime;
    }
}
```

The struct casts break down the value into seconds and a remainder. For some reason, S64Divide is not defined for native functions.

(Member functions for class GUSITime 421) +≡ (418) <423 425>

```
time_t GUSITime::Deconstruct(int64_t & rem)
{
    #if TYPE_LONGLONG
        rem = S64Mod(fTime, S64Set(1000000000));
        return U32SetU(SInt64ToUInt64(S64Div(fTime, S64Set(1000000000))));
    #else
        return U32SetU(SInt64ToUInt64(S64Divide(fTime, S64Set(1000000000), &rem)));
    #endif
}
```

```
GUSITime::operator timeval()
{
    timeval tv;
    int64_t rem;

    tv.tv_sec = Deconstruct(rem);
    tv.tv_usec = S32Set(S64Div(rem, S64Set(1000)));

    return tv;
}
```

```
GUSITime::operator timespec()
{
    timespec ts;
    int64_t rem;

    ts.tv_sec = Deconstruct(rem);
    ts.tv_nsec = S32Set(rem);

    return ts;
}
```


We construct a struct `tm` via `LongSecondsToDate`.

```

(418) <424 426>
(⟨Member functions for class GUSITime 421⟩)+≡
GUSITime::operator tm()
{
    tm          t;
    LongDateRec ldr;
    LongDateTime ldt;

    ldt = Get64(seconds);
    LongSecondsToDate(&ldt, &ldr);
    t.tm_sec  = ldr.ld.second;
    t.tm_min  = ldr.ld.minute;
    t.tm_hour = ldr.ld.hour;
    t.tm_mday = ldr.ld.day;
    t.tm_mon  = ldr.ld.month-1;
    t.tm_year = ldr.ld.year-1900;
    t.tm_wday = ldr.ld.dayOfWeek-1;
    t.tm_yday = ldr.ld.dayOfYear-1;
    t.tm_isdst = -1;

    return t;
}

```

Constructing a `GUSITime` from a struct `tm` is a bit trickier because field values are not guaranteed to be in range and `LongDateToSeconds` is not infinitely tolerant.

```

(418) <425 427>
(⟨Member functions for class GUSITime 421⟩)+≡
GUSITime::GUSITime(const tm & t)
{
    LongDateRec ldr;
    LongDateTime ldt;

    ldr.ld.era    = 0;
    ldr.ld.year   = t.tm_year+1900;
    ldr.ld.month  = t.tm_mon+1;
    ldr.ld.day    = t.tm_mday ? t.tm_mday : 1;
    ldr.ld.hour   = t.tm_hour>=0 ? t.tm_hour : 0;
    ldr.ld.minute = t.tm_min>=0 ? t.tm_min : 0;
    ldr.ld.second = t.tm_sec>=0 ? t.tm_sec : 0;

    LongDateToSeconds(&ldr, &ldt);
    if (!t.tm_mday)
        ldt = S64Subtract(ldt, S64Set(86400));
    if (t.tm_hour < 0)
        ldt = S64Add(ldt, S64Multiply(S64Set(t.tm_hour), S64Set(3600)));
    if (t.tm_min < 0)
        ldt = S64Add(ldt, S64Multiply(S64Set(t.tm_min), S64Set(60)));
    if (t.tm_sec < 0)
        ldt = S64Add(ldt, S64Set(t.tm_sec));

    Construct(ldt, seconds);
}

```

To calculate the current time, we need to combine `GetDateTime`, which gives seconds since 1904, with `Microseconds`, which gives microseconds since startup.

```

<Member functions for class GUSITime 421>+≡ (418) <426 428>
int64_t      GUSITime::sTimeOffset;
timezone     GUSITime::sTimeZone;

GUSITime GUSITime::Now()
{
    if (S64Not(sTimeOffset)) {
        GUSITime    s;
        GUSITime    zone;
        GUSITime    us;

        {
            uint32_t    secs;
            GetDateTime(&secs);
            s = GUSITime(secs, seconds);
        }
        {
            UnsignedWide    u;
            Microseconds(&u);
            us = GUSITime(UINT64_TO_SINT64(UnsignedWideToUINT64(u)), usecs);
        }
        {
            MachineLocation loc;
            ReadLocation(&loc);
            int32_t delta = loc.u.gmtDelta & 0x00FFFFFF;
            if (delta & 0x00800000)
                delta |= 0xFF000000;
            zone = GUSITime(delta, seconds);
            sTimeZone.tz_minuteswest = zone.Get(seconds) / 60;
            sTimeZone.tz_dsttime     = (loc.u.gmtDelta & 0xFF000000) != 0;
        }
        s -= zone;
        sTimeOffset = s - us;

        return s;
    } else {
        UnsignedWide    us;
        Microseconds(&us);
        GUSITime    t(UINT64_TO_SINT64(UnsignedWideToUINT64(us)), usecs);

        return S64Add(int64_t(t), sTimeOffset);
    }
}

```

The time zone is cached by Now().

```
(Member functions for class GUSITime 421)+≡ (418) <427
    timezone & GUSITime::Zone()
    {
        Now();
        return sTimeZone;
    }

GUSITime GUSITime::GMTTime()
{
    GUSITime zone(sTimeZone.tz_minuteswest, seconds);

    return *this - zone;
}

GUSITime GUSITime::LocalTime()
{
    GUSITime zone(sTimeZone.tz_minuteswest, seconds);

    return *this+zone;
}
```

12.3 Implementation of GUSITimer

GUSITimerProc wakes up the context.

```
(Member functions for class GUSITimer 429)+≡ (418) 430▷
    static void GUSITimerProc(TMTask * tm)
    {
        GUSITimer * timer = static_cast<GUSITimer *>(tm);

        GUSIProcess::A5Saver    saveA5(timer->Context());

        timer->Wakeup();
    }

GUSI_COMPLETION_PROC_A1(GUSITimerProc, TMTask)
```

The default wakeup action simply wakes up the associated context.

```
(Member functions for class GUSITimer 429)+≡ (418) <429 431▷
    void GUSITimer::Wakeup()
    {
        Context()->Wakeup();
    }
```

When a process terminates, all remaining timers get destroyed.

```
(Member functions for class GUSITimer 429)+≡ (418) <430 432>
GUSITimer::TimerQueue::~~TimerQueue()
{
    GUSIContext::LiquidateAll();
}

void GUSIKillTimers(void * timers)
{
    GUSITimer::Queue * q = static_cast<GUSITimer::Queue *>(timers);

    while (q->fTimer)
        q->fTimer->Kill();

    delete q;
}
```

On constructing the first GUSITimer in a program, we initialize sTimerProc.

```
(Member functions for class GUSITimer 429)+≡ (418) <431 433>
TimerUPP          GUSITimer::sTimerProc = (TimerUPP)0;
GUSITimer::TimerQueue  GUSITimer::sTimerQueue;

GUSITimer::GUSITimer(bool wakeup, GUSIContext * context)
{
    if (!context)
        context = GUSIContext::Current();
    if (wakeup) {
        if (!sTimerProc)
            sTimerProc = NewTimerProc(GUSITimerProcEntry);
        tmAddr = sTimerProc;
    } else
        tmAddr = 0;
    tmCount = 0;
    tmWakeUp = 0;
    tmReserved = 0;
    InsXTime(Elem());
    fQueue = sTimerQueue.get(context);
    fQueue->fContext = context;
    fNext = fQueue->fTimer;
    fQueue->fTimer = this;
}
```

We support both normal timing and drift free timers.

```
(Member functions for class GUSITimer 429)+≡ (418) <432 434>
void GUSITimer::Sleep(long ms, bool driftFree)
{
    if (!driftFree)
        tmWakeUp = 0;
    PrimeTime(Elem(), ms);
}
```

Upon destruction, we remove the time manager task. The fQueue element serves as a flag so we don't get killed twice.

```
<Member functions for class GUSITimer 429>+≡                                (418) <433
void GUSITimer::Kill()
{
    if (!fQueue)    // We are dead already
        return;

    RmvTime(Elem());
    GUSITimer ** queue = &fQueue->fTimer;

    while (*queue)
        if (*queue == this) {
            *queue = fNext;

            break;
        } else
            queue = &(*queue)->fNext;
    fNext    = 0;
    fQueue   = 0;
}

GUSITimer::~GUSITimer()
{
    Kill();
}
```


Part II

POSIX Wrappers

Chapter 13

Mapping descriptors to sockets

POSIX routines do not, of course, operate on GUSISockets but on numerical descriptors. The GUSIDescriptorTable singleton maps between descriptors and their GUSISockets.

```
<GUSIDescriptor.h 435>≡
    #ifndef _GUSIDescriptor_
    #define _GUSIDescriptor_

    #ifdef GUSI_SOURCE

    #include "GUSISocket.h"

    <Definition of class GUSIDescriptorTable 437>

    <Hooks for ANSI library interfaces 441>

    <Inline member functions for class GUSIDescriptorTable 452>

    #endif /* GUSI_SOURCE */

    #endif /* _GUSIDescriptor_ */

<GUSIDescriptor.cp 436>≡
    #include "GUSIInternal.h"
    #include "GUSIDescriptor.h"
    #include "GUSIBasics.h"
    #include "GUSIDiag.h"
    #include "GUSINull.h"

    #include <errno.h>
    #include <fcntl.h>
    #include <utility>
    #include <memory>

    GUSI_USING_STD_NAMESPACE

    <Member functions for class GUSIDescriptorTable 438>
```

13.1 Definition of GUSIDescriptorTable

A GUSIDescriptorTable is another singleton class, behaving in many aspects like an array of GUSISocket pointers. InstallSocket installs a new socket into the table, picking the first available slot with a descriptor greater than or equal to start. RemoveSocket empties one slot. GUSIDescriptorTable::LookupSocket is a shorthand for (*GUSIDescriptorTable::Instance())[fd]

<Definition of class GUSIDescriptorTable 437>≡ (435)

```
class GUSIDescriptorTable {
public:
    enum { SIZE = 64 };

    static GUSIDescriptorTable * Instance();

    int          InstallSocket(GUSISocket * sock, int start = 0);
    int          RemoveSocket(int fd);
    GUSISocket * operator[](int fd);
    static GUSISocket * LookupSocket(int fd);

    class iterator;
    friend class iterator;

    iterator &      begin();
    iterator &      end();

    ~GUSIDescriptorTable();
private:
    <Privatissima of GUSIDescriptorTable 439>
};
```

13.2 Implementation of GUSIDescriptorTable

To make sure that the instance gets destructed if it ever gets constructed, we keep it in a static auto_ptr. Am I eleet or what?

<Member functions for class GUSIDescriptorTable 438>≡ (436) 440»

```
static auto_ptr<GUSIDescriptorTable> sGUSIDescriptorTable;
```

On creation, a GUSIDescriptorTable clears all descriptors.

<Privatissima of GUSIDescriptorTable 439>≡ (437)

```
GUSISocket * fSocket[SIZE];
int          fInvalidDescriptor;
GUSIDescriptorTable();
```

<Member functions for class GUSIDescriptorTable 438>+≡ (436) «438 442»

```
GUSIDescriptorTable::GUSIDescriptorTable()
: fInvalidDescriptor(0)
{
}
```

If no instance exists yet, `GUSIDescriptorTable::Instance` creates one and calls `GUSISetupConsole` if the `setupConsole` parameter is true. `GUSISetupConsole` calls `GUSIDefaultSetupConsole`, which first calls `GUSISetupConsoleDescriptors` to set up file descriptors 0, 1, and 2, and then calls `GUSISetupConsoleStdio` to deal with the necessary initializations on the stdio level.

<Hooks for ANSI library interfaces 441>≡ (435) 446▷

```
extern "C" {
    void GUSISetupConsole();
    void GUSIDefaultSetupConsole();
    void GUSISetupConsoleDescriptors();
    void GUSISetupConsoleStdio();
}
```

<Member functions for class GUSIDescriptorTable 438>+≡ (436) ◀440 447▷

<Default implementation of GUSISetupConsole 443>

```
GUSIDescriptorTable * GUSIDescriptorTable::Instance()
{
    bool sNeedConsoleSetup = true;

    if (!sGUSIDescriptorTable.get()) {
        sGUSIDescriptorTable = auto_ptr<GUSIDescriptorTable>(new GUSIDescriptorTable());
        if (sNeedConsoleSetup) {
            sNeedConsoleSetup = false;
            GUSISetupConsole();
        }
    }
    return sGUSIDescriptorTable.get();
}
```

Application programmers may elect to override `GUSISetupConsole` to do their own initializations (notably to add further socket factories and devices), but if they chose to do so, they generally should include a call to `GUSIDefaultSetupConsole` in their version.

<Default implementation of GUSISetupConsole 443>≡ (442) 444▷

```
void GUSISetupConsole()
{
    GUSIDefaultSetupConsole();
}
```

The file descriptor part and the stdio part of the initialization functionality are usually handled in different GUSI sublibraries, therefore we separate them into different hooks. There never should be a reason to override `GUSIDefaultSetupConsole` itself.

<Default implementation of GUSISetupConsole 443>+≡ (442) ◀443 445▷

```
void GUSIDefaultSetupConsole()
{
    GUSISetupConsoleDescriptors();
    GUSISetupConsoleStdio();
}
```

It may be necessary to override this function (notably for MPW support), but this version covers the basic case. To avoid any dependence on Null devices being installed in the device table, we call the instance directly.

```

<Default implementation of GUSISetupConsole 443>+≡ (442) <444
void GUSISetupConsoleDescriptors()
{
    GUSIDescriptorTable * table = GUSIDescriptorTable::Instance();
    GUSINullDevice * null = GUSINullDevice::Instance();

    if (open("dev:console", O_RDONLY) < 0)
        table->InstallSocket(null->open());
    if (open("dev:console", O_WRONLY) < 0)
        table->InstallSocket(null->open());
    if (open("dev:console", O_WRONLY) < 0)
        table->InstallSocket(null->open());
}

```

Destructing a GUSIDescriptorTable may be a bit problematic, as this may have effects reaching up into the stdio layer. We therefore factor out the stdio aspects into the procedures StdioClose and StdioFlush which we then can redefine in other, stdio library specific, libraries.

```

<Hooks for ANSI library interfaces 441>+≡ (435) <441
extern "C" {
void GUSIStdioClose();
void GUSIStdioFlush();
}

```

```

<Member functions for class GUSIDescriptorTable 438>+≡ (436) <442 448>
void GUSIStdioClose() { }
void GUSIStdioFlush() { }
void GUSISetupConsoleStdio() { }

GUSIDescriptorTable::~GUSIDescriptorTable()
{
    GUSIStdioFlush();
    GUSIStdioClose();

    for (iterator i = begin(); i != end(); ++i)
        RemoveSocket(*i);
}

```

To keep the range of descriptor slots to search as small as possible, only descriptors smaller than `fInvalidDescriptor` are potentially valid.

```

⟨Member functions for class GUSIDescriptorTable 438⟩+≡ (436) <447 449>
int GUSIDescriptorTable::InstallSocket(GUSISocket * sock, int start)
{
    if (start<0 || start >= SIZE)
        return GUSISetPosixError(EINVAL);

    while (start<fInvalidDescriptor)
        if (!fSocket[start])
            goto found;
    else
        ++start;
    while (start > fInvalidDescriptor)
        fSocket[fInvalidDescriptor++] = static_cast<GUSISocket *>(nil);
    if (start < SIZE)
        ++fInvalidDescriptor;
    else
        return GUSISetPosixError(EMFILE);

found:
    fSocket[start] = sock;

    sock->AddReference();

    return start;
}

⟨Member functions for class GUSIDescriptorTable 438⟩+≡ (436) <448 450>
int GUSIDescriptorTable::RemoveSocket(int fd)
{
    GUSISocket * sock;

    if (fd<0 || fd >= fInvalidDescriptor || !(sock = fSocket[fd]))
        return GUSISetPosixError(EBADF);

    fSocket[fd] = nil;

    sock->RemoveReference();

    return 0;
}

⟨Member functions for class GUSIDescriptorTable 438⟩+≡ (436) <449 451>
GUSISocket * GUSIDescriptorTable::operator[](int fd)
{
    GUSISocket * sock;

    if (fd<0 || fd >= fInvalidDescriptor || !(sock = fSocket[fd]))
        return GUSISetPosixError(EBADF), static_cast<GUSISocket *>(nil);
    else
        return sock;
}

```

```

<Member functions for class GUSIDescriptorTable 438>+≡ (436) 4450
GUSISocket * GUSIDescriptorTable::LookupSocket(int fd)
{
    GUSIDescriptorTable * table = Instance();
    GUSISocket * sock;

    if (fd<0 || fd >= table->fInvalidDescriptor || !(sock = table->fSocket[fd]))
        return GUSISetPosixError(EBADF), static_cast<GUSISocket *>(nil);
    else
        return sock;
}

```

```

(Inline member functions for class GUSIDescriptorTable 452)≡ (435)
class GUSIDescriptorTable::iterator {
public:
    iterator(GUSIDescriptorTable * table, int fd = 0) : fTable(table), fFd(fd) {}
    GUSIDescriptorTable::iterator & operator++();
    GUSIDescriptorTable::iterator operator++(int);
    int operator*() { return fFd; }
    bool operator==(const GUSIDescriptorTable::iterator & other) const;
private:
    GUSIDescriptorTable * fTable;
    int fFd;
};

inline GUSIDescriptorTable::iterator & GUSIDescriptorTable::iterator::operator++()
{
    while (++fFd < fTable->fInvalidDescriptor && !fTable->fSocket[fFd])
        ;

    return *this;
}

inline GUSIDescriptorTable::iterator GUSIDescriptorTable::iterator::operator++(int)
{
    int oldFD = fFd;

    while (++fFd < fTable->fInvalidDescriptor && !fTable->fSocket[fFd])
        ;

    return GUSIDescriptorTable::iterator(fTable, oldFD);
}

inline bool GUSIDescriptorTable::iterator::operator==(const GUSIDescriptorTable::iterator & other) const
{
    return fFd == other.fFd;
}

inline GUSIDescriptorTable::iterator & GUSIDescriptorTable::begin()
{
    return iterator(this);
}

inline GUSIDescriptorTable::iterator & GUSIDescriptorTable::end()
{
    return iterator(this, fInvalidDescriptor);
}

```


Chapter 14

POSIX/Socket Wrappers

Now everything is in place to define the POSIX and socket routines themselves. As opposed to our usual practice, we don't declare the exported routines here, as they all have been declared in `unistd.h` or `sys/socket.h` already. The one exception is `remove`, which is declared in `stdio.h`, which we'd rather not include.

```
<GUSIPOSIX.h 453>≡
#ifndef _GUSIPOSIX_
#define _GUSIPOSIX_

#include <signal.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <utime.h>
#include <netdb.h>
#include <arpa/inet.h>

__BEGIN_DECLS
int remove(const char * path);
__END_DECLS

#endif /* _GUSIPOSIX_ */
```

We try to distinguish between functions mentioned in the POSIX.1 standard, BSD functions which later became part of UNIX 98, and a few MPW specific functions.

```

<GUSIPOSIX.cp 454>≡
#include "GUSIInternal.h"
#include "GUSIPOSIX.h"
#include "GUSISocket.h"
#include "GUSIFactory.h"
#include "GUSIDevice.h"
#include "GUSIDescriptor.h"
#include "GUSIPipe.h"
#include "GUSIConfig.h"
#include "GUSIBasics.h"
#include "GUSIDiag.h"
#include "GUSINetDB.h"
#include "GUSITimer.h"

#include <LowMem.h>

<POSIX function wrappers 455>
<Socket function wrappers 487>
<MPW function wrappers 538>

```

14.1 Implementation of POSIX wrappers

pipe is in fact a special case of socketpair, but we bypass the domain registry because pipe sockets are not installed as a domain.

```

<POSIX function wrappers 455>≡ (454) 456>
int pipe(int * fd)
{
    GUSIErrorSaver          saveError;
    GUSISocketFactory *     factory = GUSIPipeFactory::Instance();
    GUSIDescriptorTable *   table   = GUSIDescriptorTable::Instance();
    GUSISocket *            sock[2];

    if (!factory->socketpair(0, 0, 0, sock)) {
        if ((fd[0] = table->InstallSocket(sock[0])) > -1)
            if ((fd[1] = table->InstallSocket(sock[1])) > -1) {
                shutdown(fd[0], 1);
                shutdown(fd[1], 0);

                return 0;
            } else
                table->RemoveSocket(fd[0]);
        delete sock[0];
        delete sock[1];
    }
    if (!errno)
        return GUSISetPosixError(ENOMEM);
    else
        return -1;
}

```

fsync synchronizes a socket

```
⟨POSIX function wrappers 455⟩+≡ (454) ◀455 457▶  
int fsync(int s)  
{  
    ⟨Translate descriptor s to GUSISocket * sock or return -1 458⟩  
  
    return sock->fsync();  
}
```

close closes one file descriptor associated with a socket and deletes the socket if this was the last descriptor.

```
⟨POSIX function wrappers 455⟩+≡ (454) ◀456 459▶  
int close(int s)  
{  
    GUSIDescriptorTable * table = GUSIDescriptorTable::Instance();  
  
    return table->RemoveSocket(s);  
}
```

Many of the other routines are quite stereotypical: They translate their file descriptor argument to a GUSISocket and dispatch the call to it.

```
⟨Translate descriptor s to GUSISocket * sock or return -1 458⟩≡ (456 459–65 490–501 511–16)  
GUSISocket * sock = GUSIDescriptorTable::LookupSocket(s);  
  
if (!sock)  
    return -1;  
  
⟨POSIX function wrappers 455⟩+≡ (454) ◀457 460▶  
ssize_t read(int s, void *buffer, size_t buflen)  
{  
    ⟨Translate descriptor s to GUSISocket * sock or return -1 458⟩  
  
    return sock->read(buffer, buflen);  
}
```

<POSIX function wrappers 455>+≡ (454) <459 461>

```
static ssize_t HandleWriteErrors(ssize_t retval)
{
    if (retval == -1)
        switch (errno) {
            case EINTR:
            case EWOULDBLOCK:
            case EINPROGRESS:
            case EALREADY:
                break;
            default:
                GUSIConfiguration::Instance()->BrokenPipe();
                break;
        }

    return retval;
}
```

```
ssize_t write(int s, const void *buffer, size_t buflen)
{
    <Translate descriptor s to GUSISocket * sock or return -1 458>

    return HandleWriteErrors(sock->write(buffer, buflen));
}
```

All requests to fcntl except for F_DUPFD get dispatched to the socket.

<POSIX function wrappers 455>+≡ (454) <460 462>

```
int fcntl(int s, int cmd, ...)
{
    va_list arglist;
    va_start(arglist, cmd);

    <Translate descriptor s to GUSISocket * sock or return -1 458>

    if (cmd == F_DUPFD) {
        GUSIDescriptorTable * table = GUSIDescriptorTable::Instance();

        return table->InstallSocket(sock, va_arg(arglist, int));
    } else
        return sock->fcntl(cmd, arglist);
}
```

dup and dup2 are slight variations of the preceding.

```
⟨POSIX function wrappers 455⟩+≡ (454) ◀461 463▶
int dup(int s)
{
    GUSIDescriptorTable * table = GUSIDescriptorTable::Instance();

    ⟨Translate descriptor s to GUSISocket * sock or return -1 458⟩

    return table->InstallSocket(sock);
}

int dup2(int s, int s1)
{
    GUSIDescriptorTable * table = GUSIDescriptorTable::Instance();

    ⟨Translate descriptor s to GUSISocket * sock or return -1 458⟩

    table->RemoveSocket(s1);
    return table->InstallSocket(sock, s1);
}
```

```
⟨POSIX function wrappers 455⟩+≡ (454) ◀462 464▶
int fstat(int s, struct stat * buf)
{
    ⟨Translate descriptor s to GUSISocket * sock or return -1 458⟩

    return sock->fstat(buf);
}
```

MacOS soon will work with 64 bit file offsets, so off_t is defined as a 64 bit integer.

```
⟨POSIX function wrappers 455⟩+≡ (454) ◀463 465▶
off_t lseek(int s, off_t offset, int whence)
{
    ⟨Translate descriptor s to GUSISocket * sock or return -1 458⟩

    return sock->lseek(offset, whence);
}
```

We also should define ttyname but that will have to wait.

```
⟨POSIX function wrappers 455⟩+≡ (454) ◀464 466▶
int isatty(int s)
{
    ⟨Translate descriptor s to GUSISocket * sock or return -1 458⟩

    return sock->isatty();
}
```

Sleeping is simple, computing the rest time is not. For convenience, we also special case sleep(0) as a yield.

```
(POSIX function wrappers 455)+≡ (454) <465 467>
u_int sleep(u_int seconds)
{
    GUSIContext::Current()->ClearWakeups();

    if (!seconds) {
        GUSIContext::Yield(false);

        return 0;
    }
    GUSITime start(GUSITime::Now());
    GUSITimer * timer = new GUSITimer;
    timer->Sleep(GUSITime(seconds, GUSITime::seconds).Get(GUSITime::msecs));
    GUSIContext::Yield(true);
    delete timer;

    return (GUSITime::Now() - start).Get(GUSITime::seconds);
}
```

The next calls also need the services of GUSIDeviceRegistry. open used both the GUSIDeviceRegistry and the GUSIDescriptorTable.

```
(POSIX function wrappers 455)+≡ (454) <466 468>
int open(const char * path, int mode, ...)
{
    GUSIErrorSaver          saveError;
    GUSIDeviceRegistry *    factory = GUSIDeviceRegistry::Instance();
    GUSIDescriptorTable *   table  = GUSIDescriptorTable::Instance();
    GUSISocket *            sock;
    int                     fd;

    if (sock = factory->open(path, mode)) {
        if ((fd = table->InstallSocket(sock)) > -1)
            return fd;
        delete sock;
    }
    if (!errno)
        return GUSISetPosixError(ENOMEM);
    else
        return -1;
}
```

creat is just a special case for open.

```
(POSIX function wrappers 455)+≡ (454) <467 469>
int creat(const char * path, ...)
{
    return open(path, O_WRONLY | O_TRUNC | O_CREAT);
}
```

remove remove a file.

```
(POSIX function wrappers 455)+≡ (454) ◀468 470▶  
int remove(const char * path)  
{  
    GUSIDeviceRegistry *    factory = GUSIDeviceRegistry::Instance();  
  
    return factory->remove(path);  
}
```

unlink is the same as remove.

```
(POSIX function wrappers 455)+≡ (454) ◀469 471▶  
int unlink(const char * path)  
{  
    GUSIDeviceRegistry *    factory = GUSIDeviceRegistry::Instance();  
  
    return factory->remove(path);  
}
```

rename renames a file.

```
(POSIX function wrappers 455)+≡ (454) ◀470 472▶  
int rename(const char * oldname, const char * newname)  
{  
    GUSIDeviceRegistry *    factory = GUSIDeviceRegistry::Instance();  
  
    return factory->rename(oldname, newname);  
}
```

stat and lstat, which ultimately resolve to the same function, return information about the status of a file.

```
(POSIX function wrappers 455)+≡ (454) ◀471 473▶  
int stat(const char * path, struct stat * buf)  
{  
    GUSIDeviceRegistry *    factory = GUSIDeviceRegistry::Instance();  
  
    return factory->stat(path, buf, false);  
}  
  
int lstat(const char * path, struct stat * buf)  
{  
    GUSIDeviceRegistry *    factory = GUSIDeviceRegistry::Instance();  
  
    return factory->stat(path, buf, true);  
}
```

chmod changes file protection flags as well as this is possible under MacOS.

```
(POSIX function wrappers 455)+≡ (454) ◀472 474▶  
int chmod(const char * path, mode_t mode)  
{  
    GUSIDeviceRegistry *    factory = GUSIDeviceRegistry::Instance();  
  
    return factory->chmod(path, mode);  
}
```

utime modifies a file's modification time.

```
(POSIX function wrappers 455)+≡ (454) <473 475>
int utime(const char * path, const utimbuf * times)
{
    GUSIDeviceRegistry *    factory = GUSIDeviceRegistry::Instance();

    return factory->utime(path, times);
}
```

access checks the access permissions for a file.

```
(POSIX function wrappers 455)+≡ (454) <474 476>
int access(const char * path, int mode)
{
    GUSIDeviceRegistry *    factory = GUSIDeviceRegistry::Instance();

    return factory->access(path, mode);
}
```

mkdir and rmdir create and delete a directory, respectively.

```
(POSIX function wrappers 455)+≡ (454) <475 477>
int mkdir(const char * path, ...)
{
    GUSIDeviceRegistry *    factory = GUSIDeviceRegistry::Instance();

    return factory->mkdir(path);
}

int rmdir(const char * path)
{
    GUSIDeviceRegistry *    factory = GUSIDeviceRegistry::Instance();

    return factory->rmdir(path);
}
```

opendir returns an instance of a derived class of GUSIDirectory.

```
(POSIX function wrappers 455)+≡ (454) <476 478>
typedef GUSIDirectory * GUSIDirPtr;

DIR * opendir(const char * path)
{
    GUSIDeviceRegistry *    factory = GUSIDeviceRegistry::Instance();

    GUSIDirectory * dir = factory->opendir(path);

    return dir ? reinterpret_cast<DIR *>(new GUSIDirPtr(dir)) : 0;
}
```

The other directory functions then dispatch on the GUSIDirectory without needing the GUSIDeviceRegistry. readdir reads the next directory entry.

```
(POSIX function wrappers 455)+≡ (454) <477 479>
dirent * readdir(DIR * dir)
{
    return GUSIDirPtr(*dir)->readdir();
}
```


telldir saves the current directory position, seekdir sets it, rewinddir sets it to the beginning.

```
<POSIX function wrappers 455>+≡ (454) <478 480>
long telldir(DIR * dir)
{
    return GUSIDirPtr(*dir)->telldir();
}
```

```
void seekdir(DIR * dir, long pos)
{
    GUSIDirPtr(*dir)->seekdir(pos);
}
```

```
void rewinddir(DIR * dir)
{
    GUSIDirPtr(*dir)->seekdir(1);
}
```

closedir closes the directory stream.

```
<POSIX function wrappers 455>+≡ (454) <479 481>
int closedir(DIR * dir)
{
    delete GUSIDirPtr(*dir);
    delete dir;

    return 0;
}
```

chdir changes the default directory.

```
<POSIX function wrappers 455>+≡ (454) <480 482>
int chdir(const char * dir)
{
    GUSIFileSpec    directory(dir);

    if (!directory.Error())
        (++directory).SetDefaultDirectory();

    return GUSISetMacError(directory.Error());
}
```

getcwd returns the path to the default directory.

(POSIX function wrappers 455)+≡ (454) <481 483>

```
char *getcwd(char * buf, size_t size)
{
    GUSIFileSpec    cwd;
    char *          res;

    if (cwd.GetDefaultDirectory())
        return GUSISetMacError(cwd.Error()), static_cast<char *>(nil);

    res = (--cwd).FullPath();

    if (size < strlen(res)+1)
        return GUSISetPosixError(size > 0 ? ERANGE : EINVAL),
            static_cast<char *>(nil);
    if (!buf && !(buf = (char *) malloc(size)))
        return GUSISetPosixError(ENOMEM), static_cast<char *>(nil);

    strcpy(buf, res);

    return buf;
}
```

time returns the time in seconds since 1904. The MSL version uses 1970; the present version conforms more to my Mac traditionalistic outlook.

(POSIX function wrappers 455)+≡ (454) <482 484>

```
time_t time(time_t * timer)
{
    time_t t;

    if (!timer)
        timer = &t;

    GetDateTime(&t);

    return t;
}
```

gettimeofday returns a somewhat more accurate time than time.

(POSIX function wrappers 455)+≡ (454) <483 485>

```
int gettimeofday(struct timeval * tv, struct timezone * tz)
{
    *tv = (timeval)GUSITime::Now();
    *tz = GUSITime::Zone();

    return 0;
}
```

localtime returns the local time in a broken down record, gmtime does the same with UTC time. Our versions measure from 1904 and are thread safe.

```

<POSIX function wrappers 455>+≡ (454) <484 486>
extern "C" void GUSIKillTM(void * t)
{
    delete reinterpret_cast<tm *>(t);
}

static tm * get_tm()
{
    static GUSISpecificData<tm, GUSIKillTM> sTM;

    return sTM.get();
}

struct tm * localtime(const time_t * timer)
{
    tm * t = get_tm();
    *t = GUSITime(*timer, GUSITime::seconds);
    t->tm_isdst = GUSITime::Zone().tz_dsttime != 0;
    return t;
}

struct tm * gmtime(const time_t * timer)
{
    tm * t = get_tm();
    *t = GUSITime(*timer, GUSITime::seconds).GMTTime();
    t->tm_isdst = 0;
    return t;
}

mktime parses a struct tm.
<POSIX function wrappers 455>+≡ (454) <485
time_t mktime(struct tm *timeptr)
{
    return GUSITime(*timeptr).UGet(GUSITime::seconds);
}

```

14.2 Implementation of Socket wrappers

getdtablesize returns the size of the descriptor table.

```

<Socket function wrappers 487>≡ (454) 488>
int getdtablesize()
{
    return GUSIDescriptorTable::SIZE;
}

```

socket creates a socket and installs it in the descriptor table.

```
(Socket function wrappers 487)+≡ (454) <487 489>
int socket(int domain, int type, int protocol)
{
    GUSIErrorSaver      saveError;
    GUSISocketFactory *  factory = GUSISocketDomainRegistry::Instance();
    GUSIDescriptorTable * table  = GUSIDescriptorTable::Instance();
    GUSISocket *         sock;
    int                  fd;

    if (sock = factory->socket(domain, type, protocol)) {
        if ((fd = table->InstallSocket(sock)) > -1)
            return fd;
        delete sock;
    }
    if (!errno)
        return GUSISetPosixError(ENOMEM);
    else
        return -1;
}
```

socketpair works similar to socket, although the opportunity for problems is a bit greater.

```
(Socket function wrappers 487)+≡ (454) <488 490>
int socketpair(int domain, int type, int protocol, int * sv)
{
    GUSIErrorSaver      saveError;
    GUSISocketFactory *  factory = GUSISocketDomainRegistry::Instance();
    GUSIDescriptorTable * table  = GUSIDescriptorTable::Instance();
    GUSISocket *         sock[2];

    if (!factory->socketpair(domain, type, protocol, sock)) {
        if ((sv[0] = table->InstallSocket(sock[0])) > -1)
            if ((sv[1] = table->InstallSocket(sock[1])) > -1)
                return 0;
        else
            table->RemoveSocket(sv[0]);
        delete sock[0];
        delete sock[1];
    }
    if (!errno)
        return GUSISetPosixError(ENOMEM);
    else
        return -1;
}
```

bind binds a socket to a name.

```
(Socket function wrappers 487)+≡ (454) <489 491>
int bind(int s, const struct sockaddr *name, socklen_t namelen)
{
    {Translate descriptor s to GUSISocket * sock or return -1 458}
    return sock->bind((void *) name, namelen);
}
```

connect connects a socket to a named peer.

```
(Socket function wrappers 487)+≡ (454) ◀490 492▶  
int connect(int s, const struct sockaddr *addr, socklen_t addrlen)  
{  
    ◁Translate descriptor s to GUSISocket * sock or return -1 458▷  
    return sock->connect((void *) addr, addrlen);  
}
```

listen puts a socket in passive mode.

```
(Socket function wrappers 487)+≡ (454) ◀491 493▶  
int listen(int s, int qlen)  
{  
    ◁Translate descriptor s to GUSISocket * sock or return -1 458▷  
    return sock->listen(qlen);  
}
```

accept accepts a connection from another socket.

```
(Socket function wrappers 487)+≡ (454) ◀492 494▶  
int accept(int s, struct sockaddr *addr, socklen_t *addrlen)  
{  
    GUSIDescriptorTable * table = GUSIDescriptorTable::Instance();  
  
    ◁Translate descriptor s to GUSISocket * sock or return -1 458▷  
  
    if (sock = sock->accept(addr, addrlen))  
        if ((s = table->InstallSocket(sock)) != -1)  
            return s;  
    else  
        delete sock;  
  
    return -1;  
}
```

```
(Socket function wrappers 487)+≡ (454) ◀493 495▶  
ssize_t readv(int s, const struct iovec *iov, int count)  
{  
    ◁Translate descriptor s to GUSISocket * sock or return -1 458▷  
  
    return sock->read(GUSIScatterer(iov, count));  
}
```

```
(Socket function wrappers 487)+≡ (454) ◀494 496▶  
ssize_t recv(int s, void *buffer, size_t buflen, int flags)  
{  
    ◁Translate descriptor s to GUSISocket * sock or return -1 458▷  
  
    socklen_t fromlen = 0;  
  
    return sock->recvfrom(buffer, buflen, flags, nil, &fromlen);  
}
```

```

(Socket function wrappers 487)+≡ (454) <495 497>
    ssize_t recvfrom(int s, void *buffer, size_t buflen, int flags, struct sockaddr *from, socklen_t *f
    {
        <Translate descriptor s to GUSISocket * sock or return -1 458>

        return sock->recvfrom(buffer, buflen, flags, from, fromlen);
    }

(Socket function wrappers 487)+≡ (454) <496 498>
    ssize_t recvmsg(int s, struct msghdr *msg, int flags)
    {
        <Translate descriptor s to GUSISocket * sock or return -1 458>

        return sock->recvmsg(msg, flags);
    }

(Socket function wrappers 487)+≡ (454) <497 499>
    ssize_t writev(int s, const struct iovec *iov, int count)
    {
        <Translate descriptor s to GUSISocket * sock or return -1 458>

        return HandleWriteErrors(sock->write(GUSIGatherer(iov, count)));
    }

(Socket function wrappers 487)+≡ (454) <498 500>
    ssize_t send(int s, const void *buffer, size_t buflen, int flags)
    {
        <Translate descriptor s to GUSISocket * sock or return -1 458>

        return HandleWriteErrors(sock->sendto(buffer, buflen, flags, nil, 0));
    }

(Socket function wrappers 487)+≡ (454) <499 501>
    ssize_t sendto(int s, const void *buffer, size_t buflen, int flags, const struct sockaddr *to, sock
    {
        <Translate descriptor s to GUSISocket * sock or return -1 458>

        return HandleWriteErrors(sock->sendto(buffer, buflen, flags, to, tolen));
    }

(Socket function wrappers 487)+≡ (454) <500 502>
    ssize_t sendmsg(int s, const struct msghdr *msg, int flags)
    {
        <Translate descriptor s to GUSISocket * sock or return -1 458>

        return HandleWriteErrors(sock->sendmsg(msg, flags));
    }

```

select is quite complex, so we break it up. select_once polls all file descriptors once.

```
(Socket function wrappers 487)+≡ (454) ◀501 503▶
static int select_once(int width,
    fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
    fd_set *readres, fd_set *writeres, fd_set *exceptres)
{
    bool    r, w, e;
    bool *  canRead;
    bool *  canWrite;
    bool *  canExcept;
    int     count = 0;

    for (int s = 0; s < width ; ++s) {
        canRead = (readfds && FD_ISSET(s,readfds)) ? &r : nil;
        canWrite = (writefds && FD_ISSET(s,writefds)) ? &w : nil;
        canExcept = (exceptfds && FD_ISSET(s,exceptfds)) ? &e : nil;
        if (canRead || canWrite || canExcept) {
            GUSISocket * sock = GUSIDescriptorTable::LookupSocket(s);

            if (!GUSI_ASSERT_EXTERNAL(sock, ("Socket %d closed in select\n", s)))
                return count ? count : -1;

            r = w = e = false;
            if (sock->select(canRead, canWrite, canExcept))
                ++count;
            if (r)
                FD_SET(s,readres);
            if (w)
                FD_SET(s,writeres);
            if (e)
                FD_SET(s,exceptres);
        }
    }

    return count;
}
```

select_sleep sleeps as long as canSleep allows.

```
(Socket function wrappers 487)+≡ (454) ◀502 504▶
static bool select_sleep(bool canSleep)
{
    if (canSleep) {
        return GUSIContext::Yield(true);
    } else {
        GUSITimer    sleepTimer;
        sleepTimer.Sleep(200);
        return GUSIContext::Yield(true);
    }
}
```

select_forever keeps calling select_once until one of the file descriptors triggers.

⟨Socket function wrappers 487⟩+≡ (454) ◀503 505▶

```
static int select_forever(bool canSleep, int width,
    fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
    fd_set *readres, fd_set *writeres, fd_set *exceptres)
{
    int count;

    for (;;) {
        count =
            select_once(width,
                readfds, writefds, exceptfds,
                readres, writeres, exceptres);
        if (count)
            break;
        if (select_sleep(canSleep))
            return GUSISetPosixError(EINTR);
    }

    return count;
}
```

select_timed keeps calling select_once until one of the file descriptors triggers or the timer runs out.

⟨Socket function wrappers 487⟩+≡ (454) ◀504 506▶

```
static int select_timed(bool canSleep, int width,
    fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
    fd_set *readres, fd_set *writeres, fd_set *exceptres,
    struct timeval *timeout)
{
    int count;
    GUSITimer timer;

    timer.MicroSleep(GUSITime(*timeout).Get(GUSITime::usecs));
    for (;;) {
        count =
            select_once(width,
                readfds, writefds, exceptfds,
                readres, writeres, exceptres);
        if (count || !timer.Primed())
            break;
        if (select_sleep(canSleep))
            return GUSISetPosixError(EINTR);
    }

    return count;
}
```


Even so, select is still a heavyweight.

```
<Socket function wrappers 487>+≡                                     (454) 4505 511▷
int select(int width, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)
{
    bool        canSleep;
    int         count = 0;
    fd_set      readres;    FD_ZERO(&readres);
    fd_set      writeres;   FD_ZERO(&writeres);
    fd_set      exceptres;  FD_ZERO(&exceptres);

    <Check that all specified file descriptors are valid or return -1 507>
    <Call pre_select for all file descriptors and determine canSleep 508>
    if (!timeout) {
        count =
            select_forever(canSleep, width,
                           readfds, writefds, exceptfds,
                           &readres, &writeres, &exceptres);
    } else if (timeout->tv_sec || timeout->tv_usec) {
        count =
            select_timed(canSleep, width,
                         readfds, writefds, exceptfds,
                         &readres, &writeres, &exceptres,
                         timeout);
    } else {
        count =
            select_once(width,
                        readfds, writefds, exceptfds,
                        &readres, &writeres, &exceptres);
        GUSIContext::Yield(false);
    }
    <Call post_select for all file descriptors 509>
    <Copy internal descriptor sets to parameters 510>

    return count;
}

<Check that all specified file descriptors are valid or return -1 507>≡           (506)
for (int s = 0; s < width ; ++s)
    if ( (readfds && FD_ISSET(s,readfds))
        || (writefds && FD_ISSET(s,writefds))
        || (exceptfds && FD_ISSET(s,exceptfds))
    )
        if (!GUSIDescriptorTable::LookupSocket(s))
            return -1;

<Call pre_select for all file descriptors and determine canSleep 508>≡           (506)
for (int s = 0; s < width ; ++s)
    if (GUSISocket * sock = GUSIDescriptorTable::LookupSocket(s)) {
        bool r = readfds && FD_ISSET(s,readfds);
        bool w = writefds && FD_ISSET(s,writefds);
        bool e = exceptfds && FD_ISSET(s,exceptfds);

        if (r || w || e)
            canSleep = sock->pre_select(r, w, e) && canSleep;
    }
}
```

<Call post_select for all file descriptors 509>≡ (506)

```
for (int s = 0; s < width ; ++s)
    if (GUSISocket * sock = GUSIDescriptorTable::LookupSocket(s)) {
        bool r = readfds && FD_ISSET(s,readfds);
        bool w = writefds && FD_ISSET(s,writefds);
        bool e = exceptfds && FD_ISSET(s,exceptfds);

        if (r || w || e)
            sock->post_select(r, w, e);
    }
```

<Copy internal descriptor sets to parameters 510>≡ (506)

```
if (readfds)
    *readfds = readres;
if (writefds)
    *writefds = writeres;
if (exceptfds)
    *exceptfds = exceptres;
```

<Socket function wrappers 487>+≡ (454) <506 512>

```
int getsockname(int s, struct sockaddr *name, socklen_t *namelen)
{
    <Translate descriptor s to GUSISocket * sock or return -1 458>

    return sock->getsockname(name, namelen);
}

int getpeername(int s, struct sockaddr *name, socklen_t *namelen)
{
    <Translate descriptor s to GUSISocket * sock or return -1 458>

    return sock->getpeername(name, namelen);
}
```

<Socket function wrappers 487>+≡ (454) <511 513>

```
int shutdown(int s, int how)
{
    <Translate descriptor s to GUSISocket * sock or return -1 458>

    return sock->shutdown(how);
}
```

Bit of trivia: ioctl is actually not a POSIX function.

<Socket function wrappers 487>+≡ (454) <512 514>

```
int ioctl(int s, unsigned long request, ...)
{
    va_list arglist;
    va_start(arglist, request);

    <Translate descriptor s to GUSISocket * sock or return -1 458>

    return sock->ioctl(request, arglist);
}
```

The length argument should eventually become a socklen_t.

```
<Socket function wrappers 487>+≡ (454) <513 515>
int getsockopt(int s, int level, int optname, void *optval, socklen_t * optlen)
{
    <Translate descriptor s to GUSISocket * sock or return -1 458>
```

```
    return sock->getsockopt(level, optname, optval, optlen);
}
```

```
<Socket function wrappers 487>+≡ (454) <514 516>
int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen)
{
    <Translate descriptor s to GUSISocket * sock or return -1 458>
```

```
    return sock->setsockopt(level, optname, (void *) optval, optlen);
}
```

```
<Socket function wrappers 487>+≡ (454) <515 517>
int ftruncate(int s, off_t offset)
{
    <Translate descriptor s to GUSISocket * sock or return -1 458>
```

```
    return sock ? sock->ftruncate(offset) : -1;
}
```

truncate is implemented in terms of ftruncate.

```
<Socket function wrappers 487>+≡ (454) <516 518>
int truncate(const char * path, off_t offset)
{
    int fd = open(path, O_RDONLY);
    if (fd < 0)
        return fd;

    int res = ftruncate(fd, offset);

    close(fd);

    return res;
}
```

While sleep is a POSIX function, usleep is not.

(Socket function wrappers 487)+≡ (454) <517 519>

```
void usleep(u_int useconds)
{
    GUSIContext::Current()->ClearWakeups();

    if (!useconds) {
        GUSIContext::Yield(false);

        return;
    }
    GUSITimer * timer = new GUSITimer;
    timer->MicroSleep(useconds);
    GUSIContext::Yield(true);
    delete timer;

    return;
}
```

symlink creates a symbolic link to a file.

(Socket function wrappers 487)+≡ (454) <518 520>

```
int symlink(const char * path, const char * linkto)
{
    GUSIDeviceRegistry * factory = GUSIDeviceRegistry::Instance();

    return factory->symlink(path, linkto);
}
```

readlink reads the contents of a symbolic link.

(Socket function wrappers 487)+≡ (454) <519 521>

```
int readlink(const char * path, char * buf, int bufsize)
{
    GUSIDeviceRegistry * factory = GUSIDeviceRegistry::Instance();

    return factory->readlink(path, buf, bufsize);
}
```

inet_aton and inet_addr convert an address string to an internet address.

(Socket function wrappers 487)+≡ (454) <520 522>

```
int inet_aton(const char * addr, struct in_addr * ina)
{
    in_addr_t a = inet_addr(addr);

    if (a == INADDR_NONE)
        return 0;
    if (ina)
        ina->s_addr = a;

    return 1;
}
```

(Socket function wrappers 487)+≡ (454) <521 523>

```
in_addr_t inet_addr(const char * addr)
{
    return GUSINetDB::Instance()->inet_addr(addr);
}
```

```

⟨Socket function wrappers 487⟩+≡ (454) ◀522 524▶
char *inet_ntoa(in_addr addr)
{
    return GUSINetDB::Instance()->inet_ntoa(addr);
}

⟨Socket function wrappers 487⟩+≡ (454) ◀523 525▶
long gethostid()
{
    return GUSINetDB::Instance()->gethostid();
}

⟨Socket function wrappers 487⟩+≡ (454) ◀524 526▶
int gethostname(char *machname, int buflen)
{
    return GUSINetDB::Instance()->gethostname(machname, buflen);
}

⟨Socket function wrappers 487⟩+≡ (454) ◀525 527▶
void setprotoent(int stayopen)
{
    GUSINetDB::Instance()->setprotoent(stayopen);
}

⟨Socket function wrappers 487⟩+≡ (454) ◀526 528▶
void setserverent(int stayopen)
{
    GUSINetDB::Instance()->setserverent(stayopen);
}

⟨Socket function wrappers 487⟩+≡ (454) ◀527 529▶
void endprotoent()
{
    GUSINetDB::Instance()->endprotoent();
}

⟨Socket function wrappers 487⟩+≡ (454) ◀528 530▶
void endserverent()
{
    GUSINetDB::Instance()->endserverent();
}

⟨Socket function wrappers 487⟩+≡ (454) ◀529 531▶
hostent *gethostbyaddr(const void * addr, size_t size, int family)
{
    return GUSINetDB::Instance()->gethostbyaddr(addr, size, family);
}

⟨Socket function wrappers 487⟩+≡ (454) ◀530 532▶
hostent *gethostbyname(const char * name)
{
    return GUSINetDB::Instance()->gethostbyname(name);
}

```

```

<Socket function wrappers 487>+≡ (454) <531 533>
    protoent *getprotobyname(const char * name)
    {
        return GUSINetDB::Instance()->getprotobyname(name);
    }

<Socket function wrappers 487>+≡ (454) <532 534>
    protoent *getprotobyname(int proto)
    {
        return GUSINetDB::Instance()->getprotobyname(proto);
    }

<Socket function wrappers 487>+≡ (454) <533 535>
    protoent *getprotoent()
    {
        return GUSINetDB::Instance()->getprotoent();
    }

<Socket function wrappers 487>+≡ (454) <534 536>
    servent *getservbyname(const char * name, const char * proto)
    {
        return GUSINetDB::Instance()->getservbyname(name, proto);
    }

<Socket function wrappers 487>+≡ (454) <535 537>
    servent *getservbyport(int port, const char * proto)
    {
        return GUSINetDB::Instance()->getservbyport(port, proto);
    }

<Socket function wrappers 487>+≡ (454) <536
    servent *getservernt()
    {
        return GUSINetDB::Instance()->getservernt();
    }

```

14.3 Implementation of MPW wrappers

fgetfileinfo and fsetfileinfo manipulate the MacOS type/creator codes.

```

<MPW function wrappers 538>≡ (454) 539>
    int fgetfileinfo(const char * path, OSType * creator, OSType * type)
    {
        GUSIDeviceRegistry * factory = GUSIDeviceRegistry::Instance();

        return factory->fgetfileinfo(path, creator, type);
    }

    void fsetfileinfo(const char * path, OSType creator, OSType type)
    {
        GUSIDeviceRegistry * factory = GUSIDeviceRegistry::Instance();

        factory->fsetfileinfo(path, creator, type);
    }

```

faccess manipulates MPW properties of files.

(MPW function wrappers 538) +≡ (454) <538

```
int faccess(const char * path, unsigned * cmd, void * arg)
{
    GUSIDeviceRegistry *    factory = GUSIDeviceRegistry::Instance();

    return factory->faccess(path, cmd, arg);
}
```


Chapter 15

Pthreads Wrappers

As opposed to POSIX.1, with which I think I’m reasonable competent by now, I have little practical experience, let alone in-depth familiarity with Pthreads, so I’m going by what I learned from

- Reading B. Nicols, D. Buttlar, and Jacqueline Proulx, Farrell, “Pthreads Programming”, O’Reilly & Associates
- Taking a few glimpses at Chris Provenzano’s pthreads implementation.
- Reading the news:comp.programming.threads newsgroup.

If you believe that I’ve misunderstood Pthreads in my implementation, feel free to contact me.

As opposed to most other modules, the header files we’re generating here don’t have GUSI in its name.

```
<pthread.h 540>≡
#ifdef _PTHREAD_H_
#define _PTHREAD_H_

#include <sys/cdefs.h>
#include <sys/types.h>
#include <sys/time.h>
#include <GUSIPThread.h>

__BEGIN_DECLS
<Pthreads function declarations 552>
__END_DECLS

#endif /* _PTHREAD_H_ */
```

```

<sched.h 541>≡
    #ifndef _SCHED_H_
    #define _SCHED_H_

    #include <sys/cdefs.h>
    /* Required by UNIX 98 */
    #include <time.h>

    __BEGIN_DECLS
    <Sched function declarations 606>
    __END_DECLS

    #endif /* _SCHED_H_ */

<GUSIPThread.h 542>≡
    #ifndef _GUSIPThread_
    #define _GUSIPThread_

    #include <GUSISpecific.h>
    #include <GUSIContext.h>
    #include <GUSIContextQueue.h>

    __BEGIN_DECLS
    <Pthreads data types 544>
    __END_DECLS

    #endif /* _GUSIPThread_ */

<GUSIPThread.cp 543>≡
    #include "GUSIInternal.h"
    #include "GUSITimer.h"

    #include <pthread.h>
    #include <sched.h>

    <Implementation of Pthread attribute management 550>
    <Implementation of Pthread creation and destruction 561>
    <Implementation of Pthread thread specific data 571>
    <Implementation of Pthread varia 579>

```

15.1 Definition of Pthreads data types

A shocking revelation right at the beginning: A `pthread_t` is in fact a pointer to a `GUSIContext`, but no non-GUSI specific code shall ever know details about that.

```

<Pthreads data types 544>≡ (542) 545▶
    typedef GUSIContext * pthread_t;

```

A `pthread_attr_t` collects thread creation attributes. This is implemented as a pointer so it's easier to change the size of the underlying data structure.

```

<Pthreads data types 544>+≡ (542) ◀544 546▶
    typedef struct GUSIPThreadAttr * pthread_attr_t;

```

A `pthread_key_t` is a key to look up thread specific data.

```
(Pthreads data types 544)+≡ (542) ◀545 547▶
typedef GUSISpecific * pthread_key_t;
```

A `pthread_once_t` registers whether some routine has run once. It must always be statically initialized to `PTHREAD_ONCE_INIT` (Although in our implementation, it doesn't matter).

```
(Pthreads data types 544)+≡ (542) ◀546 548▶
typedef char pthread_once_t;

enum {
    PTHREAD_ONCE_INIT = 0
};
```

A `pthread_mutex_t` is a mutual exclusion variable, implemented as a pointer to a `GUSIContextQueue`. For initialization convenience, a 0 value means an unlocked mutex. No attributes are supported so far.

```
(Pthreads data types 544)+≡ (542) ◀547 549▶
typedef GUSIContextQueue * pthread_mutex_t;
typedef void * pthread_mutexattr_t;

#define PTHREAD_MUTEX_INITIALIZER 0
```

A `pthread_cond_t` is a condition variable, which looks rather similar to a mutex, but has different semantics. No attributes are supported so far.

```
(Pthreads data types 544)+≡ (542) ◀548
typedef GUSIContextQueue * pthread_cond_t;
typedef void * pthread_condattr_t;

#define PTHREAD_COND_INITIALIZER 0
```

15.2 PThread Attribute Management

On creation of a thread, we can specify various attributes via a thread attribute.

```
(Implementation of Pthread attribute management 550)+≡ (543) 551▶
struct GUSIPThreadAttr {
    size_t fStackSize;
    enum {
        detached = 1 << 0
    };
    int fFlags;

    static GUSIPThreadAttr sDefault;
    static pthread_attr_t sDefaultAttr;
};
```

By default, we give threads 20K of stack space.

```
(Implementation of Pthread attribute management 550)+≡ (543) ◀550 553▶
GUSIPThreadAttr GUSIPThreadAttr::sDefault = { 20480, 0 };
pthread_attr_t GUSIPThreadAttr::sDefaultAttr = &GUSIPThreadAttr::sDefault;

pthread_attr_init initializes an attribute object with the default values.
(Pthreads function declarations 552)+≡ (540) 554▶
int pthread_attr_init(pthread_attr_t * attr);
```

(Implementation of Pthread attribute management 550)+≡ (543) <551 555>

```

int pthread_attr_init(pthread_attr_t * attr)
{
    return (*attr = new GUSIPThreadAttr(GUSIPThreadAttr::sDefault))
        ? 0 : ENOMEM;
}

```

pthread_attr_destroy destroys an attribute object.

(Pthreads function declarations 552)+≡ (540) <552 556>

```

int pthread_attr_destroy(pthread_attr_t * attr);

```

(Implementation of Pthread attribute management 550)+≡ (543) <553 557>

```

int pthread_attr_destroy(pthread_attr_t * attr)
{
    delete *attr;
    *attr = nil;

    return 0;
}

```

The detach state defines whether a thread will be defined joinable or detached.

(Pthreads function declarations 552)+≡ (540) <554 558>

```

enum {
    PTHREADS_CREATE_JOINABLE,
    PTHREADS_CREATE_DETACHED
};

```

```

int pthread_attr_getdetachstate(const pthread_attr_t * attr, int * state);
int pthread_attr_setdetachstate(pthread_attr_t * attr, int state);

```

(Implementation of Pthread attribute management 550)+≡ (543) <555 559>

```

int pthread_attr_getdetachstate(const pthread_attr_t * attr, int * state)
{
    *state =
        (attr[0]->fFlags & GUSIPThreadAttr::detached)
        ? PTHREADS_CREATE_JOINABLE
        : PTHREADS_CREATE_DETACHED;

    return 0;
}

```

```

int pthread_attr_setdetachstate(pthread_attr_t * attr, int state)
{
    if (state == PTHREADS_CREATE_JOINABLE)
        attr[0]->fFlags &= ~GUSIPThreadAttr::detached;
    else
        attr[0]->fFlags |= GUSIPThreadAttr::detached;

    return 0;
}

```

The stack size defines how much stack space will be allocated. Stack overflows typically lead to utterly nasty crashes.

(Pthreads function declarations 552)+≡ (540) <556 560>

```

int pthread_attr_getstacksize(const pthread_attr_t * attr, size_t * size);
int pthread_attr_setstacksize(pthread_attr_t * attr, size_t size);

```

```

<Implementation of Pthread attribute management 550>+≡ (543) <557
int pthread_attr_getstacksize(const pthread_attr_t * attr, size_t * size)
{
    *size = attr[0]->fStackSize;

    return 0;
}

int pthread_attr_setstacksize(pthread_attr_t * attr, size_t size)
{
    attr[0]->fStackSize = size;

    return 0;
}

```

15.3 Creation and Destruction of PThreads

First, we define wrapper to map the different calling conventions of Pthreads and Macintosh threads.

```

<Pthreads function declarations 552>+≡ (540) <558 562>
__BEGIN_DECLS
typedef void * (*GUSIPThreadProc)(void *);
__END_DECLS

```

```

<Implementation of Pthread creation and destruction 561>≡ (543) 563>
struct CreateArg {
    GUSIPThreadProc fRealProc;
    void *          fRealArg;

    CreateArg(GUSIPThreadProc realProc, void * realArg)
        : fRealProc(realProc), fRealArg(realArg) {}
};

static pascal void * GUSIThreadEntryProcWrapper(CreateArg * arg)
{
    CreateArg    fixedArg = *arg;

    delete arg;
    return fixedArg.fRealProc(fixedArg.fRealArg);
}

```

pthread_create stuffs the arguments in a CreateArg and creates the context.

```

<Pthreads function declarations 552>+≡ (540) <560 564>
int pthread_create(
    pthread_t * thread,
    const pthread_attr_t * attr, GUSIPThreadProc proc, void * arg);

```

(Implementation of Pthread creation and destruction 561)+≡ (543) <561 565>

```
int pthread_create(
    pthread_t * thread,
    const pthread_attr_t * attr, GUSIPThreadProc proc, void * arg)
{
    if (!attr)
        attr = &GUSIPThreadAttr::sDefaultAttr;

    GUSIContext::Setup(true);
    *thread =
        GUSIContextFactory::Instance()->CreateContext(
            reinterpret_cast<ThreadEntryProcPtr>(GUSIThreadEntryProcWrapper),
            new CreateArg(proc, arg),
            attr[0]->fStackSize);
    switch (thread[0]->Error()) {
    case noErr:
        if (attr[0]->fFlags & GUSIPThreadAttr::detached)
            thread[0]->Detach();
        return 0;
    default:
        thread[0]->Liquidate();
        *thread = nil;

        return ENOMEM; // Most likely candidate for error
    }
}
```

A thread can either be detached, in which case it will just go away after it's done, or it can be joinable, in which case it will wait for pthread_join to be called.

(Pthreads function declarations 552)+≡ (540) <562 566>

```
int pthread_detach(pthread_t thread);
```

(Implementation of Pthread creation and destruction 561)+≡ (543) <563 567>

```
int pthread_detach(pthread_t thread)
{
    thread->Detach();

    return 0;
}
```

pthread_join waits for the thread to die and optionally returns its last words.

(Pthreads function declarations 552)+≡ (540) <564 568>

```
int pthread_join(pthread_t thread, void **value);
```

(Implementation of Pthread creation and destruction 561)+≡ (543) <565 569>

```
int pthread_join(pthread_t thread, void **value)
{
    if (thread->Done(true)) {
        if (value)
            *value = thread->Result();
        thread->Liquidate();

        return 0;
    } else
        return ESRCH;
}
```

pthread_exit ends the existence of a thread.

```
<Pthreads function declarations 552>+≡ (540) <566 570>
int pthread_exit(void *value);

<Implementation of Pthread creation and destruction 561>+≡ (543) <567
int pthread_exit(void *value)
{
    DisposeThread(GUSIContext::Current()->ID(), value, false);

    return 0;    // Not reached
}
```

15.4 Pthread thread specific data

Thread specific data offers a possibility to quickly look up a value that may be different for every thread.

pthread_key_create creates an unique key visible to all threads in a process.

```
<Pthreads function declarations 552>+≡ (540) <568 572>
__BEGIN_DECLS
typedef void (*GUSIPThreadKeyDestructor)(void *);
__END_DECLS

int pthread_key_create(pthread_key_t * key, GUSIPThreadKeyDestructor destructor);

<Implementation of Pthread thread specific data 571>+≡ (543) 573>
int pthread_key_create(pthread_key_t * key, GUSIPThreadKeyDestructor destructor)
{
    return (*key = new GUSISpecific(destructor)) ? 0 : ENOMEM;
}
```

pthread_key_delete deletes a key, but does not call any destructors.

```
<Pthreads function declarations 552>+≡ (540) <570 574>
int pthread_key_delete(pthread_key_t key);

<Implementation of Pthread thread specific data 571>+≡ (543) <571 575>
int pthread_key_delete(pthread_key_t key)
{
    delete key;

    return 0;
}
```

pthread_getspecific returns the thread specific value for a key.

```
<Pthreads function declarations 552>+≡ (540) <572 576>
void * pthread_getspecific(pthread_key_t key);

<Implementation of Pthread thread specific data 571>+≡ (543) <573 577>
void * pthread_getspecific(pthread_key_t key)
{
    return GUSIContext::Current()->GetSpecific(key);
}
```

pthread_setspecific sets a new thread specific value for a key.

```
<Pthreads function declarations 552>+≡ (540) <574 578>
int pthread_setspecific(pthread_key_t key, void * value);
```

```

<Implementation of Pthread thread specific data 571>+≡ (543) <575
int pthread_setspecific(pthread_key_t key, void * value)
{
    GUSIContext::Current()->SetSpecific(key, value);

    return 0;
}

```

15.5 Synchronization mechanisms of PThreads

Since we're only dealing with cooperative threads, all synchronization mechanisms can be implemented using means that might look naive to a student of computer science, but that actually work perfectly well in our case.

We currently don't support mutex and condition variable attributes. To minimize the amount of code changes necessary inclients, we support creating and destroying them, at least.

```

<Pthreads function declarations 552>+≡ (540) <576 580>
int pthread_mutexattr_init(pthread_mutexattr_t * attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t * attr);

<Implementation of Pthread varia 579>+≡ (543) 581>
int pthread_mutexattr_init(pthread_mutexattr_t *)
{
    return 0;
}

int pthread_mutexattr_destroy(pthread_mutexattr_t *)
{
    return 0;
}

<Pthreads function declarations 552>+≡ (540) <578 582>
int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutexattr_t *);
int pthread_mutex_destroy(pthread_mutex_t * mutex);

<Implementation of Pthread varia 579>+≡ (543) <579 583>
int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutexattr_t *)
{
    *mutex = new GUSIContextQueue;

    return 0;
}

int pthread_mutex_destroy(pthread_mutex_t * mutex)
{
    delete *mutex;
    *mutex = 0;

    return 0;
}

```


Lock may create the queue if it was allocated statically. Mutexes are implemented as a queue of context, with the frontmost context holding the lock. Simple enough, isn't it?

⟨Pthreads function declarations 552⟩+≡ (540) ◀580 584▶
 int pthread_mutex_lock(pthread_mutex_t * mutex);

⟨Implementation of Pthread varia 579⟩+≡ (543) ◀581 585▶
 int pthread_mutex_lock(pthread_mutex_t * mutex)
 {
 if (!*mutex)
 *mutex = new GUSIContextQueue;
 mutex[0]->push(GUSIContext::Current());
 while (mutex[0]->front() != GUSIContext::Current())
 GUSIContext::Yield(true);
 return 0;
 }

⟨Pthreads function declarations 552⟩+≡ (540) ◀582 586▶
 int pthread_mutex_trylock(pthread_mutex_t * mutex);

⟨Implementation of Pthread varia 579⟩+≡ (543) ◀583 587▶
 int pthread_mutex_trylock(pthread_mutex_t * mutex)
 {
 if (!*mutex)
 *mutex = new GUSIContextQueue;
 if (mutex[0]->empty() || mutex[0]->front() == GUSIContext::Current()) {
 mutex[0]->push(GUSIContext::Current());

 return 0;
 } else
 return EBUSY;
 }

Unlocking pops us off the queue and wakes up the new lock owner.

⟨Pthreads function declarations 552⟩+≡ (540) ◀584 588▶
 int pthread_mutex_unlock(pthread_mutex_t * mutex);

⟨Implementation of Pthread varia 579⟩+≡ (543) ◀585 589▶
 int pthread_mutex_unlock(pthread_mutex_t * mutex)
 {
 if (!*mutex || mutex[0]->front() != GUSIContext::Current())
 return EPERM; // We don't hold that lock
 mutex[0]->pop();
 if (GUSIContext * new_boss = mutex[0]->front())
 new_boss->Wakeup();
 GUSIContext::Yield(false);
 return 0;
 }

On to condition variable attributes, which we don't really support either.

⟨Pthreads function declarations 552⟩+≡ (540) ◀586 590▶
 int pthread_condattr_init(pthread_condattr_t * attr);
 int pthread_condattr_destroy(pthread_condattr_t * attr);

(Implementation of Pthread varia 579)+≡ (543) <587 591>

```
int pthread_condattr_init(pthread_condattr_t *)
{
    return 0;
}

int pthread_condattr_destroy(pthread_condattr_t *)
{
    return 0;
}
```

Condition variables in some respects work very similar to mutexes.

(Pthreads function declarations 552)+≡ (540) <588 592>

```
int pthread_cond_init(pthread_cond_t * cond, const pthread_condattr_t *);
int pthread_cond_destroy(pthread_cond_t * cond);
```

(Implementation of Pthread varia 579)+≡ (543) <589 593>

```
int pthread_cond_init(pthread_cond_t * cond, const pthread_condattr_t *)
{
    *cond = new GUSIContextQueue;

    return 0;
}

int pthread_cond_destroy(pthread_cond_t * cond)
{
    pthread_cond_broadcast(cond);

    delete *cond;
    *cond = 0;

    return 0;
}
```

pthread_cond_wait releases the mutex, sleeps on the condition variable, and reacquires the mutex.

(Pthreads function declarations 552)+≡ (540) <590 594>

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

(Implementation of Pthread varia 579)+≡ (543) <591 595>

```
int pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * mutex)
{
    if (!*mutex || mutex[0]->front() != GUSIContext::Current())
        return EPERM;    // We don't hold that lock
    mutex[0]->pop();
    if (GUSIContext * new_boss = mutex[0]->front())
        new_boss->Wakeup();
    if (!*cond)
        *cond = new GUSIContextQueue;
    cond[0]->push(GUSIContext::Current());
    GUSIContext::Yield(true);
    if (*cond)
        cond[0]->remove(GUSIContext::Current());
    return pthread_mutex_lock(mutex);
}
```

pthread_cond_timedwait adds a timeout value (But it still could block indefinitely trying to reacquire the mutex). Note that the timeout specifies an absolute wakeup time, not an interval.

<Pthreads function declarations 552>+≡ (540) ◀592 596▶

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

<Implementation of Pthread varia 579>+≡ (543) ◀593 597▶

```
int pthread_cond_timedwait(
    pthread_cond_t * cond,
    pthread_mutex_t * mutex,
    const struct timespec * patience)
{
    GUSITimer timer;
    if (!*mutex || mutex[0]->front() != GUSIContext::Current())
        return EPERM; // We don't hold that lock
    mutex[0]->pop();
    if (GUSIContext * new_boss = mutex[0]->front())
        new_boss->Wakeup();
    if (!*cond)
        *cond = new GUSIContextQueue;
    cond[0]->push(GUSIContext::Current());
    GUSITime interval = GUSITime(*patience)-GUSITime::Now();
    GUSIContext::Current()->ClearWakeup();
    timer.MicroSleep(interval.Get(GUSITime::usecs));
    GUSIContext::Yield(true);
    if (*cond)
        cond[0]->remove(GUSIContext::Current());
    return pthread_mutex_lock(mutex);
}
```

pthread_cond_signal wakes up a context from the queue. Since we typically still hold the associated mutex, it would be a bad idea (though not a disastrous one) to put a yield in here.

<Pthreads function declarations 552>+≡ (540) ◀594 598▶

```
int pthread_cond_signal(pthread_cond_t * cond);
```

<Implementation of Pthread varia 579>+≡ (543) ◀595 599▶

```
int pthread_cond_signal(pthread_cond_t * cond)
{
    if (!*cond)
        return 0;

    if (GUSIContext * hey = cond[0]->front())
        hey->Wakeup();
    return 0;
}
```

pthread_cond_broadcast wakes up a the entire queue (but only one context will get the mutex).

<Pthreads function declarations 552>+≡ (540) ◀596 600▶

```
int pthread_cond_broadcast(pthread_cond_t * cond);
```

(*Implementation of Pthread varia 579*) +≡ (543) ◀597 601▶

```

int pthread_cond_broadcast(pthread_cond_t * cond)
{
    if (!*cond)
        return 0;

    cond[0]->Wakeup();

    return 0;
}

```

15.6 Pthread varia

pthread_self returns the current thread.

(*Pthreads function declarations 552*) +≡ (540) ◀598 602▶

```

pthread_t pthread_self(void);

```

(*Implementation of Pthread varia 579*) +≡ (543) ◀599 603▶

```

pthread_t pthread_self()
{
    return GUSIContext::Current();
}

```

pthread_equal compares two thread handles.

(*Pthreads function declarations 552*) +≡ (540) ◀600 604▶

```

int pthread_equal(pthread_t t1, pthread_t t2);

```

(*Implementation of Pthread varia 579*) +≡ (543) ◀601 605▶

```

int pthread_equal(pthread_t t1, pthread_t t2)
{
    return t1 == t2;
}

```

pthread_once calls a routines, guaranteeing that it will be called exactly once per process.

(*Pthreads function declarations 552*) +≡ (540) ◀602

```

__BEGIN_DECLS
typedef void (*GUSIPThreadOnceProc)(void);
__END_DECLS

```

```

int pthread_once(pthread_once_t * once_block, GUSIPThreadOnceProc proc);

```

(*Implementation of Pthread varia 579*) +≡ (543) ◀603 607▶

```

int pthread_once(pthread_once_t * once_block, GUSIPThreadOnceProc proc)
{
    if (!*once_block) {
        *once_block = 1;
        proc();
    }
    return 0;
}

```

sched_yield yields the processor for other runnable threads.

(*Sched function declarations 606*) ≡ (541)

```

int sched_yield();

```

(Implementation of Pthread varia 579)+≡

(543) ◀605

```
int sched_yield()
{
    GUSIContext::Yield(false);

    return 0;
}
```


Chapter 16

Signal support

We support signals in the half assed way characteristic for GUSI's approach to asynchronous issues: Delivery is very much synchronous, basically within Yield calls. Signal handling behavior is encapsulated in the classes GUSISigContext and GUSISigProcess whose instances are manufactured by a GUSISigFactory.

```
<GUSISignal.h 608>≡
    #ifndef _GUSISIGNAL_
    #define _GUSISIGNAL_

    #include <signal.h>

    #ifdef GUSI_SOURCE
    <Definition of class GUSISigContext 611>
    <Definition of class GUSISigProcess 610>
    <Definition of class GUSISigFactory 612>
    #endif

    #endif /* _GUSISIGNAL_ */

<GUSISignal.cp 609>≡
    #include "GUSIInternal.h"
    #include "GUSISignal.h"
    #include "GUSIDiag.h"
    #include "GUSITimer.h"

    #include <stdlib.h>
    #include <unistd.h>
    #include <memory>

    GUSI_USING_STD_NAMESPACE

    <Member functions for class GUSISigContext 616>
    <Member functions for class GUSISigProcess 623>
    <Member functions for class GUSISigFactory 613>

    <POSIX functions for signal handling 635>
```

16.1 Definition of the signal handling engine

A GUSISigProcess contains the per-process signal state. GetAction and SetAction manipulate the action associated with a signal, Pending returns the set of pending signals, Post marks a signal as pending (but possibly blocked), and Raise executes a signal (which we have determined is not blocked).

```
(Definition of class GUSISigProcess 610)≡ (608)
class GUSISigContext;

class GUSISigProcess {
public:
    virtual struct sigaction & GetAction(int sig);
    virtual int SetAction(int sig, const struct sigaction & act);
    virtual sigset_t Pending() const;
    virtual void ClearPending(sigset_t clear);
    virtual void Post(int sig);
    virtual bool Raise(int sig, GUSISigContext * context);

    virtual ~GUSISigProcess();
protected:
    (Privatissima of GUSISigProcess 622)

    friend class GUSISigFactory;
    GUSISigProcess();
};
```

A GUSISigContext contains the per-thread signal state, primarily blocking info. To support pthread_kill, we have our own set of pending signals. GetBlocked and SetBlocked manipulate the set of blocking signals, Pending returns the set of pending signals, Post marks a signal as pending (but possibly blocked), and Raise executes all eligible signals.

```
(Definition of class GUSISigContext 611)≡ (608)
class GUSISigContext {
public:
    virtual sigset_t GetBlocked() const;
    virtual void SetBlocked(sigset_t sigs);
    virtual sigset_t Pending() const;
    virtual sigset_t Pending(GUSISigProcess * proc) const;
    virtual void ClearPending(sigset_t clear);
    virtual void Post(int sig);
    virtual sigset_t Ready(GUSISigProcess * proc);
    virtual bool Raise(GUSISigProcess * proc, bool allSigs = false);

    virtual ~GUSISigContext();
protected:
    (Privatissima of GUSISigContext 615)

    friend class GUSISigFactory;
    GUSISigContext(const GUSISigContext * parent);
};
```


The GUSISigFactory singleton creates the above two classes, allowing a future extension to handle more signals.

```
(Definition of class GUSISigFactory 612)≡ (608)
class GUSISigFactory {
public:
    virtual GUSISigProcess *    CreateSigProcess();
    virtual GUSISigContext *    CreateSigContext(const GUSISigContext * parent);

    virtual ~GUSISigFactory();

    static GUSISigFactory *      Instance();
    static void                  SetInstance(GUSISigFactory * instance);
protected:
    GUSISigFactory()             {}
};
```

16.2 Implementation of the signal handling engine

The GUSISigFactory is a straightforward overridable singleton.

```
(Member functions for class GUSISigFactory 613)≡ (609) 614▷
static auto_ptr<GUSISigFactory> sGUSISigFactory;

GUSISigFactory * GUSISigFactory::Instance()
{
    if (!sGUSISigFactory.get())
        SetInstance(new GUSISigFactory);

    return sGUSISigFactory.get();
}

void GUSISigFactory::SetInstance(GUSISigFactory * instance)
{
    sGUSISigFactory = auto_ptr<GUSISigFactory>(instance);
}

GUSISigFactory::~~GUSISigFactory()
{
}
```

To support more signals, override these creators to make them return subclasses of the required classes.

```
(Member functions for class GUSISigFactory 613)+≡ (609) ◀613
GUSISigProcess * GUSISigFactory::CreateSigProcess()
{
    return new GUSISigProcess;
}

GUSISigContext * GUSISigFactory::CreateSigContext(const GUSISigContext * parent)
{
    return new GUSISigContext(parent);
}
```

GUSISigContext mainly deals with a set of blocked signals, which it inherits from its parent.

```
<Privatissima of GUSISigContext 615>≡ (611) 617»
    sigset_t    fPending;
    sigset_t    fBlocked;
```

```
<Member functions for class GUSISigContext 616>≡ (609) 618»
GUSISigContext::GUSISigContext(const GUSISigContext * parent)
    : fPending(0), fBlocked(parent ? parent->fBlocked : 0)
{
}
```

```
GUSISigContext::~GUSISigContext()
{
}
```

Many signals cannot be blocked. CantBlock defines those.

```
<Privatissima of GUSISigContext 615>+≡ (611) <615
    virtual sigset_t    CantBlock();
```

```
<Member functions for class GUSISigContext 616>+≡ (609) <616 619»
    inline sigset_t SigMask(int signal) { return 1 << (signal-1); }
```

```
sigset_t GUSISigContext::CantBlock()
{
    return SigMask(SIGKILL)
        | SigMask(SIGSTOP)
        | SigMask(SIGILL)
        | SigMask(SIGFPE)
        | SigMask(SIGSEGV);
}
```

GetBlocked and SetBlocked take care of the rest.

```
<Member functions for class GUSISigContext 616>+≡ (609) <618 620»
sigset_t GUSISigContext::GetBlocked() const
{
    return fBlocked;
}

void GUSISigContext::SetBlocked(sigset_t sigs)
{
    fBlocked = sigs & ~CantBlock();
}
```

Pending and Post deal with signals pending against a thread (as a result of pthread_kill).

(Member functions for class GUSISigContext 616)+≡ (609) ◀619 621▶

```
sigset_t GUSISigContext::Pending() const
{
    return fPending;
}

void GUSISigContext::ClearPending(sigset_t clear)
{
    fPending &= ~clear;
}

void GUSISigContext::Post(int sig)
{
    sigaddset(&fPending, sig);
}
```

Raise initiates a series of signal executions and returns true if any of these should interrupt a slow system call.

(Member functions for class GUSISigContext 616)+≡ (609) ◀620

```
sigset_t GUSISigContext::Pending(GUSISigProcess * proc) const
{
    return proc->Pending() | Pending();
}

sigset_t GUSISigContext::Ready(GUSISigProcess * proc)
{
    return Pending(proc) & ~GetBlocked();
}

bool GUSISigContext::Raise(GUSISigProcess * proc, bool allSigs)
{
    bool interrupt = false;

    for (;;) {
        sigset_t todo = Ready(proc);

        if (!todo)
            return interrupt;

        proc->ClearPending(todo);
        ClearPending(todo);

        for (int sig = 1; todo; ++sig)
            if (sigismember(&todo, sig)) {
                sigdelset(&todo, sig);
                interrupt = proc->Raise(sig, this) || interrupt || allSigs;
            }
    }
}
```

GUSISigProcess stores the signal handlers and the set of signals pending against the process.

```
<Privatissima of GUSISigProcess 622>≡ (610) 625»  
    sigset_t      fPending;  
    struct sigaction fAction[NSIG-1];
```

```
<Member functions for class GUSISigProcess 623>≡ (609) 624»  
    GUSISigProcess::GUSISigProcess()  
    : fPending(0)  
    {  
        memset(&fAction, 0, sizeof(fAction));  
    }
```

```
    GUSISigProcess::~~GUSISigProcess()  
    {  
    }
```

GetAction returns a signal action.

```
<Member functions for class GUSISigProcess 623>+≡ (609) «623 626»  
    struct sigaction & GUSISigProcess::GetAction(int sig)  
    {  
        return fAction[sig-1];  
    }
```

Some actions can't be caught and/or ignored. CantCatch and CantIgnore report those.

```
<Privatissima of GUSISigProcess 622>+≡ (610) «622 631»  
    virtual bool CantCatch(int sig);  
    virtual bool CantIgnore(int sig);
```

⟨Member functions for class GUSISigProcess 623⟩+≡ (609) ◀624 627▶

```

bool GUSISigProcess::CantCatch(int sig)
{
    return (sig == SIGKILL)
        || (sig == SIGSTOP);
}

bool GUSISigProcess::CantIgnore(int sig)
{
    return (sig == SIGKILL)
        || (sig == SIGSTOP);
}

int GUSISigProcess::SetAction(int sig, const struct sigaction & act)
{
    if (act.sa_handler == SIG_IGN) {
        if (CantIgnore(sig))
            return GUSISetPosixError(EINVAL);
        sigdelset(&fPending, sig);
    } else if (act.sa_handler != SIG_DFL) {
        if (CantCatch(sig))
            return GUSISetPosixError(EINVAL);
    }
    fAction[sig-1] = act;

    return 0;
}

```

Pending and Post deal with signals pending against a process (as a result of kill or raise).

⟨Member functions for class GUSISigProcess 623⟩+≡ (609) ◀626 629▶

```

sigset_t GUSISigProcess::Pending() const
{
    return fPending;
}

void GUSISigProcess::ClearPending(sigset_t clear)
{
    fPending &= ~clear;
}

void GUSISigProcess::Post(int sig)
{
    sigaddset(&fPending, sig);
    ⟨Wake up appropriate contexts to deliver signal 628⟩
}

```

To wake up contexts, we make two passes: In the first pass, we look for contexts which have the signal unblocked. If we don't find any, we wake up all contexts in the hope that one of them is waiting in a sigwait. This corresponds more or less to the typical nonthreaded and threaded models of thread handling, respectively.

```

<Wake up appropriate contexts to deliver signal 628>≡ (627)
sigset_t    sigmask = SigMask(sig);
bool        found   = false;
for (GUSIContextQueue::iterator context = GUSIContext::begin(); context != GUSIContext::end(); ++context)
    if (!(context->SigContext()->GetBlocked() & sigmask)) {
        context->Wakeup();
        found = true;
    }
if (!found)
for (GUSIContextQueue::iterator context = GUSIContext::begin(); context != GUSIContext::end(); ++context)
    context->Wakeup();

```

Raise, finally, is the function for which the whole rest of the engine exists.

```

<Member functions for class GUSISigProcess 623>+≡ (609) <627 632>
bool GUSISigProcess::Raise(int sig, GUSISigContext * context)
{
    struct sigaction & act = GetAction(sig);

    if (act.sa_handler == SIG_IGN)
        return false; // Ignore signal
    else if (act.sa_handler == SIG_DFL)
        return DefaultAction(sig, act) && !(act.sa_flags & SA_RESTART);

    <Execute the signal handler 630>
}

```

Executing an user defined signal handler involves a rather complicated dance of blocking and unblocking signals.

```

<Execute the signal handler 630>≡ (629)
__sig_handler    handler = act.sa_handler;
sigset_t         blockset= act.sa_mask;

if (act.sa_flags & SA_RESETHAND)
    act.sa_handler = SIG_DFL;
else if (!(act.sa_flags & SA_NODEFER))
    sigaddset(&blockset, sig);

sigset_t    blocksave = context->GetBlocked();
context->SetBlocked(blocksave | blockset);
(*handler)(sig);
sigset_t    blocknew  = context->GetBlocked();
context->SetBlocked(blocksave | (blocknew & ~blockset));

return !(act.sa_flags & SA_RESTART);

```

The default behavior for many signals is to abort the process.

```

<Privatissima of GUSISigProcess 622>+≡ (610) <625
virtual bool DefaultAction(int sig, const struct sigaction & act);

```

```

⟨Member functions for class GUSISigProcess 623⟩+≡ (609) 4629
extern "C" void _exit(int status);

bool GUSISigProcess::DefaultAction(int sig, const struct sigaction &)
{
    switch (sig) {
        case SIGCHLD:
        case SIGCONT:
        case SIGSTOP:
        case SIGTSTP:
        case SIGTTIN:
        case SIGTTOU:
            break; // Ignore
        default:
            _exit(1);
    }

    return false;
}

```

16.3 Definition of the signal handling POSIX functions

A considerable number of POSIX functions is concerned with signal handling. Many of them can profit from range checking on signal numbers.

```

⟨Perform range check on signo 633⟩≡ (635 636 638)
if (!GUSI_CASSERT_CLIENT(signo>0 && signo<NSIG))
    return GUSISetPosixError(EINVAL);

⟨Perform pthreads style range check on signo 634⟩≡ (637)
if (!GUSI_CASSERT_CLIENT(signo>0 && signo<NSIG))
    return EINVAL;

```

First of all, we define out of line versions of the signal set manipulators, giving them the added benefit of a range check.

(POSIX functions for signal handling 635)≡ (609) 636▶

```
int (sigaddset)(sigset_t *set, int signo)
{
    ⟨Perform range check on signo 633⟩
    return sigaddset(set, signo);
}

int (sigdelset)(sigset_t *set, int signo)
{
    ⟨Perform range check on signo 633⟩
    return sigdelset(set, signo);
}

int (sigemptyset)(sigset_t *set)
{
    return sigemptyset(set);
}

int (sigfillset)(sigset_t *set)
{
    return sigfillset(set);
}

int (sigismember)(const sigset_t *set, int signo)
{
    ⟨Perform range check on signo 633⟩
    return sigismember(set, signo);
}
```

raise posts a signal against the current process.

(POSIX functions for signal handling 635)+≡ (609) ◀635 637▶

```
int raise(int signo)
{
    ⟨Perform range check on signo 633⟩
    GUSIProcess::Instance()->SigProcess()->Post(signo);

    return 0;
}
```

pthread_kill posts a signal against the specified thread.

(POSIX functions for signal handling 635)+≡ (609) ◀636 638▶

```
int pthread_kill(pthread_t thread, int signo)
{
    if (!signo)
        return thread ? 0 : ESRCH;
    ⟨Perform pthreads style range check on signo 634⟩
    thread->SigContext()->Post(signo);
    thread->Wakeup();

    return 0;
}
```


sigaction manipulates the signal action table.

```
(POSIX functions for signal handling 635)+≡ (609) <637 639>
int sigaction(int signo, const struct sigaction * act, struct sigaction * oact)
{
    <Perform range check on signo 633>
    GUSISigProcess * proc = GUSIPProcess::Instance()->SigProcess();
    if (oact)
        *oact = proc->GetAction(signo);
    if (act)
        if (proc->SetAction(signo, *act))
            return -1;
        else if (act->sa_handler==SIG_IGN)
            GUSIContext::Current()->SigContext()->ClearPending(SigMask(signo));
    return 0;
}
```

signal is the historical and rather inconvenient API for sigaction.

```
(POSIX functions for signal handling 635)+≡ (609) <638 640>
__sig_handler signal(int signo, __sig_handler newhandler)
{
    struct sigaction oact;
    struct sigaction nact;

    nact.sa_handler = newhandler;
    nact.sa_mask = 0;
    nact.sa_flags = SA_RESETHAND;

    if (sigaction(signo, &nact, &oact))
        return reinterpret_cast<__sig_handler>(0);
    else
        return oact.sa_handler;
}
```

sigpending returns the list of pending signals.

```
(POSIX functions for signal handling 635)+≡ (609) <639 641>
int sigpending(sigset_t * pending)
{
    if (pending)
        *pending = GUSIPProcess::Instance()->SigProcess()->Pending()
            | GUSIContext::Current()->SigContext()->Pending();
    return 0;
}
```

pthread_sigmask and sigprocmask manipulate the signal mask.

(POSIX functions for signal handling 635)+≡ (609) <640 642>

```
int pthread_sigmask(int how, const sigset_t * mask, sigset_t * omask)
{
    GUSISigContext * context = GUSIContext::Current()->SigContext();
    if (omask)
        *omask = context->GetBlocked();
    if (mask)
        switch (how) {
            case SIG_BLOCK:
                context->SetBlocked(context->GetBlocked() | *mask);
                break;
            case SIG_SETMASK:
                context->SetBlocked(*mask);
                break;
            case SIG_UNBLOCK:
                context->SetBlocked(context->GetBlocked() & ~*mask);
                break;
            default:
                return EINVAL;
        }
    GUSIContext::Raise();
    return 0;
}

int sigprocmask(int how, const sigset_t * mask, sigset_t * omask)
{
    return GUSISetPosixError(pthread_sigmask(how, mask, omask));
}
```

sigsuspend waits for a signal to arrive. This is one of the few POSIX functions which always returns an error.

(POSIX functions for signal handling 635)+≡ (609) <641 643>

```
int sigsuspend(const sigset_t * mask)
{
    GUSISigContext * context = GUSIContext::Current()->SigContext();
    sigset_t    blocksave = context->GetBlocked();
    context->SetBlocked(*mask);
    GUSIContext::SigSuspend();
    sigset_t    blocknew = context->GetBlocked();
    context->SetBlocked(blocksave | (blocknew & ~*mask));

    return GUSISetPosixError(EINTR);
}
```

sigwait waits for a blocked signal.

```
(POSIX functions for signal handling 635)+≡ (609) <642 644>
int sigwait(const sigset_t * sigs, int * signo)
{
    if (!GUSI_CASSERT_CLIENT(sigs && signo && !(*sigs & ~GUSIContext::Blocked())))
        return GUSISetPosixError(EINVAL);
    GUSIContext::SigWait(*sigs);
    const sigset_t cursigs = *sigs & GUSIContext::Pending();
    for (*signo = 1; !sigismember(&cursigs, *signo); ++*signo)
        ;
    GUSIContext::Current()->SigContext()->ClearPending(SigMask(*signo));
    GUSIProcess::Instance()->SigProcess()->ClearPending(SigMask(*signo));

    return 0;
}
```

abort raises SIGABRT and calls _exit.

```
(POSIX functions for signal handling 635)+≡ (609) <643 645>
void abort(void)
{
    raise(SIGABRT);

    _exit(2);
}
```

_exit is similar to exit, but doesn't call handlers established with atexit. MPW already has a working implementation for _exit.

```
(POSIX functions for signal handling 635)+≡ (609) <644 646>
#ifdef __MWERKS__
extern int __aborting;

extern "C" void _exit(int code)
{
    __aborting = 1;

    exit(code);
}
#endif
```

To handle alarms, we define the auxiliary class GUSIAAlarm.

```
(POSIX functions for signal handling 635)+≡ (609) <645 647>
class GUSIAAlarm : public GUSITimer {
public:
    GUSIAAlarm(long interval = 0) : GUSITimer(true), fInterval(interval) {}

    virtual void    Wakeup();
    long           Restart(long interval = 0);
private:
    long           fInterval;
};

void GUSIAAlarm::Wakeup()
{
    GUSIPProcess::Instance()->SigProcess()->Post(SIGALRM);
    if (fInterval)
        Sleep(fInterval, true);
}

long GUSIAAlarm::Restart(long interval)
{
    fInterval = interval;
    RmvTime(Elem());
    long rest = tmCount;
    if (rest < 0)
        rest = -rest;
    else
        rest *= 1000;
    tmCount    = 0;
    tmWakeUp   = 0;
    tmReserved = 0;
    InsXTime(Elem());

    return rest;
}

static auto_ptr<GUSIAAlarm> sGUSIAAlarm;

alarm raises a SIGALRM after a specified number of seconds has elapsed.
```

```
(POSIX functions for signal handling 635)+≡ (609) <646 648>
unsigned int alarm(unsigned int delay)
{
    unsigned int togo = 0;

    GUSIAAlarm * a = sGUSIAAlarm.get();
    if (a)
        togo = static_cast<unsigned int>(a->Restart() / 1000000);
    else
        sGUSIAAlarm = auto_ptr<GUSIAAlarm>(a = new GUSIAAlarm);
    if (a && delay)
        a->Sleep(delay*1000);

    return togo;
}
```

ualarm provides a finer resolution and optionally offers the possibility to generate repeated signals.

(POSIX functions for signal handling 635) +≡ (609) <647

```
useconds_t ualarm(useconds_t delay, useconds_t interval)
{
    useconds_t togo = 0;

    GUSIAalarm * a = sGUSIAalarm.get();
    if (a)
        togo = static_cast<useconds_t>(a->Restart(static_cast<long>(interval)));
    else
        sGUSIAalarm = auto_ptr<GUSIAalarm>(a = new GUSIAalarm(interval));
    if (a && delay)
        a->MicroSleep(delay);

    return togo;
}
```


Part III

Domain Specific Code

Chapter 17

Mixin Classes for Sockets

This section contains some building block classes for sockets:

- GUSISMBlocking implements the blocking/nonblocking flag.
- GUSISMState implements a state variable.
- GUSISMinputBuffer provides a GUSIBuffer for input.
- GUSISMOutputBuffer provides a GUSIBuffer for output.
- GUSISMAsyncError provides storage for asynchronous errors.

```
<GUSISocketMixins.h 649>≡
#ifndef _GUSISocketMixins_
#define _GUSISocketMixins_

#ifdef GUSI_INTERNAL

#include "GUSISocket.h"
#include "GUSIBuffer.h"

#include <fcntl.h>
#include <sys/ioctl.h>

<Definition of class GUSISMBlocking 651>
<Definition of class GUSISMState 652>
<Definition of class GUSISMinputBuffer 653>
<Definition of class GUSISMOutputBuffer 654>
<Definition of class GUSISMAsyncError 655>

<Inline member functions for class GUSISMBlocking 656>
<Inline member functions for class GUSISMState 658>
<Inline member functions for class GUSISMinputBuffer 660>
<Inline member functions for class GUSISMOutputBuffer 664>
<Inline member functions for class GUSISMAsyncError 667>

#endif /* GUSI_INTERNAL */

#endif /* _GUSISocketMixins_ */
```

Almost all member functions are inline, so this file is mostly a placeholder.

```
<GUSISocketMixins.cp 650>≡  
#include "GUSIInternal.h"  
#include "GUSISocketMixins.h"
```

17.1 Definition of GUSISocketMixins

GUSISMBlocking implements the fBlocking flags and the DoIoctl() and DoFcntl() variants to manipulate it. These two functions work like their GUSISocket member function counterparts, but handle the return value differently: The POSIX function result is stored in *result, while the return value indicates whether the request was handled.

```
<Definition of class GUSISMBlocking 651>≡ (649)  
class GUSISMBlocking {  
public:  
    GUSISMBlocking();  
    bool    fBlocking;  
    bool    DoFcntl(int * result, int cmd, va_list arg);  
    bool    DoIoctl(int * result, unsigned int request, va_list arg);  
};
```

GUSISMState captures the state of a socket over its lifetime. It starts out as Unbound. bind() will put it into Unconnected state, though few socket classes care about this distinction. listen() will put it into Listening state. accept() starts a Connected new socket. connect() puts an Unconnected socket into Connecting state from where it emerges Connected. fReadShutdown and fWriteShutdown record shutdown promises.

```
<Definition of class GUSISMState 652>≡ (649)  
class GUSISMState {  
public:  
    enum State {  
        Unbound,  
        Unconnected,  
        Listening,  
        Connecting,  
        Connected,  
        Closing  
    };  
    GUSISMState();  
    State    fState;  
    bool    fReadShutdown;  
    bool    fWriteShutdown;  
    void    Shutdown(int how);  
};
```

GUSISMSInputBuffer defines the input buffer and some socket options that go with it. DoGetSockOpt() and DoSetSockOpt() work the same way as DoFcntl() and DoIoctl() above.

```
(Definition of class GUSISMSInputBuffer 653)≡ (649)
class GUSISMSInputBuffer {
public:
    GUSIRingBuffer fInputBuffer;
    GUSISMSInputBuffer();
    bool          DoGetSockOpt(
        int * result, int level, int optname,
        void *optval, socklen_t * optlen);
    bool          DoSetSockOpt(
        int * result, int level, int optname,
        void *optval, socklen_t optlen);
    bool          DoIoctl(int * result, unsigned int request, va_list arg);
};
```

GUSISMSOutputBuffer defines the output buffer and some socket options that go with it.

```
(Definition of class GUSISMSOutputBuffer 654)≡ (649)
class GUSISMSOutputBuffer {
public:
    GUSIRingBuffer fOutputBuffer;
    GUSISMSOutputBuffer();
    bool          DoGetSockOpt(
        int * result, int level, int optname,
        void *optval, socklen_t * optlen);
    bool          DoSetSockOpt(
        int * result, int level, int optname,
        void *optval, socklen_t optlen);
};
```

GUSISMASyncError stores asynchronous errors and makes them available via getsockopt. GetAsyncError returns the error and resets the stored value.

```
(Definition of class GUSISMASyncError 655)≡ (649)
class GUSISMASyncError {
public:
    GUSISMASyncError();
    int          fAsyncError;
    int          SetAsyncPosixError(int error);
    int          SetAsyncMacError(OSErr error);
    int          GetAsyncError();
    bool          DoGetSockOpt(
        int * result, int level, int optname,
        void *optval, socklen_t * optlen);
};
```

17.2 Implementation of GUSISocketMixins

Because all the member functions are simple and called in few places, it makes sense to inline them.

All sockets start out blocking.

```
(Inline member functions for class GUSISMBlocking 656)≡ (649) 657▶  
inline GUSISMBlocking::GUSISMBlocking() : fBlocking(true) {}
```

For historical reasons, there is both an `ioctl()` and a `fcntl()` interface to the blocking flag.

```
(Inline member functions for class GUSISMBlocking 656)+≡ (649) ◀656  
inline bool GUSISMBlocking::DoFcntl(int * result, int cmd, va_list arg)  
{  
    switch(cmd) {  
    case F_GETFL :  
        return (*result = fBlocking ? 0: FNDELAY), true;  
    case F_SETFL :  
        fBlocking = !(va_arg(arg, int) & FNDELAY);  
  
        return (*result = 0), true;  
    }  
    return false;  
}  
inline bool GUSISMBlocking::DoIoctl(int * result, unsigned int request, va_list arg)  
{  
    if (request == FIONBIO) {  
        fBlocking = !*va_arg(arg, int *);  
        return (*result = 0), true;  
    }  
    return false;  
}
```

Sockets start out as unconnected.

```
(Inline member functions for class GUSISMState 658)≡ (649) 659▶  
inline GUSISMState::GUSISMState() :  
    fState(Unbound), fReadShutdown(false), fWriteShutdown(false) {}
```

```
(Inline member functions for class GUSISMState 658)+≡ (649) ◀658  
inline void GUSISMState::Shutdown(int how)  
{  
    if (!(how & 1))  
        fReadShutdown = true;  
    if (how > 0)  
        fWriteShutdown = true;  
}
```

Buffers initially are 8K.

```
(Inline member functions for class GUSISMInputBuffer 660)≡ (649) 661▶  
inline GUSISMInputBuffer::GUSISMInputBuffer() : fInputBuffer(8192) {}
```

getsockopt() is used to obtain the buffer size.

(Inline member functions for class GUSISMSInputBuffer 660)+≡ (649) <660 662>

```
inline bool GUSISMSInputBuffer::DoGetSockOpt(
    int * result, int level, int optname,
    void *optval, socklen_t *)
{
    if (level == SOL_SOCKET && optname == SO_RCVBUF) {
        *(int *)optval = (int)fInputBuffer.Size();

        return (*result = 0), true;
    }
    return false;
}
```

setsockopt() modifies the buffer size.

(Inline member functions for class GUSISMSInputBuffer 660)+≡ (649) <661 663>

```
inline bool GUSISMSInputBuffer::DoSetSockOpt(
    int * result, int level, int optname,
    void *optval, socklen_t)
{
    if (level == SOL_SOCKET && optname == SO_RCVBUF) {
        fInputBuffer.SwitchBuffer(*(int *)optval);

        return (*result = 0), true;
    }
    return false;
}
```

ioctl() returns the number of available bytes.

(Inline member functions for class GUSISMSInputBuffer 660)+≡ (649) <662

```
inline bool GUSISMSInputBuffer::DoIoctl(int * result, unsigned int request, va_list arg)
{
    if (request == FIONREAD) {
        *va_arg(arg, long *) = fInputBuffer.Valid();
        return (*result = 0), true;
    }
    return false;
}
```

GUSISMSOutputBuffer works identically to the input buffer.

(Inline member functions for class GUSISMSOutputBuffer 664)≡ (649) 665>

```
inline GUSISMSOutputBuffer::GUSISMSOutputBuffer() : fOutputBuffer(8192) {}
```

getsockopt() is used to obtain the buffer size.

(Inline member functions for class GUSISMSOutputBuffer 664)+≡ (649) <664 666>

```
inline bool GUSISMSOutputBuffer::DoGetSockOpt(
    int * result, int level, int optname,
    void *optval, socklen_t *)
{
    if (level == SOL_SOCKET && optname == SO_SNDBUF) {
        *(int *)optval = (int)fOutputBuffer.Size();

        return (*result = 0), true;
    }
    return false;
}
```

setsockopt() is modifies the buffer size.

(Inline member functions for class GUSISMSOutputBuffer 664)+≡ (649) <665

```
inline bool GUSISMSOutputBuffer::DoSetSockOpt(
    int * result, int level, int optname,
    void *optval, socklen_t)
{
    if (level == SOL_SOCKET && optname == SO_SNDBUF) {
        fOutputBuffer.SwitchBuffer(*(int *)optval);

        return (*result = 0), true;
    }
    return false;
}
```

(Inline member functions for class GUSISMASyncError 667)≡ (649) 668»

```
inline GUSISMASyncError::GUSISMASyncError()
: fAsyncError(0)
{
}
```

The central member functions of GUSISMASyncError are SetAsyncXXXError and GetAsyncError.

(Inline member functions for class GUSISMASyncError 667)+≡ (649) <667 669»

```
inline int GUSISMASyncError::SetAsyncPosixError(int error)
{
    if (error) {
        fAsyncError = error;

        return -1;
    }
    return 0;
}
inline int GUSISMASyncError::GetAsyncError()
{
    int err = fAsyncError;
    fAsyncError = 0;
    return err;
}
```

For some reason, the CW Pro 4 compilers generated bad code for this in some combination, so we make it out of line.

(Inline member functions for class GUSISMASyncError 667)+≡ (649) <668 670»

```
inline int GUSISMASyncError::SetAsyncMacError(OSErr error)
{
    if (error) {
        fAsyncError = GUSIMapMacError(error);

        return -1;
    }
    return 0;
}
```

DoGetSockOpt only handles SO_ERROR (hi Philippe!).

```
<Inline member functions for class GUSISMAsyncError 667>+≡ (649) <669
inline bool GUSISMAsyncError::DoGetSockOpt(
    int * result, int level, int optname,
    void *optval, socklen_t * optlen)
{
    if (level == SOL_SOCKET && optname == SO_ERROR) {
        *(int *)optval = GetAsyncError();
        *optlen        = sizeof(int);

        return (*result = 0), true;
    }
    return false;
}
```


Chapter 18

Null device

A `GUSINullSocket` implements the null socket class for MacTCP. All instances of `GUSINullSocket` are created by the `GUSINullDevice` singleton, so there is no point in exporting the class itself.

```
<GUSINull.h 671>≡
    #ifndef _GUSINull_
    #define _GUSINull_

    #ifdef GUSI_INTERNAL

    #include "GUSIDevice.h"

    <Definition of class GUSINullDevice 673>

    <Inline member functions for class GUSINullDevice 677>

    #endif /* GUSI_INTERNAL */

    #endif /* _GUSINull_ */

<GUSINull.cp 672>≡
    #include "GUSIInternal.h"
    #include "GUSINull.h"
    #include "GUSIBasics.h"
    #include "GUSIDdiag.h"

    #include <fcntl.h>
    #include <stddef.h>
    #include <sys/stat.h>

    #include <Devices.h>

    <Definition of class GUSINullSocket 674>
    <Member functions for class GUSINullDevice 675>
    <Member functions for class GUSINullSocket 681>
```

18.1 Definition of GUSINullDevice

GUSINullDevice is a singleton subclass of GUSIDevice.

<Definition of class GUSINullDevice 673>≡ (671)

```
class GUSINullDevice : public GUSIDevice {
public:
    static GUSINullDevice * Instance();
    virtual bool    Want(GUSIFileToken & file);
    virtual GUSISocket * open(GUSIFileToken & file, int flags);
    virtual int stat(GUSIFileToken & file, struct stat * buf);
    GUSISocket * open();
protected:
    GUSINullDevice() {}
    static GUSINullDevice * sInstance;
};
```

18.2 Definition of GUSINullSocket

<Definition of class GUSINullSocket 674>≡ (672)

```
class GUSINullSocket :
    public GUSISocket
{
public:
    GUSINullSocket();
    <Overridden member functions for GUSINullSocket 682>
};
```

18.3 Implementation of GUSINullDevice

You can use GUSINullSockets directly from C++, but the usual way to use them is to call GUSIwithNullSockets to have "Dev:Null" mapped to them.

<Member functions for class GUSINullDevice 675>≡ (672) 676»

```
extern "C" void GUSIwithNullSockets()
{
    GUSIDeviceRegistry::Instance()->AddDevice(GUSINullDevice::Instance());
}
```

<Member functions for class GUSINullDevice 675>+≡ (672) <675 678»

```
GUSINullDevice * GUSINullDevice::sInstance = nil;
```

<Inline member functions for class GUSINullDevice 677>≡ (671)

```
inline GUSINullDevice * GUSINullDevice::Instance()
{
    if (!sInstance)
        sInstance = new GUSINullDevice;
    return sInstance;
}
```

GUSINullDevice will handle only the open request.

```
(Member functions for class GUSINullDevice 675)+≡ (672) <676 679>
bool GUSINullDevice::Want(GUSIFileToken & file)
{
    const char * path = file.Path();

    switch (file.WhichRequest()) {
    case GUSIFileToken::kWillOpen:
    case GUSIFileToken::kWillStat:
        return file.StrFragEqual(path+4, "null") && !path[8];
    default:
        return false;
    }
}
```

Open will never fail except for lack of memory.

```
(Member functions for class GUSINullDevice 675)+≡ (672) <678 680>
GUSISocket * GUSINullDevice::open(GUSIFileToken &, int)
{
    GUSISocket * sock = new GUSINullSocket;
    if (!sock)
        GUSISetPosixError(ENOMEM);
    return sock;
}

GUSISocket * GUSINullDevice::open()
{
    GUSISocket * sock = new GUSINullSocket;
    if (!sock)
        GUSISetPosixError(ENOMEM);
    return sock;
}
```

The only original part o a null socket's reply to stat and fstat is the device/inode.

```
<Member functions for class GUSINullDevice 675>+≡                                (672) <679
static int GUSINullStat(struct stat * buf)
{
    buf->st_dev      =   'dev ' ;
    buf->st_ino      =   'null' ;
    buf->st_mode      =   S_IFCHR | 0666 ;
    buf->st_nlink     =   1;
    buf->st_uid       =   0;
    buf->st_gid       =   0;
    buf->st_rdev      =   0;
    buf->st_size      =   1;
    buf->st_atime     =   time(NULL);
    buf->st_mtime     =   time(NULL);
    buf->st_ctime     =   time(NULL);
    buf->st_blksize   =   0;
    buf->st_blocks    =   1;

    return 0;
}

int GUSINullDevice::stat(GUSIFileToken &, struct stat * buf)
{
    return GUSINullStat(buf);
}
```

18.4 Implementation of GUSINullSocket

The implementation of GUSINullSocket is trivial.

```
<Member functions for class GUSINullSocket 681>≡                                (672) 683»
GUSINullSocket::GUSINullSocket()
{
}
```

Read always returns EOF.

```
<Overridden member functions for GUSINullSocket 682>≡                                (674) 684»
virtual ssize_t read(const GUSIScatterer & buffer);
```

```
<Member functions for class GUSINullSocket 681>+≡                                (672) <681 685»
ssize_t GUSINullSocket::read(const GUSIScatterer &)
{
    return 0;
}
```

Writes always succeed.

```
<Overridden member functions for GUSINullSocket 682>+≡                                (674) <682 686»
virtual ssize_t write(const GUSIGatherer & buffer);
```

```
<Member functions for class GUSINullSocket 681>+≡                                (672) <683 687»
ssize_t GUSINullSocket::write(const GUSIGatherer & buffer)
{
    return buffer.Length();
}
```

Stat returns our special stat block.

```
⟨Overridden member functions for GUSINullSocket 682⟩+≡ (674) ◀684 688▶  
virtual int fstat(struct stat * buf);
```

```
⟨Member functions for class GUSINullSocket 681⟩+≡ (672) ◀685 689▶  
int GUSINullSocket::fstat(struct stat * buf)  
{  
    return GUSINullStat(buf);  
}
```

Null sockets implement simple calls.

```
⟨Overridden member functions for GUSINullSocket 682⟩+≡ (674) ◀686  
virtual bool Supports(ConfigOption config);
```

```
⟨Member functions for class GUSINullSocket 681⟩+≡ (672) ◀687  
bool GUSINullSocket::Supports(ConfigOption config)  
{  
    return config == kSimpleCalls;  
}
```


Chapter 19

The GUSI Pipe Socket Class

Pipes and socket pairs are implemented with the `GUSIPipeSocket` class. The `GUSIPipeFactory` singleton creates pairs of `GUSIPipeSockets`.

```
<GUSIPipe.h 690>≡
    #ifndef _GUSIPipe_
    #define _GUSIPipe_

    #ifdef GUSI_INTERNAL

    #include "GUSISocket.h"
    #include "GUSIFactory.h"

    <Definition of class GUSIPipeFactory 692>

    <Inline member functions for class GUSIPipeFactory 694>

    #endif /* GUSI_INTERNAL */

    #endif /* _GUSIPipe_ */

<GUSIPipe.cp 691>≡
    #include "GUSIInternal.h"
    #include "GUSIPipe.h"
    #include "GUSIBasics.h"
    #include "GUSIBuffer.h"

    #include <errno.h>

    <Definition of class GUSIPipeSocket 697>
    <Inline member functions for class GUSIPipeSocket 701>
    <Member functions for class GUSIPipeFactory 693>
    <Member functions for class GUSIPipeSocket 700>
```

19.1 Definition of GUSIPipeFactory

GUSIPipeFactory is a singleton subclass of GUSISocketFactory.

```
<Definition of class GUSIPipeFactory 692>≡ (690)
class GUSIPipeFactory : public GUSISocketFactory {
public:
    static GUSISocketFactory * Instance();
    virtual GUSISocket * socket(int domain, int type, int protocol);
    virtual int socketpair(int domain, int type, int protocol, GUSISocket * s[2]);
private:
    GUSIPipeFactory() {}
    static GUSISocketFactory * sInstance;
};
```

19.2 Implementation of GUSIPipeFactory

```
<Member functions for class GUSIPipeFactory 693>≡ (691) 695▷
GUSISocketFactory * GUSIPipeFactory::sInstance = nil;
```

```
<Inline member functions for class GUSIPipeFactory 694>≡ (690)
inline GUSISocketFactory * GUSIPipeFactory::Instance()
{
    if (!sInstance)
        sInstance = new GUSIPipeFactory;
    return sInstance;
}
```

GUSIPipeFactory is odd in that socket is not meaningful.

```
<Member functions for class GUSIPipeFactory 693>+≡ (691) ◀693 696▷
GUSISocket * GUSIPipeFactory::socket(int, int, int)
{
    return GUSISetPosixError(EOPNOTSUPP), static_cast<GUSISocket *>(nil);
}
```


socketpair is meaningful, however.

```

<Member functions for class GUSIPipeFactory 693>+≡ (691) <695
int GUSIPipeFactory::socketpair(int, int, int, GUSISocket * s[2])
{
    GUSIErrorSaver      saveError;
    GUSIPipeSocket *    sock[2];

    if (s[0] = sock[0] = new GUSIPipeSocket)
        if (s[1] = sock[1] = new GUSIPipeSocket) {
            sock[0]->SetPeer(sock[1]);
            sock[1]->SetPeer(sock[0]);

            return 0;
        } else
            delete s[0];

    if (!errno)
        return GUSISetPosixError(ENOMEM);
    else
        return -1;
}

```

19.3 Definition of GUSIPipeSocket

A GUSIPipeSocket is implemented with a simple GUSIBuffer.

```

<Definition of class GUSIPipeSocket 697>≡ (691)
class GUSIPipeSocket : public GUSISocket {
public:
    GUSIPipeSocket();
    virtual ~GUSIPipeSocket();
    <Overridden member functions for GUSIPipeSocket 704>
    <Peer management for GUSIPipeSocket 698>
protected:
    <Privatissima of GUSIPipeSocket 699>
};

```

Each GUSIPipeSocket has a peer which is set with SetPeer.

```

<Peer management for GUSIPipeSocket 698>≡ (697)
void SetPeer(GUSIPipeSocket * peer);

```

19.4 Implementation of GUSIPipeSocket

```

<Privatissima of GUSIPipeSocket 699>≡ (697) 702▷
GUSIRingBuffer fBuffer;
bool fWriteShutdown;
bool fBlocking;
GUSIPipeSocket * fPeer;

```

```

⟨Member functions for class GUSIPipeSocket 700⟩≡ (691) 703▸
    GUSIPipeSocket::GUSIPipeSocket()
        : fBuffer(8192), fWriteShutdown(false), fBlocking(true), fPeer(nil)
    {
    }

```

We don't have much to do with peers except setting them and waking them up.

```

⟨Inline member functions for class GUSIPipeSocket 701⟩≡ (691)
    inline void GUSIPipeSocket::SetPeer(GUSIPipeSocket * peer) { fPeer = peer; }

```

```

⟨Privatissima of GUSIPipeSocket 699⟩+≡ (697) ◀699 708▸
    void WakeupPeer();

```

```

⟨Member functions for class GUSIPipeSocket 700⟩+≡ (691) ◀700 705▸
    void GUSIPipeSocket::WakeupPeer()
    {
        if (fPeer)
            fPeer->Wakeup();
    }

```

Currently we only implement the simple calls.

```

⟨Overridden member functions for GUSIPipeSocket 704⟩≡ (697) 706▸
    virtual bool Supports(ConfigOption config);

```

```

⟨Member functions for class GUSIPipeSocket 700⟩+≡ (691) ◀703 707▸
    bool GUSIPipeSocket::Supports(ConfigOption config)
    {
        return config == kSimpleCalls;
    }

```

```

⟨Overridden member functions for GUSIPipeSocket 704⟩+≡ (697) ◀704 711▸
    virtual ssize_t read(const GUSIScatterer & buffer);

```

```

⟨Member functions for class GUSIPipeSocket 700⟩+≡ (691) ◀705 712▸
    ssize_t GUSIPipeSocket::read(const GUSIScatterer & buffer)
    {
        size_t len = buffer.Length();

        if (!fBuffer.Valid()) {
            ⟨No more data in GUSIPipeSocket, check for EOF and consider waiting 709⟩
        }
        fBuffer.Consume(buffer, len);
        WakeupPeer();

        return (int) len;
    }

```

Since there is currently no data, the simplest strategy won't work and we need something a bit more elaborate.

```

⟨Privatissima of GUSIPipeSocket 699⟩+≡ (697) ◀702 710▸
    bool Eof() { return !fPeer || fPeer->fWriteShutdown; }

```

```

<No more data in GUSIPipeSocket, check for EOF and consider waiting 709>≡ (707)
    if (Eof())
        return 0;
    if (!fBlocking)
        return GUSISetPosixError(EWOULDBLOCK);
    bool signal = false;
    AddContext();
    do {
        if (signal = GUSIContext::Yield(true))
            break;
    } while (!Eof() && !fBuffer.Valid());
    RemoveContext();
    if (signal)
        return GUSISetPosixError(EINTR);

<Privatissima of GUSIPipeSocket 699>+≡ (697) ◀708
    bool Shutdown() { return !fPeer || fWriteShutdown; }

<Overridden member functions for GUSIPipeSocket 704>+≡ (697) ◀706 714▷
    virtual ssize_t write(const GUSIGatherer & buffer);

<Member functions for class GUSIPipeSocket 700>+≡ (691) ◀707 715▷
    ssize_t GUSIPipeSocket::write(const GUSIGatherer & buffer)
    {
        size_t buflen = buffer.Length();
        size_t len;
        size_t offset = 0;

restart:
        if (Shutdown())
            return GUSISetPosixError(EPIPE);
        if (fPeer->fBuffer.Free() < buflen) {
            <Too much data in GUSIPipeSocket, consider writing in portions 713>
        }
        fPeer->fBuffer.Produce(buffer, buflen, offset);
        WakeupPeer();

        return (size_t) buffer.Length();
    }

```

This could get painful. We write a portion and then wait for free buffer space.

```

<Too much data in GUSIPipeSocket, consider writing in portions 713>≡ (712)
    if (!fBlocking && !fPeer->fBuffer.Free())
        return offset ? (int) offset : GUSISetPosixError(EWOULDBLOCK);
    len = buflen;
    fPeer->fBuffer.Produce(buffer, len, offset);
    buflen -= len;
    WakeupPeer();
    bool signal = false;
    AddContext();
    while (!Shutdown() && !fPeer->fBuffer.Free()) {
        if (signal = GUSIContext::Yield(true))
            break;
    }
    RemoveContext();
    if (signal)
        return offset ? (int) offset : GUSISetPosixError(EINTR);
    goto restart;

```

Of course, we also have `select()`. `canWrite` will be sort of unsatisfactory, since it says nothing about the size of `write()` possible.

```

<Overridden member functions for GUSIPipeSocket 704>+≡ (697) <711 717>
    virtual bool    select(bool * canRead, bool * canWrite, bool * exception);

<Member functions for class GUSIPipeSocket 700>+≡ (691) <712 716>
    bool GUSIPipeSocket::select(bool * canRead, bool * canWrite, bool *)
    {
        bool cond = false;
        if (canRead)
            if (*canRead = Eof() || fBuffer.Valid())
                cond = true;
        if (canWrite)
            if (*canWrite = Shutdown() || fPeer->fBuffer.Free())
                cond = true;

        return cond;
    }

```

When we're destroyed, we sever the links to our peer.

```

<Member functions for class GUSIPipeSocket 700>+≡ (691) <715 718>
    GUSIPipeSocket::~GUSIPipeSocket()
    {
        if (fPeer)
            fPeer->fPeer = nil;
        WakeupPeer();
    }

```

Some similar functions are necessary for `shutdown()`.

```

<Overridden member functions for GUSIPipeSocket 704>+≡ (697) <714
    virtual int shutdown(int how);

```

(Member functions for class GUSIPipeSocket 700)+≡

(691) ◀716

```
int GUSIPipeSocket::shutdown(int how)
{
    if (how)                                // write
        fWriteShutdown = true;
    if (!(how & 1))                          // read
        if (fPeer)
            fPeer->fWriteShutdown = true;

    return 0;
}
```


Chapter 20

TCP/IP shared infrastructure

Both the MacTCP and the forthcoming open transport implementation of TCP/IP sockets share a common registry.

```
(GUSIInet.h 719)≡
#ifndef _GUSIInet_
#define _GUSIInet_

#ifdef GUSI_SOURCE

#include <sys/cdefs.h>

__BEGIN_DECLS
(Definition of GUSIwithInetSockets 721)
__END_DECLS

#ifdef GUSI_INTERNAL

#include "GUSIFactory.h"

extern GUSISocketTypeRegistry gGUSIInetFactories;

#endif /* GUSI_INTERNAL */

#endif /* GUSI_SOURCE */

#endif /* _GUSIInet_ */
```

```

<GUSIInet.cp 720>≡
#include "GUSIInternal.h"
#include "GUSIInet.h"
#include "GUSIMTInet.h"
#include "GUSIOpenTransport.h"
#include "GUSIOTInet.h"

#include <sys/types.h>
#include <sys/socket.h>

GUSISocketTypeRegistry gGUSIInetFactories(AF_INET, 8);

<Implementation of GUSIwithInetSockets 722>

<Definition of GUSIwithInetSockets 721>≡ (719)
void GUSIwithInetSockets();

<Implementation of GUSIwithInetSockets 722>≡ (720)
void GUSIwithInetSockets()
{
    if (GUSIOTFactory::Initialize())
        GUSIwithOTInetSockets();
    else
        GUSIwithMTInetSockets();
}

```


Chapter 21

Converting Between Names and IP Addresses

The GUSINetDB class coordinates access to the domain name server database.

The GUSIServiceDB class is responsible for a database of TCP/IP service name to port number mappings.

The hostent and servent classes are somewhat inconvenient to set up as they reference extra chunks of memory, so we define the wrapper classes GUSIhostent and GUSIservent.

```
<GUSINetDB.h 723>≡
#ifdef _GUSINetDB_
#define _GUSINetDB_

#ifdef GUSI_SOURCE
#include "GUSISpecific.h"

#include <sys/types.h>
#include <netdb.h>
#include <arpa/inet.h>

<Definition of class GUSIhostent 733>
<Definition of class GUSIservent 734>
<Definition of class GUSIServiceDB 730>
<Definition of class GUSINetDB 725>

#ifdef GUSI_INTERNAL

<Inline member functions for class GUSIServiceDB 752>

#endif /* GUSI_INTERNAL */

#endif /* GUSI_SOURCE */

#endif /* _GUSINetDB_ */
```

```

<GUSINetDB.cp 724>≡
#include "GUSInternal.h"
#include "GUSINetDB.h"
#include "GUSIFileSpec.h"
#include "GUSIFactory.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

#include <Resources.h>
#include <Memory.h>

<Member functions for class GUSINetDB 736>
<Member functions for class GUSIServiceDB 750>
<Member functions for class GUSIhostent 760>
<Member functions for class GUSIservent 761>

```

21.1 Definition of GUSINetDB

```

<Definition of class GUSINetDB 725>≡ (723)
class GUSINetDB {
public:
    <Constructing instances of GUSINetDB 726>
    <GUSINetDB host database 727>
    <GUSINetDB service database 728>
    <GUSINetDB protocol database 729>
protected:
    GUSINetDB();
    virtual ~GUSINetDB() {}
    <Privatissima of GUSINetDB 735>
};

```

GUSINetDB is a singleton, but usually instantiated by an instance of a derived class.

```

<Constructing instances of GUSINetDB 726>≡ (725)
static GUSINetDB * Instance();

```

The public interface of GUSINetDB consists of three areas. The first set of calls is concerned with host names and IP numbers.

```

<GUSINetDB host database 727>≡ (725)
virtual hostent * gethostbyname(const char * name);
virtual hostent * gethostbyaddr(const void * addr, size_t len, int type);
virtual char * inet_ntoa(in_addr inaddr);
virtual in_addr_t inet_addr(const char *address);
virtual long gethostid();
virtual int gethostname(char *machname, int buflen);

```

The next set of calls is concerned with TCP and UDP services.

```

<GUSINetDB service database 728>≡ (725)
virtual servent *  getservbyname(const char * name, const char * proto);
virtual servent *  getservbyport(int port, const char * proto);
virtual servent *  getservent();
virtual void       setservent(int stayopen);
virtual void       endservent();

```

Finally, there is a set of calls concerned with protocols.

```

<GUSINetDB protocol database 729>≡ (725)
virtual protoent * getprotobyname(const char * name);
virtual protoent * getprotobyname(int proto);
virtual protoent * getprotoent();
virtual void       setprotoent(int stayopen);
virtual void       endprotoent();

```

21.2 Definition of GUSIServiceDB

GUSIServiceDB is a singleton, used as a primitive iterator. The semantics of these iterators conform only very superficially to real iterators:

- Only a single instance of the iterator is supported.
- Comparison operators all compare against `end()`, no matter what arguments are passed.

```

<Definition of class GUSIServiceDB 730>≡ (723)
extern "C" void GUSIKillServiceDBData(void * entry);

class GUSIServiceDB {
public:
    static GUSIServiceDB * Instance();
    <Iterating over the GUSIServiceDB 731>
protected:
    static GUSIServiceDB * sInstance;
                                GUSIServiceDB()    {}
                                ~GUSIServiceDB()    {}

    virtual

    friend void GUSIKillServiceDBData(void * entry);

    <Internal iterator protocol of GUSIServiceDB 732>
};

```

Iterating is accomplished by a public interface conforming to STL iterator protocols.

```

<Iterating over the GUSIServiceDB 731>≡ (730)
class iterator {
public:
    inline bool      operator==(const iterator & other);
    inline bool      operator!=(const iterator & other);
    inline iterator & operator++;
    inline servent * operator*();
};

inline static iterator begin();
inline static iterator end();

```

This interface does not access any data elements in the iterator, but directly calls through to a private interface in the GUSIServiceDB, which explains the limitations in the iterator implementation.

```

<Internal iterator protocol of GUSIServiceDB 732>≡ (730)
    friend class iterator;

    class Data {
    public:
        Data() : fCurrent(0) {}

        servent *    fCurrent;
        GUSIservent fServent;
    };
    static GUSISpecificData<Data, GUSIKillServiceDBData>    sData;

    virtual void    Reset() = 0;
    virtual void    Next()  = 0;

```

21.3 Definition of GUSIhostent and GUSIservent

A GUSIhostent may need a lot of data, so we allocate the name data dynamically.

```

<Definition of class GUSIhostent 733>≡ (723)
    class GUSIhostent : public hostent {
    public:
        GUSIhostent();

        void    Alloc(size_t size);

        char *  fAlias[16];
        char *  fAddressList[16];
        char *  fName;
        size_t  fAlloc;
        char    fAddrString[16];
    };

    extern "C" void GUSIKillHostEnt(void * hostent);

```

A GUSIservent typically will remain more modest in its needs, so the data is allocated statically.

```

<Definition of class GUSIservent 734>≡ (723)
    class GUSIservent : public servent {
    public:
        GUSIservent();

        char *  fAlias[8];
        char    fName[256];
    };

```

21.4 Implementation of GUSINetDB

GUSINetDB is a singleton, but typically implemented by an instance of a subclass (stored into `fInstance` by that subclass) rather than the base class.

```
<Privatissima of GUSINetDB 735>≡ (725) 741▷
    static GUSINetDB *      sInstance;

<Member functions for class GUSINetDB 736>≡ (724) 737▷

GUSINetDB *  GUSINetDB::Instance()
{
    if (!sInstance) {
        GUSISocketDomainRegistry::Instance();
        if (!sInstance)
            sInstance = new GUSINetDB();
    }
    return sInstance;
}
GUSINetDB::GUSINetDB()
{
    GUSIContext::Setup(false);

    <Initialize fields of GUSINetDB 742>
}
```

The host functions are not implemented unless the programmer adds support for either MacTCP or OpenTransport net database support.

```

<Member functions for class GUSINetDB 736>+≡ (724) <736 738>
    hostent * GUSINetDB::gethostbyname(const char *)
    {
        return (h_errno = NO_RECOVERY), static_cast<hostent *>(nil);
    }
    hostent * GUSINetDB::gethostbyaddr(const void *, size_t, int)
    {
        return (h_errno = NO_RECOVERY), static_cast<hostent *>(nil);
    }
    char * GUSINetDB::inet_ntoa(in_addr)
    {
        return (h_errno = NO_RECOVERY), static_cast<char *>(nil);
    }
    in_addr_t GUSINetDB::inet_addr(const char * address)
    {
        int a[4];
        if (sscanf(address, "%d.%d.%d.%d", a, a+1, a+2, a+3) != 4)
            return static_cast<in_addr_t>(GUSISetHostError(NO_RECOVERY));
        else if ((a[0] & 0xFFFFFFFF00) || (a[1] & 0xFFFFFFFF00)
                || (a[2] & 0xFFFFFFFF00) || (a[3] & 0xFFFFFFFF00))
            return static_cast<in_addr_t>(GUSISetHostError(NO_RECOVERY));
        else
            return (in_addr_t)((a[0] << 24) | (a[1] << 16) | (a[2] << 8) | a[3]);
    }
    long GUSINetDB::gethostid()
    {
        return 0;
    }

```

Unlike the other functions, gethostname is defined entirely in terms of the other functions and thus network independent.

```

<Member functions for class GUSINetDB 736>+≡ (724) <737 743>
    int GUSINetDB::gethostname(char *machname, int buflen)
    {
        static char * sHostName = nil;

        if (!sHostName) {
            in_addr ipaddr;
            if (!(ipaddr.s_addr = static_cast<in_addr_t>(gethostid()))) {
                <No TCP/IP support, determine host name from chooser 739>
            } else {
                <Translate host IP number to host name 740>
            }
        }
        strncpy(machname, sHostName, unsigned(buflen));
        machname[buflen-1] = 0; /* extra safeguard */

        return 0;
    }

```

If there is no TCP/IP support, we still can get the system's idea of what the Mac is named.

```

<No TCP/IP support, determine host name from chooser 739>≡ (738)
    Handle  hostString  = GetResource('STR ', -16413);
    char    hsState     = HGetState(hostString);
    HLock(hostString);
    sHostName = new char[**hostString+1];
    memcpy(sHostName, *hostString, **hostString);
    sHostName[**hostString] = 0;
    HSetState(hostString, hsState);

<Translate host IP number to host name 740>≡ (738)
    char * name = inet_ntoa(ipaddr);
    sHostName = new char[strlen(name)+1];
    strcpy(sHostName, name);

```

The service database is implemented in terms of GUSIServiceDB. Only getservent and setservent accesses GUSIServiceDB directly, however.

```

<Privatissima of GUSINetDB 735>+≡ (725) ◀735 745▶
    bool                fServiceOpen;
    GUSIServiceDB::iterator fServiceIter;

<Initialize fields of GUSINetDB 742>≡ (736) 746▶
    fServiceOpen = false;

<Member functions for class GUSINetDB 736>+≡ (724) ◀738 744▶
    servent * GUSINetDB::getservent()
    {
        if (!fServiceOpen)
            setservent(0);
        else
            ++fServiceIter;
        if (fServiceIter == GUSIServiceDB::end())
            endservent();
        return *fServiceIter;
    }
    void GUSINetDB::setservent(int)
    {
        fServiceIter = GUSIServiceDB::begin();
        fServiceOpen = true;
    }
    void GUSINetDB::endservent()
    {
        fServiceOpen = false;
    }

```


Under these circumstances, the iterator functions reduce to triviality.

```

<Member functions for class GUSINetDB 736>+≡ (724) ◀747 749▷
protoent * GUSINetDB::getprotoent()
{
    fNextProtocol = !fNextProtocol;
    return sProtocols+!fNextProtocol;
}
void GUSINetDB::setprotoent(int)
{
    fNextProtocol = 0;
}
void GUSINetDB::endprotoent()
{
    fNextProtocol = 0;
}

<Member functions for class GUSINetDB 736>+≡ (724) ◀748
protoent * GUSINetDB::getprotobyname(const char * name)
{
    for (int i = 0; i<2; ++i)
        if (!strcmp(name, sProtocols[i].p_name))
            return sProtocols+i;
    return static_cast<protoent *>(nil);
}
protoent * GUSINetDB::getprotobyname(int proto)
{
    for (int i = 0; i<2; ++i)
        if (proto == sProtocols[i].p_proto)
            return sProtocols+i;
    return static_cast<protoent *>(nil);
}

```

21.5 Implementation of GUSIServiceDB

GUSIServiceDB is a singleton which is always instantiated by an instance of a derived class. The derived classes GUSIFileServiceDB and GUSIBuiltinServiceDB are defined here, but others are possible as well.

```

<Member functions for class GUSIServiceDB 750>≡ (724) 751▷
  <Definition of class GUSIFileServiceDB 756>
  <Definition of class GUSIBuiltinServiceDB 753>

  <Member functions for class GUSIFileServiceDB 757>
  <Member functions for class GUSIBuiltinServiceDB 754>

```

By default, we try to construct a GUSIFileServiceDB and fall back to the GUSIBuiltinServiceDB if that fails.

```
(Member functions for class GUSIServiceDB 750) +≡ (724) 4750
GUSISpecificData<GUSIServiceDB::Data, GUSIKillServiceDBData> GUSIServiceDB::sData;

void GUSIKillServiceDBData(void * data)
{
    delete reinterpret_cast<GUSIServiceDB::Data *>(data);
}

GUSIServiceDB * GUSIServiceDB::sInstance;

GUSIServiceDB * GUSIServiceDB::Instance()
{
    if (!sInstance)
        if (!(sInstance = GUSIFileServiceDB::Instance()))
            sInstance = GUSIBuiltinServiceDB::Instance();
    return sInstance;
}
```

Iterators can be defined without regard to the implementation of the GUSIServiceDB currently used.

```
(Inline member functions for class GUSIServiceDB 752) ≡ (723)
GUSIServiceDB::iterator GUSIServiceDB::begin()
{
    Instance()->Reset();
    Instance()->Next();

    return iterator();
}
GUSIServiceDB::iterator GUSIServiceDB::end()
{
    return iterator();
}
bool GUSIServiceDB::iterator::operator==(const GUSIServiceDB::iterator &)
{
    return !GUSIServiceDB::sData->fCurrent;
}
bool GUSIServiceDB::iterator::operator!=(const GUSIServiceDB::iterator &)
{
    return GUSIServiceDB::sData->fCurrent
        == static_cast<servent *>(nil);
}
GUSIServiceDB::iterator & GUSIServiceDB::iterator::operator++()
{
    GUSIServiceDB::Instance()->Next();
    return *this;
}
servent * GUSIServiceDB::iterator::operator*()
{
    return GUSIServiceDB::sData->fCurrent;
}
```

GUSIBuiltinServiceDB is the most simple implementation of the service database, based on a compiled-in table of the most important services.

(Definition of class GUSIBuiltinServiceDB 753)≡

(750)

```
extern "C" void GUSIKillBuiltinServiceDBEntry(void * entry);

class GUSIBuiltinServiceDB : public GUSIServiceDB {
public:
    static GUSIBuiltinServiceDB * Instance() { return new GUSIBuiltinServiceDB; }
protected:
    friend void GUSIKillBuiltinServiceDBEntry(void * entry);

    struct Entry {
        Entry() : fValue(0) {}

        int fValue;
    };
    static GUSISpecificData<Entry, GUSIKillBuiltinServiceDBEntry> sEntry;

    virtual void    Reset();
    virtual void    Next();

    static servant  sServices[];
};
```

(Member functions for class GUSIBuiltinServiceDB 754)≡

(750) 755»

GUSISpecificData<GUSIBuiltinServiceDB::Entry, GUSIKillBuiltinServiceDBEntry> GUSIBuiltinServiceDB::

```
void GUSIKillBuiltinServiceDBEntry(void * entry)
{
    delete reinterpret_cast<GUSIBuiltinServiceDB::Entry *>(entry);
}
```

```
servent GUSIBuiltinServiceDB::sServices[] =
```

```
{
    { "echo",      {NULL}, 7, "udp"},
    { "discard",   {NULL}, 9, "udp"},
    { "time",      {NULL}, 37, "udp"},
    { "domain",    {NULL}, 53, "udp"},
    { "sunrpc",     {NULL}, 111, "udp"},
    { "tftp",      {NULL}, 69, "udp"},
    { "biff",      {NULL}, 512, "udp"},
    { "who",       {NULL}, 513, "udp"},
    { "talk",      {NULL}, 517, "udp"},

    { "ftp-data",  {NULL}, 20, "tcp"},
    { "ftp",       {NULL}, 21, "tcp"},
    { "telnet",    {NULL}, 23, "tcp"},
    { "smtp",      {NULL}, 25, "tcp"},
    { "time",      {NULL}, 37, "tcp"},
    { "whois",     {NULL}, 43, "tcp"},
    { "domain",    {NULL}, 53, "tcp"},
    { "hostnames", {NULL}, 101, "tcp"},
    { "nntp",      {NULL}, 119, "tcp"},
    { "finger",    {NULL}, 79, "tcp"},
    { "http",      {NULL}, 80, "tcp"},
    { "ntp",       {NULL}, 123, "tcp"},
    { "uucp",      {NULL}, 540, "tcp"},
    { NULL, {NULL}, -1, NULL}
};
```

(Member functions for class GUSIBuiltinServiceDB 754)+≡

(750) «754

```
void GUSIBuiltinServiceDB::Reset()
{
    sEntry->fValue = 0;
}
void GUSIBuiltinServiceDB::Next()
{
    if (!sServices[sEntry->fValue].s_name)
        sData->fCurrent = static_cast<servent *>(nil);
    else
        sData->fCurrent = sServices + sEntry->fValue++;
}
```

GUSIFileServiceDB is the traditional implementation, based on reading a file from disk. This class is not 100% thread safe: The normal functions are OK, but getservent will not be safe.

(Definition of class GUSIFileServiceDB 756)≡ (750)

```
class GUSIFileServiceDB : public GUSIServiceDB {
public:
    static GUSIFileServiceDB * Instance();
protected:
    FILE *          fFile;
    pthread_mutex_t fLock;

    GUSIFileServiceDB(FILE * file)
        : fFile(file), fLock(0) { }

    virtual void    Reset();
    virtual void    Next();
};
```

Constructing a GUSIFileServiceDB can only succeed if the services file exists.

(Member functions for class GUSIFileServiceDB 757)≡ (750) 758▷

```
GUSIFileServiceDB * GUSIFileServiceDB::Instance()
{
    GUSIFileSpec services(kPreferencesFolderType, kOnSystemDisk);

    if (services.Error())
        return static_cast<GUSIFileServiceDB *>(nil);

    services.SetName("\p/etc/services");

    FILE * f = fopen(services.FullPath(), "r");

    return f ? new GUSIFileServiceDB(f) : static_cast<GUSIFileServiceDB *>(nil);
}
```

Reset simply rewinds the file.

(Member functions for class GUSIFileServiceDB 757)+≡ (750) ◀757 759▷

```
void GUSIFileServiceDB::Reset()
{
    rewind(fFile);
}
```

Next has to parse the lines of the file.

(*Member functions for class GUSIFileServiceDB* 757)+≡

(750) 4758

```
void GUSIFileServiceDB::Next()
{
    GUSIServent & service = sData->fServent;

    pthread_mutex_lock(&fLock);
    sData->fCurrent = static_cast<servent *>(nil);
    while (fgets(service.fName, 128, fFile)) {
        char * p;
        if (p = strpbrk(service.fName, "#\n\r")) // Line terminated at newline or '#'
            *p = 0;
        if (!(service.s_name = strtok(service.fName, " \t"))
            || !(p = strtok(NULL, " \t"))
            || !(service.s_proto = strpbrk(p, "/,")
        )
            continue;

        *service.s_proto++ = 0;
        service.s_port = atoi(p);

        int aliascount;
        for (aliascount = 0; aliascount < 15; )
            if (!(service.s_aliases[aliascount++] = strtok(NULL, " \t")))
                break;
        service.s_aliases[aliascount] = NULL;
        sData->fCurrent = &service;

        break;
    }
    pthread_mutex_unlock(&fLock);
}
```

21.6 Implementation of GUSIhostent and GUSIservent

The wrapper classes are rather simple. Note that a GUSIhostent never shrinks.

```
(Member functions for class GUSIhostent 760)≡ (724)
GUSIhostent::GUSIhostent()
: fName(nil), fAlloc(0)
{
    h_aliases = fAlias;
    h_addr_list = fAddressList;
}
void GUSIhostent::Alloc(size_t size)
{
    if (size > fAlloc) {
        if (fName)
            delete fName;
        h_name = fName = new char[fAlloc = size];
    }
}
void GUSIKillHostEnt(void * hostent)
{
    delete reinterpret_cast<GUSIhostent *>(hostent);
}

(Member functions for class GUSIservent 761)≡ (724)
GUSIservent::GUSIservent()
{
    s_aliases = fAlias;
}
```


Chapter 22

Basic MacTCP code

A `GUSIMTInetSocket` defines the infrastructure shared between MacTCP TCP and UDP sockets.

```
(GUSIMTInet.h 762)≡
#ifdef _GUSIMTInet_
#define _GUSIMTInet_

#ifdef GUSI_SOURCE

#include <sys/cdefs.h>

__BEGIN_DECLS
(Definition of GUSIwithMTInetSockets 790)
__END_DECLS

#ifdef GUSI_INTERNAL

#include "GUSISocket.h"
#include "GUSISocketMixins.h"

#include <netinet/in.h>
#include <MacTCP.h>

(Definition of class GUSIMTInetSocket 764)

#endif /* GUSI_INTERNAL */

#endif /* GUSI_SOURCE */

#endif /* _GUSIMTInet_ */
```

```

<GUSIMTInet.cp 763>≡
#include "GUSIInternal.h"
#include "GUSIMTInet.h"
#include "GUSIMTTcp.h"
#include "GUSIMTUdp.h"
#include "GUSIDiag.h"
#include "GUSIFSWrappers.h"

#include <stdlib.h>
#include <errno.h>
#include <algorithm>

#include <Devices.h>

<Member functions for class GUSIMTInetSocket 769>

```

22.1 Definition of GUSIMTInetSocket

MacTCP related sockets are buffered, have a standard state model, and can be non-blocking.

```

<Definition of class GUSIMTInetSocket 764>≡ (762)
class GUSIMTInetSocket :
    public      GUSISocket,
    protected   GUSISMBlocking,
    protected   GUSISMState,
    protected   GUSISMInputBuffer,
    protected   GUSISMOutputBuffer,
    protected   GUSISMAsyncError
{
public:
    GUSIMTInetSocket();
    <Overridden member functions for GUSIMTInetSocket 773>
    <Definition of classes MiniWDS and MidiWDS 766>
    <MacTCP driver management 767>
protected:
    <Data members for GUSIMTInetSocket 765>
};

```

All MacTCP related sockets need a StreamPtr; they store their own and their peer's address away, and the save errors reported at interrupt time in an fAsyncError field.

```

<Data members for GUSIMTInetSocket 765>≡ (764) 768▶
StreamPtr  fStream;
sockaddr_in fSockAddr;
sockaddr_in fPeerAddr;

```

MacTCP I/O calls communicate by means of read and write data structures, of which we need only the most primitive variants.

```

<Definition of classes MiniWDS and MidiWDS 766>≡ (764)
    #if PRAGMA_STRUCT_ALIGN
        #pragma options align=mac68k
    #endif
    class MiniWDS {
    public:
        u_short fLength;
        char * fDataPtr;
        u_short fZero;

        MiniWDS() : fZero(0) {}
        Ptr operator &() { return (Ptr)this; }
    };
    class MidiWDS {
    public:
        u_short fLength;
        char * fDataPtr;
        u_short fLength2;
        char * fDataPtr2;
        u_short fZero;

        MidiWDS() : fZero(0) {}
        Ptr operator &() { return (Ptr)this; }
    };
    #if PRAGMA_STRUCT_ALIGN
        #pragma options align=reset
    #endif

```

The only other interesting bit in the interface is the driver management, which arranges to open the MacTCP driver and domain name resolver at most once, as late as possible in the program (If you open some SLIP or PPP drivers before the Toolbox is initialized, you'll wish you'd curled up by the fireside with a nice Lovecraft novel instead). `Driver()` returns the driver reference number of the MacTCP driver. `HostAddr()` returns our host's IP address.

```

<MacTCP driver management 767>≡ (764)
    static short    Driver();
    static u_long   HostAddr();

```

22.2 Implementation of GUSIMTInetSocket

`Driver()` preserves error status in an `OSErr` variable, initially 1 to convey unresolved-ness.

```

<Data members for GUSIMTInetSocket 765>+≡ (764) <765
    static short    sDrvrRefNum;
    static OSErr    sDrvrState;
    static u_long   sHostAddress;

<Member functions for class GUSIMTInetSocket 769>≡ (763) 770▷
    short    GUSIMTInetSocket::sDrvrRefNum    = 0;
    OSErr    GUSIMTInetSocket::sDrvrState     = 1;
    u_long   GUSIMTInetSocket::sHostAddress   = 0;

```

Driver() opens the driver if necessary and stores its refnum.

<Member functions for class GUSIMTInetSocket 769>+≡ (763) <769 771>

```
short GUSIMTInetSocket::Driver()
{
    if (sDrvrState == 1)
        sDrvrState = GUSIFSOpenDriver("\p.IPP", &sDrvrRefNum);

    return sDrvrState ? 0 : sDrvrRefNum;
}
```

HostAddr() does an ipctlGetAddr control call if necessary.

<Member functions for class GUSIMTInetSocket 769>+≡ (763) <770 772>

```
u_long GUSIMTInetSocket::HostAddr()
{
    if (!sHostAddress && Driver()) {
        GUSIIOPBWrapper<GetAddrParamBlock> ga;

        ga->ioCRefNum = Driver();
        ga->csCode = ipctlGetAddr;

        if (!ga.Control())
            sHostAddress = ga->ourAddress;
    }
    return sHostAddress;
}
```

Initial values should be fairly obvious.

<Member functions for class GUSIMTInetSocket 769>+≡ (763) <771 774>

```
GUSIMTInetSocket::GUSIMTInetSocket()
: fStream(nil)
{
    memset(&fSockAddr, 0, sizeof(sockaddr_in));
    fSockAddr.sin_family = AF_INET;

    fPeerAddr = fSockAddr;
}
```

bind() for MacTCP sockets has the fatal flaw that it is totally unable to reserve a socket.

<Overridden member functions for GUSIMTInetSocket 773>≡ (764) 776>

```
virtual int bind(void * addr, socklen_t namelen);
```

<Member functions for class GUSIMTInetSocket 769>+≡ (763) <772 777>

```
int GUSIMTInetSocket::bind(void * addr, socklen_t namelen)
{
    struct sockaddr_in *name = (struct sockaddr_in *)addr;
    <Sanity checks for GUSIMTInetSocket::bind() 775>
    fSockAddr.sin_addr.s_addr = name->sin_addr.s_addr;
    fSockAddr.sin_port = name->sin_port;

    return 0;
}
```

The address to be passed must be up to a minimal standard of decency. For instance, the host address must be either the real IP number of our host or one of the two legitimate pseudo-addresses for "localhost".

```

<Sanity checks for GUSIMTInetSocket::bind() 775>≡ (774)
    if (!GUSI_ASSERT_CLIENT(
        namelen >= sizeof(struct sockaddr_in),
        ("bind: address len %d < %d\n", namelen, sizeof(struct sockaddr_in)))
    )
        return GUSISetPosixError(EINVAL);
    if (!GUSI_ASSERT_CLIENT(
        name->sin_family == AF_INET,
        ("bind: family %d != %d\n", name->sin_family, AF_INET))
    )
        return GUSISetPosixError(EAFNOSUPPORT);
    if (!GUSI_ASSERT_CLIENT(!fSockAddr.sin_port, "bind: Socket already bound\n"))
        return GUSISetPosixError(EINVAL);
    switch (name->sin_addr.s_addr) {
    default:
        if (!GUSI_ASSERT_CLIENT(
            name->sin_addr.s_addr == HostAddr(),
            ("bind: addr %08X != %08X\n", name->sin_addr.s_addr, HostAddr())
        )
        )
            return GUSISetPosixError(EADDRNOTAVAIL);
    case 0:
    case 0x7F000001:
        break;
    }

```

getsockname() and getpeername() return the stored values.

```

<Overridden member functions for GUSIMTInetSocket 773>+≡ (764) <773 778>
    virtual int getsockname(void * addr, socklen_t * namelen);
    virtual int getpeername(void * addr, socklen_t * namelen);

<Member functions for class GUSIMTInetSocket 769>+≡ (763) <774 779>
    int GUSIMTInetSocket::getsockname(void *name, socklen_t *namelen)
    {
        if (!GUSI_ASSERT_CLIENT(*namelen >= 0, "getsockname: passed negative length\n"))
            return GUSISetPosixError(EINVAL);

        memcpy(name, &fSockAddr, *namelen = min(*namelen, socklen_t(sizeof(sockaddr_in))));
        return 0;
    }
    int GUSIMTInetSocket::getpeername(void *name, socklen_t *namelen)
    {
        if (!GUSI_ASSERT_CLIENT(*namelen >= 0, "getpeername: passed negative length\n"))
            return GUSISetPosixError(EINVAL);

        memcpy(name, &fPeerAddr, *namelen = min(*namelen, socklen_t(sizeof(sockaddr_in))));
        return 0;
    }

```

shutdown() just delegates to GUSISMState.

```

<Overridden member functions for GUSIMTInetSocket 773>+≡ (764) <776 780>
    virtual int shutdown(int how);

```

(Member functions for class GUSIMTInetSocket 769)+≡ (763) <777 781>
int GUSIMTInetSocket::shutdown(int how)
{
if (!GUSI_SASSERT_CLIENT(how >= 0 && how < 3, "shutdown: 0,1, or 2\n"))
return GUSISetPosixError(EINVAL);

GUSISMState::Shutdown(how);

return 0;
}

fcntl() handles the blocking support.

(Overridden member functions for GUSIMTInetSocket 773)+≡ (764) <778 782>
virtual int fcntl(int cmd, va_list arg);

(Member functions for class GUSIMTInetSocket 769)+≡ (763) <779 783>
int GUSIMTInetSocket::fcntl(int cmd, va_list arg)
{
int result;

if (GUSISMBlocking::DoFcntl(&result, cmd, arg))
return result;

GUSI_ASSERT_CLIENT(false, ("fcntl: illegal request %d\n", cmd));

return GUSISetPosixError(EOPNOTSUPP);
}

ioctl() deals with blocking support and with FIONREAD.

(Overridden member functions for GUSIMTInetSocket 773)+≡ (764) <780 784>
virtual int ioctl(unsigned int request, va_list arg);

(Member functions for class GUSIMTInetSocket 769)+≡ (763) <781 785>
int GUSIMTInetSocket::ioctl(unsigned int request, va_list arg)
{
int result;

if (GUSISMBlocking::DoIoctl(&result, request, arg)
|| GUSISMinputBuffer::DoIoctl(&result, request, arg)
)
return result;

GUSI_ASSERT_CLIENT(false, ("ioctl: illegal request %d\n", request));

return GUSISetPosixError(EOPNOTSUPP);
}

getsockopt and setsockopt are available for setting buffer sizes and getting asynchronous errors.

(Overridden member functions for GUSIMTInetSocket 773)+≡ (764) <782 786>
virtual int getsockopt(int level, int optname, void *optval, socklen_t *optlen);

<Member functions for class GUSIMTInetSocket 769>+≡ (763) ¶783 787▷
int GUSIMTInetSocket::getsockopt(int level, int optname, void *optval, socklen_t * optlen)
{
int result;

if (GUSISMSInputBuffer::DoGetSockOpt(&result, level, optname, optval, optlen)
|| GUSISMSOutputBuffer::DoGetSockOpt(&result, level, optname, optval, optlen)
|| GUSISMSAsyncError::DoGetSockOpt(&result, level, optname, optval, optlen)
)
return result;

GUSI_ASSERT_CLIENT(false, ("getsockopt: illegal request %d\n", optname));

return GUSISetPosixError(EOPNOTSUPP);
}

<Overridden member functions for GUSIMTInetSocket 773>+≡ (764) ¶784 788▷
virtual int setsockopt(int level, int optname, void *optval, socklen_t optlen);

<Member functions for class GUSIMTInetSocket 769>+≡ (763) ¶785 789▷
int GUSIMTInetSocket::setsockopt(int level, int optname, void *optval, socklen_t optlen)
{
int result;

if (GUSISMSInputBuffer::DoSetSockOpt(&result, level, optname, optval, optlen)
|| GUSISMSOutputBuffer::DoSetSockOpt(&result, level, optname, optval, optlen)
)
return result;

GUSI_ASSERT_CLIENT(false, ("setsockopt: illegal request %d\n", optname));

return GUSISetPosixError(EOPNOTSUPP);
}

MacTCP sockets implement socket style calls.

<Overridden member functions for GUSIMTInetSocket 773>+≡ (764) ¶786
virtual bool Supports(ConfigOption config);

<Member functions for class GUSIMTInetSocket 769>+≡ (763) ¶787 791▷
bool GUSIMTInetSocket::Supports(ConfigOption config)
{
return config == kSocketCalls;
}

<Definition of GUSIwithMTInetSockets 790>≡ (762)
void GUSIwithMTInetSockets();

<Member functions for class GUSIMTInetSocket 769>+≡ (763) ¶789
void GUSIwithMTInetSockets()
{
GUSIwithMTTcpSockets();
GUSIwithMTUdpSockets();
}

Chapter 23

MacTCP TCP sockets

A `GUSIMTTcpSocket` implements the TCP socket class for MacTCP. All instances of `GUSIMTTcpSocket` are created by the `GUSIMTTcpFactory` singleton, so there is no point in exporting the class itself.

```
<GUSIMTTcp.h 792>≡
#ifndef _GUSIMTTcp_
#define _GUSIMTTcp_

#ifdef GUSI_SOURCE

#include <sys/cdefs.h>

__BEGIN_DECLS
<Definition of GUSIwithMTTcpSockets 794>
__END_DECLS

#ifdef GUSI_INTERNAL

#include "GUSIFactory.h"

<Definition of class GUSIMTTcpFactory 795>

<Inline member functions for class GUSIMTTcpFactory 797>

#endif /* GUSI_INTERNAL */

#endif /* GUSI_SOURCE */

#endif /* _GUSIMTTcp_ */
```

```

<GUSIMTTcp.cp 793>≡
#include "GUSIInternal.h"
#include "GUSIMTTcp.h"
#include "GUSIMTInet.h"
#include "GUSIMTNetDB.h"
#include "GUSIInet.h"
#include "GUSIDdiag.h"

#include <errno.h>

#include <Devices.h>

#include <algorithm>

<Definition of class GUSIMTTcpSocket 800>
<Interrupt level routines for GUSIMTTcpSocket 805>
<Member functions for class GUSIMTTcpSocket 802>
<Member functions for class GUSIMTTcpFactory 796>

```

23.1 Definition of GUSIMTTcpFactory

GUSIMTTcpFactory is a singleton subclass of GUSISocketFactory.

```

<Definition of GUSIwithMTTcSockets 794>≡ (792)
void GUSIwithMTTcSockets();

<Definition of class GUSIMTTcpFactory 795>≡ (792)
class GUSIMTTcpFactory : public GUSISocketFactory {
public:
    static GUSISocketFactory * Instance();
    virtual GUSISocket * socket(int domain, int type, int protocol);
private:
    GUSIMTTcpFactory() {}
    static GUSISocketFactory * instance;
};

```

23.2 Implementation of GUSIMTTcpFactory

```

<Member functions for class GUSIMTTcpFactory 796>≡ (793) 798▷
GUSISocketFactory * GUSIMTTcpFactory::instance = nil;

<Inline member functions for class GUSIMTTcpFactory 797>≡ (792)
inline GUSISocketFactory * GUSIMTTcpFactory::Instance()
{
    if (!instance)
        instance = new GUSIMTTcpFactory;
    return instance;
}

<Member functions for class GUSIMTTcpFactory 796>+≡ (793) ◀796 799▷
GUSISocket * GUSIMTTcpFactory::socket(int, int, int)
{
    return new GUSIMTTcpSocket();
}

```

```

<Member functions for class GUSIMTTcpFactory 796>+≡ (793) 4798
void GUSIwithMTTCpSockets()
{
    gGUSIInetFactories.AddFactory(SOCK_STREAM, 0, GUSIMTTcpFactory::Instance());
    GUSIMTNetDB::Instantiate();
}

```

23.3 Definition of GUSIMTTcpSocket

The only specific data member, fSelf, serves as the userDataPtr of the TCP notification procedure. Since accept() associates a different GUSIMTTcpSocket with the same TCP StreamPtr, passing this for this purpose would be fatal.

```

<Definition of class GUSIMTTcpSocket 800>≡ (793)
class GUSIMTTcpSocket : public GUSIMTInetSocket {
public:
    GUSIMTTcpSocket();
    ~GUSIMTTcpSocket();
    <Overridden member functions for GUSIMTTcpSocket 828>
private:
    GUSIMTTcpSocket ** fSelf;
    <Privatissima of GUSIMTTcpSocket 801>
};

```

23.4 Implementation of GUSIMTTcpSocket

The implementation of GUSIMTTcpSocket consists of a synchronous high level part which mostly deals with GUSIRingBuffers and an asynchronous low level part. The low level procedures in their single-mindedness are actually simpler to explain, so we start with them.

23.4.1 Interrupt level routines for GUSIMTTcpSocket

Both GUSIMTTSendDone() and GUSIMTTRecvDone() are always called with the same TCPiopb in a GUSIMTTcpSocket so they can easily find out the address of the socket itself. GUSIMTTSend and GUSIMTTRecv set up send and receive calls.

```

<Privatissima of GUSIMTTcpSocket 801>≡ (800) 809▷
TCPiopb      fSendPB;
MiniWDS      fSendWDS;
TCPiopb      fRecvPB;
friend void   GUSIMTTSend(GUSIMTTcpSocket * sock);
friend void   GUSIMTTRecv(GUSIMTTcpSocket * sock);
friend void   GUSIMTTSendDone(TCPiopb * pb);
friend void   GUSIMTTRecvDone(TCPiopb * pb);
static TCPIOCompletionUPP sSendProc;
static TCPIOCompletionUPP sRecvProc;

```

The UPPs for the completion procedures are set up the first time a socket is constructed.

```

<Member functions for class GUSIMTTcpSocket 802>≡ (793) 810▷
TCPIOCompletionUPP GUSIMTTcpSocket::sSendProc = 0;
TCPIOCompletionUPP GUSIMTTcpSocket::sRecvProc = 0;

```

```

<Initialize fields of GUSIMTTcpSocket 803>≡ (827 840) 804»
if (!sSendProc)
    sSendProc = NewTCPIOCompletionProc(GUSIMTTSendDone);
if (!sRecvProc)
    sRecvProc = NewTCPIOCompletionProc(GUSIMTTRecvDone);

```

The send and receive parameter blocks are highly specialized and never really change during the existence of a socket.

```

<Initialize fields of GUSIMTTcpSocket 803>+≡ (827 840) <803 811»
fSendPB.ioCRefNum = GUSIMTInetSocket::Driver();
fRecvPB.ioCRefNum = GUSIMTInetSocket::Driver();

```

GUSIMTTSendDone() does all its work between fSendPB and fOutputBuffer. If a send fails, the whole send buffer is cleared.

```

<Interrupt level routines for GUSIMTTcpSocket 805>≡ (793) 806»
void GUSIMTTSendDone(TCPiobp * pb)
{
    GUSIMTTcpSocket * sock =
        reinterpret_cast<GUSIMTTcpSocket *>((char *)pb-offsetof(GUSIMTTcpSocket, fSendPB));
    if (sock->fOutputBuffer.Locked())
        sock->fOutputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIMTTSendDone), pb);
    else {
        sock->fOutputBuffer.ClearDefer();
        sock->fOutputBuffer.FreeBuffer(sock->fSendWDS.fDataPtr, sock->fSendWDS.fLength);
        if (sock->SetAsyncMacError(sock->fSendPB.ioResult)) {
            for (long valid; valid = sock->fOutputBuffer.Valid(); )
                sock->fOutputBuffer.FreeBuffer(nil, valid);
            sock->fWriteShutdown = true;
        }
        GUSIMTTSend(sock);
        sock->Wakeup();
    }
}

```

GUSIMTTSend() starts a tcp send call if there is data to send and otherwise sets itself up as the deferred procedure of the output buffer (and thus is guaranteed to be called the next time data is deposited in the buffer again). If all data has been delivered and a shutdown is requested, send one.

```

<Interrupt level routines for GUSIMTTcpSocket 805>+≡ (793) <805 807>
void GUSIMTTSend(GUSIMTTcpSocket * sock)
{
    size_t valid = sock->fOutputBuffer.Valid();

    sock->fOutputBuffer.ClearDefer();
    if (!valid) {
        if (!sock->fWriteShutdown)
            sock->fOutputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIMTTSend), sock);
        else if (sock->fState == GUSIMTTcpSocket::Connected) {
            sock->fState = GUSISMState::Closing;
            sock->fSendPB.ioCompletion = nil;
            sock->fSendPB.csCode = TCPClose;
            sock->fSendPB.csParam.close.validityFlags = timeoutValue | timeoutAction;
            sock->fSendPB.csParam.close.ulpTimeoutValue = 60 /* seconds */;
            sock->fSendPB.csParam.close.ulpTimeoutAction = 0 /* 0:abort 1:report */;

            PBControlAsync(ParmBlkPtr(&sock->fSendPB));
        }
    } else {
        valid = min(valid, min((size_t)65535, sock->fOutputBuffer.Size() >> 1));

        sock->fSendWDS.fDataPtr =
            static_cast<Ptr>(sock->fOutputBuffer.ConsumeBuffer(valid));
        sock->fSendWDS.fLength = (u_short) valid;

        sock->fSendPB.ioCompletion = sock->sSendProc;
        sock->fSendPB.csCode = TCPSend;
        sock->fSendPB.csParam.send.validityFlags = timeoutValue | timeoutAction;
        sock->fSendPB.csParam.send.ulpTimeoutValue = 60 /* seconds */;
        sock->fSendPB.csParam.send.ulpTimeoutAction = 0 /* 0:abort 1:report */;
        sock->fSendPB.csParam.send.wdsPtr = &sock->fSendWDS;
        sock->fSendPB.csParam.send.sendFree = 0;
        sock->fSendPB.csParam.send.sendLength = 0;
        sock->fSendPB.csParam.send.urgentFlag = 0;
        sock->fSendPB.csParam.send.pushFlag =
            valid == sock->fOutputBuffer.Valid();

        PBControlAsync(ParmBlkPtr(&sock->fSendPB));
    }
}

```

GUSIMTTRecvDone() does all its work between fRecvPB and fInputBuffer.

```

<Interrupt level routines for GUSIMTTcpSocket 805>+≡ (793) <806 808>
void GUSIMTTRecvDone(TCPIopb * pb)
{
    GUSIMTTcpSocket * sock =
        reinterpret_cast<GUSIMTTcpSocket *>((char *)pb-offsetof(GUSIMTTcpSocket, fRecvPB));
    if (sock->fInputBuffer.Locked())
        sock->fInputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIMTTRecvDone), pb);
    else {
        sock->fInputBuffer.ClearDefer();
        switch (sock->fRecvPB.ioResult) {
            case noErr:
                sock->fInputBuffer.ValidBuffer(
                    sock->fRecvPB.csParam.receive.rcvBuff,
                    sock->fRecvPB.csParam.receive.rcvBuffLen);
                // Fall through
            case commandTimeout:
                GUSIMTTRecv(sock);
                break;
            default:
                sock->SetAsyncMacError(sock->fRecvPB.ioResult);
            case connectionClosing:
                sock->fReadShutdown = true;
                break;
        }
        sock->Wakeup();
    }
}

```

GUSIMTTRecv() starts a tcp receive call if there is room left and otherwise sets itself up as the deferred procedure of the output buffer (and thus is guaranteed to be called the next time there is free space in the buffer again).

```

<Interrupt level routines for GUSIMTTcpSocket 805>+≡ (793) <807 812>
void GUSIMTTRecv(GUSIMTTcpSocket * sock)
{
    size_t free = sock->fInputBuffer.Free();
    if (!free)
        sock->fInputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIMTTRecv), sock);
    else {
        sock->fInputBuffer.ClearDefer();
        free = min(free, min((size_t)65535, sock->fInputBuffer.Size() >> 1));

        sock->fRecvPB.ioCompletion          = sock->sRecvProc;
        sock->fRecvPB.csCode                 = TCPRcv;
        sock->fRecvPB.csParam.receive.rcvBuff =
            static_cast<Ptr>(sock->fInputBuffer.ProduceBuffer(free));
        sock->fRecvPB.csParam.receive.rcvBuffLen= free;
        sock->fRecvPB.csParam.receive.commandTimeoutValue = 120;

        PBControlAsync(ParmBlkPtr(&sock->fRecvPB));
    }
}

```

For some global events, MacTCP calls a notification procedure.

```

<Privatissima of GUSIMTTcpSocket 801>+≡ (800) <801 813>
friend pascal void GUSIMTTNotify(
    StreamPtr, u_short, GUSIMTTcpSocket **,
    u_short, struct ICMPReport *);
static TCPNotifyUPP sNotifyProc;

```

The UPP for the notification procedure is set up the first time a socket is constructed.

```

<Member functions for class GUSIMTTcpSocket 802>+≡ (793) <802 814>
TCPNotifyUPP GUSIMTTcpSocket::sNotifyProc = 0;

<Initialize fields of GUSIMTTcpSocket 803>+≡ (827 840) <804 815>
fSelf = nil;
if (!sNotifyProc)
    sNotifyProc = NewTCPNotifyProc(TCPNotifyProcPtr(GUSIMTTNotify));

<Interrupt level routines for GUSIMTTcpSocket 805>+≡ (793) <808 816>
pascal void GUSIMTTNotify(
    StreamPtr,
    u_short eventCode, GUSIMTTcpSocket ** sp, u_short, struct ICMPReport *)
{
    GUSIMTTcpSocket * sock = *sp;

    switch (eventCode) {
    case TCPClosing:
        sock->fReadShutdown = true;
        break;
    case TCPTerminate:
        sock->fReadShutdown = true;
        sock->fWriteShutdown = true;
        sock->fState = GUSISMSState::Unconnected;
        break;
    }
    sock->Wakeup();
}

```

When a connect() completes, we can fill in the peer address. connect() uses fSendPB.

```

<Privatissima of GUSIMTTcpSocket 801>+≡ (800) <809 817>
friend void GUSIMTTConnectDone(TCPiopb * pb);
static TCPIOCompletionUPP sConnectProc;

<Member functions for class GUSIMTTcpSocket 802>+≡ (793) <810 818>
TCPIOCompletionUPP GUSIMTTcpSocket::sConnectProc = 0;

<Initialize fields of GUSIMTTcpSocket 803>+≡ (827 840) <811 819>
if (!sConnectProc)
    sConnectProc = NewTCPIOCompletionProc(GUSIMTTConnectDone);

```

```

<Interrupt level routines for GUSIMTTcpSocket 805>+≡ (793) <812 824>
void GUSIMTTConnectDone(TCPiopb * pb)
{
    GUSIMTTcpSocket * sock =
        (GUSIMTTcpSocket *)((char *)pb-offsetof(GUSIMTTcpSocket, fSendPB));
    if (!sock->SetAsyncMacError(pb->ioResult)) {
        sock->fSockAddr.sin_family = AF_INET;
        sock->fSockAddr.sin_addr.s_addr = pb->csParam.open.localHost;
        sock->fSockAddr.sin_port = pb->csParam.open.localPort;
        sock->fPeerAddr.sin_family = AF_INET;
        sock->fPeerAddr.sin_addr.s_addr = pb->csParam.open.remoteHost;
        sock->fPeerAddr.sin_port = pb->csParam.open.remotePort;
        sock->fState = GUSISMState::Connected;

        GUSIMTTSend(sock);
        GUSIMTTRecv(sock);
    } else
        sock->fState = GUSISMState::Unconnected;
    GUSI_MESSAGE(("Connect %x\n", sock));
    sock->Wakeup();
}

```

Passive opens work similarly, but it is necessary to build a backlog if the interrupt level gets opens faster than the high level routines can accept them.

```

<Privatissima of GUSIMTTcpSocket 801>+≡ (800) <813 820>
struct Listener {
    StreamPtr          fTcp;
    GUSIMTTcpSocket ** fRef;
    sockaddr_in        fSockAddr;
    sockaddr_in        fPeerAddr;
    bool               fBusy;
};
Listener * fListeners;
bool       fRestartListen;
char       fNumListeners;
char       fCurListener;
char       fNextListener;
friend void GUSIMTTListenDone(TCPiopb * pb);
friend void GUSIMTTListen(GUSIMTTcpSocket * sock);
static TCPIOCompletionUPP sListenProc;

```

```

<Member functions for class GUSIMTTcpSocket 802>+≡ (793) <814 821>
TCPIOCompletionUPP GUSIMTTcpSocket::sListenProc = 0;

```

```

<Initialize fields of GUSIMTTcpSocket 803>+≡ (827 840) <815
fListeners = nil;
fRestartListen = true;
fNumListeners = 0;
fCurListener = 0;
fNextListener = 0;
if (!sListenProc)
    sListenProc = NewTCPIOCompletionProc(GUSIMTTListenDone);

```

CreateStream() creates a TCP stream.

```

<Privatissima of GUSIMTTcpSocket 801>+≡ (800) <817 822>
StreamPtr CreateStream(GUSIMTTcpSocket ** socketRef);

```


<Member functions for class GUSIMTTcpSocket 802>+≡ (793) 4818 823▷

StreamPtr GUSIMTTcpSocket::CreateStream(GUSIMTTcpSocket ** socketRef)

```
{
    fSendPB.ioCompletion      = nil;
    fSendPB.csCode            = TCPCreate;
    fSendPB.csParam.create.rcvBuff = (char *)NewPtr(8192);
    fSendPB.csParam.create.rcvBuffLen = 8192;
    fSendPB.csParam.create.notifyProc = sNotifyProc;
    fSendPB.csParam.create.userDataPtr = Ptr(socketRef);

    PBControlSync(ParmBlkPtr(&fSendPB));

    if (fSendPB.ioResult)
        return nil;
    else
        return fSendPB.tcpStream;
}
```

SetupListener() prepares a Listener.

<Privatissima of GUSIMTTcpSocket 801>+≡ (800) 4820 839▷

void SetupListener(Listener & listener);

<Member functions for class GUSIMTTcpSocket 802>+≡ (793) 4821 827▷

void GUSIMTTcpSocket::SetupListener(Listener & listener)

```
{
    listener.fRef = new (GUSIMTTcpSocket *);
    *listener.fRef = this;
    listener.fTcp = CreateStream(listener.fRef);
    listener.fBusy = false;
}
```

GUSIMTTListenDone() saves the connection parameters and starts the next passive open, if possible. Blocking on fInputBuffer is somewhat bizarre; we're not actually using the buffer, just its lock. The only times this lock is used is while we're waiting for a local socket number to be assigned to an unbound listener socket and when the socket is shutting down.

```

<Interrupt level routines for GUSIMTTcpSocket 805>+≡ (793) <816 825>
void GUSIMTTListenDone(TCPiopb * pb)
{
    bool allowRestart = true;
    GUSIMTTcpSocket * sock =
        (GUSIMTTcpSocket *)((char *)pb-offsetof(GUSIMTTcpSocket, fRecvPB));
    switch (pb->ioResult) {
        case commandTimeout:
        case openFailed:
            break;
        default:
            if (!sock->SetAsyncMacError(pb->ioResult)) {
                GUSIMTTcpSocket::Listener & listener = sock->fListeners[sock->fCurListener];
                listener.fSockAddr.sin_family = AF_INET;
                listener.fSockAddr.sin_addr.s_addr = pb->csParam.open.localHost;
                listener.fSockAddr.sin_port = pb->csParam.open.localPort;
                listener.fPeerAddr.sin_family = AF_INET;
                listener.fPeerAddr.sin_addr.s_addr = pb->csParam.open.remoteHost;
                listener.fPeerAddr.sin_port = pb->csParam.open.remotePort;
                listener.fBusy = true;
                sock->fCurListener = (sock->fCurListener+1) % sock->fNumListeners;
                GUSI_MESSAGE(("Listen %x\n", &listener));
            } else
                allowRestart = false;
    }
    sock->Wakeup();
    if (allowRestart)
        if (sock->fInputBuffer.Locked())
            sock->fInputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIMTTListenDone), pb);
        else {
            sock->fInputBuffer.ClearDefer();
            GUSIMTTListen(sock);
        }
}

```

GUSIMTTListen() initiates a passive open.

```

<Interrupt level routines for GUSIMTTcpSocket 805>+≡ (793) <824
void GUSIMTTListen(GUSIMTTcpSocket * sock)
{
    if (sock->fRestartListen = sock->fListeners[sock->fCurListener].fBusy)
        return;
    sock->fRecvPB.tcpStream = sock->fListeners[sock->fCurListener].fTcp;
    sock->fRecvPB.ioCompletion = sock->sListenProc;
    sock->fRecvPB.csCode = TCPPassiveOpen;
    sock->fRecvPB.csParam.open.validityFlags = timeoutValue | timeoutAction;
    sock->fRecvPB.csParam.open.ulpTimeoutValue = 300 /* seconds */;
    sock->fRecvPB.csParam.open.ulpTimeoutAction = 1 /* 1:abort 0:report */;
    sock->fRecvPB.csParam.open.commandTimeoutValue = 0 /* infinity */;
    sock->fRecvPB.csParam.open.remoteHost = 0;
    sock->fRecvPB.csParam.open.remotePort = 0;
    sock->fRecvPB.csParam.open.localHost = sock->fSockAddr.sin_addr.s_addr;
    sock->fRecvPB.csParam.open.localPort = sock->fSockAddr.sin_port;
    sock->fRecvPB.csParam.open.dontFrag = 0;
    sock->fRecvPB.csParam.open.timeToLive = 0;
    sock->fRecvPB.csParam.open.security = 0;
    sock->fRecvPB.csParam.open.optionCnt = 0;

    <Do the TCPPassiveOpen and pick up port number if necessary 826>
}

```

If we do a listen() on an unbound socket, MacTCP assigns a socket number, which strangely happens at some time *after* the asynchronous call has initially returned but *before* the call has completed. Thus, we lock to prevent further opens from starting and then do a sort-of-busy wait. Thanks to Peter Lewis for his explanations on that point.

```

<Do the TCPPassiveOpen and pick up port number if necessary 826>≡ (825)
if (!sock->fSockAddr.sin_port) {
    sock->fInputBuffer.Lock();
    PBControlAsync(ParmBlkPtr(&sock->fRecvPB));
    while (!sock->fRecvPB.csParam.open.localPort)
        GUSIContext::Yield(false);
    sock->fSockAddr.sin_port = sock->fRecvPB.csParam.open.localPort;
    sock->fInputBuffer.Release();
} else
    PBControlAsync(ParmBlkPtr(&sock->fRecvPB));

```

23.4.2 High level interface for GUSIMTTcpSocket

The constructor has to initialize a rather large number of data fields, and as a side effect opens the MacTCP driver if necessary. No other interesting activity occurs.

```

<Member functions for class GUSIMTTcpSocket 802>+≡ (793) <823 829>
GUSIMTTcpSocket::GUSIMTTcpSocket()
{
    <Initialize fields of GUSIMTTcpSocket 803>
}

```

connect() opens a connection actively.

```

<Overridden member functions for GUSIMTTcpSocket 828>≡ (800) 831>
virtual int connect(void * address, socklen_t addrlen);

```

```

<Member functions for class GUSIMTTcpSocket 802>+≡ (793) <827 832>
int GUSIMTTcpSocket::connect(void * address, socklen_t addrlen)
{
    sockaddr_in * addr = (sockaddr_in *) address;

    <Sanity checks for GUSIMTTcpSocket::connect() 830>

    if (!fSelf) {
        fSelf = new (GUSIMTTcpSocket *);
        *fSelf = this;
    }
    if (!fStream)
        if (!(fStream = CreateStream(fSelf)))
            return GUSISetPosixError(ENFILE);

    fSendPB.tcpStream = fStream;
    fRecvPB.tcpStream = fStream;

    fSendPB.ioCompletion = sConnectProc;
    fSendPB.csCode = TCPActiveOpen;
    fSendPB.csParam.open.validityFlags = timeoutValue | timeoutAction;
    fSendPB.csParam.open.ulpTimeoutValue = 60 /* seconds */;
    fSendPB.csParam.open.ulpTimeoutAction = 1 /* 1:abort 0:report */;
    fSendPB.csParam.open.commandTimeoutValue= 0;
    fSendPB.csParam.open.remoteHost = addr->sin_addr.s_addr;
    fSendPB.csParam.open.remotePort = addr->sin_port;
    fSendPB.csParam.open.localHost = fSockAddr.sin_addr.s_addr;
    fSendPB.csParam.open.localPort = fSockAddr.sin_port;
    fSendPB.csParam.open.dontFrag = 0;
    fSendPB.csParam.open.timeToLive = 0;
    fSendPB.csParam.open.security = 0;
    fSendPB.csParam.open.optionCnt = 0;
    fState = Connecting;

    SetAsyncPosixError(0);
    if (GUSISetMacError(PBControlAsync(ParmBlkPtr(&fSendPB))))
        return -1;

    if (!fBlocking)
        return GUSISetPosixError(EINPROGRESS);

    AddContext();
    while (fSendPB.ioResult == inProgress)
        GUSIContext::Yield(true);
    RemoveContext();

    return GUSISetPosixError(GetAsyncError());
}

```

```

<Sanity checks for GUSIMTTcpSocket::connect() 830>≡ (829)
if (!GUSI_CASSERT_CLIENT(addrlen >= int(sizeof(sockaddr_in))))
    return GUSISetPosixError(EINVAL);
if (!GUSI_CASSERT_CLIENT(addr->sin_family == AF_INET))
    return GUSISetPosixError(EAFNOSUPPORT);
if (GUSISetPosixError(GetAsyncError())) // non-blocking connect failed
    return -1;
switch (fState) {
case Connecting:
    return GUSISetPosixError(EALREADY); // non-blocking connect in progress
case Unbound:
case Unconnected:
    break; // Go ahead
default:
    return GUSISetPosixError(EISCONN); // Already connected in some form
}

```

Most of the dirty work of listen() is already handled in GUSIMTTListen.

```

<Overridden member functions for GUSIMTTcpSocket 828>+≡ (800) <828 835>
virtual int listen(int queueLength);

```

```

<Member functions for class GUSIMTTcpSocket 802>+≡ (793) <829 836>
int GUSIMTTcpSocket::listen(int queueLength)
{
    <Sanity checks for GUSIMTTcpSocket::listen() 833>
    <Adjust queueLength according to BSD definition 834>

    fInputBuffer.SwitchBuffer(0);
    fOutputBuffer.SwitchBuffer(0);
    fState = Listening;
    fListeners = new Listener[fNumListeners = queueLength];
    while (queueLength--)
        SetupListener(fListeners[queueLength]);

    GUSIMTTListen(this);

    return 0;
}

```

```

<Sanity checks for GUSIMTTcpSocket::listen() 833>≡ (832)
if (!GUSI_CASSERT_CLIENT(fState <= Unconnected))
    return GUSISetPosixError(EISCONN);

```

For some weird reason, BSD multiplies queue lengths with a fudge factor.

```

<Adjust queueLength according to BSD definition 834>≡ (832)
if (queueLength < 1)
    queueLength = 1;
else if (queueLength > 4)
    queueLength = 8;
else
    queueLength = ((queueLength * 3) >> 1) + 1;

```

accept() also is able to delegate most of the hard work to GUSIMTTListen.

```

<Overridden member functions for GUSIMTTcpSocket 828>+≡ (800) <831 841>
virtual GUSISocket * accept(void *from, socklen_t *fromlen);

```

```

<Member functions for class GUSIMTTcpSocket 802>+≡ (793) <832 840>
GUSISocket * GUSIMTTcpSocket::accept(void *from, socklen_t *fromlen)
{
    GUSIMTTcpSocket * sock;

    <Sanity checks for GUSIMTTcpSocket::accept() 837>
    <Wait for a connection to arrive for the GUSIMTTcpSocket 838>

    sock = new GUSIMTTcpSocket(fListeners[fNextListener]);

    SetupListener(fListeners[fNextListener]);
    fNextListener = (fNextListener+1) % fNumListeners;

    if (fRestartListen)
        GUSIMTTListen(this);

    if (sock && from)
        sock->getpeername(from, fromlen);

    return sock;
}

<Sanity checks for GUSIMTTcpSocket::accept() 837>≡ (836)
if (!GUSI_CASSERT_CLIENT(fState == Listening)) {
    GUSISetPosixError(ENOTCONN);
    return nil;
}

```

Listener slots are filled one by one, so we simply check whether the next listener block has been filled yet.

```

<Wait for a connection to arrive for the GUSIMTTcpSocket 838>≡ (836)
if (!fListeners[fNextListener].fBusy && !fReadShutdown) {
    if (!fBlocking)
        return GUSISetPosixError(EWOULDBLOCK), static_cast<GUSISocket *>(0);
    bool signal = false;
    AddContext();
    while (!fListeners[fNextListener].fBusy && !fReadShutdown)
        if (signal = GUSIContext::Yield(true))
            break;
    RemoveContext();
    if (signal)
        return GUSISetPosixError(EINTR), static_cast<GUSISocket *>(0);
}
if (!fListeners[fNextListener].fBusy && fReadShutdown) {
    GUSISetPosixError(ESHUTDOWN);
    return nil;
}

```

accept() uses a special constructor of GUSIMTTcpSocket which constructs a socket directly from a Listener.

```

<Privatissima of GUSIMTTcpSocket 801>+≡ (800) <822
GUSIMTTcpSocket(Listener & listener);

```

```

<Member functions for class GUSIMTTcpSocket 802>+≡ (793) ◀836 842▶
GUSIMTTcpSocket::GUSIMTTcpSocket(Listener & listener)
{
    <Initialize fields of GUSIMTTcpSocket 803>
    fSockAddr      = listener.fSockAddr;
    fPeerAddr      = listener.fPeerAddr;
    fSelf          = listener.fRef;
    *fSelf         = this;
    fStream        = listener.fTcp;
    fState         = Connected;
    fSendPB.tcpStream = fStream;
    fRecvPB.tcpStream = fStream;

    GUSIMTTSend(this);
    GUSIMTTRecv(this);
}

recvfrom() reads from fInputBuffer.

<Overridden member functions for GUSIMTTcpSocket 828>+≡ (800) ◀835 845▶
virtual ssize_t recvfrom(const GUSIScatterer & buffer, int, void * from, socklen_t * fromlen);

<Member functions for class GUSIMTTcpSocket 802>+≡ (793) ◀840 846▶
ssize_t GUSIMTTcpSocket::recvfrom(const GUSIScatterer & buffer, int flags, void * from, socklen_t * fromlen)
{
    if (from)
        getpeername(from, fromlen);

    <Sanity checks for GUSIMTTcpSocket::recvfrom() 843>
    <Wait for valid data on GUSIMTTcpSocket 844>

    size_t len = buffer.Length();
    if (flags & MSG_PEEK)
        fInputBuffer.Peek(buffer, len);
    else
        fInputBuffer.Consume(buffer, len);

    return (ssize_t)len;
}

<Sanity checks for GUSIMTTcpSocket::recvfrom() 843>≡ (842)
if (!fInputBuffer.Valid())
    if (GUSISetPosixError(GetAsyncError()))
        return -1;
    else if (fReadShutdown)
        return 0;
switch (fState) {
case Unbound:
case Unconnected:
case Listening:
    return GUSISetPosixError(ENOTCONN);
case Closing:
case Connecting:
case Connected:
    break;
}

```

The socket needs to be in Connected or Closing state and the input buffer needs to be nonempty before a read can succeed.

```

<Wait for valid data on GUSIMTTcpSocket 844>≡ (842)
    if (!fReadShutdown && (fState == Connecting || fState == Connected || fState == Closing) && !fInput
        if (!fBlocking)
            return GUSISetPosixError(EWOULDBLOCK);
        bool signal = false;
        AddContext();
        while (!fReadShutdown && (fState == Connecting || fState == Connected || fState == Closing) &&
            if (signal = GUSIContext::Yield(true))
                break;
        RemoveContext();
        if (signal)
            return GUSISetPosixError(EINTR);
    }

```

sendto() writes to fOutputBuffer. As opposed to reads, writes have to be executed fully. This leads to a problem when a nonblocking write wants to write more data than the total length of the buffer. In this case, GUSI disregards the nonblocking flag.

```

<Overridden member functions for GUSIMTTcpSocket 828>+≡ (800) <841 849>
    virtual ssize_t sendto(const GUSIGatherer & buffer, int flags, const void * to, socklen_t);

<Member functions for class GUSIMTTcpSocket 802>+≡ (793) <842 850>
    ssize_t GUSIMTTcpSocket::sendto(const GUSIGatherer & buffer, int flags, const void * to, socklen_t)
    {
        <Sanity checks for GUSIMTTcpSocket::sendto() 847>

        size_t rest = buffer.Length();
        size_t offset = 0;
        while (rest) {
            size_t len = rest;
            <Wait for free buffer space on GUSIMTTcpSocket 848>
            fOutputBuffer.Produce(buffer, len, offset);
            rest -= len;
        }

        return offset;
    }

```


<Sanity checks for GUSIMTTcpSocket::sendto() 847>≡ (846)

```

if (GUSISetPosixError(GetAsyncError()))
    return -1;
if (fWriteShutdown)
    return GUSISetPosixError(ESHUTDOWN);

if (!GUSI_SASSERT_CLIENT(!to, "Can't sendto() on a stream socket"))
    return GUSISetPosixError(EOPNOTSUPP);
switch (fState) {
case Unbound:
case Unconnected:
case Listening:
    return GUSISetPosixError(ENOTCONN);
case Closing:
case Connecting:
case Connected:
    break;
}

if (!fBlocking && (fState == Connecting || !fOutputBuffer.Free()))
    return GUSISetPosixError(EWOULDBLOCK);

```

<Wait for free buffer space on GUSIMTTcpSocket 848>≡ (846)

```

if (!fBlocking && !fOutputBuffer.Free())
    break;
if (!fWriteShutdown && (fState == Connecting || fState == Connected) && !fOutputBuffer.Free()) {
    bool signal = false;
    AddContext();
    while (!fWriteShutdown && (fState == Connecting || fState == Connected) && !fOutputBuffer.Free())
        if (signal = GUSIContext::Yield(true))
            break;
    RemoveContext();
    if (signal)
        if (offset)
            break;
        else
            GUSISetPosixError(EINTR);
}
if (fWriteShutdown && !fOutputBuffer.Free())
    if (offset)
        break;
    else
        return GUSISetPosixError(ESHUTDOWN);

```

select() checks for various conditions on the socket.

<Overridden member functions for GUSIMTTcpSocket 828>+≡ (800) ◀845 851▶

```

virtual bool    select(bool * canRead, bool * canWrite, bool * exception);

```

<Member functions for class GUSIMTTcpSocket 802>+≡ (793) <846 852>

```

bool GUSIMTTcpSocket::select(bool * canRead, bool * canWrite, bool *)
{
    bool cond = false;
    if (canRead)
        if (*canRead = fReadShutdown ||
            fAsyncError ||
            (fState == Listening
             ? fListeners[fNextListener].fBusy
             : fInputBuffer.Valid() > 0
            )
        )
            cond = true;
    if (canWrite)
        if (*canWrite = fWriteShutdown || fAsyncError || fOutputBuffer.Free())
            cond = true;

    if (cond)
        GUSI_MESSAGE(("Select%s%s\n",
                     (canRead && *canRead ? " read" : ""),
                     (canWrite && *canWrite ? " write" : "")));

    return cond;
}

```

shutdown() for writing sends a closing notice. fOutputBuffer is locked and released so the TCPclose is sent.

<Overridden member functions for GUSIMTTcpSocket 828>+≡ (800) <849

```

virtual int shutdown(int how);

```

<Member functions for class GUSIMTTcpSocket 802>+≡ (793) <850 853>

```

int GUSIMTTcpSocket::shutdown(int how)
{
    if (GUSIMTInetSocket::shutdown(how))
        return -1;
    fOutputBuffer.Lock();
    fOutputBuffer.Release();

    return 0;
}

```

MacTCP has ways to make you feel very sorry for yourself if you don't close streams.

```

<Member functions for class GUSIMTTcpSocket 802>+≡ (793) <852
GUSIMTTcpSocket::~GUSIMTTcpSocket()
{
    TCPiopb pb;

    pb.ioCRefNum    = GUSIMTInetSocket::Driver();
    pb.csCode       = TCPRelease;

    if (fState == Listening) {
        <Shut down listening GUSIMTTcpSocket 854>
    } else if (fStream) {
        pb.tcpStream = fStream;
        switch (fState) {
            case Connecting:
            case Connected:
                shutdown(2);
        }
        AddContext();
        while (fState > Unconnected)
            GUSIContext::Yield(true);
        RemoveContext();

        if (PBControlSync(ParmBlkPtr(&pb)))
            return;

        DisposePtr(pb.csParam.create.rcvBuff); /* there is no release pb */
    }
}

<Shut down listening GUSIMTTcpSocket 854>≡ (853)
fInputBuffer.Lock();
for (int i = 0; i < fNumListeners; i++) {
    pb.tcpStream = fListeners[i].fTcp;
    if (PBControlSync(ParmBlkPtr(&pb)))
        continue;
    DisposePtr(pb.csParam.create.rcvBuff); /* there is no release pb */
}

```


Chapter 24

MacTCP UDP sockets

A `GUSIMTUDpSocket` implements the UDP socket class for MacTCP. All instances of `GUSIMTUDpSocket` are created by the `GUSIMTUDpFactory` singleton, so there is no point in exporting the class itself.

```
<GUSIMTUDp.h 855>≡
#ifdef _GUSIMTUDp_
#define _GUSIMTUDp_

#ifdef GUSI_SOURCE

#include <sys/cdefs.h>

__BEGIN_DECLS
<Definition of GUSIwithMTUDpSockets 857>
__END_DECLS

#ifdef GUSI_INTERNAL

#include "GUSIFactory.h"

<Definition of class GUSIMTUDpFactory 858>

<Inline member functions for class GUSIMTUDpFactory 860>

#endif /* GUSI_INTERNAL */

#endif /* GUSI_SOURCE */

#endif /* _GUSIMTUDp_ */
```

```

<GUSIMTUDP.cp 856>≡
#include "GUSIInternal.h"
#include "GUSIMTUDP.h"
#include "GUSIMTInet.h"
#include "GUSIMTNetDB.h"
#include "GUSIInet.h"
#include "GUSIDiag.h"

#include <errno.h>

#include <algorithm>

#include <Devices.h>

<Definition of class GUSIMTUDPsocket 863>
<Interrupt level routines for GUSIMTUDPsocket 868>
<Member functions for class GUSIMTUDPsocket 865>
<Member functions for class GUSIMTUDPFactory 859>

```

24.1 Definition of GUSIMTUDPFactory

GUSIMTUDPFactory is a singleton subclass of GUSISocketFactory.

```

<Definition of GUSIwithMTUDPSockets 857>≡ (855)
void GUSIwithMTUDPSockets();

<Definition of class GUSIMTUDPFactory 858>≡ (855)
class GUSIMTUDPFactory : public GUSISocketFactory {
public:
    static GUSISocketFactory * Instance();
    virtual GUSISocket * socket(int domain, int type, int protocol);
private:
    GUSIMTUDPFactory() {}
    static GUSISocketFactory * instance;
};

```

24.2 Implementation of GUSIMTUDPFactory

```

<Member functions for class GUSIMTUDPFactory 859>≡ (856) 861»
GUSISocketFactory * GUSIMTUDPFactory::instance = nil;

<Inline member functions for class GUSIMTUDPFactory 860>≡ (855)
inline GUSISocketFactory * GUSIMTUDPFactory::Instance()
{
    if (!instance)
        instance = new GUSIMTUDPFactory;
    return instance;
}

<Member functions for class GUSIMTUDPFactory 859>+≡ (856) <859 862»
GUSISocket * GUSIMTUDPFactory::socket(int, int, int)
{
    return new GUSIMTUDPsocket();
}

```

```

⟨Member functions for class GUSIMTudpFactory 859⟩+≡ (856) ◀861
void GUSIwithMTUdpSockets()
{
    gGUSIInetFactories.AddFactory(SOCK_DGRAM, 0, GUSIMTudpFactory::Instance());
    GUSIMTNetDB::Instantiate();
}

```

24.3 Definition of GUSIMTudpSocket

GUSIMTudpSocket have no interesting data of their own.

```

⟨Definition of class GUSIMTudpSocket 863⟩≡ (856)
class GUSIMTudpSocket : public GUSIMTInetSocket {
public:
    GUSIMTudpSocket();
    ~GUSIMTudpSocket();
    ⟨Overridden member functions for GUSIMTudpSocket 878⟩
private:
    ⟨Privatissima of GUSIMTudpSocket 864⟩
};

```

24.4 Implementation of GUSIMTudpSocket

The implementation of GUSIMTudpSocket consists of a synchronous high level part which mostly deals with GUSIRingBuffers and an asynchronous low level part. We'll start with the low level procedures.

24.4.1 Interrupt level routines for GUSIMTudpSocket

Both GUSIMTUSendDone() and GUSIMTUREcvDone() are always called with the same UDPIopb in a GUSIMTudpSocket so they can easily find out the address of the socket itself. GUSIMTUSend and GUSIMTUREcv set up send and receive calls.

```

⟨Privatissima of GUSIMTudpSocket 864⟩≡ (863) 875▷
UDPIopb          fSendPB;
MidiWDS          fSendWDS;
UDPIopb          fRecvPB;
friend void      GUSIMTUSend(GUSIMTudpSocket * sock);
friend void      GUSIMTUREcv(GUSIMTudpSocket * sock);
friend void      GUSIMTUSendDone(UDPIopb * pb);
friend void      GUSIMTUREcvDone(UDPIopb * pb);
static UDPIOCompletionUPP sSendProc;
static UDPIOCompletionUPP sRecvProc;

```

The UPPs for the completion procedures are set up the first time a socket is constructed.

```

⟨Member functions for class GUSIMTudpSocket 865⟩≡ (856) 876▷
UDPIOCompletionUPP GUSIMTudpSocket::sSendProc = 0;
UDPIOCompletionUPP GUSIMTudpSocket::sRecvProc = 0;

```

(Initialize fields of GUSIMTUDpSocket 866)≡ (877) 867▷

```
if (!sSendProc)
    sSendProc = NewUDPIOCompletionProc(GUSIMTUSendDone);
if (!sRecvProc)
    sRecvProc = NewUDPIOCompletionProc(GUSIMTUSendDone);
```

The send and receive parameter blocks are highly specialized and never really change during the existence of a socket.

(Initialize fields of GUSIMTUDpSocket 866)+≡ (877) ◀866

```
fSendPB.ioCRefNum = GUSIMTInetSocket::Driver();
fRecvPB.ioCRefNum = GUSIMTInetSocket::Driver();
```

GUSIMTUSendDone() does all its work between fSendPB and fOutputBuffer. If a send fails, the whole send buffer is cleared.

(Interrupt level routines for GUSIMTUDpSocket 868)≡ (856) 869▷

```
void GUSIMTUSendDone(UDPiopb * pb)
{
    GUSIMTUDpSocket * sock =
        reinterpret_cast<GUSIMTUDpSocket *>((char *)pb-offsetof(GUSIMTUDpSocket, fSendPB));
    if (sock->fOutputBuffer.Locked())
        sock->fOutputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIMTUSendDone), pb);
    else {
        sock->fOutputBuffer.ClearDefer();
        sock->fOutputBuffer.FreeBuffer(sock->fSendWDS.fDataPtr, sock->fSendWDS.fLength);
        switch (sock->fSendPB.ioResult) {
            case noErr:
                break; /* Everything ok */
            case insufficientResources:
                break; /* Queue overflow, discard packet but proceed */
            default:
                sock->SetAsyncMacError(sock->fSendPB.ioResult);
                for (long valid; valid = sock->fOutputBuffer.Valid(); )
                    sock->fOutputBuffer.FreeBuffer(nil, valid);
                sock->fWriteShutdown = true;
        }
        GUSIMTUSend(sock);
        sock->Wakeup();
    }
}
```

Since UDP consists of distinct packets, each packet is prefixed by a header when stored in a GUSIBuffer.

(Interrupt level routines for GUSIMTUDpSocket 868)+≡ (856) ◀868 870▷

```
struct GUSIUDPHeader {
    in_addr_t    fPeerAddr;
    in_port_t    fPeerPort;
    uint16_t     fLength;
};
```


GUSIMTUSend() starts an UDP send call if there is data to send and otherwise sets itself up as the deferred procedure of the output buffer (and thus is guaranteed to be called the next time data is deposited in the buffer again). If all data has been delivered and a shutdown is requested, send one.

To avoid race conditions, we insist that a valid request consist of at least 1 byte of packet data.

```

<Interrupt level routines for GUSIMTudpSocket 868>+≡ (856) <869 872>
void GUSIMTUSend(GUSIMTudpSocket * sock)
{
    sock->fOutputBuffer.ClearDefer();
    if (!sock->fOutputBuffer.Valid() || sock->fOutputBuffer.Locked()) {
        if (!sock->fWriteShutdown)
            sock->fOutputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIMTUSend), sock);
        else
            sock->fState = GUSISMState::Closing;
    } else {
        sock->fOutputBuffer.ClearDefer();

        GUSIUDPHeader header;
        size_t len = sizeof(GUSIUDPHeader);

        sock->fOutputBuffer.Consume(&header, len);

        <Set up fSendWDS 871>

        sock->fSendPB.ioCompletion = sock->sSendProc;
        sock->fSendPB.csCode = UDPWrite;
        sock->fSendPB.csParam.send.remoteHost = header.fPeerAddr;
        sock->fSendPB.csParam.send.remotePort = header.fPeerPort;
        sock->fSendPB.csParam.send.wdsPtr = &sock->fSendWDS;
        sock->fSendPB.csParam.send.checkSum = true;
        sock->fSendPB.csParam.send.sendLength = 0;

        PBControlAsync(ParmBlkPtr(&sock->fSendPB));
    }
}

```

Since we have to preserve packet sizes and GUSIBuffer does not, we sometimes might get the packet data in two pieces.

```

<Set up fSendWDS 871>≡ (870)
len = header.fLength;
sock->fSendWDS.fDataPtr =
    static_cast<Ptr>(sock->fOutputBuffer.ConsumeBuffer(len));
sock->fSendWDS.fLength = (u_short) len;
if (len < header.fLength) {
    len = header.fLength - len;
    sock->fSendWDS.fDataPtr2 =
        static_cast<Ptr>(sock->fOutputBuffer.ConsumeBuffer(len));
    sock->fSendWDS.fLength2 = (u_short) len;
} else
    sock->fSendWDS.fLength2 = 0;

```

GUSIMTRecvDone() does all its work between fRecvPB and fInputBuffer.

```
<Interrupt level routines for GUSIMTudpSocket 868>+≡                                (856) <870 874>
void GUSIMTRecvDone(UDPiopb * pb)
{
    GUSIMTudpSocket * sock =
        reinterpret_cast<GUSIMTudpSocket *>((char *)pb-offsetof(GUSIMTudpSocket, fRecvPB));
    if (sock->fInputBuffer.Locked())
        sock->fInputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIMTRecvDone), pb);
    else {
        sock->fInputBuffer.ClearDefer();
        switch (sock->fRecvPB.ioResult) {
            case noErr:
                <Check and store received UDP packet 873>
                // Fall through
            case commandTimeout:
                GUSIMTRecv(sock);
                break;
            default:
                sock->SetAsyncMacError(sock->fRecvPB.ioResult);
                sock->fReadShutdown = true;
                break;
        }
        sock->Wakeup();
    }
}
```

If the packet is too big, or if the socket is bound and a packet arrives from a different peer, we drop it. Whether we drop or not, we must release the buffers. Since we reuse the same completion procedure, we also must check whether we arrived here on a read or on a buffer return.

```

<Check and store received UDP packet 873>≡ (872)
    if (sock->fRecvPB.csCode == UDPRead) {
        long needed = sock->fRecvPB.csParam.receive.rcvBuffLen+8;
        if (sock->fInputBuffer.Size() < needed)
            ; // Drop
        else if (sock->fInputBuffer.Free() < needed) {
            sock->fInputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIMTurecvDone), pb);
            return;
        } else if (sock->fPeerAddr.sin_port &&
            ( sock->fRecvPB.csParam.receive.remoteHost != sock->fPeerAddr.sin_addr.s_addr
              || sock->fRecvPB.csParam.receive.remotePort != sock->fPeerAddr.sin_port
            )
        ) {
            ; // Drop
        } else {
            GUSIUDPHeader header;
            size_t len = sizeof(GUSIUDPHeader);

            header.fPeerAddr = sock->fRecvPB.csParam.receive.remoteHost;
            header.fPeerPort = sock->fRecvPB.csParam.receive.remotePort;
            header.fLength = sock->fRecvPB.csParam.receive.rcvBuffLen;

            sock->fInputBuffer.Produce(&header, len);
            len = header.fLength;
            sock->fInputBuffer.Produce(
                sock->fRecvPB.csParam.receive.rcvBuff, len);
        }
        sock->fRecvPB.csCode = UDPBfrReturn;
        PBControlAsync(ParmBlkPtr(&sock->fRecvPB));

        return;
    }

```

GUSIMTurecv() starts an UDP receive call. No free space is necessary in the buffer.

```

<Interrupt level routines for GUSIMTudpSocket 868>+≡ (856) <872
void GUSIMTurecv(GUSIMTudpSocket * sock)
{
    sock->fRecvPB.ioCompletion = sock->sRecvProc;
    sock->fRecvPB.csCode = UDPRead;
    sock->fRecvPB.csParam.receive.timeOut = 0;
    sock->fRecvPB.csParam.receive.secondTimeStamp = 0;

    PBControlAsync(ParmBlkPtr(&sock->fRecvPB));
}

```

CreateStream() creates a UDP stream.

```

<Privatissima of GUSIMTudpSocket 864>+≡ (863) <864
int CreateStream();

```

```

<Member functions for class GUSIMTUDpSocket 865>+≡ (856) <865 877>
int GUSIMTUDpSocket::CreateStream()
{
    fSendPB.ioCompletion          = nil;
    fSendPB.csCode                = UDPCreate;
    fSendPB.csParam.create.rcvBuff = (char *)NewPtr(4096);
    fSendPB.csParam.create.rcvBuffLen = 4096;
    fSendPB.csParam.create.notifyProc = nil;
    fSendPB.csParam.create.userDataPtr = nil;
    fSendPB.csParam.create.localPort  = fSockAddr.sin_port;

    PBControlSync(ParmBlkPtr(&fSendPB));

    if (fSendPB.ioResult)
        return GUSISetMacError(fSendPB.ioResult);

    fState          = Connected;
    fStream          = fRecvPB.udpStream = fSendPB.udpStream;
    fSockAddr.sin_port = fSendPB.csParam.create.localPort;

    GUSIMTUSend(this);
    GUSIMTUCrecv(this);

    return 0;
}

```

24.4.2 High level interface for GUSIMTUDpSocket

The constructor has to initialize a rather large number of data fields, and as a side effect opens the MacUDP driver if necessary. No other interesting activity occurs.

```

<Member functions for class GUSIMTUDpSocket 865>+≡ (856) <876 879>
GUSIMTUDpSocket::GUSIMTUDpSocket()
{
    <Initialize fields of GUSIMTUDpSocket 866>
}

connect() installs a peer address.

<Overridden member functions for GUSIMTUDpSocket 878>≡ (863) 882>
virtual int connect(void * address, socklen_t addrlen);

<Member functions for class GUSIMTUDpSocket 865>+≡ (856) <877 883>
int GUSIMTUDpSocket::connect(void * address, socklen_t addrlen)
{
    sockaddr_in * addr = (sockaddr_in *) address;

    <Sanity checks for GUSIMTUDpSocket::connect() 880>
    <Create UDP stream if necessary 881>

    fPeerAddr = *addr;

    return 0;
}

```

```

    <Sanity checks for GUSIMTUDpSocket::connect() 880>≡ (879)
    if (!GUSI_CASSERT_CLIENT(addrlen >= int(sizeof(sockaddr_in))))
        return GUSISetPosixError(EINVAL);
    if (!GUSI_CASSERT_CLIENT(addr->sin_family == AF_INET)) {
        memset(&fPeerAddr, 0, sizeof(sockaddr_in));
        return GUSISetPosixError(EAFNOSUPPORT);
    }

    <Create UDP stream if necessary 881>≡ (879 883 887 891)
    if (!fStream && CreateStream())
        return -1;

    recvfrom() reads from fInputBuffer.

    <Overridden member functions for GUSIMTUDpSocket 878>+≡ (863) <878 886>
    virtual ssize_t recvfrom(const GUSIScatterer & buffer, int, void * from, socklen_t * fromlen);

    <Member functions for class GUSIMTUDpSocket 865>+≡ (856) <879 887>
    ssize_t GUSIMTUDpSocket::recvfrom(const GUSIScatterer & buffer, int flags, void * from, socklen_t * fromlen)
    {
        <Sanity checks for GUSIMTUDpSocket::recvfrom() 884>
        <Create UDP stream if necessary 881>
        <Wait for valid data on GUSIMTUDpSocket 885>

        GUSIUDPHeader header;
        size_t len = sizeof(GUSIUDPHeader);

        if (flags & MSG_PEEK) {
            GUSIRingBuffer::Peeker peeker(fInputBuffer);

            peeker.Peek(&header, len);
            len = min(buffer.Length(), (int) header.fLength);
            peeker.Peek(buffer, len);
        } else {
            fInputBuffer.Consume(&header, len);

            len = min(buffer.Length(), (int)header.fLength);
            fInputBuffer.Consume(buffer, len);
        }

        if (from) {
            sockaddr_in * addr = (sockaddr_in *) from;
            *fromlen = sizeof(sockaddr_in);
            addr->sin_family = AF_INET;
            addr->sin_addr.s_addr = header.fPeerAddr;
            addr->sin_port = header.fPeerPort;
        }

        return (int)len;
    }

```

<Sanity checks for GUSIMTUDpSocket::recvfrom() 884>≡ (883)

```

if (!fInputBuffer.Valid())
    if (GUSISetPosixError(GetAsyncError()))
        return -1;
    else if (fReadShutdown)
        return 0;
if (!GUSI_CASSERT_CLIENT(!from || *fromlen >= int(sizeof(sockaddr_in))))
    return GUSISetPosixError(EINVAL);

```

The input buffer needs to be nonempty before a read can succeed.

<Wait for valid data on GUSIMTUDpSocket 885>≡ (883)

```

if (!fReadShutdown && !fInputBuffer.Valid()) {
    if (!fBlocking)
        return GUSISetPosixError(EWOULDBLOCK);
    bool signal = false;
    AddContext();
    while (!fReadShutdown && !fInputBuffer.Valid())
        if (signal = GUSIContext::Yield(true))
            break;
    RemoveContext();
    if (signal)
        return GUSISetPosixError(EINTR);
}

```

sendto() writes to fOutputBuffer. It is very important that the output buffer be locked between writing the header and writing the data.

<Overridden member functions for GUSIMTUDpSocket 878>+≡ (863) <882 890>

```

virtual ssize_t sendto(const GUSIGatherer & buffer, int flags, const void * to, socklen_t addrlen);

```

```

<Member functions for class GUSIMTUDpSocket 865>+≡ (856) 4883 891▷
ssize_t GUSIMTUDpSocket::sendto(const GUSIGatherer & buffer, int flags, const void * to, socklen_t addrlen)
{
    <Sanity checks for GUSIMTUDpSocket::sendto() 888>
    <Create UDP stream if necessary 881>
    <Wait for free buffer space on GUSIMTUDpSocket 889>

    GUSIUdpHeader header;
    if (fPeerAddr.sin_port) {
        header.fPeerAddr = fPeerAddr.sin_addr.s_addr;
        header.fPeerPort = fPeerAddr.sin_port;
    } else if (to) {
        header.fPeerAddr = static_cast<const sockaddr_in *>(to)->sin_addr.s_addr;
        header.fPeerPort = static_cast<const sockaddr_in *>(to)->sin_port;
    } else
        return GUSISetPosixError(ENOTCONN);
    header.fLength = (uint16_t)buffer.Length();

    fOutputBuffer.Lock();
    size_t len = sizeof(GUSIUdpHeader);
    fOutputBuffer.Produce(&header, len);
    len = buffer.Length();
    size_t offset = 0;
    fOutputBuffer.Produce(buffer, len, offset);
    fOutputBuffer.Release();

    return len;
}

<Sanity checks for GUSIMTUDpSocket::sendto() 888>≡ (887)
if (GUSISetPosixError(GetAsyncError()))
    return -1;
if (!GUSI_CASSERT_CLIENT(!to || addrlen >= int(sizeof(sockaddr_in))))
    return GUSISetPosixError(EINVAL);
if (!GUSI_CASSERT_CLIENT(buffer.Length() < 0x10000))
    return GUSISetPosixError(EINVAL);

<Wait for free buffer space on GUSIMTUDpSocket 889>≡ (887)
size_t needed = buffer.Length()+8;
if (!fBlocking && fOutputBuffer.Free() < needed)
    return GUSISetPosixError(EWOULDBLOCK);
if (!fWriteShutdown && fOutputBuffer.Free() < needed) {
    if (fOutputBuffer.Size() < needed)
        return GUSISetPosixError(EINVAL);
    bool signal = false;
    AddContext();
    while (!fWriteShutdown && fOutputBuffer.Free() < needed)
        if (signal = GUSIContext::Yield(true))
            break;
    RemoveContext();
    if (signal)
        return GUSISetPosixError(EINTR);
}
if (fWriteShutdown)
    return GUSISetPosixError(ESHUTDOWN);

```

select() checks for various conditions on the socket.

```
<Overridden member functions for GUSIMTUDpSocket 878>+≡ (863) <886 892>
virtual bool select(bool * canRead, bool * canWrite, bool * exception);

<Member functions for class GUSIMTUDpSocket 865>+≡ (856) <887 893>
bool GUSIMTUDpSocket::select(bool * canRead, bool * canWrite, bool *)
{
    <Create UDP stream if necessary 881>

    bool cond = false;
    if (canRead)
        if (*canRead = fReadShutdown || fAsyncError || fInputBuffer.Valid() > 0)
            cond = true;
    if (canWrite)
        if (*canWrite = fWriteShutdown || fAsyncError || fOutputBuffer.Free()>1600)
            cond = true;

    return cond;
}
```

shutdown() for writing sends a closing notice. fOutputBuffer is locked and released so the send procedure is called.

```
<Overridden member functions for GUSIMTUDpSocket 878>+≡ (863) <890
virtual int shutdown(int how);

<Member functions for class GUSIMTUDpSocket 865>+≡ (856) <891 894>
int GUSIMTUDpSocket::shutdown(int how)
{
    if (GUSIMTInetSocket::shutdown(how))
        return -1;
    fOutputBuffer.Lock();
    fOutputBuffer.Release();

    return 0;
}
```


MacTCP has ways to make you feel very sorry for yourself if you don't close streams.

```
<Member functions for class GUSIMTudpSocket 865>+≡ (856) <893
GUSIMTudpSocket::~GUSIMTudpSocket()
{
    if (fStream) {
        UDPIopb pb;

        pb.ioCRefNum    = GUSIMTInetSocket::Driver();
        pb.csCode        = UDPRRelease;
        pb.udpStream     = fStream;

        if (fState != Closing) {
            shutdown(2);

            AddContext();
            while (fState != Closing)
                GUSIContext::Yield(true);
            RemoveContext();
        }

        if (PBControlSync(ParmBlkPtr(&pb)))
            return;

        DisposePtr(pb.csParam.create.rcvBuff); /* there is no release pb */
    }
}
```


Chapter 25

IP Name Lookup in MacTCP

```
<GUSIMTNetDB.h 895>≡
    #ifndef _GUSIMTNetDB_
    #define _GUSIMTNetDB_

    #ifdef GUSI_INTERNAL
    #include "GUSINetDB.h"

<Definition of class GUSIMTNetDB 897>

    #endif /* GUSI_INTERNAL */

    #endif /* _GUSIMTNetDB_ */

<GUSIMTNetDB.cp 896>≡
    #include "GUSIInternal.h"
    #include "GUSIMTNetDB.h"
    #include "GUSIMTInet.h"
    #include "GUSIContext.h"

    #include <MacTCP.h>

    #include <sys/socket.h>
    #include <string.h>
    #include <stdlib.h>

<MacTCP DNR code 898>
<Member functions for class GUSIMTNetDB 900>
```

25.1 Definition of GUSIMTNetDB

```
(Definition of class GUSIMTNetDB 897)≡ (895)
class GUSIMTNetDB : public GUSINetDB {
public:
    static void Instantiate();
    static bool Resolver();

    (Overridden member functions for GUSIMTNetDB 903)
private:
    GUSIMTNetDB() {}
    GUSISpecificData<GUSIhostent,GUSIKillHostEnt> fHost;
    static OSerr sResolverState;
};
```

25.2 Implementation of GUSIMTNetDB

To implement this, we need to include the MacTCP DNR code, a somewhat unpleasant operation. If possible, we wrap within an anonymous namespace.

```
(MacTCP DNR code 898)≡ (896) 899▷
#ifdef GUSI_COMPILER_HAS_NAMESPACE
namespace {
#include <AddressXlation.h>

#include "dnr.c"
}
#else
#include "dnr.c"
#endif
```

When a DNR call terminates, it wakes up the thread issuing the call.

```
(MacTCP DNR code 898)+≡ (896) 4898
static pascal void DNRDone(struct hostInfo *, GUSIContext * context)
{
    context->Wakeup();
}

#if GENERATINGCFM
RoutineDescriptor uDNRDone =
    BUILD_ROUTINE_DESCRIPTOR(uppResultProcInfo, DNRDone);
#else
#define uDNRDone DNRDone
#endif
```

The MacTCP DNR code is installed by calling `Instantiate`, which initializes the singleton instance of `GUSINetDB`.

```
(Member functions for class GUSIMTNetDB 900)≡ (896) 901▷
void GUSIMTNetDB::Instantiate()
{
    if (!sInstance)
        sInstance = new GUSIMTNetDB;
}
```

```

⟨Member functions for class GUSIMTNetDB 900⟩+≡ (896) ◀900 902▷
    OSErr GUSIMTNetDB::sResolverState = 1;

    bool GUSIMTNetDB::Resolver()
    {
        GUSIMTInetSocket::Driver();
        if (sResolverState == 1)
            sResolverState = OpenResolver(nil);

        return !sResolverState;
    }

```

Naturally, MacTCP keeps its host data in different structures than the socket standard.

```

⟨Member functions for class GUSIMTNetDB 900⟩+≡ (896) ◀901 904▷
    static void CopyHost(hostInfo & macHost, GUSIhostent & unixHost)
    {
        /* for some reason there is a dot at the end of the name */
        size_t len = strlen(macHost.cname);
        if (macHost.cname[len-1] == '.')
            macHost.cname[--len] = 0;
        len = (len+4) & ~3;

        unixHost.Alloc(len+NUM_ALT_ADDRS*4);
        strcpy(unixHost.h_name, macHost.cname);
        unixHost.h_aliases[0] = NULL; // Aliases not supported
        unixHost.h_addrtype = AF_INET;
        unixHost.h_length = 4;

        int addrs = 0;
        for (int i=0; i<NUM_ALT_ADDRS && macHost.addr[addrs]!=0; ++i, ++addrs) {
            unixHost.h_addr_list[addrs] = unixHost.fName + len;
            len += 4;
            memcpy(unixHost.h_addr_list[addrs], &macHost.addr[addrs], 4);
        }
        unixHost.h_addr_list[addrs] = NULL;
    }
}

⟨Overridden member functions for GUSIMTNetDB 903⟩≡ (897) 907▷
    virtual hostent * gethostbyname(const char * name);

```

```

<Member functions for class GUSIMTNetDB 900>+≡ (896) <902 908>
    hostent * GUSIMTNetDB::gethostbyname(const char * name)
    {
        <Open MacTCP DNR or fail lookup 905>
        if (!strcmp(name, "localhost")) {
            long ipaddr = gethostid();
            if (ipaddr)
                return gethostbyaddr((char *) &ipaddr, sizeof(in_addr), AF_INET);
            return GUSISetHostError(HOST_NOT_FOUND), static_cast<hostent *>(nil);
        }
        <Declare and initialize macHost and unixHost 906>
        if (StrToAddr(const_cast<char *>(name), &macHost, ResultUPP(&uDNRDone),
            reinterpret_cast<Ptr>(GUSIContext::Current()))
            == cacheFault
        )
        {
            do {
                GUSIContext::Yield(true);
            } while (macHost.rtnCode == cacheFault);
        }
        if (macHost.rtnCode)
            return GUSISetMacHostError(macHost.rtnCode), static_cast<hostent *>(nil);
        if (macHost.cname[0] == 0)
            return GUSISetHostError(HOST_NOT_FOUND), static_cast<hostent *>(nil);

        CopyHost(macHost, unixHost);

        return &unixHost;
    }

<Open MacTCP DNR or fail lookup 905>≡ (904 908)
    if (!Resolver())
        return GUSISetHostError(NO_RECOVERY), static_cast<hostent *>(nil);

<Declare and initialize macHost and unixHost 906>≡ (904 908)
    hostInfo      macHost;
    GUSIhostent & unixHost = *fHost;

    for (int i=0; i<NUM_ALT_ADDRS; ++i)
        macHost.addr[i] = 0;

<Overridden member functions for GUSIMTNetDB 903>+≡ (897) <903 909>
    virtual hostent * gethostbyaddr(const void * addr, size_t len, int type);

```

```

(896) 4904 910▶
(Member functions for class GUSIMTNetDB 900)+≡
    hostent * GUSIMTNetDB::gethostbyaddr(const void * addrP, size_t, int)
    {
        (Open MacTCP DNR or fail lookup 905)
        (Declare and initialize macHost and unixHost 906)

        ip_addr addr = *(ip_addr *)addrP;
        if (addr == 0x7F000001)
            addr = static_cast<ip_addr>(gethostid());

        if (AddrToName(addr, &macHost, ResultUPP(&uDNRDone),
            reinterpret_cast<Ptr>(GUSIContext::Current())
            ) == cacheFault
        )
            do {
                GUSIContext::Yield(true);
            } while (macHost.rtnCode == cacheFault);

        if (macHost.rtnCode)
            return GUSISetMacHostError(macHost.rtnCode), static_cast<hostent *>(nil);
        /* For some reason, the IP address usually seems to be set to 0 */
        if (!macHost.addr[0])
            macHost.addr[0] = addr;
        CopyHost(macHost, unixHost);

        return &unixHost;
    }

(897) 4907 911▶
(Overridden member functions for GUSIMTNetDB 903)+≡
    virtual char * inet_ntoa(in_addr inaddr);

(896) 4908 912▶
(Member functions for class GUSIMTNetDB 900)+≡
    char * GUSIMTNetDB::inet_ntoa(in_addr inaddr)
    {
        GUSIhostent & unixHost = *fHost;
        sprintf(unixHost.fAddrString, "%d.%d.%d.%d",
            (inaddr.s_addr >> 24) & 0xFF,
            (inaddr.s_addr >> 16) & 0xFF,
            (inaddr.s_addr >> 8) & 0xFF,
            inaddr.s_addr & 0xFF);
        return unixHost.fAddrString;
    }

(897) 4909
(Overridden member functions for GUSIMTNetDB 903)+≡
    virtual long gethostid();

(896) 4910
(Member functions for class GUSIMTNetDB 900)+≡
    long GUSIMTNetDB::gethostid()
    {
        return static_cast<long>(GUSIMTInetSocket::HostAddr());
    }

```


Chapter 26

Open Transport socket infrastructure

A GUSIOTSocket defines a class of Open Transport sockets. Since most families differ only in a few details, like address representation, we abstract the typical differences in a strategy class GUSIOTStrategy.

```
<GUSIOpenTransport.h 913>≡
    #ifndef _GUSIOpenTransport_
    #define _GUSIOpenTransport_

    #ifdef GUSI_INTERNAL

    #include "GUSISocket.h"
    #include "GUSISocketMixins.h"
    #include "GUSIFactory.h"

    #include <netinet/in.h>
    #include <netinet/tcp.h>

    #undef O_ASYNC
    #undef O_NDELAY
    #undef O_NONBLOCK
    #undef SIGHUP
    #undef SIGURG
    #undef AF_INET
    #undef TCP_KEEPAIVE

    #include <OpenTransport.h>
    #include <OpenTptInternet.h>

    <Definition of class GUSIOTStrategy 918>
    <Definition of class GUSIOTFactory 915>
    <Definition of class GUSIOTStreamFactory 916>
    <Definition of class GUSIOTDatagramFactory 917>
    <Definition of template GUSIOT 922>
    <Definition of class GUSIOTSocket 923>
    <Definition of class GUSIOTStreamSocket 924>
    <Definition of class GUSIOTDatagramSocket 925>
```

```

#endif /* GUSI_INTERNAL */

#endif /* _GUSIOpenTransport_ */

(GUSIOpenTransport.cp 914)≡
#define GUSI_MESSAGE_LEVEL 1

#include "GUSIInternal.h"
#include "GUSIOpenTransport.h"
#include "GUSIDdiag.h"
#include "GUSITimer.h"

#include <stdlib.h>
#include <errno.h>

#include <algorithm>

(Asynchronous notifier function for Open Transport 938)
(Member functions for class GUSIOTFactory 927)
(Member functions for class GUSIOTStreamFactory 929)
(Member functions for class GUSIOTDatagramFactory 931)
(Member functions for class GUSIOTStrategy 933)
(Member functions for class GUSIOTSocket 941)
(Member functions for class GUSIOTStreamSocket 970)
(Member functions for class GUSIOTDatagramSocket 1001)

```

26.1 Definition of GUSIOTFactory and descendants

A GUSIOTFactory is an abstract class combining a socket creation mechanism with a strategy instance. To clarify our intent, we isolate the latter in GUSIOTStrategy.

```

(Definition of class GUSIOTFactory 915)≡ (913)
class GUSIOTFactory : public GUSISocketFactory {
public:
    static bool          Initialize();
protected:
    virtual GUSIOTStrategy *Strategy(int domain, int type, int protocol) = 0;
private:
    (Privatissima of GUSIOTFactory 926)
};

(Definition of class GUSIOTStreamFactory 916)≡ (913)
class GUSIOTStreamFactory : public GUSIOTFactory {
public:
    GUSISocket * socket(int domain, int type, int protocol);
};

(Definition of class GUSIOTDatagramFactory 917)≡ (913)
class GUSIOTDatagramFactory : public GUSIOTFactory {
public:
    GUSISocket * socket(int domain, int type, int protocol);
};

```

26.2 Definition of GUSIOTStrategy

A GUSIOTStrategy contains all the tricky parts of each Open Transport family.

(Definition of class GUSIOTStrategy 918)≡ (913)

```
class GUSIOTStrategy {
public:
    (Strategic interfaces for GUSIOTStrategy 919)
protected:
    (Privatissima of GUSIOTStrategy 920)
};
```

CreateConfiguration creates an appropriate OTConfiguration. This method is not virtual, as it relies on the strategy method ConfigPath.

(Strategic interfaces for GUSIOTStrategy 919)≡ (918) 921▷

```
OTConfiguration *      CreateConfiguration();
```

(Privatissima of GUSIOTStrategy 920)≡ (918) 932▷

```
virtual const char *    ConfigPath() = 0;
```

PackAddress converts a socket address into an OT address, and UnpackAddress performs the reverse step. CopyAddress copies an address.

(Strategic interfaces for GUSIOTStrategy 919)+≡ (918) ◀919

```
virtual int PackAddress(const void * address, socklen_t len, TNetbuf & addr, bool non_null = false) = 0;
virtual int UnpackAddress(const TNetbuf & addr, void * address, socklen_t * len) = 0;
virtual int CopyAddress(const TNetbuf & from, TNetbuf & to);
```

26.3 Definition of GUSIOT

Open Transport allocates and deallocates many data structures, which we simplify with a smart template. Allocation works via class allocation operators, which is a bit weird admittedly.

```
(Definition of template GUSIOT 922)≡ (913)
template <class T, int tag> class GUSIOT : public T {
public:
    void * operator new(size_t, EndpointRef ref)
    { OSStatus err; return OTAlloc(ref, tag, T_ALL, &err); }
    void * operator new(size_t, EndpointRef ref, int fields)
    { OSStatus err; return OTAlloc(ref, tag, fields, &err); }
    void operator delete(void * o)
    { if (o) OTFree(o, tag); }
};
template <class T, int tag> class GUSIOTAddr : public GUSIOT<T, tag> {
public:
    int Pack(GUSIOTStrategy * strategy, const void * address, socklen_t len, bool non_null=false)
    { return strategy->PackAddress(address, len, addr, non_null); }
    int Unpack(GUSIOTStrategy * strategy, void * address, socklen_t * len)
    { return strategy->UnpackAddress(addr, address, len); }
    int Copy(GUSIOTStrategy * strategy, GUSIOTAddr<T, tag> * to)
    { return strategy->CopyAddress(addr, to->addr); }
};

typedef GUSIOTAddr<TBind,      T_BIND>      GUSIOTTBind;
typedef GUSIOTAddr<TCall,      T_CALL>      GUSIOTTCall;
typedef GUSIOTAddr<TUnitData,  T_UNITDATA>  GUSIOTTUnitData;
typedef GUSIOTAddr<TUDErr,      T_UDERROR>  GUSIOTTUDErr;
typedef GUSIOT<TDiscon,         T_DIS>      GUSIOTTDiscon;
typedef GUSIOT<TOptMgmt,        T_OPTMGMT>  GUSIOTTOptMgmt;
```

26.4 Definition of GUSIOTSocket and descendants

Open Transport sockets are rather lightweight, since OT is rather similar to sockets already.

⟨Definition of class GUSIOTSocket 923⟩≡ (913)

```
class GUSIOTSocket :
    public      GUSISocket,
    protected  GUSISMBlocking,
    protected  GUSISMState,
    protected  GUSISMAsyncError
{
public:
    ⟨Overridden member functions for GUSIOTSocket 951⟩
protected:
    GUSIOTSocket(GUSIOTStrategy * strategy);

    ⟨Privatissima of GUSIOTSocket 935⟩

    friend class GUSIOTStreamSocket;
    friend class GUSIOTDatagramSocket;
};
```

⟨Definition of class GUSIOTStreamSocket 924⟩≡ (913)

```
class GUSIOTStreamSocket : public GUSIOTSocket {
public:
    ⟨Overridden member functions for GUSIOTStreamSocket 971⟩
protected:
    GUSIOTStreamSocket(GUSIOTStrategy * strategy);

    ⟨Privatissima of GUSIOTStreamSocket 928⟩
};
```

⟨Definition of class GUSIOTDatagramSocket 925⟩≡ (913)

```
class GUSIOTDatagramSocket : public GUSIOTSocket {
public:
    ⟨Overridden member functions for GUSIOTDatagramSocket 1002⟩
protected:
    GUSIOTDatagramSocket(GUSIOTStrategy * strategy);

    ⟨Privatissima of GUSIOTDatagramSocket 930⟩
};
```

26.5 Implementation of GUSIOTFactory and descendants

⟨Privatissima of GUSIOTFactory 926⟩≡ (915)

```
static bool sOK;
```

<Member functions for class GUSIOTFactory 927>≡ (914)
 bool GUSIOTFactory::sOK = false;

```
bool GUSIOTFactory::Initialize()
{
    if (!sOK)
        sOK = !InitOpenTransport();
    return sOK;
}
```

Since all we need to know is in the GUSIOTStrategy, it often suffices simply to create a GUSIOTSocket. Stream and datagram sockets differ merely in the descendant they create.

<Privatissima of GUSIOTStreamSocket 928>≡ (924) 936»
 friend class GUSIOTStreamFactory;

<Member functions for class GUSIOTStreamFactory 929>≡ (914)
 GUSISocket * GUSIOTStreamFactory::socket(int domain, int type, int protocol)
 {
 GUSIOTStrategy * strategy = Strategy(domain, type, protocol);
 if (Initialize() && strategy)
 return new GUSIOTStreamSocket(strategy);
 else
 return static_cast<GUSISocket *>(0);
 }

<Privatissima of GUSIOTDatagramSocket 930>≡ (925) 1006»
 friend class GUSIOTDatagramFactory;

<Member functions for class GUSIOTDatagramFactory 931>≡ (914)
 GUSISocket * GUSIOTDatagramFactory::socket(int domain, int type, int protocol)
 {
 GUSIOTStrategy * strategy = Strategy(domain, type, protocol);
 if (Initialize() && strategy)
 return new GUSIOTDatagramSocket(strategy);
 else
 return static_cast<GUSISocket *>(0);
 }

26.6 Implementation of GUSIOTStrategy

GUSIOTStrategy is mostly abstract, except for the CreateConfiguration and CopyAddress methods.

<Privatissima of GUSIOTStrategy 920>+≡ (918) 4920
 OTConfiguration * fConfig;
 GUSIOTStrategy() : fConfig(nil) {}
 virtual ~GUSIOTStrategy();

```

<Member functions for class GUSIOTStrategy 933>≡ (914) 934▷
    OTConfiguration * GUSIOTStrategy::CreateConfiguration()
    {
        if (!fConfig)
            fConfig = OTCreateConfiguration(ConfigPath());

        return OTCloneConfiguration(fConfig);
    }

    GUSIOTStrategy::~GUSIOTStrategy()
    {
        if (fConfig)
            OTDestroyConfiguration(fConfig);
    }

```

While you might have to override this for sophisticated addresses, this should do for most.

```

<Member functions for class GUSIOTStrategy 933>+≡ (914) ◀933
    int GUSIOTStrategy::CopyAddress(const TNetbuf & from, TNetbuf & to)
    {
        memcpy(to.buf, from.buf, to.len = from.len);

        return 0;
    }

```

26.7 Implementation of GUSIOTSocket

Open Transport may call this routine for dozens and dozens of different reasons. Pretty much every call results in a wakeup of the threads attached to the socket. We save some of the more interesting events in bitsets. in MopupEvents.

```

<Privatissima of GUSIOTSocket 935>≡ (923) 939▷
    uint16_t      fNewEvent;
    uint16_t      fCurEvent;
    uint16_t      fEvent;
    uint32_t      fNewCompletion;
    uint32_t      fCurCompletion;
    uint32_t      fCompletion;
    friend pascal void GUSIOTNotify(GUSIOTSocket *, OTEventCode, OTResult, void *);

<Privatissima of GUSIOTStreamSocket 928>+≡ (924) ◀928 980▷
    friend pascal void GUSIOTNotify(GUSIOTSocket *, OTEventCode, OTResult, void *);

<Initialize fields of GUSIOTSocket 937>≡ (944) 943▷
    fNewEvent      = 0;
    fEvent         = 0;
    fNewCompletion = 0;
    fCompletion     = 0;

```

(Asynchronous notifier function for Open Transport 938)≡ (914)

```
inline uint32_t CompleteMask(OTEventCode code)
{
    return 1 << (code & 0x1F);
}

pascal void GUSIOTNotify(
    GUSIOTSocket * sock, OTEventCode code, OTResult result, void *cookie)
{
    GUSIOTStreamSocket * ss = dynamic_cast<GUSIOTStreamSocket *>(sock);

    switch (code & 0x7f000000L) {
    case 0:
        sock->fNewEvent |= code;
        break;
    case kCOMPLETEEVENT:
        if (!(code & 0x00FFFFE0))
            sock->fNewCompletion |= CompleteMask(code);
        if (code == T_OPENCOMPLETE)
            sock->fEndpoint = static_cast<EndpointRef>(cookie);
        break;
    }
    switch (result) {
    case kOTLookErr:
        sock->SetAsyncPosixError(kOTLookErr);
        break;
    default:
        sock->SetAsyncMacError(result);
    }
    sock->Wakeup();
}
```

To avoid race conditions with the notifier, we sometimes need a lock.

(Privatissima of GUSIOTSocket 935)+≡ (923) <935 940>

```
class Lock {
public:
    Lock(EndpointRef end) : fEndpoint(end) { OTEnterNotifier(fEndpoint); }
    ~Lock() { OTLeaveNotifier(fEndpoint); }
private:
    EndpointRef fEndpoint;
};
```

For some events, we have to take a followup action at a more convenient time.

(Privatissima of GUSIOTSocket 935)+≡ (923) <939 942>

```
virtual void MopupEvents();
```


<Member functions for class GUSIOTSocket 941>≡ (914) 944▷

```
void GUSIOTSocket::MopupEvents()
{
    {
        Lock    lock(fEndpoint);

        fEvent      |= (fCurEvent      = fNewEvent);
        fCompletion  |= (fCurCompletion = fNewCompletion);
        fNewEvent    = 0;
        fNewCompletion = 0;
    }

    if (fCurEvent & T_UDERR) {
        GUSIOTTUDErr * udErr = new (fEndpoint) GUSIOTTUDErr;

        if (!OTRcvUDErr(fEndpoint, udErr) && udErr)
            SetAsyncMacError(udErr->error);

        delete udErr;
    }
    if (fCurEvent & (T_DISCONNECT | T_ORDREL)) {
        fReadShutdown = true;
    }
}
```

GUSIOTSocket creates an asynchronous endpoint for the appropriate provider.

<Privatissima of GUSIOTSocket 935>+≡ (923) ◀940 945▷

```
GUSIOTStrategy * fStrategy;
EndpointRef      fEndpoint;
linger           fLinger;
```

<Initialize fields of GUSIOTSocket 937>+≡ (944) ◀937 950▷

```
fStrategy      = strategy;
fEndpoint      = nil;
fLinger.l_onoff = false;
fLinger.l_linger = 0;
```

```

⟨Member functions for class GUSIOTSocket 941⟩+≡ (914) ◀941 946▶
GUSIOTSocket::GUSIOTSocket(GUSIOTStrategy * strategy)
{
    ⟨Initialize fields of GUSIOTSocket 937⟩
    SetAsyncMacError(
        OTAsyncOpenEndpoint(
            fStrategy->CreateConfiguration(),
            0, nil,
            reinterpret_cast<OTNotifyProcPtr>(GUSIOTNotify),
            this));
    AddContext();
    MopupEvents();
    while (!fAsyncError && !(fCompletion & CompleteMask(T_OPENCOMPLETE))) {
        GUSIContext::Yield(true);
        MopupEvents();
    }
    RemoveContext();
    if (!fEndpoint)
        GUSISetPosixError(GetAsyncError());
}

```

The destructor tears down the connection as gracefully as possible. It also respects the linger settings.

```

⟨Privatissima of GUSIOTSocket 935⟩+≡ (923) ◀942 948▶
virtual void close();
virtual ~GUSIOTSocket();

⟨Member functions for class GUSIOTSocket 941⟩+≡ (914) ◀944 952▶
void GUSIOTSocket::close()
{
    if (OTGetEndpointState(fEndpoint) != T_IDLE) {
        ⟨Disconnect the GUSIOTSocket, dammit 947⟩
    }

    Unbind();
    OTCloseProvider(fEndpoint);

    GUSISocket::close();
}

GUSIOTSocket::~~GUSIOTSocket()
{
    delete fSockName;
}

```

Under desperate circumstances, we are prepared to employ quite a bit of violence to disconnect the socket.

```

<Disconnect the GUSIOTSocket, dammit 947>≡ (946)
    fCompletion &= ~(CompleteMask(T_DISCONNECTCOMPLETE));
    GUSIOTTCall * call = new (fEndpoint, 0) GUSIOTTCall;
    SetAsyncMacError(OTSndDisconnect(fEndpoint, call));
    delete call;
    AddContext();
    MopupEvents();
    while (!fAsyncError && !(fCompletion & CompleteMask(T_DISCONNECTCOMPLETE))) {
        GUSIContext::Yield(true);
        MopupEvents();
    }
    RemoveContext();
    fAsyncError = 0;

```

Clone creates another socket of the same class.

```

<Privatissima of GUSIOTSocket 935>+≡ (923) <945 949>
    virtual GUSIOTSocket * Clone() = 0;

```

At the time the socket function bind is called, we are not really ready yet to call OTBind, but if we don't call it, we can't report whether the address was free.

```

<Privatissima of GUSIOTSocket 935>+≡ (923) <948 954>
    GUSIOTTCall * fSockName;
    int BindToAddress(GUSIOTTCall * addr);

```

```

<Initialize fields of GUSIOTSocket 937>+≡ (944) <943
    fSockName = nil;

```

```

<Overridden member functions for GUSIOTSocket 951>≡ (923) 956>
    virtual int bind(void * name, socklen_t namelen);

```

```

<Member functions for class GUSIOTSocket 941>+≡ (914) <946 953>
    int GUSIOTSocket::bind(void * name, socklen_t namelen)
    {
        if (fSockName)
            return GUSISetPosixError(EINVAL);

        int res = -1;
        GUSIOTTCall * inName = new (fEndpoint) GUSIOTTCall;
        if (!inName)
            return GUSISetPosixError(ENOMEM);
        if (inName->Pack(fStrategy, name, namelen))
            goto freeInName;
        inName->qlen = 0;

        res = BindToAddress(inName);
    freeInName:
        delete inName;

        return res;
    }

```

```

<Member functions for class GUSIOTSocket 941>+≡ (914) <952 955>
int GUSIOTSocket::BindToAddress(GUSIOTTBind * addr)
{
    fSockName = new (fEndpoint) GUSIOTTBind;
    if (!fSockName)
        return GUSISetPosixError(ENOMEM);
    fCompletion &= ~CompleteMask(T_BINDCOMPLETE);
    SetAsyncMacError(OTBind(fEndpoint, addr, fSockName));
    AddContext();
    MopupEvents();
    while (!fAsyncError && !(fCompletion & CompleteMask(T_BINDCOMPLETE))) {
        GUSIContext::Yield(true);
        MopupEvents();
    }
    RemoveContext();
    if (GUSISetPosixError(GetAsyncError())) {
        delete fSockName;
        fSockName = nil;

        return -1;
    } else
        return 0;
}

```

Open Transport takes unbinding a lot more serious than MacTCP.

```

<Privatissima of GUSIOTSocket 935>+≡ (923) <949
void Unbind();

<Member functions for class GUSIOTSocket 941>+≡ (914) <953 957>
void GUSIOTSocket::Unbind()
{
    fCompletion &= ~(CompleteMask(T_BINDCOMPLETE) | CompleteMask(T_UNBINDCOMPLETE));
    SetAsyncMacError(OTUnbind(fEndpoint));
    AddContext();
    MopupEvents();
    while (!fAsyncError && !(fCompletion & CompleteMask(T_UNBINDCOMPLETE))) {
        GUSIContext::Yield(true);
        MopupEvents();
    }
    RemoveContext();
    fAsyncError = 0;
}

```

getsockname and getpeername unpack the stored addresses. Note that the reaction to nil addresses is a bit different.

```

<Overridden member functions for GUSIOTSocket 951>+≡ (923) <951 958>
virtual int getsockname(void * name, socklen_t * namelen);

<Member functions for class GUSIOTSocket 941>+≡ (914) <955 959>
int GUSIOTSocket::getsockname(void * name, socklen_t * namelen)
{
    if (!fSockName)
        BindToAddress(nil);

    return fSockName->Unpack(fStrategy, name, namelen);
}

```

shutdown() just delegates to GUSISMState.

<Overridden member functions for GUSIOTSocket 951>+≡ (923) <956 960>
virtual int shutdown(int how);

<Member functions for class GUSIOTSocket 941>+≡ (914) <957 961>
int GUSIOTSocket::shutdown(int how)
{
 if (!GUSI_SASSERT_CLIENT(how >= 0 && how < 3, "shutdown: 0,1, or 2\n"))
 return GUSISetPosixError(EINVAL);

 switch (how) {
 case SHUT_RD:
 case SHUT_RDWR:
 Unbind();
 }

 GUSISMState::Shutdown(how);

 return 0;
}

fcntl() handles the blocking support.

<Overridden member functions for GUSIOTSocket 951>+≡ (923) <958 962>
virtual int fcntl(int cmd, va_list arg);

<Member functions for class GUSIOTSocket 941>+≡ (914) <959 963>
int GUSIOTSocket::fcntl(int cmd, va_list arg)
{
 int result;

 if (GUSISMBlocking::DoFcntl(&result, cmd, arg))
 return result;

 GUSI_ASSERT_CLIENT(false, ("fcntl: illegal request %d\n", cmd));

 return GUSISetPosixError(EOPNOTSUPP);
}

ioctl() deals with blocking support and with FIONREAD.

<Overridden member functions for GUSIOTSocket 951>+≡ (923) <960 964>
virtual int ioctl(unsigned int request, va_list arg);

(Member functions for class GUSIOTSocket 941)+≡ (914) <961 965>

```
int GUSIOTSocket::ioctl(unsigned int request, va_list arg)
{
    int result;

    if (GUSISMBlocking::DoIoctl(&result, request, arg))
        return result;
    if (request == FIONREAD) {
        size_t res;
        if (OTCountDataBytes(fEndpoint, &res))
            res = 0;
        *va_arg(arg, size_t *) = res;
        return 0;
    }

    GUSI_ASSERT_CLIENT(false, ("ioctl: illegal request %d\n", request));

    return GUSISetPosixError(EOPNOTSUPP);
}
```

getsockopt and setsockopt are available for a variety of options.

(Overridden member functions for GUSIOTSocket 951)+≡ (923) <962 966>

```
virtual int getsockopt(int level, int optname, void *optval, socklen_t * optlen);
```

```

(Member functions for class GUSIOTSocket 941)⋮≡ (914) ⋮963 967⋮
int GUSIOTSocket::getsockopt(int level, int optname, void *optval, socklen_t * optlen)
{
    int result;

    MopupEvents();

    if (GUSISMAsyncError::DoGetSockOpt(&result, level, optname, optval, optlen))
        return result;

    TOptMgmt    optReq;
    UInt8       optBuffer[ kOTOOptionHeaderSize + sizeof(struct linger) ];
    TOption*    opt      = (TOption*)optBuffer;
    int         len;

    optReq.flags = T_CURRENT;
    optReq.opt.buf = (UInt8*) optBuffer;

    opt->level   = XTI_GENERIC;
    opt->name    = optname;

    switch (level) {
    case SOL_SOCKET:
        switch (optname) {
        case SO_KEEPALIVE:
            len = sizeof(struct t_kpalive);
            break;
        case SO_DEBUG:
        case SO_RCVBUF:
        case SO_SNDBUF:
        case SO_RCVLOWAT:
        case SO_SNDLOWAT:
            len = 4;
            break;
        case SO_LINGER:
            memcpy(optval, &fLinger, sizeof(struct linger));

            return 0;
        default:
            goto notSupported;
        }
        break;
    case IPPROTO_TCP:
        switch (optname) {
        case TCP_KEEPALIVE:
            opt->name = OPT_KEEPALIVE;
            len = sizeof(struct t_kpalive);
            break;
        default:
            goto notSupported;
        }
        break;
    default:
        goto notSupported;
}

```

```

    }
    optReq.opt.len = opt->len = kOTOOptionHeaderSize+len;
    if (GUSISetMacError(OTOOptionManagement(fEndpoint, &optReq, &optReq)))
        return -1;
    switch (optname) {
    case TCP_KEEPALIVE:
        if (level == IPPROTO_TCP)
            *(int *)optval = reinterpret_cast<t_kpalive *>(opt->value)->kp_timeout*60;
        else
            *(int *)optval = reinterpret_cast<t_kpalive *>(opt->value)->kp_onoff;
        *optlen = 4;
        break;
    default:
        memcpy(optval, opt->value, len);
        *optlen = len;
    }

    return 0;
notSupported:
    GUSI_ASSERT_CLIENT(false, ("getsockopt: illegal request %d\n", optname));

    return GUSISetPosixError(EOPNOTSUPP);
}

<Overridden member functions for GUSIOTSocket 951>+≡ (923) <964 968>
virtual int setsockopt(int level, int optname, void *optval, socklen_t optlen);

```



```

(Member functions for class GUSIOTSocket 941)+≡ (914) 4965 969▸
int GUSIOTSocket::setsockopt(int level, int optname, void *optval, socklen_t)
{
    MopupEvents();

    TOptMgmt          optReq;
    UInt8             optBuffer[ kTOptionHeaderSize + sizeof(struct linger) ];
    TOption*          opt      = (TOption*)optBuffer;
    t_kpalive          kpal     = {1, 120};
    int               len;

    optReq.flags = T_NEGOTIATE;
    optReq.opt.buf = (UInt8*) optBuffer;

    opt->level = INET_IP;
    opt->name = optname;

    switch (level) {
    case SOL_SOCKET:
        switch (optname) {
        case SO_KEEPALIVE:
            len = sizeof(struct t_kpalive);
            kpal.kp_onoff = *reinterpret_cast<int *>(optval);
            optval = &kpal;
            break;
        case SO_DEBUG:
        case SO_RCVBUF:
        case SO_SNDBUF:
        case SO_RCVLOWAT:
        case SO_SNDLOWAT:
            len = 4;
            break;
        case SO_LINGER:
            memcpy(&fLinger, optval, sizeof(struct linger));
            break;
        default:
            goto notSupported;
        }
        break;
    case IPPROTO_TCP:
        switch (optname) {
        case TCP_KEEPALIVE:
            len = sizeof(struct t_kpalive);
            kpal.kp_timeout = (*reinterpret_cast<int *>(optval) + 30) / 60;
            optval = &kpal;
            break;
        default:
            goto notSupported;
        }
        break;
    default:
        goto notSupported;
    }
    optReq.opt.len = opt->len = kTOptionHeaderSize+len;
}

```

```

memcpy(opt->value, optval, len);

return GUSISetMacError(OTOptionManagement(fEndpoint, &optReq, &optReq));
notSupported:
GUSI_ASSERT_CLIENT(false, ("setsockopt: illegal request %d\n", optname));

return GUSISetPosixError(EOPNOTSUPP);
}

```

Open Transport sockets implement socket style calls.

(Overridden member functions for GUSIOTSocket 951)+≡ (923) <966
virtual bool Supports(ConfigOption config);

(Member functions for class GUSIOTSocket 941)+≡ (914) <967
bool GUSIOTSocket::Supports(ConfigOption config)
{
return config == kSocketCalls;
}

26.8 Implementation of GUSIOTStreamSocket

GUSIOTStreamSockets have to be connected and send data in a continuous stream.

(Member functions for class GUSIOTStreamSocket 970)+≡ (914) 972»
GUSIOTStreamSocket::GUSIOTStreamSocket(GUSIOTStrategy * strategy)
: GUSIOTSocket(strategy)
{
(Initialize fields of GUSIOTStreamSocket 981)
}

Clone creates another socket of the same class.

(Overridden member functions for GUSIOTStreamSocket 971)+≡ (924) 973»
virtual GUSIOTSocket * Clone();

(Member functions for class GUSIOTStreamSocket 970)+≡ (914) <970 974»
GUSIOTSocket * GUSIOTStreamSocket::Clone()
{
return new GUSIOTStreamSocket(fStrategy);
}

(Overridden member functions for GUSIOTStreamSocket 971)+≡ (924) <971 975»
virtual void close();
~GUSIOTStreamSocket();

```

{Member functions for class GUSIOTStreamSocket 970}+≡ (914) ◀972 976▶
void GUSIOTStreamSocket::close()
{
    OTSndOrderlyDisconnect(fEndpoint);

    GUSITimer    lingerer;

    if (fLinger.l_onoff)
        lingerer.Sleep(fLinger.l_linger*1000+1);

    AddContext();
    MopupEvents();
    while (OTGetEndpointState(fEndpoint) > T_IDLE)
        if (fLinger.l_onoff && !lingerer.Primed()) // Linger timer ran out
            break;
        else if (GUSIContext::Yield(true))
            break;
        else
            MopupEvents();
    RemoveContext();

    GUSIOTSocket::close();
}

GUSIOTStreamSocket::~GUSIOTStreamSocket()
{
    delete fPeerName;
}

```

Stream sockets include a mopup action for connect and disconnect.

```

{Overridden member functions for GUSIOTStreamSocket 971}+≡ (924) ◀973 977▶
virtual void MopupEvents();

```

```

<Member functions for class GUSIOTStreamSocket 970>+≡ (914) <974 978>
void GUSIOTStreamSocket::MopupEvents()
{
    GUSIOTSocket::MopupEvents();

    if (fCurEvent & T_CONNECT) {
        GUSI_MESSAGE(("Connect\n"));
        OTRcvConnect(fEndpoint, fPeerName);
    }
    if (fCurEvent & T_ORDREL) {
        OTRcvOrderlyDisconnect(fEndpoint);
        GUSI_MESSAGE(("Orderly Disconnect\n"));
    }
    if (fCurEvent & T_DISCONNECT) {
        GUSIOTDiscon * discon = new (fEndpoint) GUSIOTDiscon;

        OTRcvDisconnect(fEndpoint, discon);
        if (discon) {
            GUSI_MESSAGE(("Disconnect %d\n", XTI2OSSStatus(discon->reason)));
            SetAsyncMacError(XTI2OSSStatus(discon->reason));

            delete discon;
        } else {
            GUSI_MESSAGE(("Disconnect\n"));
        }
    }
}

```

listen is a bit embarrassing, because we already committed ourselves to a queue length of 0, so we have to unbind and rebind ourselves.

```

<Overridden member functions for GUSIOTStreamSocket 971>+≡ (924) <975 982>
virtual int listen(int qlen);

<Member functions for class GUSIOTStreamSocket 970>+≡ (914) <976 983>
int GUSIOTStreamSocket::listen(int queueLength)
{
    Unbind();
    <Fudge queueLength 979>
    fSockName->qlen = queueLength;
    SetAsyncMacError(OTBind(fEndpoint, fSockName, nil));
    AddContext();
    MopupEvents();
    while (!fAsyncError && !(fCompletion & CompleteMask(T_BINDCOMPLETE))) {
        GUSIContext::Yield(true);
        MopupEvents();
    }
    RemoveContext();

    return GUSISetPosixError(GetAsyncError());
}

```

For some weird reason, BSD multiplies queue lengths with a fudge factor.

```

<Fudge queueLength 979>≡ (978)
    if (queueLength < 1)
        queueLength = 1;
    else if (queueLength > 4)
        queueLength = 8;
    else
        queueLength = ((queueLength * 3) >> 1) + 1;

```

The peer address for a GUSIOTStreamSocket is stored in a GUSIOTTCall structure.

```

<Privatissima of GUSIOTStreamSocket 928>+≡ (924) <936 985>
    GUSIOTTCall * fPeerName;

```

```

<Initialize fields of GUSIOTStreamSocket 981>≡ (970) 986>
    fPeerName = nil;

```

```

<Overridden member functions for GUSIOTStreamSocket 971>+≡ (924) <977 984>
    virtual int getpeername(void * name, socklen_t * namelen);

```

```

<Member functions for class GUSIOTStreamSocket 970>+≡ (914) <978 987>
    int GUSIOTStreamSocket::getpeername(void * name, socklen_t * namelen)
    {
        MopupEvents();

        if (!fPeerName)
            return GUSISetPosixError(ENOTCONN);
        return fPeerName->Unpack(fStrategy, name, namelen);
    }

```

accept may become quite complex, because connections could nest. The listening socket calls OTListen, queues candidates by their fNextListener field, and then tries calling OTAccept on the first candidate.

```

<Overridden member functions for GUSIOTStreamSocket 971>+≡ (924) <982 991>
    virtual GUSISocket * accept(void * address, socklen_t * addrlen);

```

```

<Privatissima of GUSIOTStreamSocket 928>+≡ (924) <980
    GUSIOTStreamSocket * fNextListener;
    GUSIOTStreamSocket * fLastListener;

```

```

<Initialize fields of GUSIOTStreamSocket 981>+≡ (970) <981
    fNextListener = nil;
    fLastListener = nil;

```

```

<Member functions for class GUSIOTStreamSocket 970>+≡ (914) <983 992>
GUSISocket * GUSIOTStreamSocket::accept(void * address, socklen_t * addrlen)
{
    MopupEvents();

    bool    mustListen = false;
    for (;;) {
        if (mustListen || !fNextListener) {
            <Call OTListen and queue a candidate 988>
            mustListen = false;
        }
        if (fNextListener) {
            <Call OTAccept and return if successful 990>
        }
    }
}

<Call OTListen and queue a candidate 988>≡ (987)
GUSIOTTCall * call = new (fEndpoint) GUSIOTTCall;
if (!call)
    return GUSISetPosixError(ENOMEM), static_cast<GUSISocket *>(0);
OTResult err;
bool signal = false;
AddContext();
for (;;) {
    err = OTListen(fEndpoint, call);
    if (!fBlocking || (err != kOTNoDataErr))
        break;
    if (signal = GUSIContext::Yield(true))
        break;
}
RemoveContext();
<return an appropriate error if GUSIOTStreamSocket::accept failed 989>
GUSIOTStreamSocket * candidate = dynamic_cast<GUSIOTStreamSocket *>(Clone());
candidate->fPeerName = call;
if (!fNextListener)
    fNextListener = candidate;
else
    fLastListener->fNextListener = candidate;
fLastListener = candidate;

<return an appropriate error if GUSIOTStreamSocket::accept failed 989>≡ (988)
if (signal)
    return GUSISetPosixError(EINTR), static_cast<GUSISocket *>(0);
if (err) {
    delete call;

    if (err==kOTNoDataErr)
        GUSISetPosixError(EWOULDBLOCK);
    else
        GUSISetMacError(err);

    return static_cast<GUSISocket *>(0);
}

```

```

( Call OTAccept and return if successful 990 )≡ (987)
GUSIOTSocket * sock = fNextListener;
fCompletion &= ~(CompleteMask(T_ACCEPTCOMPLETE));
SetAsyncMacError(OTAccept(fEndpoint, fNextListener->fEndpoint, fNextListener->fPeerName));
AddContext();
MopupEvents();
while (!fAsyncError && !(fCompletion & CompleteMask(T_ACCEPTCOMPLETE))) {
    GUSIContext::Yield(true);
    MopupEvents();
}
RemoveContext();
switch (GetAsyncError()) {
case kOTLookErr:
    switch (OTLook(fEndpoint)) {
    case T_LISTEN: // Another connection has arrived
        mustListen = true;
        continue;
    case T_DISCONNECT: // Peer disconnected already
        goto deleteCandidate;
    }
    break;
case 0:
    fNextListener = fNextListener->fNextListener;

    sock->getpeername(address, addrlen);
    sock->fSockName = new (fEndpoint) GUSIOTTBind;
    if (sock->fSockName && !fSockName->Copy(fStrategy, sock->fSockName))
        return sock;

    // Fall through
default:
deleteCandidate:
    fNextListener = fNextListener->fNextListener;

    delete sock;
}

connect is comparatively simple.
( Overridden member functions for GUSIOTStreamSocket 971 )+≡ (924) 4984 993▷
virtual int connect(void * address, socklen_t addrlen);

```

(Member functions for class GUSIOTStreamSocket 970)+≡ (914) <987 994>

```
int GUSIOTStreamSocket::connect(void * address, socklen_t addrlen)
{
    MopupEvents();
    OTResult res = 0;
restart:
    switch (OTGetEndpointState(fEndpoint)) {
    case T_OUTCON:
        if (!fBlocking)
            return GUSISetPosixError(EALREADY);
        break;
    case T_UNBND:
        if (BindToAddress(nil))
            return -1;
        // Fall through
    case T_IDLE:
        fPeerName = new (fEndpoint) GUSIOTTCall;
        if (!fPeerName)
            return GUSISetPosixError(GetAsyncError());
        if (fPeerName->Pack(fStrategy, address, addrlen, true))
            goto freePeerName;
        fEvent &= ~(T_CONNECT|T_DISCONNECT);
        res = OTConnect(fEndpoint, fPeerName, nil);
        MopupEvents();
        break;
    default:
        return GUSISetPosixError(EISCONN);
    }
    if (!fBlocking && !(fEvent & (T_CONNECT|T_DISCONNECT)))
        return GUSISetPosixError(EINPROGRESS);
    {
        bool signal = false;
        AddContext();
        MopupEvents();
        while (!signal && !(fEvent & (T_CONNECT|T_DISCONNECT))) {
            signal = GUSIContext::Yield(true);
            MopupEvents();
        }
        RemoveContext();
        if (signal)
            return GUSISetPosixError(EINTR);
    }
    if (fEvent & T_CONNECT)
        return 0;
    GUSISetPosixError(GetAsyncError());
freePeerName:
    delete fPeerName;
    fPeerName = nil;

    return -1;
}
```


Data transfer is simple as well. Here is the version for stream protocols.

```
<Overridden member functions for GUSIOTStreamSocket 971>+≡ (924) <991 995>
virtual ssize_t recvfrom(const GUSIScatterer & buffer, int flags, void * from, socklen_t * fromlen);

<Member functions for class GUSIOTStreamSocket 970>+≡ (914) <992 996>
ssize_t GUSIOTStreamSocket::recvfrom(const GUSIScatterer & buffer, int flags, void * from, socklen_t * fromlen)
{
    MopupEvents();
    OTFlags otflags;
    uint16_t exp = fEvent & T_EXDATA;
    fEvent ^= exp;
    OTResult res = OTRcv(fEndpoint, buffer.Buffer(), buffer.Length(), &otflags);
    if (res == kOTNoDataErr) {
        if (GUSISetPosixError(GetAsyncError()))
            return -1;
        if (fReadShutdown)
            return 0;
        if (!fBlocking)
            return GUSISetPosixError(EWOULDBLOCK);
        bool signal = false;
        AddContext();
        MopupEvents();
        while (res == kOTNoDataErr && !(res = fAsyncError)) {
            signal = GUSIContext::Yield(true);
            MopupEvents();
            if (signal)
                break;
            res = OTRcv(fEndpoint, buffer.Buffer(), buffer.Length(), &otflags);
        }
        RemoveContext();
        if (signal)
            return GUSISetPosixError(EINTR);
    }
    if (res < 0)
        return GUSISetMacError(res);
    else
        buffer.SetLength(res);
    if (from)
        fPeerName->Unpack(fStrategy, from, fromlen);
    if (exp && (otflags & (T_EXPEDITED|T_MORE)) != T_EXPEDITED) {
        fEvent |= exp;
    }
    return res;
}

<Overridden member functions for GUSIOTStreamSocket 971>+≡ (924) <993 997>
virtual ssize_t sendto(const GUSIGatherer & buffer, int flags, const void * to, socklen_t tolen);
```

(Member functions for class GUSIOTStreamSocket 970)+≡

(914) <994 998>

```

ssize_t GUSIOTStreamSocket::sendto(const GUSIGatherer & buffer, int flags, const void * to, socklen_t
{
    MopupEvents();
    if (GUSISetPosixError(GetAsyncError()))
        return -1;
    long done= 0;
    char * buf = static_cast<char *>(buffer.Buffer());
    long len = buffer.Length();
    while (len) {
        OTResult res = OTSnd(fEndpoint, buf, len, 0);
        if (res <= 0)
            if (res == kOTFlowErr) {
                if (!fBlocking)
                    return done ? done : GUSISetPosixError(EWOULDBLOCK);
                AddContext();
                bool signal = GUSIContext::Yield(true);
                MopupEvents();
                RemoveContext();
                if (signal)
                    return done ? done : GUSISetPosixError(EINTR);
            } else
                return GUSISetMacError(res);
        else {
            buf += res;
            len -= res;
            done+= res;
        }
    }
    return buffer.Length();
}

```

select for stream sockets intermingles data information and connection information as usual.

(Overridden member functions for GUSIOTStreamSocket 971)+≡

(924) <995 999>

```

virtual bool select(bool * canRead, bool * canWrite, bool * except);

```

<Member functions for class GUSIOTStreamSocket 970>+≡ (914) <996 1000>

```
bool GUSIOTStreamSocket::select(bool * canRead, bool * canWrite, bool * except)
{
    MopupEvents();

    bool      res      = false;
    OTResult   state    = OTGetEndpointState(fEndpoint);

    if (canRead) {
        size_t sz;
        if (*canRead =
            fReadShutdown                // EOF
            || fAsyncError                // Asynchronous error
            || (!OTCountDataBytes(fEndpoint, &sz) && sz) // Data available
            || state == T_INCON           // Connection pending
            || fNextListener              // Connection pending
        )
            res = true;
    }
    if (canWrite)
        if (fWriteShutdown || fAsyncError)
            res = *canWrite = true;
        else
            switch (state) {
            case T_DATAXFER:
            case T_INREL:
                *canWrite = true;
                res        = true;
                break;
            default:
                *canWrite = false;
            }
    if (except)
        if (*except = (fEvent & T_EXDATA) != 0)
            res = true;
    return res;
}
```

shutdown() for stream sockets has to send an orderly disconnect.

<Overridden member functions for GUSIOTStreamSocket 971>+≡ (924) <997

```
virtual int shutdown(int how);
```

<Member functions for class GUSIOTStreamSocket 970>+≡ (914) <998

```
int GUSIOTStreamSocket::shutdown(int how)
{
    switch (how) {
    case SHUT_WR:
    case SHUT_RDWR:
        OTSndOrderlyDisconnect(fEndpoint);
    }

    return GUSIOTSocket::shutdown(how);
}
```

26.9 Implementation of GUSIOTDatagramSocket

GUSIOTDatagramSockets don't need to be connected.

```
<Member functions for class GUSIOTDatagramSocket 1001>≡ (914) 1003»
    GUSIOTDatagramSocket::GUSIOTDatagramSocket(GUSIOTStrategy * strategy)
        : GUSIOTSocket(strategy)
    {
        <Initialize fields of GUSIOTDatagramSocket 1009>
    }
```

Clone creates another socket of the same class.

```
<Overridden member functions for GUSIOTDatagramSocket 1002>≡ (925) 1004»
    virtual GUSIOTSocket * Clone();

<Member functions for class GUSIOTDatagramSocket 1001>+≡ (914) <1001 1005»
    GUSIOTSocket * GUSIOTDatagramSocket::Clone()
    {
        return new GUSIOTDatagramSocket(fStrategy);
    }

<Overridden member functions for GUSIOTDatagramSocket 1002>+≡ (925) <1002 1010»
    ~GUSIOTDatagramSocket();

<Member functions for class GUSIOTDatagramSocket 1001>+≡ (914) <1003 1007»
    GUSIOTDatagramSocket::~GUSIOTDatagramSocket()
    {
        delete fPeerName;
    }
```

Datagram sockets might be bound at rather arbitrary times.

```
<Privatissima of GUSIOTDatagramSocket 930>+≡ (925) <930 1008»
    int BindIfUnbound();

<Member functions for class GUSIOTDatagramSocket 1001>+≡ (914) <1005 1011»
    int GUSIOTDatagramSocket::BindIfUnbound()
    {
        if (OTGetEndpointState(fEndpoint) == T_UNBND)
            return BindToAddress(nil);
        return 0;
    }
```

The peer address for a GUSIOTDatagramSocket is stored in a GUSIOTTBind structure.

```
<Privatissima of GUSIOTDatagramSocket 930>+≡ (925) <1006
    GUSIOTTBind * fPeerName;

<Initialize fields of GUSIOTDatagramSocket 1009>≡ (1001)
    fPeerName = nil;

<Overridden member functions for GUSIOTDatagramSocket 1002>+≡ (925) <1004 1012»
    virtual int getpeername(void * name, socklen_t * namelen);

<Member functions for class GUSIOTDatagramSocket 1001>+≡ (914) <1007 1013»
    int GUSIOTDatagramSocket::getpeername(void * name, socklen_t * namelen)
    {
        if (!fPeerName)
            return GUSISetPosixError(ENOTCONN);
        return fPeerName->Unpack(fStrategy, name, namelen);
    }
```

A datagram socket can connect as many times as it wants.

⟨Overridden member functions for GUSIOTDatagramSocket 1002⟩+≡ (925) ◀1010 1014▶
virtual int connect(void * address, socklen_t addrlen);

⟨Member functions for class GUSIOTDatagramSocket 1001⟩+≡ (914) ◀1011 1015▶
int GUSIOTDatagramSocket::connect(void * address, socklen_t addrlen)
{
 MopupEvents();
 if (BindIfUnbound())
 return -1;
 if (!fPeerName) {
 fPeerName = new (fEndpoint) GUSIOTTBind;
 if (!fPeerName)
 return GUSISetPosixError(ENOMEM);
 }
 if (fPeerName->Pack(fStrategy, address, addrlen, true)) {
 delete fPeerName;
 fPeerName = nil;

 return -1;
 }
 return 0;
}

Datagram protocols use slightly different calls for data transfers.

⟨Overridden member functions for GUSIOTDatagramSocket 1002⟩+≡ (925) ◀1012 1018▶
virtual ssize_t recvfrom(const GUSIScatterer & buffer, int flags, void * from, socklen_t * fromlen);

```

<Member functions for class GUSIOTDatagramSocket 1001>+≡ (914) <1013 1019>
ssize_t GUSIOTDatagramSocket::recvfrom(const GUSIScatterer & buffer, int flags, void * from, socklen_t * fromlen)
{
    MopupEvents();
    GUSIOTTUnitData * data = new (fEndpoint, T_ADDR|T_OPT) GUSIOTTUnitData;
    if (!data)
        return GUSISetPosixError(ENOMEM);
    data->udata.len = data->udata.maxlen = buffer.Length();
    data->udata.buf = static_cast<UInt8 *>(buffer.Buffer());
retry:
    OTFlags otflags;
    OTResult res;
    while ((res = OTRcvUData(fEndpoint, data, &otflags)) == kOTNoDataErr) {
        if (GUSISetPosixError(GetAsyncError()))
            res = -1;
        else if (fReadShutdown)
            res = 0;
        else if (!fBlocking)
            res = GUSISetPosixError(EWOULDBLOCK);
        else {
            AddContext();
            bool signal = GUSIContext::Yield(true);
            MopupEvents();
            RemoveContext();
            if (signal)
                return GUSISetPosixError(EINTR);
            continue;
        }
        goto killDataAndReturn;
    }
    if (res < 0) {
        GUSISetMacError(res);
        res = -1;
    } else {
        <Compare received address to stored peer address 1016>
        if (from)
            data->Unpack(fStrategy, from, fromlen);
    }
    if (res >= 0) {
        res = buffer.SetLength(data->udata.len);
        <Read rest of datagram if buffer allocation was too stingy 1017>
    }
killDataAndReturn:
    data->udata.buf = nil;
    delete data;
    return res;
}

```

If we called connect on the socket, we want to restrict peer addresses.

⟨Compare received address to stored peer address 1016⟩≡ (1015)

```
if (fPeerName)
    if (memcmp(
        fPeerName->addr.buf, data->addr.buf,
        min(fPeerName->addr.len, data->addr.len))
    )
        goto retry; // No match
```

Open Transport requires one to read the entire datagram, so if there was a rest, we use a temporary buffer. This part may be called recursively.

⟨Read rest of datagram if buffer allocation was too stingy 1017⟩≡ (1015)

```
if (otflags & T_MORE) {
    char * buf = new char[1024];
    recvfrom(GUSIScatterer(buf, 1024), 0, nil, nil);
}
```

sendto needs either a valid to address or a stored peer address set by connect.

⟨Overridden member functions for GUSIOTDatagramSocket 1002⟩+≡ (925) ◀1014 1020▶

```
virtual ssize_t sendto(const GUSIGatherer & buffer, int flags, const void * to, socklen_t tolen);
```

```

<Member functions for class GUSIOTDatagramSocket 1001>+≡ (914) <1015 1021>
ssize_t GUSIOTDatagramSocket::sendto(const GUSIGatherer & buffer, int flags, const void * to, sockl
{
    MopupEvents();
    if (GUSISetPosixError(GetAsyncError()))
        return -1;
    if (!to && !fPeerName)
        return GUSISetPosixError(ENOTCONN);
    GUSIOTTUnitData * data = new (fEndpoint, to ? T_ADDR|T_OPT : T_OPT) GUSIOTTUnitData;
    if (!data)
        return GUSISetPosixError(ENOMEM);
    if (to) {
        if (data->Pack(fStrategy, to, tolen, true))
            return -1;
    } else if (fPeerName) {
        data->addr = fPeerName->addr;
    }
    data->udata.len = data->udata.maxlen = buffer.Length();
    data->udata.buf = static_cast<UInt8 *>(buffer.Buffer());
    OTResult res;
    while ((res = OTSndUDData(fEndpoint, data)) == kOTFlowErr) {
        if (!fBlocking)
            return GUSISetPosixError(EWOULDBLOCK);
        AddContext();
        bool signal = GUSIContext::Yield(true);
        MopupEvents();
        RemoveContext();
        if (signal)
            return GUSISetPosixError(EINTR);
    }
    data->udata.buf = nil;
    if (!to)
        data->addr.buf = nil;
    if (res < 0)
        return GUSISetMacError(res);
    else
        res = data->udata.len;
    delete data;

    return res;
}

```

select for datagram sockets returns data information only.

```

<Overridden member functions for GUSIOTDatagramSocket 1002>+≡ (925) <1018
virtual bool select(bool * canRead, bool * canWrite, bool * except);

```



```

<Member functions for class GUSIOTDatagramSocket 1001>+≡ (914) <1019
bool GUSIOTDatagramSocket::select(bool * canRead, bool * canWrite, bool * except)
{
    MopupEvents();
    bool      res      = false;
    OTResult   state    = OTGetEndpointState(fEndpoint);

    if (canRead) {
        size_t sz;
        if (*canRead = fAsyncError || (!OTCountDataBytes(fEndpoint, &sz) && sz))
            res = true;
    }
    if (canWrite)
        switch (state) {
            case T_IDLE:
            case T_DATAXFER:
            case T_INREL:
                *canWrite = true;
                res      = true;
                break;
            default:
                *canWrite = fAsyncError != 0;
        }
    if (except)
        *except = false;

    return res;
}

```


Chapter 27

Open Transport TCP/IP sockets

For TCP and UDP, the strategy classes do most of the work, the derived socket classes only have to do option management.

```
<GUSIOTInet.h 1022>≡
  #ifndef _GUSIOTInet_
  #define _GUSIOTInet_

  #ifdef GUSI_SOURCE

  #include <sys/cdefs.h>

  __BEGIN_DECLS
  <Definition of GUSIwithOTInetSockets 1024>
  __END_DECLS

  #endif /* GUSI_SOURCE */

  #endif /* _GUSIOTInet_ */
```

```

<GUSIOTInet.cp 1023>≡
#include "GUSIInternal.h"
#include "GUSIOTInet.h"
#include "GUSIOpenTransport.h"
#include "GUSIOTNetDB.h"
#include "GUSIInet.h"
#include "GUSIDdiag.h"

#include <stdlib.h>
#include <errno.h>
#include <netinet/in.h>
#include <net/if.h>

<Definition of class GUSIOTTcpFactory 1025>
<Definition of class GUSIOTUdpFactory 1026>
<Definition of class GUSIOTInetStrategy 1027>
<Definition of class GUSIOTMInetOptions 1029>
<Definition of class GUSIOTTcpStrategy 1028>
<Definition of class GUSIOTTcpSocket 1032>
<Definition of class GUSIOTUdpStrategy 1030>
<Definition of class GUSIOTUdpSocket 1031>
<Member functions for class GUSIOTTcpFactory 1034>
<Member functions for class GUSIOTUdpFactory 1038>
<Member functions for class GUSIOTInetStrategy 1041>
<Member functions for class GUSIOTMInetOptions 1045>
<Member functions for class GUSIOTTcpStrategy 1051>
<Member functions for class GUSIOTTcpSocket 1053>
<Member functions for class GUSIOTUdpStrategy 1058>
<Member functions for class GUSIOTUdpSocket 1060>
<Implementation of GUSIwithOTInetSockets 1065>

```

27.1 Definition of Open Transport Internet hooks

```

<Definition of GUSIwithOTInetSockets 1024>≡ (1022)
void GUSIwithOTInetSockets();
void GUSIwithOTTcpSockets();
void GUSIwithOTUdpSockets();

```

27.2 Definition of GUSIOTTcpFactory

The only way that the GUSIOTTcpFactory differs from the GUSIOTFactory is that it defines a strategy and an instantiation mechanism.

(Definition of class GUSIOTTcpFactory 1025)≡ (1023)

```
class GUSIOTTcpFactory : public GUSIOTStreamFactory {
public:
    static GUSISocketFactory * Instance();
protected:
    GUSIOTTcpFactory() {}

    GUSISocket * socket(int domain, int type, int protocol);
    virtual GUSIOTStrategy *Strategy(int domain, int type, int protocol);
private:
    (Privatissima of GUSIOTTcpFactory 1033)
};
```

27.3 Definition of GUSIOTUdpFactory

The GUSIOTUdpFactory is the datagram equivalent of GUSIOTTcpFactory. It returns its own socket typ to support multicasting.

(Definition of class GUSIOTUdpFactory 1026)≡ (1023)

```
class GUSIOTUdpFactory : public GUSIOTDatagramFactory {
public:
    static GUSISocketFactory * Instance();
protected:
    GUSIOTUdpFactory() {}

    GUSISocket * socket(int domain, int type, int protocol);
    virtual GUSIOTStrategy *Strategy(int domain, int type, int protocol);
private:
    (Privatissima of GUSIOTUdpFactory 1037)
};
```

27.4 Definition of GUSIOTInetStrategy

TCP and UDP sockets use different creation configurations, but the same address packing routines.

(Definition of class GUSIOTInetStrategy 1027)≡ (1023)

```
class GUSIOTInetStrategy : public GUSIOTStrategy {
protected:
    virtual int PackAddress(const void * address, socklen_t len, TNetbuf & addr, bool non_null);
    virtual int UnpackAddress(const TNetbuf & addr, void * address, socklen_t * len);
};
```

27.5 Definition of GUSIOTTcpStrategy

The GUSIOTTcpStrategy defines the configuration for TCP sockets

```
(Definition of class GUSIOTTcpStrategy 1028)≡ (1023)
class GUSIOTTcpStrategy : public GUSIOTInetStrategy {
protected:
    virtual const char *    ConfigPath();
};
```

27.6 Definition of GUSIOTMInetOptions

The options common to TCP and UDP sockets are implemented in the mixin class GUSIOTMInetOptions.

```
(Definition of class GUSIOTMInetOptions 1029)≡ (1023)
class GUSIOTMInetOptions {
protected:
    bool    DoGetSockOpt(
        int * result, EndpointRef endpoint, int level, int optname,
        void *optval, socklen_t * optlen);
    bool    DoSetSockOpt(
        int * result, EndpointRef endpoint, int level, int optname,
        void *optval, socklen_t optlen);
    bool    DoIoctl(int * result, unsigned int request, va_list arg);
};
```

27.7 Definition of GUSIOTUdpStrategy

The GUSIOTUdpStrategy defines the configuration for UDP sockets

```
(Definition of class GUSIOTUdpStrategy 1030)≡ (1023)
class GUSIOTUdpStrategy : public GUSIOTInetStrategy {
protected:
    virtual const char *    ConfigPath();
};
```

27.8 Definition of GUSIOTUdpSocket

A GUSIOTUdpSocket supports UDP datagram services over Open Transport.

```
(Definition of class GUSIOTUdpSocket 1031)≡ (1023)
class GUSIOTUdpSocket : public GUSIOTDatagramSocket, protected GUSIOTMInetOptions {
public:
    (Overridden member functions for GUSIOTUdpSocket 1059)
protected:
    GUSIOTUdpSocket(GUSIOTStrategy * strategy) : GUSIOTDatagramSocket(strategy) {}

    friend class GUSIOTUdpFactory;
};
```

27.9 Definition of GUSIOTTcpSocket

A GUSIOTUdpSocket supports TCP stream services over Open Transport.

```
<Definition of class GUSIOTTcpSocket 1032>≡ (1023)
class GUSIOTTcpSocket : public GUSIOTStreamSocket, protected GUSIOTMInetOptions {
public:
    <Overridden member functions for GUSIOTTcpSocket 1052>
protected:
    GUSIOTTcpSocket(GUSIOTStrategy * strategy) : GUSIOTStreamSocket(strategy) {}

    friend class GUSIOTTcpFactory;
};
```

27.10 Implementation of GUSIOTTcpFactory

```
<Privatissima of GUSIOTTcpFactory 1033>≡ (1025)
static GUSISocketFactory * sInstance;
```

```
<Member functions for class GUSIOTTcpFactory 1034>≡ (1023) 1035▶
GUSISocketFactory * GUSIOTTcpFactory::sInstance;
```

```
GUSISocketFactory * GUSIOTTcpFactory::Instance()
{
    if (!sInstance)
        sInstance = new GUSIOTTcpFactory;
    return sInstance;
}
```

The real smarts of GUSIOTTcpFactory are in the Strategy member function.

```
<Member functions for class GUSIOTTcpFactory 1034>+≡ (1023) ◀1034 1036▶
GUSIOTStrategy * GUSIOTTcpFactory::Strategy(int, int, int)
{
    static GUSIOTStrategy * tcpStrategy = new GUSIOTTcpStrategy;

    return tcpStrategy;
}
```

OpenTransport TCP sockets have a little bit of added functionality over stream sockets.

```
<Member functions for class GUSIOTTcpFactory 1034>+≡ (1023) ◀1035
GUSISocket * GUSIOTTcpFactory::socket(int domain, int type, int protocol)
{
    GUSIOTStrategy * strategy = Strategy(domain, type, protocol);
    if (Initialize() && strategy)
        return new GUSIOTTcpSocket(strategy);
    else
        return static_cast<GUSISocket *>(0);
}
```

27.11 Implementation of GUSIOTUdpFactory

```
<Privatissima of GUSIOTUdpFactory 1037>≡ (1026)
static GUSISocketFactory * sInstance;
```

(Member functions for class GUSIOTUdpFactory 1038)≡ (1023) 1039»
 GUSISocketFactory * GUSIOTUdpFactory::sInstance;

```
GUSISocketFactory * GUSIOTUdpFactory::Instance()
{
    if (!sInstance)
        sInstance = new GUSIOTUdpFactory;
    return sInstance;
}
```

The real smarts of GUSIOTUdpFactory are in the Strategy member function.

(Member functions for class GUSIOTUdpFactory 1038)+≡ (1023) «1038 1040»
 GUSIOTStrategy * GUSIOTUdpFactory::Strategy(int, int, int)

```
{
    static GUSIOTStrategy * udpStrategy = new GUSIOTUdpStrategy;

    return udpStrategy;
}
```

OpenTransport UDP sockets have a little bit of added functionality over datagram sockets.

(Member functions for class GUSIOTUdpFactory 1038)+≡ (1023) «1039
 GUSISocket * GUSIOTUdpFactory::socket(int domain, int type, int protocol)

```
{
    GUSIOTStrategy * strategy = Strategy(domain, type, protocol);
    if (Initialize() && strategy)
        return new GUSIOTUdpSocket(strategy);
    else
        return static_cast<GUSISocket *>(0);
}
```

27.12 Implementation of GUSIOTInetStrategy

GUSIOTInetStrategy defines the packing and unpacking functions.

(Member functions for class GUSIOTInetStrategy 1041)≡ (1023) 1043»

```
int GUSIOTInetStrategy::PackAddress(const void * address, socklen_t len, TNetbuf & addr, bool non_r
{
    const sockaddr_in *name = (const sockaddr_in *)address;
    (Sanity checks for GUSIOTInetStrategy::PackAddress 1042)
    OTInitInetAddress(
        reinterpret_cast<InetAddress *>(addr.buf),
        name->sin_port, name->sin_addr.s_addr);
    addr.len = 16;

    return 0;
}
```



```

<Sanity checks for GUSIOTInetStrategy::PackAddress 1042>≡ (1041)
    if (!GUSI_ASSERT_CLIENT(
        len >= sizeof(struct sockaddr_in),
        ("PackAddress: address len %d < %d\n", len, sizeof(struct sockaddr_in)))
    )
        return GUSISetPosixError(EINVAL);
    if (!GUSI_ASSERT_CLIENT(
        name->sin_family == AF_INET,
        ("PackAddress: family %d != %d\n", name->sin_family, AF_INET))
    )
        return GUSISetPosixError(EAFNOSUPPORT);
    if (non_null && (!name->sin_addr.s_addr || !name->sin_port))
        return GUSISetPosixError(EADDRNOTAVAIL);

<Member functions for class GUSIOTInetStrategy 1041>+≡ (1023) <1041
    int GUSIOTInetStrategy::UnpackAddress(const TNetbuf & addr, void * address, socklen_t * len)
    {
        sockaddr_in *name = (sockaddr_in *)address;
        <Sanity checks for GUSIOTInetStrategy::UnpackAddress 1044>
        const InetAddress * otaddr = reinterpret_cast<InetAddress *>(addr.buf);
        name->sin_family = AF_INET;
        name->sin_port = otaddr->fPort;
        name->sin_addr.s_addr = otaddr->fHost;
        *len = sizeof(struct sockaddr_in);

        return 0;
    }

<Sanity checks for GUSIOTInetStrategy::UnpackAddress 1044>≡ (1043)
    if (!GUSI_ASSERT_CLIENT(
        *len >= sizeof(struct sockaddr_in),
        ("UnpackAddress: address len %d < %d\n", len, sizeof(struct sockaddr_in)))
    )
        return GUSISetPosixError(EINVAL);

```

27.13 Implementation of GUSIOTMInetOptions

```
(Member functions for class GUSIOTMInetOptions 1045)≡ (1023) 1046»
bool GUSIOTMInetOptions::DoGetSockOpt(
    int * result, EndpointRef endpoint, int level, int optname,
    void *optval, socklen_t * optlen)
{
    TOptMgmt    optReq;
    UInt8       optBuffer[ kTOptionHeaderSize + 50 ];
    TOption*    opt      = (TOption*)optBuffer;
    int         len;

    optReq.flags = T_CURRENT;
    optReq.opt.buf = (UInt8*) optBuffer;

    opt->level = INET_IP;
    opt->name = optname;

    switch (level) {
    case SOL_SOCKET:
        switch (optname) {
        case SO_REUSEPORT:
            opt->name = SO_REUSEADDR;
            // Fall through
        case SO_REUSEADDR:
        case SO_DONTROUTE:
            len = 4;
            break;
        default:
            goto notSupported;
        }
        break;
    case IPPROTO_IP:
        switch (optname) {
        case IP_OPTIONS:
            len = *optlen;
            break;
        case IP_TOS:
        case IP_TTL:
            len = 1;
            break;
        default:
            goto notSupported;
        }
        break;
    default:
        goto notSupported;
    }
    optReq.opt.len = opt->len = kTOptionHeaderSize+len;
    if (*result = GUSISetMacError(OTOptionManagement(endpoint, &optReq, &optReq)))
        return true;
    switch (optname) {
    case IP_TOS:
    case IP_TTL:
```

```

        *reinterpret_cast<int *>(optval) = *reinterpret_cast<char *>(opt->value);
        *optlen = 4;
        break;
    case IP_OPTIONS:
        len = optReq.opt.len;
        // Fall through
    default:
        memcpy(optval, opt->value, len);
        *optlen = len;
        break;
    }

    return true;
notSupported:
    return false;
}

```

```

<Member functions for class GUSIOTMInetOptions 1045>+≡ (1023) <1045 1047>
bool GUSIOTMInetOptions::DoSetSockOpt(
    int * result, EndpointRef endpoint, int level, int optname,
    void *optval, socklen_t optlen)
{
    TOptMgmt          optReq;
    UInt8             optBuffer[ kTOptionHeaderSize + sizeof(struct linger) ];
    TOption*          opt      = (TOption*)optBuffer;
    t_kpalive          kpal     = {1, 120};
    int                len;
    char               val;

    optReq.flags = T_NEGOTIATE;
    optReq.opt.buf = (UInt8*) optBuffer;

    opt->level = INET_IP;
    opt->name = optname;

    switch (level) {
    case SOL_SOCKET:
        switch (optname) {
        case SO_REUSEPORT:
            opt->name = SO_REUSEADDR;
            // Fall through
        case SO_REUSEADDR:
        case SO_DONTROUTE:
            len = 4;
            break;
        default:
            goto notSupported;
        }
        break;
    case IPPROTO_IP:
        switch (optname) {
        case IP_OPTIONS:
            len = optlen;
            break;
        case IP_TOS:
        case IP_TTL:
            val = *reinterpret_cast<int *>(optval);
            optval = &val;
            len = 1;
            break;
        default:
            goto notSupported;
        }
        break;
    default:
        goto notSupported;
    }
    optReq.opt.len = opt->len = kTOptionHeaderSize+len;
    memcpy(opt->value, optval, len);

    *result = GUSISetMacError(OTOptionManagement(endpoint, &optReq, &optReq));
}

```

```

    return true;
notSupported:
    return false;
}

```

Internet sockets have to support a fairly complex set of `ioctl` options to obtain information about hardware interfaces. We don't support changing any of the information.

`SetInterface` fills in an `ifreq` structure. The name is synthesized, encoding the interface index and secondary IP address for fast lookup.

```

<Member functions for class GUSIOTMInetOptions 1045>+≡ (1023) <1046 1048>
static void SetAddress(sockaddr_in * sa, InetHost addr)
{
    sa->sin_family      = AF_INET;
    sa->sin_port         = 0;
    sa->sin_addr.s_addr = addr;
}

static void SetInterface(ifreq * ifr, int ifNum, int aliasNum, InetHost addr)
{
    sprintf(ifr->ifr_name, (aliasNum ? "ot%d:%d" : "ot%d"), ifNum, aliasNum);
    SetAddress(reinterpret_cast<sockaddr_in *>(&ifr->ifr_addr), addr);
}

```

GetInterfaceList gets a list of all IP addresses.

(Member functions for class GUSIOTMInetOptions 1045)+≡

(1023) <1047 1049>

```
static int GetInterfaceList(ifconf * conf)
{
    InetInterfaceInfo  info;
    int                maxInterfaces  = conf->ifc_len / sizeof(ifreq);
    int                numInterfaces  = 0;
    SInt32             interface     = 0;
    ifreq *            ifr = conf->ifc_req;

    if (!maxInterfaces)
        return GUSISetPosixError(EINVAL);

    while (!OTInetGetInterfaceInfo(&info, numInterfaces)) {
        SetInterface(ifr++, numInterfaces, 0, info.fAddress);
        if (++interface == maxInterfaces)
            goto bufferFull;
        if (info.fIPSecondaryCount) {
            InetHost * secondaries = new InetHost[info.fIPSecondaryCount];
            OTInetGetSecondaryAddresses(secondaries, &info.fIPSecondaryCount, numInterfaces);
            for (int i = 0; i < info.fIPSecondaryCount; ++i) {
                SetInterface(ifr++, numInterfaces, i+1, secondaries[i]);
                if (++interface == maxInterfaces)
                    goto bufferFull;
            }
            delete secondaries;
        }
        ++numInterfaces;
    }
bufferFull:
    conf->ifc_len = interface * sizeof(ifreq);

    return 0;
}
```

GetInterfaceParam gets a parameter for an interface. Secondary IP addresses are only retrieved for SIOCGIFADDR.

```
<Member functions for class GUSIOTMInetOptions 1045>+≡ (1023) <1048 1050>
static int GetInterfaceParam(ifreq * ifr, unsigned int request)
{
    int ifnum = atoi(ifr->ifr_name+2);
    int ifalias = 0;
    char * alias = strchr(ifr->ifr_name, ':');
    if (alias)
        ifalias = atoi(alias+1);

    InetInterfaceInfo info;
    if (OTInetGetInterfaceInfo(&info, ifnum) || ifalias > info.fIPSecondaryCount)
        return GUSISetPosixError(ENOENT);
    if (ifalias && request == SIOCGIFADDR) {
        InetHost * secondaries = new InetHost[info.fIPSecondaryCount];
        OTInetGetSecondaryAddresses(secondaries, &info.fIPSecondaryCount, ifnum);
        info.fAddress = secondaries[ifalias-1];
        delete secondaries;
    }
    switch (request) {
    case SIOCGIFADDR:
        SetAddress(reinterpret_cast<sockaddr_in *>(&ifr->ifr_addr), info.fAddress);
        break;
    case SIOCGIFFLAGS:
        ifr->ifr_flags = IFF_UP | IFF_BROADCAST | IFF_MULTICAST;
        break;
    case SIOCGIFBRDADDR:
        SetAddress(reinterpret_cast<sockaddr_in *>(&ifr->ifr_addr), info.fBroadcastAddr);
        break;
    case SIOCGIFNETMASK:
        SetAddress(reinterpret_cast<sockaddr_in *>(&ifr->ifr_addr), info.fNetmask);
        break;
    }
    return 0;
}
```

```

<Member functions for class GUSIOTMInetOptions 1045>+≡ (1023) <1049
bool GUSIOTMInetOptions::DoIoctl(int * result, unsigned int request, va_list arg)
{
    switch (request) {
    case SIOCGIFCONF:
        *result = GetInterfaceList(va_arg(arg, ifconf *));

        return true;
    case SIOCGIFADDR:
    case SIOCGIFFLAGS:
    case SIOCGIFBRDADDR:
    case SIOCGIFNETMASK:
        *result = GetInterfaceParam(va_arg(arg, ifreq *), request);

        return true;
    }

    return false;
}

```

27.14 Implementation of GUSIOTTcpStrategy

GUSIOTTcpStrategy merely needs to define the creation options.

```

<Member functions for class GUSIOTTcpStrategy 1051>≡ (1023)
const char * GUSIOTTcpStrategy::ConfigPath()
{
    return kTCPName;
}

```

27.15 Implementation of GUSIOTTcpSocket

getsockopt and setsockopt for GUSIOTTcpSocket add the options for multicast.

```

<Overridden member functions for GUSIOTTcpSocket 1052>≡ (1032) 1054>
virtual int getsockopt(int level, int optname, void *optval, socklen_t * optlen);

```



```

(1023) 1055⤵
<Member functions for class GUSIOTTcpSocket 1053>≡
int GUSIOTTcpSocket::getsockopt(int level, int optname, void *optval, socklen_t * optlen)
{
    int result = GUSIOTSocket::getsockopt(level, optname, optval, optlen);

    if (!result || errno != EOPNOTSUPP
        || GUSIOTMInetOptions::DoGetSockOpt(&result, fEndpoint, level, optname, optval, optlen)
        )
        return result;

    TOptMgmt    optReq;
    UInt8       optBuffer[ kOTOOptionHeaderSize + sizeof(long) ];
    TOption*    opt      = (TOption*)optBuffer;
    int         len;

    optReq.flags = T_CURRENT;
    optReq.opt.buf = (UInt8*) optBuffer;

    opt->level    = INET_TCP;
    opt->name     = optname;

    switch (level) {
    case IPPROTO_TCP:
        switch (optname) {
            case TCP_MAXSEG:
            case TCP_NODELAY:
                len = 4;
                break;
            default:
                goto notSupported;
        }
        break;
    default:
        goto notSupported;
    }

    optReq.opt.len = opt->len = kOTOOptionHeaderSize+len;
    if (GUSISetMacError(OTOOptionManagement(fEndpoint, &optReq, &optReq)))
        return -1;
    memcpy(optval, opt->value, len);
    *optlen = len;

    return 0;
notSupported:
    GUSI_ASSERT_CLIENT(false, ("getsockopt: illegal request %d\n", optname));

    return GUSISetPosixError(EOPNOTSUPP);
}

(1032) 1052 1056⤵
<Overridden member functions for GUSIOTTcpSocket 1052>+≡
virtual int setsockopt(int level, int optname, void *optval, socklen_t optlen);

```

```

<Member functions for class GUSIOTTcpSocket 1053>+≡ (1023) <1053 1057>
int GUSIOTTcpSocket::setsockopt(int level, int optname, void *optval, socklen_t optlen)
{
    int result = GUSIOTSocket::setsockopt(level, optname, optval, optlen);

    if (!result || errno != EOPNOTSUPP
        || GUSIOTMInetOptions::DoSetSockOpt(&result, fEndpoint, level, optname, optval, optlen)
        )
        return result;

    TOptMgmt          optReq;
    UInt8             optBuffer[ kTOptionHeaderSize + sizeof(TIPAddMulticast) ];
    TOption*          opt      = (TOption*)optBuffer;
    int               len;

    optReq.flags = T_NEGOTIATE;
    optReq.opt.buf = (UInt8*) optBuffer;

    opt->level  = INET_TCP;
    opt->name    = optname;

    switch (level) {
    case IPPROTO_TCP:
        switch (optname) {
            case TCP_MAXSEG:
            case TCP_NODELAY:
                len = 4;
                break;
            default:
                goto notSupported;
        }
        break;
    default:
        goto notSupported;
    }
    optReq.opt.len = opt->len = kTOptionHeaderSize+len;
    memcpy(opt->value, optval, len);

    return GUSISetMacError(OTOptionManagement(fEndpoint, &optReq, &optReq));
notSupported:
    GUSI_ASSERT_CLIENT(false, ("setsockopt: illegal request %d\n", optname));

    return GUSISetPosixError(EOPNOTSUPP);
}

```

ioctl() deals with interface access.

```

<Overridden member functions for GUSIOTTcpSocket 1052>+≡ (1032) <1054
virtual int ioctl(unsigned int request, va_list arg);

```

```

<Member functions for class GUSIOTTcpSocket 1053>+≡ (1023) <1055
int GUSIOTTcpSocket::ioctl(unsigned int request, va_list arg)
{
    int result;

    if (GUSIOTMInetOptions::DoIoctl(&result, request, arg))
        return result;
    else
        return GUSIOTSocket::ioctl(request, arg);
}

```

27.16 Implementation of GUSIOTUdpStrategy

GUSIOTUdpStrategy merely needs to define the creation options.

```

<Member functions for class GUSIOTUdpStrategy 1058>≡ (1023)
const char * GUSIOTUdpStrategy::ConfigPath()
{
    return kUDPName;
}

```

27.17 Implementation of GUSIOTUdpSocket

getsockopt and setsockopt for GUSIOTUdpSocket add the options for multicast.

```

<Overridden member functions for GUSIOTUdpSocket 1059>≡ (1031) 1061▷
virtual int getsockopt(int level, int optname, void *optval, socklen_t * optlen);

```

```

(Member functions for class GUSIOTUdpSocket 1060)≡ (1023) 1062»
int GUSIOTUdpSocket::getsockopt(int level, int optname, void *optval, socklen_t * optlen)
{
    int result = GUSIOTSocket::getsockopt(level, optname, optval, optlen);

    if (!result || errno != EOPNOTSUPP
        || GUSIOTMInetOptions::DoGetSockOpt(&result, fEndpoint, level, optname, optval, optlen)
        )
        return result;

    TOptMgmt    optReq;
    UInt8       optBuffer[ kTOptionHeaderSize + sizeof(long) ];
    TOption*    opt      = (TOption*)optBuffer;
    int         len;

    optReq.flags = T_CURRENT;
    optReq.opt.buf = (UInt8*) optBuffer;

    opt->name     = optname;

    switch (level) {
    case SOL_SOCKET:
        opt->level = INET_IP;
        switch (optname) {
        case SO_BROADCAST:
            len = 4;
            break;
        default:
            goto notSupported;
        }
        break;
    case IPPROTO_IP:
        opt->level = INET_IP;
        switch (optname) {
        case IP_HDRINCL:
        case IP_RCVDSTADDR:
        case IP_MULTICAST_IF:
            len = 4;
            break;
        case IP_MULTICAST_TTL:
        case IP_MULTICAST_LOOP:
            len = 1;
            break;
        default:
            goto notSupported;
        }
        break;
    default:
        goto notSupported;
    }
    optReq.opt.len = opt->len = kTOptionHeaderSize+len;
    if (GUSISetMacError(OTOptionManagement(fEndpoint, &optReq, &optReq)))
        return -1;
    switch (optname) {

```

```

    case IP_MULTICAST_TTL:
    case IP_MULTICAST_LOOP:
        *reinterpret_cast<int *>(optval) = *reinterpret_cast<char *>(opt->value);
        *optlen = 4;
        break;
    case IP_HDRINCL:
    case IP_RCVDSTADDR:
    case SO_BROADCAST:
    case IP_MULTICAST_IF:
        memcpy(optval, opt->value, len);
        *optlen = len;
        break;
    }

    return 0;
notSupported:
    GUSI_ASSERT_CLIENT(false, ("getsockopt: illegal request %d\n", optname));

    return GUSISetPosixError(EOPNOTSUPP);
}

```

{Overridden member functions for GUSIOTUdpSocket 1059}+≡ (1031) ◀1059 1063▶
 virtual int setsockopt(int level, int optname, void *optval, socklen_t optlen);

```

<Member functions for class GUSIOTUdpSocket 1060>+≡ (1023) <1060 1064>
int GUSIOTUdpSocket::setsockopt(int level, int optname, void *optval, socklen_t optlen)
{
    int result = GUSIOTSocket::setsockopt(level, optname, optval, optlen);

    if (!result || errno != EOPNOTSUPP
        || GUSIOTMInetOptions::DoSetSockOpt(&result, fEndpoint, level, optname, optval, optlen)
        )
        return result;

    TOptMgmt          optReq;
    UInt8             optBuffer[ kTOptionHeaderSize + sizeof(TIPAddMulticast) ];
    TOption*          opt      = (TOption*)optBuffer;
    char              val;

    optReq.flags = T_NEGOTIATE;
    optReq.opt.buf = (UInt8*) optBuffer;

    opt->level  = INET_IP;
    opt->name    = optname;

    int len;
    switch (level) {
    case SOL_SOCKET:
        switch (optname) {
        case SO_BROADCAST:
            len = 4;
            break;
        default:
            goto notSupported;
        }
        break;
    case IPPROTO_IP:
        switch (optname) {
        case IP_HDRINCL:
        case IP_RCVDSTADDR:
        case IP_MULTICAST_IF:
            len = 4;
            break;
        case IP_MULTICAST_TTL:
        case IP_MULTICAST_LOOP:
            val = *reinterpret_cast<long *>(optval) != 0;
            optval = &val;
            len = 1;
            break;
        case IP_ADD_MEMBERSHIP:
        case IP_DROP_MEMBERSHIP:
            len = 8;
            break;
        default:
            goto notSupported;
        }
        break;
    }
}

```

```

    optReq.opt.len = opt->len = kOTOptionHeaderSize+len;
    memcpy(opt->value, optval, len);

    return GUSISetMacError(OTOptionManagement(fEndpoint, &optReq, &optReq));
notSupported:
    GUSI_ASSERT_CLIENT(false, ("setsockopt: illegal request %d\n", optname));

    return GUSISetPosixError(EOPNOTSUPP);
}

ioctl() deals with interface access.
<Overridden member functions for GUSIOTUdpSocket 1059>+≡ (1031) <1061
virtual int ioctl(unsigned int request, va_list arg);

<Member functions for class GUSIOTUdpSocket 1060>+≡ (1023) <1062
int GUSIOTUdpSocket::ioctl(unsigned int request, va_list arg)
{
    int result;

    if (GUSIOTMInetOptions::DoIoctl(&result, request, arg))
        return result;
    else
        return GUSIOTSocket::ioctl(request, arg);
}

```

27.18 Implementation of Open Transport Internet hooks

```

<Implementation of GUSIwithOTInetSockets 1065>≡ (1023)
void GUSIwithOTTcpSockets()
{
    gGUSIInetFactories.AddFactory(SOCK_STREAM, 0, GUSIOTTcpFactory::Instance());
    GUSIOTNetDB::Instantiate();
}

void GUSIwithOTUdpSockets()
{
    gGUSIInetFactories.AddFactory(SOCK_DGRAM, 0, GUSIOTUdpFactory::Instance());
    GUSIOTNetDB::Instantiate();
}

void GUSIwithOTInetSockets()
{
    GUSIwithOTTcpSockets();
    GUSIwithOTUdpSockets();
}

```


Chapter 28

IP Name Lookup in Open Transport

```
<GUSIOTNetDB.h 1066>≡
#ifdef _GUSIOTNetDB_
#define _GUSIOTNetDB_

#ifdef GUSI_INTERNAL
#include "GUSINetDB.h"
#include "GUSIContext.h"
#include "GUSIOpenTransport.h"

<Name dropping for file GUSIOTNetDB 1068>

<Definition of class GUSIOTNetDB 1069>

#endif /* GUSI_INTERNAL */

#endif /* _GUSIOTNetDB_ */
```

```

<GUSIOTNetDB.cp 1067>≡
#include "GUSIInternal.h"
#include "GUSIOTNetDB.h"
#include "GUSIOTInet.h"

#include <sys/socket.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#undef AF_INET
#undef IP_OPTIONS
#undef IP_TOS
#undef IP_TTL
#undef IP_HDRINCL
#undef IP_MULTICAST_IF
#undef IP_MULTICAST_TTL
#undef IP_MULTICAST_LOOP
#undef IP_ADD_MEMBERSHIP
#undef IP_DROP_MEMBERSHIP

#include <OpenTptInternet.h>

<Asynchronous notifier function for GUSIOTNetDB 1074>
<Member functions for class GUSIOTNetDB 1071>

```

28.1 Definition of GUSIOTNetDB

We don't want to open the Open Transport headers files in our public header, but we need InetSvcRef.

```

<Name dropping for file GUSIOTNetDB 1068>≡ (1066)
class TInternetServices;
typedef TInternetServices* InetSvcRef;

<Definition of class GUSIOTNetDB 1069>≡ (1066)
class GUSIOTNetDB : public GUSINetDB {
public:
    static void Instantiate();
    bool Resolver();

    <Overridden member functions for GUSIOTNetDB 1078>
private:
    GUSISpecificData<GUSIhostent, GUSIKillHostEnt> fHost;
    <Privatissima of GUSIOTNetDB 1070>
};

```

28.2 Implementation of GUSIOTNetDB

```

<Privatissima of GUSIOTNetDB 1070>≡ (1069) 1072»
GUSIOTNetDB();

```

```

⟨Member functions for class GUSIOTNetDB 1071⟩≡ (1067) 1075▸
GUSIOTNetDB::GUSIOTNetDB()
{
    ⟨Initialize fields of GUSIOTNetDB 1073⟩
}

```

The GUSIOTNetDB notifier operates similarly to the GUSIOTSocket notifier, but it has to get the context to wake up somehow from its parameters.

```

⟨Privatissima of GUSIOTNetDB 1070⟩+≡ (1069) ◀1070
uint16_t      fEvent;
uint32_t      fCompletion;
OSStatus      fAsyncError;
InetSvcRef    fSvc;
GUSIContext * fCreationContext;
friend pascal void GUSIOTNetDBNotify(GUSIOTNetDB *, OTEventCode, OTResult, void *);

⟨Initialize fields of GUSIOTNetDB 1073⟩≡ (1071)
fEvent          = 0;
fCompletion      = 0;
fAsyncError      = 0;
fSvc             = 0;
fCreationContext = 0;

```

<Asynchronous notifier function for GUSIOTNetDB 1074>≡ (1067)

```

inline uint32_t CompleteMask(OTEventCode code)
{
    return 1 << (code & 0x1F);
}

pascal void GUSIOTNetDBNotify(
    GUSIOTNetDB * netdb, OTEventCode code, OTResult result, void *cookie)
{
    GUSIContext * context = netdb->fCreationContext;

    switch (code & 0x7F000000L) {
    case 0:
        netdb->fEvent |= code;
        break;
    case kCOMPLETEEVENT:
    case kPRIVATEEVENT:
        if (!(code & 0x00FFFFE0))
            netdb->fCompletion |= CompleteMask(code);
        switch (code) {
        case T_OPENCOMPLETE:
            netdb->fSvc = static_cast<InetSvcRef>(cookie);
            break;
        case T_DNRSTRINGTOADDRCOMPLETE:
        case T_DNRADDRTONAMECOMPLETE:
            context = static_cast<GUSIContext **>(cookie)[-1];
            break;
        }
        break;
    }
    if (result)
        netdb->fAsyncError = result;
    context->Wakeup();
}

```

The Open Transport DNR code is installed by calling `Instantiate`, which initializes the singleton instance of `GUSINetDB`.

<Member functions for class GUSIOTNetDB 1071>+≡ (1067) <1071 1076>

```

void GUSIOTNetDB::Instantiate()
{
    if (!sInstance)
        sInstance = new GUSIOTNetDB;
}

```

```

⟨Member functions for class GUSIOTNetDB 1071⟩+≡ (1067) ◀1075 1077▶
bool GUSIOTNetDB::Resolver()
{
    if (!fCreationContext) {
        fCreationContext = GUSIContext::Current();
        fAsyncError = OTAsyncOpenInternetServices(
            kDefaultInternetServicesPath,
            0,
            reinterpret_cast<OTNotifyProcPtr>(GUSIOTNetDBNotify),
            this);
        while (!fAsyncError && !(fCompletion & CompleteMask(T_OPENCOMPLETE)))
            GUSIContext::Yield(true);
    }
    return fSvc != 0;
}

```

Naturally, Open Transport keeps its host data in different structures than the socket standard.

```

⟨Member functions for class GUSIOTNetDB 1071⟩+≡ (1067) ◀1076 1079▶
static void CopyHost(InetHostInfo & otHost, GUSIHostent & unixHost)
{
    size_t len = strlen(otHost.name);
    len = (len+4) & ~3;

    unixHost.Alloc(len+kMaxHostAddrs*4);
    strcpy(unixHost.h_name, otHost.name);
    unixHost.h_aliases[0] = NULL; // Aliases not supported
    unixHost.h_addrtype = AF_INET;
    unixHost.h_length = 4;

    int addrs = 0;
    for (int i=0; i<kMaxHostAddrs && otHost.addrs[addrs]!=0; ++i, ++addrs) {
        unixHost.h_addr_list[addrs] = unixHost.fName + len;
        len += 4;
        memcpy(unixHost.h_addr_list[addrs], &otHost.addrs[addrs], 4);
    }
    unixHost.h_addr_list[addrs] = NULL;
}

```

```

⟨Overridden member functions for GUSIOTNetDB 1078⟩≡ (1069) 1082▶
virtual hostent * gethostbyname(const char * name);

```

```

<Member functions for class GUSIOTNetDB 1071>+≡ (1067) <1077 1083>
    hostent * GUSIOTNetDB::gethostbyname(const char * name)
    {
        <Open Open Transport DNR or fail lookup 1080>
        if (!strcmp(name, "localhost")) {
            long ipaddr = gethostid();
            if (ipaddr)
                return gethostbyaddr((char *) &ipaddr, sizeof(in_addr), AF_INET);
            return GUSISetHostError(HOST_NOT_FOUND), static_cast<hostent *>(nil);
        }
        <Declare and initialize otHost and unixHost 1081>
        fCompletion &= ~CompleteMask(T_DNRSTRINGTOADDRCOMPLETE);
        fAsyncError = OTInetStringToAddress(fSvc, const_cast<char *>(name), &otHost.fInfo);
        while (!fAsyncError && !(fCompletion & CompleteMask(T_DNRSTRINGTOADDRCOMPLETE)))
            GUSIContext::Yield(true);

        if (fAsyncError)
            return GUSISetHostError(NO_RECOVERY), static_cast<hostent *>(nil);

        CopyHost(otHost.fInfo, unixHost);

        return &unixHost;
    }

<Open Open Transport DNR or fail lookup 1080>≡ (1079 1083)
    if (!Resolver())
        return GUSISetHostError(NO_RECOVERY), static_cast<hostent *>(nil);

<Declare and initialize otHost and unixHost 1081>≡ (1079 1083)
    struct {
        GUSIContext * fContext;
        InetHostInfo fInfo;
    } otHost;
    GUSIhostent & unixHost = *fHost;
    otHost.fContext = GUSIContext::Current();

<Overridden member functions for GUSIOTNetDB 1078>+≡ (1069) <1078 1084>
    virtual hostent * gethostbyaddr(const void * addr, size_t len, int type);

```

```

<Member functions for class GUSIOTNetDB 1071>+≡ (1067) <1079 1085>
    hostent * GUSIOTNetDB::gethostbyaddr(const void * addrP, size_t len, int)
    {
        <Open Open Transport DNR or fail lookup 1080>
        <Declare and initialize otHost and unixHost 1081>

        InetHost addr = *(InetHost *)addrP;
        if (addr == 0x7F000001)
            addr = static_cast<InetHost>(gethostid());

        fCompletion &= ~CompleteMask(T_DNRADDRTONAMECOMPLETE);
        fAsyncError = OTInetAddressToName(fSvc, addr, otHost.fInfo.name);
        while (!fAsyncError && !(fCompletion & CompleteMask(T_DNRADDRTONAMECOMPLETE)))
            GUSIContext::Yield(true);
        if (fAsyncError)
            return GUSISetHostError(NO_RECOVERY), static_cast<hostent *>(nil);

        memset(otHost.fInfo.addrs, 0, kMaxHostAddrs*4);
        otHost.fInfo.addrs[0] = addr;
        CopyHost(otHost.fInfo, unixHost);

        return &unixHost;
    }

<Overridden member functions for GUSIOTNetDB 1078>+≡ (1069) <1082 1086>
    virtual char * inet_ntoa(in_addr inaddr);

<Member functions for class GUSIOTNetDB 1071>+≡ (1067) <1083 1087>
    char * GUSIOTNetDB::inet_ntoa(in_addr inaddr)
    {
        GUSIhostent & unixHost = *fHost;
        sprintf(unixHost.fAddrString, "%d.%d.%d.%d",
            (inaddr.s_addr >> 24) & 0xFF,
            (inaddr.s_addr >> 16) & 0xFF,
            (inaddr.s_addr >> 8) & 0xFF,
            inaddr.s_addr & 0xFF);
        return unixHost.fAddrString;
    }

<Overridden member functions for GUSIOTNetDB 1078>+≡ (1069) <1084
    virtual long gethostid();

<Member functions for class GUSIOTNetDB 1071>+≡ (1067) <1085
    long GUSIOTNetDB::gethostid()
    {
        if (!Resolver())
            return GUSISetHostError(NO_RECOVERY), 0;
        InetInterfaceInfo info;
        OTInetGetInterfaceInfo(&info, kDefaultInetInterface);

        return static_cast<long>(info.fAddress);
    }

```


Chapter 29

Pseudo-synchronous File System Calls

MacOS offers a wide variety of file system APIs, but the most convenient of them—the `FSpXXX` calls only works synchronously. The routines defined here offer a version of these calls, executed asynchronously and embedded into the `GUSIContext` switching model.

```
(GUSIFSWrappers.h 1088)≡
#ifdef _GUSIFSWrappers_
#define _GUSIFSWrappers_

#ifdef GUSI_SOURCE

#include <Files.h>

#include <sys/cdefs.h>

#include "GUSIFFileSpec.h"

__BEGIN_DECLS
<Declarations of C GUSIFSWrappers 1097>
__END_DECLS
#ifdef __cplusplus
<Declarations of C++ GUSIFSWrappers 1090>
#endif

#endif /* GUSI_SOURCE */

#endif /* GUSIFSWrappers */
```

```

<GUSIFSWrappers.cp 1089>≡
#include "GUSIInternal.h"
#include "GUSIFSWrappers.h"
#include "GUSIContext.h"

#include <PLStringFuncs.h>
#include <Devices.h>
#include <Script.h>

<Implementation of GUSIFSWrappers 1091>

<Declarations of C++ GUSIFSWrappers 1090>≡ (1088) 1092»
    OSErr GUSIFSGetCatInfo(GUSIIOPBWrapper<GUSICatInfo> * info);
    OSErr GUSIFSSetCatInfo(GUSIIOPBWrapper<GUSICatInfo> * info);

<Implementation of GUSIFSWrappers 1091>≡ (1089) 1093»
    OSErr GUSIFSGetCatInfo(GUSIIOPBWrapper<GUSICatInfo> * info)
    {
        info->StartIO();
        PBGetCatInfoAsync(&info->fPB.Info());
        return info->FinishIO();
    }

    OSErr GUSIFSSetCatInfo(GUSIIOPBWrapper<GUSICatInfo> * info)
    {
        info->StartIO();
        PBSetCatInfoAsync(&info->fPB.Info());
        return info->FinishIO();
    }

<Declarations of C++ GUSIFSWrappers 1090>+≡ (1088) <1090 1094»
    OSErr GUSIFSGetFCBInfo(GUSIIOPBWrapper<FCBPBRec> * fcb);

<Implementation of GUSIFSWrappers 1091>+≡ (1089) <1091 1095»
    OSErr GUSIFSGetFCBInfo(GUSIIOPBWrapper<FCBPBRec> * fcb)
    {
        fcb->StartIO();
        PBGetFCBInfoAsync(&fcb->fPB);
        return fcb->FinishIO();
    }

<Declarations of C++ GUSIFSWrappers 1090>+≡ (1088) <1092 1096»
    OSErr GUSIFSGetVInfo(GUSIIOPBWrapper<ParamBlockRec> * pb);
    OSErr GUSIFSHGetVInfo(GUSIIOPBWrapper<HParamBlockRec> * pb);

```

```

<Implementation of GUSIFSWrappers 1091>+≡ (1089) ◁1093 1098▷
    OSErr GUSIFSGetVInfo(GUSIIOPBWrapper<ParamBlockRec> * pb)
    {
        pb->StartIO();
        PBGetVInfoAsync(&pb->fPB);
        return pb->FinishIO();
    }
    OSErr GUSIFSHGetVInfo(GUSIIOPBWrapper<HParamBlockRec> * pb)
    {
        pb->StartIO();
        PBHGetVInfoAsync(&pb->fPB);
        return pb->FinishIO();
    }

<Declarations of C++ GUSIFSWrappers 1090>+≡ (1088) ◁1094 1099▷
    OSErr GUSIFSOpen(GUSIIOPBWrapper<ParamBlockRec> * pb);

<Declarations of C GUSIFSWrappers 1097>≡ (1088) 1100▷
    OSErr GUSIFSOpenDriver(StringPtr name, short * refNum);

<Implementation of GUSIFSWrappers 1091>+≡ (1089) ◁1095 1101▷
    OSErr GUSIFSOpen(GUSIIOPBWrapper<ParamBlockRec> * pb)
    {
        pb->StartIO();
        PBOpenAsync(&pb->fPB);
        return pb->FinishIO();
    }
    OSErr GUSIFSOpenDriver(StringPtr name, short * refNum)
    {
        GUSIIOPBWrapper<ParamBlockRec> pb;

        pb->ioParam.ioNamePtr = name;
        pb->ioParam.ioPermssn = fsCurPerm;

        OSErr err = GUSIFSOpen(&pb);
        *refNum = pb->ioParam.ioRefNum;

        return err;
    }

<Declarations of C++ GUSIFSWrappers 1090>+≡ (1088) ◁1096 1104▷
    OSErr GUSIFSHGetFInfo(GUSIIOPBWrapper<HParamBlockRec> * pb);
    OSErr GUSIFSHSetFInfo(GUSIIOPBWrapper<HParamBlockRec> * pb);

<Declarations of C GUSIFSWrappers 1097>+≡ (1088) ◁1097 1102▷
    OSErr GUSIFSGetFInfo(const FSSpec * spec, FInfo * info);
    OSErr GUSIFSSetFInfo(const FSSpec * spec, const FInfo * info);

```

```

(Implementation of GUSIFSWrappers 1091)+≡ (1089) <1098 1103>
OSErr GUSIFSHGetFInfo(GUSIIOPBWrapper<HParamBlockRec> * pb)
{
    pb->StartIO();
    PBHGetFInfoAsync(&pb->fPB);
    return pb->FinishIO();
}
OSErr GUSIFSHSetFInfo(GUSIIOPBWrapper<HParamBlockRec> * pb)
{
    pb->StartIO();
    PBHSetFInfoAsync(&pb->fPB);
    return pb->FinishIO();
}
OSErr GUSIFSGetFInfo(const FSSpec * spec, FInfo * info)
{
    GUSIIOPBWrapper<HParamBlockRec> pb;

    pb->fileParam.ioVRefNum = spec->vRefNum;
    pb->fileParam.ioDirID = spec->parID;
    pb->fileParam.ioNamePtr = const_cast<StringPtr>(spec->name);
    pb->fileParam.ioFDirIndex = 0;

    OSErr err = GUSIFSHGetFInfo(&pb);
    if (!err)
        *info = pb->fileParam.ioFlFndrInfo;

    return err;
}
OSErr GUSIFSSetFInfo(const FSSpec * spec, const FInfo * info)
{
    GUSIIOPBWrapper<HParamBlockRec> pb;

    pb->fileParam.ioVRefNum = spec->vRefNum;
    pb->fileParam.ioDirID = spec->parID;
    pb->fileParam.ioNamePtr = const_cast<StringPtr>(spec->name);
    pb->fileParam.ioFDirIndex = 0;

    OSErr err = GUSIFSHGetFInfo(&pb);
    if (!err) {
        pb->fileParam.ioDirID = spec->parID; /* Gets overwritten by PBHGetInfo */
        pb->fileParam.ioFlFndrInfo = *info;
        err = GUSIFSHSetFInfo(&pb);
    }

    return err;
}

(Declarations of C GUSIFSWrappers 1097)+≡ (1088) <1100 1105>
OSErr GUSIFSOpenDF(const FSSpec * spec, char permission, short * refNum);
OSErr GUSIFSOpenRF(const FSSpec * spec, char permission, short * refNum);

```

<Implementation of GUSIFSWrappers 1091>+≡ (1089) <1101 1106>

```

OSError GUSIFSOpenDF(const FSSpec * spec, char permission, short * refNum)
{
    GUSIIOPBWrapper<HParamBlockRec>    pb;

    pb->fileParam.ioVRefNum =    spec->vRefNum;
    pb->fileParam.ioDirID   =    spec->parID;
    pb->fileParam.ioNamePtr =    const_cast<StringPtr>(spec->name);
    pb->ioParam.ioPermssn   =    permission;

    pb.StartIO();
    PBHOpenDFAsync(&pb.fPB);
    OSError err = pb.FinishIO();
    if (!err)
        *refNum = pb->ioParam.ioRefNum;

    return err;
}

```

OSError GUSIFSOpenRF(const FSSpec * spec, char permission, short * refNum)

```

{
    GUSIIOPBWrapper<HParamBlockRec>    pb;

    pb->fileParam.ioVRefNum =    spec->vRefNum;
    pb->fileParam.ioDirID   =    spec->parID;
    pb->fileParam.ioNamePtr =    const_cast<StringPtr>(spec->name);
    pb->ioParam.ioPermssn   =    permission;

    pb.StartIO();
    PBHOpenRFAsync(&pb.fPB);
    OSError err = pb.FinishIO();
    if (!err)
        *refNum = pb->ioParam.ioRefNum;

    return err;
}

```

<Declarations of C++ GUSIFSWrappers 1090>+≡ (1088) <1099 1108>

```

OSError GUSIFSHGetVolParms(GUSIIOPBWrapper<HParamBlockRec> * pb);

```

<Declarations of C GUSIFSWrappers 1097>+≡ (1088) <1102 1107>

```

OSError GUSIFSGetVolParms(short vRefNum, GetVolParmsInfoBuffer * volParms);

```

```

<Implementation of GUSIFSWrappers 1091>+≡ (1089) <1103 1109>
    OSErr GUSIFSHGetVolParms(GUSIIOPBWrapper<HParamBlockRec> * pb)
    {
        pb->StartIO();
        PBHGetVolParmsAsync(&pb->fPB);
        return pb->FinishIO();
    }
    OSErr GUSIFSGetVolParms(short vRefNum, GetVolParmsInfoBuffer * volParms)
    {
        GUSIIOPBWrapper<HParamBlockRec> pb;

        pb->ioParam.ioNamePtr = nil;
        pb->ioParam.ioVRefNum = vRefNum;
        pb->ioParam.ioBuffer = Ptr(&volParms);
        pb->ioParam.ioReqCount = sizeof(GetVolParmsInfoBuffer);

        return GUSIFSHGetVolParms(&pb);
    }

<Declarations of C GUSIFSWrappers 1097>+≡ (1088) <1105 1110>
    OSErr GUSIFSCreate(const FSSpec * spec, OSType creator, OSType type, ScriptCode script);

<Declarations of C++ GUSIFSWrappers 1090>+≡ (1088) <1104 1118>
    OSErr GUSIFSCreate(const FSSpec * spec);

```

⟨Implementation of GUSIFSWrappers 1091⟩+≡

(1089) ◀1106 1111▶

```
OSErr GUSIFSCreate(const FSSpec * spec)
{
    GUSIIOPBWrapper<HParamBlockRec>    pb;

    pb->fileParam.ioVRefNum =    spec->vRefNum;
    pb->fileParam.ioDirID   =    spec->parID;
    pb->fileParam.ioNamePtr =    const_cast<StringPtr>(spec->name);

    pb.StartIO();
    PBHCreateAsync(&pb.fPB);
    return pb.FinishIO();
}

OSErr GUSIFSCreate(const FSSpec * spec, OSType creator, OSType type, ScriptCode script)
{
    OSErr err;

    if (err = GUSIFSCreate(spec))
        return err;

    GUSIIOPBWrapper<GUSICatInfo>    info;
    info->FileInfo().ioVRefNum =    spec->vRefNum;
    info->FileInfo().ioDirID   =    spec->parID;
    info->FileInfo().ioNamePtr =    const_cast<StringPtr>(spec->name);

    if (err = GUSIFSGetCatInfo(&info))
        goto nuke;

    info->FInfo().fdCreator    =    creator;
    info->FInfo().fdType      =    type;

    if (script == smSystemScript)
        info->FXInfo().fdScript =    0;
    else
        info->FXInfo().fdScript =    script | 0x80;

    if (err = GUSIFSSetCatInfo(&info))
        goto nuke;

    return noErr;
nuke:
    GUSIFSDelete(spec);

    return err;
}
```

⟨Declarations of C GUSIFSWrappers 1097⟩+≡

(1088) ◀1107 1112▶

```
OSErr GUSIFSDelete(const FSSpec * spec);
```

```

(Implementation of C GUSIFSWrappers 1091)+≡ (1089) <1109 1113>
    OSErr GUSIFSDDelete(const FSSpec * spec)
    {
        GUSIIOPBWrapper<HParamBlockRec>    pb;

        pb->fileParam.ioVRefNum =    spec->vRefNum;
        pb->fileParam.ioDirID    =    spec->parID;
        pb->fileParam.ioNamePtr =    const_cast<StringPtr>(spec->name);

        pb.StartIO();
        PBHDeleteAsync(&pb.fPB);
        return pb.FinishIO();
    }

(Declarations of C GUSIFSWrappers 1097)+≡ (1088) <1110 1114>
    OSErr GUSIFSDirCreate(const FSSpec * spec);

(Implementation of C GUSIFSWrappers 1091)+≡ (1089) <1111 1115>
    OSErr GUSIFSDirCreate(const FSSpec * spec)
    {
        GUSIIOPBWrapper<HParamBlockRec>    pb;

        pb->fileParam.ioVRefNum =    spec->vRefNum;
        pb->fileParam.ioDirID    =    spec->parID;
        pb->fileParam.ioNamePtr =    const_cast<StringPtr>(spec->name);

        pb.StartIO();
        PBDirCreateAsync(&pb.fPB);
        return pb.FinishIO();
    }

(Declarations of C GUSIFSWrappers 1097)+≡ (1088) <1112 1116>
    OSErr GUSIFSSetFLock(const FSSpec * spec);
    OSErr GUSIFSRstFLock(const FSSpec * spec);

```


<Implementation of GUSIFSWrappers 1091>+≡ (1089) <1113 1117>

```

OSError GUSIFSSetFlock(const FSSpec * spec)
{
    GUSIIOPBWrapper<HParamBlockRec>    pb;

    pb->fileParam.ioVRefNum =    spec->vRefNum;
    pb->fileParam.ioDirID   =    spec->parID;
    pb->fileParam.ioNamePtr =    const_cast<StringPtr>(spec->name);

    pb.StartIO();
    PBHSetFlockAsync(&pb.fPB);
    return pb.FinishIO();
}

OSError GUSIFSRstFlock(const FSSpec * spec)
{
    GUSIIOPBWrapper<HParamBlockRec>    pb;

    pb->fileParam.ioVRefNum =    spec->vRefNum;
    pb->fileParam.ioDirID   =    spec->parID;
    pb->fileParam.ioNamePtr =    const_cast<StringPtr>(spec->name);

    pb.StartIO();
    PBHRstFlockAsync(&pb.fPB);
    return pb.FinishIO();
}

```

<Declarations of C GUSIFSWrappers 1097>+≡ (1088) <1114 1119>

```

OSError GUSIFSRename(const FSSpec * spec, ConstStr255Param newname);

```

<Implementation of GUSIFSWrappers 1091>+≡ (1089) <1115 1120>

```

OSError GUSIFSRename(const FSSpec * spec, ConstStr255Param newname)
{
    GUSIIOPBWrapper<HParamBlockRec>    pb;

    pb->fileParam.ioVRefNum =    spec->vRefNum;
    pb->fileParam.ioDirID   =    spec->parID;
    pb->fileParam.ioNamePtr =    const_cast<StringPtr>(spec->name);
    pb->ioParam.ioMisc      =    (Ptr)const_cast<StringPtr>(newname);

    pb.StartIO();
    PBHRenameAsync(&pb.fPB);
    return pb.FinishIO();
}

```

<Declarations of C++ GUSIFSWrappers 1090>+≡ (1088) <1108

```

OSError GUSIFSCatMove(const FSSpec * spec, long dest);

```

<Declarations of C GUSIFSWrappers 1097>+≡ (1088) <1116 1121>

```

OSError GUSIFSCatMove(const FSSpec * spec, const FSSpec * dest);

```

(Implementation of GUSIFSWrappers 1091) +≡ (1089) <1117 1122>

```
static OSerr GUSIFSCatMove1(GUSIIOPBWrapper<CMovePBRec> * pb)
{
    pb->StartIO();
    PBCatMoveAsync(&pb->fPB);
    return pb->FinishIO();
}
OSerr GUSIFSCatMove(const FSSpec * spec, long dest)
{
    GUSIIOPBWrapper<CMovePBRec> pb;

    pb->ioVRefNum = spec->vRefNum;
    pb->ioDirID = spec->parID;
    pb->ioNamePtr = const_cast<StringPtr>(spec->name);
    pb->ioNewName = nil;
    pb->ioNewDirID = dest;

    return GUSIFSCatMove1(&pb);
}
OSerr GUSIFSCatMove(const FSSpec * spec, const FSSpec * dest)
{
    GUSIIOPBWrapper<CMovePBRec> pb;

    pb->ioVRefNum = spec->vRefNum;
    pb->ioDirID = spec->parID;
    pb->ioNamePtr = const_cast<StringPtr>(spec->name);
    pb->ioNewName = const_cast<StringPtr>(dest->name);
    pb->ioNewDirID = dest->parID;

    return GUSIFSCatMove1(&pb);
}
```

(Declarations of C GUSIFSWrappers 1097) +≡ (1088) <1119

```
OSerr GUSIFSMoveRename(const FSSpec * spec, const FSSpec * dest);
```

<Implementation of GUSIFSWrappers 1091>+≡ (1089) <1120

```
static OSErr GUSIFSMoveRename1(const FSSpec * spec, const FSSpec * dest)
{
    OSErr err;

    if (err = GUSIFSCatMove(spec, dest->parID))
        return err;
    FSSpec corner(*spec);
    corner.parID = dest->parID;
    if (err = GUSIFSRename(&corner, dest->name))
        GUSIFSCatMove(&corner, spec->parID);
    return err;
}
OSErr GUSIFSMoveRename(const FSSpec * spec, const FSSpec * dest)
{
    OSErr err;
    GUSIIOPBWrapper<HParamBlockRec> pb;

    <Try PBMoveRename if available 1123>
    <Move and rename, moving the corner file out of the way if necessary 1124>

    return err;
}
```

If possible, we use the combined call.

<Try PBMoveRename if available 1123>≡ (1122)

```
GetVolParmsInfoBuffer volParms;

if (!GUSIFSGetVolParms(spec->vRefNum, &volParms)
    && (volParms.vMAttrib & (1 << bHasMoveRename)))
{
    pb->copyParam.ioVRefNum = spec->vRefNum;
    pb->copyParam.ioDirID = spec->parID;
    pb->copyParam.ioNamePtr = const_cast<StringPtr>(spec->name);
    pb->copyParam.ioNewName = nil;
    pb->copyParam.ioNewDirID = dest->parID;
    pb->copyParam.ioCopyName = const_cast<StringPtr>(dest->name);

    pb.StartIO();
    PBHMoveRenameAsync(&pb.fPB);
    err = pb.FinishIO();

    if (err != paramErr)
        return err;
}
```

```

(Move and rename, moving the corner file out of the way if necessary 1124)≡ (1122)
GUSIFileSpec corner(*spec);
corner.SetParID(dest->parID);
if (corner.Exists()) {
    GUSITempFileSpec temp(dest->vRefNum, dest->parID);
    err = GUSIFSRename(&corner, temp->name);
    if (!err) {
        err = GUSIFSMoveRename1(spec, dest);
        GUSIFSRename(&temp, corner->name);
    }
} else
    err = GUSIFSMoveRename1(spec, dest);

```

Chapter 30

File specifications

While the Macintosh toolbox has a convenient type `FSSpec` to pass to file system routines, it lacks manipulation functions. We provide them here. This module has been known under a different name and with different data types in GUSI 1.

```
<GUSIFileSpec.h 1125>≡
#ifndef _GUSIFileSpec_
#define _GUSIFileSpec_

#include <MacTypes.h>
#include <Files.h>
#include <Folders.h>

#include <string.h>

// <<Plain C declarations for [[GUSIFileSpec]]>>

#ifdef GUSI_SOURCE

#include "GUSIBasics.h"
#include "GUSIContext.h"

<Definition of class GUSICatInfo 1127>
<Definition of class GUSIFileSpec 1128>

<Inline member functions for class GUSICatInfo 1151>
<Inline member functions for class GUSIFileSpec 1153>

#endif /* GUSI_SOURCE */

#endif /* GUSIFileSpec */
```

```

(GUSIFileSpec.cp 1126)≡
#include "GUSIInternal.h"
#include "GUSIFileSpec.h"
#include "GUSIFSWrappers.h"

#include <PLStringFuncs.h>
#include <Errors.h>
#include <TextUtils.h>
#include <Resources.h>
#include <Memory.h>
#include <Aliases.h>

#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>

(Auxiliary functions for class GUSIFileSpec 1166)
(Member functions for class GUSIFileSpec 1155)

```

30.1 Definition of GUSICatInfo

GUSICatInfo is a wrapper for CInfoPBRec. Since the latter is a union, we cannot inherit from it.

```

(Definition of class GUSICatInfo 1127)≡ (1125)
class GUSICatInfo {
    CInfoPBRec fInfo;
public:
    bool        IsFile() const;
    bool        IsAlias() const;
    bool        DirIsExported() const;
    bool        DirIsMounted() const;
    bool        DirIsShared() const;
    bool        HasRdPerm() const;
    bool        HasWrPerm() const;
    bool        Locked() const;

    CInfoPBRec & Info() { return fInfo; }
    operator CInfoPBRec &() { return fInfo; }
    struct HFileInfo & FileInfo() { return fInfo.hFileInfo; }
    struct DirInfo & DirInfo() { return fInfo.dirInfo; }
    struct FInfo & FInfo() { return fInfo.hFileInfo.ioFlFndrInfo; }
    struct FXInfo & FXInfo() { return fInfo.hFileInfo.ioFlXFndrInfo; }

    const CInfoPBRec & Info() const { return fInfo; }
    operator const CInfoPBRec &() const { return fInfo; }
    const struct HFileInfo & FileInfo() const { return fInfo.hFileInfo; }
    const struct DirInfo & DirInfo() const { return fInfo.dirInfo; }
    const struct FInfo & FInfo() const { return fInfo.hFileInfo.ioFlFndrInfo; }
    const struct FXInfo & FXInfo() const { return fInfo.hFileInfo.ioFlXFndrInfo; }
};

```

30.2 Definition of GUSIFileSpec

A GUSIFileSpec is a manipulation friendly version of an FSSpec. Due to conversion operators, a GUSIFileSpec can be used wherever a const FSSpec is specified. For any long term storage, FSSpec should be used rather than the much bulkier GUSIFileSpec.

```
(Definition of class GUSIFileSpec 1128)≡ (1125) 1150▷
class GUSIFileSpec : private FSSpec {
public:
    (Constructors for GUSIFileSpec 1129)
    (Error handling in GUSIFileSpec 1133)
    (Default directory handling in GUSIFileSpec 1134)
    (Converting a GUSIFileSpec to a FSSpec 1135)
    (Getting the GUSICatInfo for a GUSIFileSpec 1136)
    (Getting path names from a GUSIFileSpec 1137)
    (Alias resolution for a GUSIFileSpec 1139)
    (Manipulating a GUSIFileSpec 1141)
    (Comparing two GUSIFileSpec objects 1148)
protected:
    (Privatissima of GUSIFileSpec 1152)
};
```

A GUSIFileSpec can be constructed in lots of different ways. Most of the constructors have a final argument useAlias that, when true, suppresses resolution of leaf aliases.

First, two trivial cases: The default constructor and the copy constructor, which do exactly what you'd expect them to do.

```
(Constructors for GUSIFileSpec 1129)≡ (1128) 1130▷
GUSIFileSpec() {}
GUSIFileSpec(const GUSIFileSpec & spec);
```

The FSSpec conversion is almost a copy constructor, but by default resolves leaf aliases.

```
(Constructors for GUSIFileSpec 1129)+≡ (1128) ◀1129 1131▷
GUSIFileSpec(const FSSpec & spec, bool useAlias = false);
```

A number of convenience constructors let you construct a GUSIFileSpec from various components.

```
(Constructors for GUSIFileSpec 1129)+≡ (1128) ◀1130 1132▷
// Construct from volume reference number, directory ID & file name
GUSIFileSpec(short vRefNum, long parID, ConstStr31Param name, bool useAlias = false);

// Construct from working directory & file name
GUSIFileSpec(short wd, ConstStr31Param name, bool useAlias = false);

// Construct from full or relative path
GUSIFileSpec(const char * path, bool useAlias = false);

// Construct from open file reference number
explicit GUSIFileSpec(short fRefNum);
```

Finally, there is a constructor for constructing a GUSIFileSpec for one of the folders obtainable with FindFolder().

```
(Constructors for GUSIFileSpec 1129)+≡ (1128) ◀1131
GUSIFileSpec(OSType object, short vol = kOnSystemDisk);
```

All GUSIFileSpecs have an error variable from which the last error is available.

```
(Error handling in GUSIFileSpec 1133)≡ (1128)
OSErr      Error();
            operator void*() const;
bool       operator!() const;
```

While earlier versions of GUSI maintained a notion of "current directory" that was independent of the HFS default directory, there is no such distinction anymore in GUSI 2. SetDefaultDirectory sets the default directory to vRefNum, dirID. GetDefaultDirectory sets vRefNum, dirID to the default directory. Neither routine affects the name. GetVolume gets the named or indexed volume.

```
(Default directory handling in GUSIFileSpec 1134)≡ (1128)
OSErr  SetDefaultDirectory();
OSErr  GetDefaultDirectory();
OSErr  GetVolume(short index = -1);
```

Conversions of GUSIFileSpec to FSSpec values and references is, of course, simple. Pointers are a bit trickier; we define an custom operator& and two smart pointer classes. operator->(), however, is simpler.

```
(Converting a GUSIFileSpec to a FSSpec 1135)≡ (1128)
operator const FSSpec &() const;

class pointer {
public:
    pointer(GUSIFileSpec * ptr);
    operator GUSIFileSpec *() const;
    operator const FSSpec *() const;
private:
    GUSIFileSpec * ptr;
};
pointer operator&();

class const_pointer {
public:
    const_pointer(const GUSIFileSpec * ptr);
    operator const GUSIFileSpec *() const;
    operator const FSSpec *() const;
private:
    const GUSIFileSpec * ptr;
};
const_pointer operator&() const;

const FSSpec * operator->() const;
```


Each `GUSIFileSpec` has an implicit `GUSICatInfo` record associated with it. `CatInfo()` calls `GetCatInfo()` if the record is not already valid and returns a pointer to the record. `DirInfo()` replaces the `GUSIFileSpec` by the specification of its parent directory and returns a pointer to information about the parent. Both calls return a nil pointer if the object does not exist. If all you are interested in is whether the object exists, call `Exists()`.

```
<Getting the GUSICatInfo for a GUSIFileSpec 1136>≡ (1128)
const GUSICatInfo * CatInfo(short index);
const GUSICatInfo * DirInfo();
const GUSICatInfo * CatInfo() const;
bool Exists() const;
```

In many POSIXish contexts, it's necessary to specify path names with C string. Although this practice is dangerous on a Mac, we of course have to conform to it. The basic operations are getting a full path and getting a path relative to a directory (the current directory by default).

```
<Getting path names from a GUSIFileSpec 1137>≡ (1128) 1138>
char * FullPath() const;
char * RelativePath() const;
char * RelativePath(const FSSpec & dir) const;
```

If the path ultimately is going to flow back into a GUSI routine again, it is possible to simply encode the `GUSIFileSpec` in ASCII. This is fast and reliable, but you should of course not employ it with any non-GUSI destination routine and should never store such a part across a reboot. The standard `GUSIFileSpec` constructors for paths will accept encoded paths.

The encoding is defined as:

- 1 byte: DC1 (ASCII 0x11)
- 4 bytes: Volume reference number in zero-padded hex
- 8 bytes: Directory ID in zero-padded hex
- n bytes: Partial pathname, starting with ':'

```
<Getting path names from a GUSIFileSpec 1137>+≡ (1128) <1137
char * EncodedPath() const;
```

Most aliases are resolved implicitly, but occasionally you may want to do explicit resolution. `Resolve` resolves an alias. If `gently` is set, nonexistent target files of aliases don't cause an error to be returned.

```
<Alias resolution for a GUSIFileSpec 1139>≡ (1128) 1140>
OSErr Resolve(bool gently = true);
```

`AliasPath` returns the path an alias points to without mounting any volumes.

```
<Alias resolution for a GUSIFileSpec 1139>+≡ (1128) <1139
char * AliasPath() const;
```

A major feature of the `GUSIFileSpec` class is the set of operators for manipulating a file specification.

`operator--` replaces a file specification with the directory specification of its parent.

```
<Manipulating a GUSIFileSpec 1141>≡ (1128) 1142>
GUSIFileSpec & operator--();
```

operator++ sets the dirID of a directory specification to the ID of the directory.

```
(Manipulating a GUSIFileSpec 1141)+≡ (1128) <1141 1143>
GUSIFileSpec & operator++();
```

The two versions of operator+=, which internally both call AddPathComponent, replace a directory specification with the specification

```
(Manipulating a GUSIFileSpec 1141)+≡ (1128) <1142 1144>
GUSIFileSpec & AddPathComponent(const char * name, int length, bool fullSpec);
GUSIFileSpec & operator+=(ConstStr31Param name);
GUSIFileSpec & operator+=(const char * name);
```

operator+ provides a non-destructive variant of operator+=.

```
(Manipulating a GUSIFileSpec 1141)+≡ (1128) <1143 1145>
friend GUSIFileSpec operator+(const FSSpec & spec, ConstStr31Param name);
friend GUSIFileSpec operator+(const FSSpec & spec, const char * name);
```

Array access replaces the file specification with the indexth object in the *parent directory* of the specification (allowing the same specification to be reused for reading a directory).

```
(Manipulating a GUSIFileSpec 1141)+≡ (1128) <1144 1146>
GUSIFileSpec & operator[](short index);
```

To manipulate the fields of the file specification directly, there is a procedural interface.

```
(Manipulating a GUSIFileSpec 1141)+≡ (1128) <1145 1147>
void SetVRef(short vref);
void SetParID(long parid);
void SetName(ConstStr63Param nam);
```

For some modifications to propagate quickly, the surrounding folder needs to have its date modified.

```
(Manipulating a GUSIFileSpec 1141)+≡ (1128) <1146
OSErr TouchFolder();
```

Two GUSIFileSpecs can be compared for (in)equality.

```
(Comparing two GUSIFileSpec objects 1148)≡ (1128) 1149>
friend bool operator==(const GUSIFileSpec & one, const GUSIFileSpec & other);
```

IsParentOf determines whether the object specifies a parent directory of other.

```
(Comparing two GUSIFileSpec objects 1148)+≡ (1128) <1148
bool IsParentOf(const GUSIFileSpec & other) const;
```

A GUSITempFileSpec is just a GUSIFileSpec with constructors to construct filenames for temporary files.

```
(Definition of class GUSIFileSpec 1128)+≡ (1125) <1128
class GUSITempFileSpec : public GUSIFileSpec {
public:
    GUSITempFileSpec(short vRefNum = kOnSystemDisk);
    GUSITempFileSpec(short vRefNum, long parID);
    GUSITempFileSpec(ConstStr31Param basename);
    GUSITempFileSpec(short vRefNum, ConstStr31Param basename);
    GUSITempFileSpec(short vRefNum, long parID, ConstStr31Param basename);
private:
    void TempName();
    void TempName(ConstStr31Param basename);

    static int sID;
};
```

30.3 Implementation of GUSICatInfo

All of the member functions for GUSICatInfo are inline.

```
(Inline member functions for class GUSICatInfo 1151)≡ (1125)
inline bool GUSICatInfo::IsFile() const
{
    return !(DirInfo().ioFlAttrib & 0x10);
}

inline bool GUSICatInfo::IsAlias() const
{
    return
        !(FileInfo().ioFlAttrib & 0x10) &&
        (FInfo().fdFlags & (1 << 15));
}

inline bool GUSICatInfo::DirIsExported() const
{
    return (FileInfo().ioFlAttrib & 0x20) != 0;
}

inline bool GUSICatInfo::DirIsMounted() const
{
    return (FileInfo().ioFlAttrib & 0x08) != 0;
}

inline bool GUSICatInfo::DirIsShared() const
{
    return (FileInfo().ioFlAttrib & 0x04) != 0;
}

inline bool GUSICatInfo::HasRdPerm() const
{
    return !(DirInfo().ioACUser & 0x02) != 0;
}

inline bool GUSICatInfo::HasWrPerm() const
{
    return !(DirInfo().ioACUser & 0x04) != 0;
}

inline bool GUSICatInfo::Locked() const
{
    return (FileInfo().ioFlAttrib & 0x11) == 0x01;
}
```

30.4 Implementation of GUSIFileSpec

sError contains the error code for failed calls. Error returns the value.

```
(Privatissima of GUSIFileSpec 1152)≡ (1128) 1154▷
mutable OSErr fError;
```

```

<Inline member functions for class GUSIFileSpec 1153>≡ (1125) 1156»
inline OSErr GUSIFileSpec::Error()
{
    return fError;
}

inline GUSIFileSpec::operator void*() const
{
    return (void *)!fError;
}

inline bool GUSIFileSpec::operator!() const
{
    return fError!=0;
}

```

For path name constructions, a sometimes large scratch area is needed which is maintained in sScratch. A scratch request preserves a preexisting scratch area at the end of the new area if extend is nonzero.

```

<Privatissima of GUSIFileSpec 1152>+≡ (1128) «1152 1158»
static char *      sScratch;
static long        sScratchSize;

static char *      CScratch(bool extend = false);
static StringPtr   PScratch();

<Member functions for class GUSIFileSpec 1155>≡ (1126) 1157»
char *  GUSIFileSpec::sScratch;
long    GUSIFileSpec::sScratchSize;

<Inline member functions for class GUSIFileSpec 1153>+≡ (1125) «1153 1171»
inline StringPtr GUSIFileSpec::PScratch()
{
    return (StringPtr) CScratch();
}

<Member functions for class GUSIFileSpec 1155>+≡ (1126) «1155 1159»
char * GUSIFileSpec::CScratch(bool extend)
{
    if (extend) {
        char * newScratch = NewPtr(sScratchSize + 64);
        if (!newScratch)
            return nil;
        BlockMoveData(sScratch, newScratch+64, sScratchSize);
        sScratchSize += 64;
        DisposePtr(sScratch);

        return sScratch = newScratch;
    } else if (!sScratchSize)
        if (sScratch = NewPtr(256))
            sScratchSize = 256;

    return sScratch;
}

```

A GUSIFileSpec has a GUSICatInfo embedded and a flag fValidInfo indicating that it is valid.

```

<Privatissima of GUSIFileSpec 1152>+≡ (1128) <1154 1173>
    GUSIIOBPWrapper<GUSICatInfo>    fInfo;
    bool                             fValidInfo;

<Member functions for class GUSIFileSpec 1155>+≡ (1126) <1157 1160>
    GUSIFileSpec::GUSIFileSpec(const GUSIFileSpec & spec)
        : fValidInfo(false)
    {
        vRefNum = spec.vRefNum;
        parID   = spec.parID;
        if (*spec.name < 64)
            PLstrcpy(name, spec.name);
        fError = noErr;
    }

<Member functions for class GUSIFileSpec 1155>+≡ (1126) <1159 1161>
    GUSIFileSpec::GUSIFileSpec(const FSSpec & spec, bool useAlias)
        : fValidInfo(false)
    {
        vRefNum = spec.vRefNum;
        parID   = spec.parID;
        if (*spec.name < 64)
            PLstrcpy(name, spec.name);
        fError = noErr;

        if (!useAlias)
            Resolve();
    }

<Member functions for class GUSIFileSpec 1155>+≡ (1126) <1160 1162>
    GUSIFileSpec::GUSIFileSpec(short vRefNum, long parID, ConstStr31Param name, bool useAlias)
        : fValidInfo(false)
    {
        OSErr err;

        if ((err = FSMakeFSSpec(vRefNum, parID, name, this)) && (err != fnfErr)) {
            this->vRefNum = vRefNum;
            this->parID   = parID;
            memcpy(this->name, name, *name+1);
        }
        fError = noErr;

        if (!useAlias)
            Resolve();

        if (EqualString(this->name, name, false, true))
            memcpy(this->name, name, *name+1);
    }

```

```

<Member functions for class GUSIFileSpec 1155>+≡ (1126) <1161 1163>
GUSIFileSpec::GUSIFileSpec(short wd, ConstStr31Param name, bool useAlias)
: fValidInfo(false)
{
    if ((fError = FMakeFSSpec(wd, 0, name, this)) && (fError != fnfErr))
        return;

    if (!useAlias)
        Resolve();

    if (EqualString(this->name, name, false, true))
        memcpy(this->name, name, *name+1);
}

<Member functions for class GUSIFileSpec 1155>+≡ (1126) <1162 1164>
GUSIFileSpec::GUSIFileSpec(OSType object, short vol)
: fValidInfo(false)
{
    fError = FindFolder(vol, object, true, &vRefNum, &parID);
}

<Member functions for class GUSIFileSpec 1155>+≡ (1126) <1163 1165>
GUSIFileSpec::GUSIFileSpec(short fRefNum)
: fValidInfo(false)
{
    GUSIIOPBWrapper<FCBPBRec> fcb;

    fcb->ioNamePtr = name;
    fcb->ioRefNum = fRefNum;
    fcb->ioFCBIndx = 0;

    if (fError = GUSIFSGetFCBInfo(&fcb))
        return;

    vRefNum = fcb->ioFCBVRefNum;
    parID = fcb->ioFCBParID;
}

```

The pathname constructor is by far the most complex constructor. This code is also probably not portable to Rhapsody, whatever and whenever that will be.

To minimize the number of CatInfo calls necessary, the fullSpec variable reflects whether the entire FSSpec under construction or only the vRefNum and parID are valid.

```

<Member functions for class GUSIFFileSpec 1155>+≡ (1126) <1164 1172>
GUSIFFileSpec::GUSIFFileSpec(const char * path, bool useAlias)
    : fValidInfo(false)
{
    const char *    nextPath;
    bool            fullSpec    =    false;

    vRefNum = 0;
    parID   = 0;

    <Try decoding the path as an encoded FSSpec 1167>
    <Try converting the path with FSMakeFSSpec and return 1168>

    <Determine the starting directory of the path 1169>

    while (*path && !fError) {
        if (*path == ':') {
            <Walk directories upwards 1170>
        } else {
            fullSpec = true;
            if (nextPath = strchr(path, ':')) {
                AddPathComponent(path, nextPath-path, fullSpec);
                path = nextPath+1;
            } else {
                AddPathComponent(path, strlen(path), fullSpec);
                break;
            }
        }
    }
    if (!fError && !fullSpec)
        --(*this);
    if (!fError && !useAlias)
        Resolve();
}

```

First, we test whether the path represents an encoded [FSSpec]]. This is potentially ambiguous, but no sane person would start a disk name with DC1. The ReadHex function, which only works on big endian machines, reads a specified number of hex digits.

(Auxiliary functions for class GUSIFFileSpec 1166)≡ (1126)

```
bool ReadHex(const char * path, int bytes, char * result)
{
    char hexbyte[3];
    hexbyte[2] = 0;
    while (bytes--) {
        hexbyte[0] = *path++; hexbyte[1] = *path++;
        if (isxdigit(hexbyte[0]) && isxdigit(hexbyte[1]))
            *result = (char) strtol(hexbyte, nil, 16);
        else
            return false;
    }
    return true;
}
```

(Try decoding the path as an encoded FSSpec 1167)≡ (1165)

```
if (*path == 0x11 && path[13] == ':') // Magic DC1 character
    if (
        !ReadHex(path+1, 2, (char *)&vRefNum) || !ReadHex(path+5, 4, (char *)&parID)
    ) {
        vRefNum = 0;
        parID = 0;
    } else
        path += 13;
```

(Try converting the path with FSMakeFSSpec and return 1168)≡ (1165)

```
int pathLen = (int)strlen(path);
StringPtr ppath = PScratch();
if (pathLen < 256 && ppath) {
    memcpy(ppath+1, path, pathLen);

    switch (fError = FSMakeFSSpec(vRefNum, parID, ppath, this)) {
    case fnfErr:
        fError = noErr;

        return;
    case noErr:
        if (!useAlias)
            Resolve();

        while ((nextPath = strchr(path, ':')) && nextPath[1])
            path = nextPath+1;
        memcpy(ppath+1, path, ppath[0] = strlen(path) - (nextPath != 0));
        if (EqualString(name, ppath, false, true))
            memcpy(name, ppath, ppath[0]+1);

        return;
    default:
        break;
    }
}
```



```

⟨Determine the starting directory of the path 1169⟩≡ (1165)
if (path[0] == ':' || !(nextPath = strchr(path, ':'))) {
    if (!vRefNum && !parID)
        GetDefaultDirectory();

    if (*path == ':')
        ++path;
} else {
    if (nextPath - (char *) path > 62) {
        fError = bdNamErr;

        return;
    }

    memcpy(this->name+1, (char *) path, *this->name = nextPath - path);

    if (GetVolume())
        return;

    path = nextPath + 1;
}

```

Going upwards is a bit more complicated than might seem necessary at first, because if "a:b" is an alias pointing to "c:d:e:", we want a:b::"] to be the same as "c:d:".

```

⟨Walk directories upwards 1170⟩≡ (1165)
if (!fullSpec)
    --(*this);
if (!fError)
    Resolve();
fullSpec = false;
while (!fError && *++path == ':')
    --(*this);

⟨Inline member functions for class GUSIFileSpec 1153⟩+≡ (1125) ◀1156 1177▶
inline OSErr GUSIFileSpec::SetDefaultDirectory()
{
    return fError = HSetVol(nil, vRefNum, parID);
}

inline OSErr GUSIFileSpec::GetDefaultDirectory()
{
    name[0]      = 0;
    fValidInfo   = false;
    return fError = HGetVol(nil, &vRefNum, &parID);
}

```

If name is used, we have to make sure that it ends with a colon.

```

<Member functions for class GUSIFFileSpec 1155>+≡ (1126) <1165 1174>
  OSErr GUSIFFileSpec::GetVolume(short index)
  {
    fValidInfo = false;
    if (name[0] || index>=0) {
      GUSIIOPBWrapper<ParamBlockRec> vol;

      vol->volumeParam.ioVRefNum = vRefNum;
      vol->volumeParam.ioNamePtr = name;
      vol->volumeParam.ioVolIndex = index;

      if (index < 0 && name[name[0]] != ':')
        name[++name[0]] = ':';

      if (fError = GUSIFSGetVInfo(&vol))
        return fError;

      vRefNum = vol->volumeParam.ioVRefNum;
    } else {
      fError = noErr;
      vRefNum = 0;
    }
    parID = fsRtParID;

    return fError;
  }

```

For convenience, we define a fictitious parent directory of all volumes.

```

<Privatissima of GUSIFFileSpec 1152>+≡ (1128) <1158 1185>
  enum { ROOT_MAGIC_COOKIE = 666 };

```

operator-- replaces file specifications with their parent directory, volumes with the root pseudo directory.

```

<Member functions for class GUSIFFileSpec 1155>+≡ (1126) <1172 1175>
  GUSIFFileSpec & GUSIFFileSpec::operator--()
  {
    if (parID == fsRtParID) {
      vRefNum = ROOT_MAGIC_COOKIE;
      parID = 0;
      name[0] = 0;
      fError = noErr;
      fValidInfo = false;
    } else
      DirInfo();

    return *this;
  }

```

operator++ walks down one path component.

```
(1126) <1174 1176>
<Member functions for class GUSIFFileSpec 1155>+≡
GUSIFFileSpec & GUSIFFileSpec::operator++()
{
    if (!parID && vRefNum == ROOT_MAGIC_COOKIE) {
        vRefNum = 0;
        parID = fsRtParID;
        name[0] = 0;

        goto punt;
    }

    if (!CatInfo())
        goto punt;
    if (fInfo->IsAlias())
        if (Resolve() || !CatInfo())
            goto punt;
    if (fInfo->IsFile()) {
        fError = afpObjectTypeErr;

        goto punt;
    }

    parID = fInfo->DirInfo().ioDrDirID;
    name[0] = 0;
    fValidInfo = false;

punt:
    return *this;
}
```

[AddPathComponent]] is the basic operation for descending a directory hierarchy.

```
(1126) <1175 1178>
<Member functions for class GUSIFFileSpec 1155>+≡
GUSIFFileSpec & GUSIFFileSpec::AddPathComponent(const char * name, int length, bool fullSpec)
{
    if (length > 63) {
        fError = bdNamErr;

        goto punt;
    }

    if (fullSpec)
        if (!++(*this))
            goto punt;

    memcpy(this->name+1, name, *this->name = length);

    if (parID == fsRtParID)
        GetVolume();

punt:
    return *this;
}
```

operator+= and operator+ are merely wrappers around AddPathComponent.

```
<Inline member functions for class GUSIFileSpec 1153>+≡ (1125) <1171 1183>
inline GUSIFileSpec & GUSIFileSpec::operator+=(ConstStr31Param name)
{
    return AddPathComponent((char *) name+1, *name, true);
}

inline GUSIFileSpec & GUSIFileSpec::operator+=(const char * name)
{
    return AddPathComponent(name, strlen(name), true);
}

<Member functions for class GUSIFileSpec 1155>+≡ (1126) <1176 1179>
GUSIFileSpec operator+(const FSSpec & spec, ConstStr31Param name)
{
    GUSIFileSpec s(spec);

    return s += name;
}

GUSIFileSpec operator+(const FSSpec & spec, const char * name)
{
    GUSIFileSpec s(spec);

    return s += name;
}

<Member functions for class GUSIFileSpec 1155>+≡ (1126) <1178 1180>
GUSIFileSpec & GUSIFileSpec::operator[](short index)
{
    if (parID == fsRtParID)
        GetVolume(index);
    else
        CatInfo(index);

    return *this;
}
```

⟨*Member functions for class GUSIFileSpec 1155*⟩+≡ (1126) ◀1179 1181▶

```
void GUSIFileSpec::SetVRef(short vref)
{
    vRefNum      = vref;
    fValidInfo   = false;
}

void GUSIFileSpec::SetParID(long parid)
{
    parID        = parid;
    fValidInfo   = false;
}

void GUSIFileSpec::SetName(ConstStr63Param nam)
{
    PLstrcpy(name, nam);
    fValidInfo   = false;
}
```

⟨*Member functions for class GUSIFileSpec 1155*⟩+≡ (1126) ◀1180 1182▶

```
OSErr GUSIFileSpec::TouchFolder()
{
    GUSIFileSpec    folder(*this, true);

    if (!folder.DirInfo())
        return fError = folder.Error();

    GetDateTime(&folder.fInfo->DirInfo().ioDrMdDat);
    folder.fInfo->DirInfo().ioDrDirID = folder.fInfo->DirInfo().ioDrParID;

    return fError = GUSIFSSetCatInfo(&folder.fInfo);
}
```

The sort of information obtained depends on the index parameter.

⟨*Member functions for class GUSIFileSpec 1155*⟩+≡ (1126) ◀1181 1184▶

```
const GUSICatInfo * GUSIFileSpec::CatInfo(short index)
{
    if (fValidInfo && !index)    // Valid already
        return &fInfo.fPB;

    fInfo->DirInfo().ioVRefNum = vRefNum;
    fInfo->DirInfo().ioDrDirID = parID;
    fInfo->DirInfo().ioNamePtr = (StringPtr) name;
    fInfo->DirInfo().ioFDirIndex = index;
    fInfo->DirInfo().ioACUser = 0;

    fValidInfo = !(fError = GUSIFSSetCatInfo(&fInfo)) && index>=0;

    return fError ? nil : &fInfo.fPB;
}
```

The other variations of the call are simple.

```

<Inline member functions for class GUSIFileSpec 1153>+≡ (1125) <1177 1196>
inline const GUSICatInfo * GUSIFileSpec::CatInfo() const
{
    return const_cast<GUSIFileSpec *>(this)->CatInfo(0);
}

inline const GUSICatInfo * GUSIFileSpec::DirInfo()
{
    if (CatInfo(-1)) {
        parID      = fInfo->DirInfo().ioDrParID;
        fValidInfo = true;

        return &fInfo.fPB;
    } else
        return nil;
}

inline bool GUSIFileSpec::Exists() const
{
    return CatInfo() != nil;
}

```

We start the path getting functions with the full path. The full and relative path functions are quite similar, iterating a file specification current upward, accumulating the path in path.

```

<Member functions for class GUSIFileSpec 1155>+≡ (1126) <1182 1186>
char * GUSIFileSpec::FullPath() const
{
    char *          path = CScratch();
    GUSIFileSpec    current(*this);
    const GUSICatInfo * info      = current.CatInfo();
    bool            directory    = info && !info->IsFile();

    if (!path)
        return nil;
    *(path += sScratchSize-1) = 0;

    for (;;) {
        if (PrependPathComponent(path, current.name, directory))
            return nil;
        if (current.parID == fsRtParID)
            return path;
        if (!--current)
            return nil;
        directory = true;
    }
}

```

Each accumulation step is preformed in PrependPathComponent.

```

<Privatissima of GUSIFileSpec 1152>+≡ (1128) <1173
OSErr PrependPathComponent(char *&path, ConstStr63Param component, bool colon) const;

```

```

<Member functions for class GUSIFileSpec 1155>+≡ (1126) <1184 1187>
OSError GUSIFileSpec::PrependPathComponent(char *&path, ConstStr63Param component, bool colon) const
{
    if (colon)
        *--path = ':';
    if (path-sScratch < *component+1) {
        long length = sScratch+sScratchSize-path;
        if (!CScratch(true))
            return GUSI_MUTABLE(GUSIFileSpec, fError) = -108;
        path = sScratch+sScratchSize-length;
    }
    memcpy(path -= *component, component+1, *component);
    return noErr;
}

```

RelativePath works similarly, but has to compare with the stop directory at each step. GetVolume is called to translate all drive numbers to real volume numbers.

```

<Member functions for class GUSIFileSpec 1155>+≡ (1126) <1186 1188>
char * GUSIFileSpec::RelativePath(const FSSpec & dir) const
{
    GUSIFileSpec current(dir);
    if (current.GetVolume(0))
        return FullPath();
    short relVRef = current.vRefNum;
    long relDirID= current.parID;
    current = *this;
    if (current.GetVolume(0) || current.vRefNum != relVRef)
        return FullPath();

    char * path = CScratch();
    const GUSICatInfo * info = current.CatInfo();
    bool directory = info && !info->IsFile();

    if (!path)
        return nil;
    if (directory && info->DirInfo().ioDrDirID == relDirID)
        return strcpy(path, ":");
    *(path += sScratchSize-1) = 0;

    for (;;) {
        if (PrependPathComponent(path, current.name, directory))
            return nil;
        if (current.parID == relDirID) {
            if (directory)
                *--path = ':';
            return path;
        }
        if (current.parID == fsRtParID)
            return path;
        if (!--current)
            return nil;
        directory = true;
    }
}

```

```

<Member functions for class GUSIFileSpec 1155>+≡ (1126) <1187 1189>
char * GUSIFileSpec::RelativePath() const
{
    GUSIFileSpec here;
    here.GetDefaultDirectory();
    return RelativePath(here);
}

```

Encoding is simple. Note the use of the "%#s" Metrowerks specific extension.

```

<Member functions for class GUSIFileSpec 1155>+≡ (1126) <1188 1190>
char * GUSIFileSpec::EncodedPath() const
{
    if (!CScratch())
        return nil;

    sprintf(sScratch, "\\021%04hX%08X:%#s", vRefNum, parID, name);

    return sScratch;
}

```

```

<Member functions for class GUSIFileSpec 1155>+≡ (1126) <1189 1191>
OSErr GUSIFileSpec::Resolve(bool gently)
{
    const GUSICatInfo * info = CatInfo();

    if (!info || (!info->IsAlias() && (fError = resFNotFound)))
        if (gently)
            return fError = noErr;
        else
            return fError;

    Boolean isFolder;
    Boolean wasAlias;

    return fError = ResolveAliasFile(this, true, &isFolder, &wasAlias);
}

```


Getting the target of an alias without resolving it is quite tricky. We have to guess whether the target is a file or a directory.

```

⟨Member functions for class GUSIFileSpec 1155⟩+≡ (1126) ◀1190 1194▶
char * GUSIFileSpec::AliasPath() const
{
    const GUSICatInfo * info = CatInfo();
    if (!info || (!info->IsAlias() && (GUSI_MUTABLE(GUSIFileSpec, fError) = resFNotFound)))
        return nil;

    char * path = CScratch();
    if (!path)
        return nil;
    *(path += sScratchSize-1) = 0;

    AliasHandle alias;
    ⟨Get the AliasHandle for the alias file and detach it 1192⟩
    bool directory;
    ⟨Find out if alias points at a file or a directory 1193⟩
    Str63 component;

    /*
     Build path from target up to root. Separate with colons.
    */
    for (short index = 0; !(GUSI_MUTABLE(GUSIFileSpec, fError) = GetAliasInfo(alias, index, component)); ++index)
        if (!*component || PrependPathComponent(path, component, directory))
            break;
        else
            directory = true;

    if (!fError && !(GUSI_MUTABLE(GUSIFileSpec, fError) = GetAliasInfo(alias, asiVolumeName, component)))
        PrependPathComponent(path, component, true);

    DisposeHandle((Handle) alias);

    return fError ? nil : path;
}

```

(Get the AliasHandle for the alias file and detach it 1192)≡ (1191)

```

short  oldRes = CurResFile();
short  res    = FSpOpenResFile(this, fsRdPerm);
if (res == -1) {
    GUSI_MUTABLE(GUSIFileSpec, fError) = ResError();
    alias = nil;
} else {
    if (alias = (AliasHandle) Get1Resource('alis', 0))
        DetachResource((Handle) alias);
    else
        GUSI_MUTABLE(GUSIFileSpec, fError) = ResError();

    CloseResFile(res);
}
UseResFile(oldRes);

if (!alias)
    return nil;

```

We have to guess whether the alias points at a file or a directory, but if the alias was constructed by GUSI or by the Finder, this guess should be correct (though not necessarily up to date).

(Find out if alias points at a file or a directory 1193)≡ (1191)

```

directory = false;
if (info->FInfo().fdCreator == 'MACS')
    switch (info->FInfo().fdType) {
        case 'srvr':
        case 'fadr':
        case 'faet':
        case 'hdk':
        case 'famn':
        case 'fash':
        case 'fdrp':
            directory = true;
    }

```

Comparisons are again a bit tricky because of the volume numbers.

(Member functions for class GUSIFileSpec 1155)+≡ (1126) <1191 1195>

```

bool operator==(const GUSIFileSpec & one, const GUSIFileSpec & other)
{
    if (one.parID != other.parID || !EqualString(one.name, other.name, false, true))
        return false;
    if (one.vRefNum == other.vRefNum)
        return true;
    GUSIFileSpec current;
    current = one;
    current.GetVolume(0);
    short vRef1 = current.vRefNum;
    current = other;
    current.GetVolume(0);
    short vRef2 = current.vRefNum;

    return vRef1 == vRef2;
}

```

(Member functions for class GUSIFileSpec 1155)+≡ (1126) ◀1194 1199▶

```

bool    GUSIFileSpec::IsParentOf(const GUSIFileSpec & other) const
{
    for (GUSIFileSpec current(other); !(--current).Error();)
        if (current == *this)
            return true;

    return false;
}

```

Reference conversion is straightforward, as is operator->.

(Inline member functions for class GUSIFileSpec 1153)+≡ (1125) ◀1183 1197▶

```

inline GUSIFileSpec::operator const FSSpec &() const
{
    return *this;
}
inline const FSSpec * GUSIFileSpec::operator->() const
{
    return this;
}

```

Pointers, however, are a trickier issue, as they might be used either as a GUSIFileSpec * or as an FSSpec *.

(Inline member functions for class GUSIFileSpec 1153)+≡ (1125) ◀1196 1198▶

```

inline GUSIFileSpec::const_pointer::const_pointer(const GUSIFileSpec * ptr)
    : ptr(ptr)
{
}
inline GUSIFileSpec::const_pointer::operator const GUSIFileSpec *() const
{
    return ptr;
}
inline GUSIFileSpec::const_pointer::operator const FSSpec *() const
{
    return &static_cast<const FSSpec &>(*ptr);
}
inline GUSIFileSpec::const_pointer GUSIFileSpec::operator&() const
{
    return const_pointer(this);
}

```

GUSIFileSpec::pointer is the non-constant equivalent to GUSIFileSpec::const_pointer.

```
(Inline member functions for class GUSIFileSpec 1153) +≡ (1125) <1197
inline GUSIFileSpec::pointer::pointer(GUSIFileSpec * ptr)
    : ptr(ptr)
{
}
inline GUSIFileSpec::pointer::operator GUSIFileSpec *() const
{
    return ptr;
}
inline GUSIFileSpec::pointer::operator const FSSpec *() const
{
    return &static_cast<const FSSpec &>(*ptr);
}
inline GUSIFileSpec::pointer GUSIFileSpec::operator&()
{
    return pointer(this);
}
}
```

The constructors for GUSITempFileSpec differ in the base constructors they call.

```
(Member functions for class GUSIFileSpec 1155) +≡ (1126) <1195 1200>
GUSITempFileSpec::GUSITempFileSpec(short vRefNum)
    : GUSIFileSpec(kTemporaryFolderType, vRefNum)
{
    TempName();
}
GUSITempFileSpec::GUSITempFileSpec(short vRefNum, long parID)
{
    SetVRef(vRefNum);
    SetParID(parID);

    TempName();
}
}
```

Optionally, a base name can be specified for a temporary name.

```
(Member functions for class GUSIFileSpec 1155) +≡ (1126) <1199 1201>
GUSITempFileSpec::GUSITempFileSpec(ConstStr63Param basename)
    : GUSIFileSpec(kTemporaryFolderType, kOnSystemDisk)
{
    TempName(basename);
}
GUSITempFileSpec::GUSITempFileSpec(short vRefNum, ConstStr31Param basename)
    : GUSIFileSpec(kTemporaryFolderType, vRefNum)
{
    TempName(basename);
}
GUSITempFileSpec::GUSITempFileSpec(short vRefNum, long parID, ConstStr31Param basename)
{
    SetVRef(vRefNum);
    SetParID(parID);

    TempName(basename);
}
}
```

The names are searched by TempName. To avoid excessive searching, each search starts at a new location.

(Member functions for class GUSIFileSpec 1155) +≡ (1126) ◀1200 1202▶

```
int GUSITempFileSpec::sID = 0;

void GUSITempFileSpec::TempName()
{
    for (;;) {
        Str31    name;

        sprintf((char *)name, (char *)"\ptmp%04d", sID++);
        SetName(name);

        sID        %= 10000;
        fValidInfo  = false;
        if (!Exists()) {
            if (fError == fnfErr)
                fError = noErr;
            return;
        }
    }
}
```

The search with an existing base name is a slight variation on the above.

```

<Member functions for class GUSIFileSpec 1155>+≡ (1126) <1201
void GUSITempFileSpec::TempName(ConstStr31Param basename)
{
    for (int id = 0; ++id) {
        Str32 name;
        int len = 2;

        if (id < 10)
            ;
        else if (id < 100)
            len = 3;
        else if (id < 1000)
            len = 4;
        else if (id < 10000)
            len = 5;
        else {
            fError = fnfErr;
            return;
        }

        <Insert id at the appropriate place into basename 1203>
        SetName(name);

        fValidInfo = false;
        if (!Exists()) {
            if (fError == fnfErr)
                fError = noErr;
            return;
        }
    }
}

```

I think it looks a bit prettier if the number is stuck before the first dot, instead of at the end.

```

<Insert id at the appropriate place into basename 1203>≡ (1202)
if (!id) {
    memcpy(name, basename, *basename+1);
} else if (*basename + len > 31) {
    name[0] = 30;
    memcpy(name+1, basename+1, 31-len);
    sprintf(reinterpret_cast<char *>(name+32-len), "%d", id);
} else {
    name[0] = *basename + len;
    unsigned char * period = reinterpret_cast<unsigned char *>(memchr(basename+1, '.', *basename));
    int prelen = period ? (period-basename)-1 : *basename;
    memcpy(name+1, basename+1, prelen);
    sprintf(reinterpret_cast<char *>(name+1+prelen), ".%d", id);
    memcpy(name+1+prelen+len, period, *basename-prelen);
}

```

Chapter 31

Disk files

A `GUSIMacFileSocket` implements the operations on mac files. All instances of `GUSIMacFileSocket` are created by the `GUSIMacFileDevice` singleton, so there is no point in exporting the class itself.

A `GUSIMacDirectory` implements directory handles on mac directories.

```
<GUSIMacFile.h 1204>≡
#ifdef _GUSIMacFile_
#define _GUSIMacFile_

#ifdef GUSI_INTERNAL

#include "GUSIDevice.h"

<Definition of class GUSIMacFileDevice 1206>

#endif /* GUSI_INTERNAL */

#endif /* _GUSIMacFile_ */
```

```

<GUSIMacFile.cp 1205>≡
#define GUSI_MESSAGE_LEVEL 5

#include "GUSIInternal.h"
#include "GUSIMacFile.h"
#include "GUSIFSWrappers.h"
#include "GUSISocketMixins.h"
#include "GUSIBasics.h"
#include "GUSIDdiag.h"
#include "GUSIConfig.h"

#include <fcntl.h>
#include <sys/stat.h>
#include <stddef.h>
#include <unistd.h>
#include <algorithm>
#include <memory>

#include <Aliases.h>
#include <Devices.h>
#include <TextUtils.h>
#include <Resources.h>

<Prototypes for GUSIMacFileSocket interrupt level routines 1308>
<Definition of class GUSIMacFileSocket 1207>
<Definition of class GUSIMacDirectory 1208>
<Interrupt level routines for GUSIMacFileSocket 1314>
<Interrupt routine wrappers for GUSIMacFileSocket 1310>
<Member functions for class GUSIMacFileDevice 1209>
<Member functions for class GUSIMacFileSocket 1267>
<Member functions for class GUSIMacDirectory 1323>

```


31.1 Definition of GUSIMacFileDevice

GUSIMacFileDevice is a singleton subclass of GUSIDevice.

```
(Definition of class GUSIMacFileDevice 1206)≡ (1204)
class GUSIMacFileDevice : public GUSIDevice {
public:
    static GUSIMacFileDevice * Instance();
    virtual bool Want(GUSIFileToken & file);

    (Overridden member functions for GUSIMacFileDevice 1213)

    GUSISocket * open(short fileRef, int flags);

    int MarkTemporary(const FSSpec & file);
    void CleanupTemporaries(bool giveup);

    ~GUSIMacFileDevice();
protected:
    GUSIMacFileDevice() : fTemporaries(0) {}

    (Privatissima of GUSIMacFileDevice 1223)
};
```

31.2 Definition of GUSIMacFileSocket

```
(Definition of class GUSIMacFileSocket 1207)≡ (1205)
class GUSIMacFileSocket :
    public GUSISocket,
    protected GUSISMAsyncError,
    protected GUSISMBlocking,
    protected GUSISMState,
    protected GUSISMInputBuffer,
    protected GUSISMOutputBuffer
{
public:
    GUSIMacFileSocket(short fileRef, bool append, int mode);
    ~GUSIMacFileSocket();
    (Overridden member functions for GUSIMacFileSocket 1279)
private:
    (Privatissima of GUSIMacFileSocket 1266)
};
```

31.3 Definition of GUSIMacDirectory

```
<Definition of class GUSIMacDirectory 1208>≡ (1205)
class GUSIMacDirectory : public GUSIDirectory {
public:
    GUSIMacDirectory(const FSSpec & spec);

    <Overridden member functions for GUSIMacDirectory 1324>
private:
    <Privatissima of GUSIMacDirectory 1322>
};
```

31.4 Implementation of GUSIMacFileDevice

```
<Member functions for class GUSIMacFileDevice 1209>≡ (1205) 1210»
static auto_ptr<GUSIMacFileDevice> sGUSIMacFileDevice;

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) «1209 1211»
GUSIMacFileDevice * GUSIMacFileDevice::Instance()
{
    if (!sGUSIMacFileDevice.get())
        sGUSIMacFileDevice = auto_ptr<GUSIMacFileDevice>(new GUSIMacFileDevice);
    return sGUSIMacFileDevice.get();
}

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) «1210 1212»
GUSIMacFileDevice::~~GUSIMacFileDevice()
{
    CleanupTemporaries(true);
}

GUSIMacFileDevice will handle all file requests thrown at it, but no device re-
quests.

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) «1211 1214»
bool GUSIMacFileDevice::Want(GUSIFileToken & file)
{
    return file.IsFile();
}

<Overridden member functions for GUSIMacFileDevice 1213>≡ (1206) 1221»
virtual GUSISocket * open(GUSIFileToken & file, int flags);
```

```

⟨Member functions for class GUSIMacFileDevice 1209⟩+≡ (1205) ◀1212 1220▶
GUSISocket * GUSIMacFileDevice::open(GUSIFileToken & file, int flags)
{
    OSErr          err;
    char           permission;
    short          fileRef;

    ⟨Translate flags into permission 1215⟩
tryOpen:
    ⟨Attempt Open into fileRef 1216⟩
    ⟨If file didn't exist, try to create it and goto tryOpen if successful 1217⟩
    ⟨Handle errors from Open 1218⟩
    ⟨Handle O_EXCL and O_TRUNC 1219⟩

    return open(fileRef, flags);
}

```

First, we need to translate the POSIX open flags into a Mac file permission.

```

⟨Translate flags into permission 1215⟩≡ (1214)
switch (flags & 3) {
case O_RDONLY:
    permission = fsRdPerm;
    break;
case O_WRONLY:
case O_RDWR:
    permission =
        GUSIConfiguration::Instance()->fSharedOpen ? fsRdWrShPerm : fsRdWrPerm;
    break;
}

```

Now, we attempt to open the data or resource fork of the file.

```

⟨Attempt Open into fileRef 1216⟩≡ (1214)
if (flags & O_RSRC)
    err = GUSIFSOpenRF(&file, permission, &fileRef);
else
    err = GUSIFSOpenDF(&file, permission, &fileRef);

```

If the file did not exist, we try to create it. We don't need to worry about special casing for O_RSRC here, I think.

```

⟨If file didn't exist, try to create it and goto tryOpen if successful 1217⟩≡ (1214)
if (err == fnfErr && (flags & O_CREAT)) {
    if (!(err = GUSIFSCreate(&file))) {
        GUSIConfiguration::Instance()->SetDefaultFType(file);
        file.TouchFolder();
    }
    if (!err || err == dupFNErr) {
        flags &= ~(O_CREAT | O_EXCL); // Avoid loop or false positive
        goto tryOpen;                // Retry
    }
}

```

Now we check whether the Open ultimately succeeded.

```

<Handle errors from Open 1218>≡ (1214)
switch (err) {
case afpObjectTypeErr:
    return GUSISetPosixError(EISDIR), static_cast<GUSIMacFileSocket *>(nil);
default:
    return GUSISetMacError(err), static_cast<GUSIMacFileSocket *>(nil);
case noErr:
    break;
}

```

If we got to this point with O_EXCL still set, the file must have existed, but it shouldn't have. If O_TRUNC is set, truncate the file.

```

<Handle O_EXCL and O_TRUNC 1219>≡ (1214)
if (flags & O_EXCL) {
    FSClose(fileRef);
    return GUSISetPosixError(EEXIST), static_cast<GUSIMacFileSocket *>(nil);
}

if (flags & O_TRUNC)
    SetEOF(fileRef, 0);

```

There is also a special case backdoor open, available only for GUSIMacFileDevice to handle files which are already open somehow. This open will never fail except for lack of memory.

```

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) <1214 1222>
GUSISocket * GUSIMacFileDevice::open(short fileRef, int flags)
{
    GUSISocket * sock;

    if (!(sock =
        new GUSIMacFileSocket(fileRef, (flags & O_APPEND) != 0, flags & 3)
    )) {
        FSClose(fileRef);
        GUSISetPosixError(ENOMEM);
    }

    return sock;
}

```

The normal case of remove is straightforward, but we also want to support the removing of open files, which is frequently used in POSIX code, as much as possible.

```

<Overridden member functions for GUSIMacFileDevice 1213>+≡ (1206) <1213 1226>
virtual int remove(GUSIFileToken & file);

```

```

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) <1220 1224>
int GUSIMacFileDevice::remove(GUSIFFileToken & file)
{
    OSErr err;

    switch (err = GUSIFSDDelete(&file)) {
    case noErr:
        return 0;
    case fBsyErr:
        return MarkTemporary(static_cast<const FSSpec &>(file));
    default:
        return GUSISetMacError(err);
    }
}

```

MarkTemporary moves the file to the temporary folder and puts it on a list of death candidates.

```

<Privatissima of GUSIMacFileDevice 1223>≡ (1206)
struct TempQueue {
    TempQueue * fNext;
    FSSpec      fSpec;
} * fTemporaries;

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) <1222 1225>
int GUSIMacFileDevice::MarkTemporary(const FSSpec & file)
{
    OSErr          err;
    GUSITempFileSpec temp(file.vRefNum, file.name);

    if (err = GUSIFSMoveRename(&file, &temp))
        return GUSISetMacError(err);
    TempQueue * q = new TempQueue;
    q->fNext      = fTemporaries;
    q->fSpec       = static_cast<const FSSpec &>(temp);
    fTemporaries  = q;

    return 0;
}

```

CleanupTemporaries tries at promising times to delete as many of the temporary files as possible.

```

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) <1224 1227>
void GUSIMacFileDevice::CleanupTemporaries(bool giveup)
{
    TempQueue ** p = &fTemporaries;

    for (TempQueue * q = *p; q; q = *p)
        if (GUSIFSDelete(&q->fSpec) != fBsyErr || giveup) {
            // Delete entry
            *p = q->fNext;
            delete q;
        } else {
            // Keep entry
            p = &q->fNext;
        }
}

```

rename can be a surprisingly difficult operation.

```

<Overridden member functions for GUSIMacFileDevice 1213>+≡ (1206) <1221 1232>
virtual int rename(GUSIFFileToken & file, const char * newname);

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) <1225 1233>
int GUSIMacFileDevice::rename(GUSIFFileToken & file, const char * newname)
{
    OSErr          err;
    GUSIFFileSpec  newfile(newname, true);

    <Ensure that file to rename exists and new name resides on same volume 1228>
    bool  samename = EqualString(file->name, newfile->name, true, true);
    bool  samedir  = file->parID == newfile->parID;
    bool  newexists= newfile.Exists();
    <Handle special renaming cases by recursing 1229>
    <Move existing target out of the way 1230>
    if (samedir)
        err = GUSIFSRename(&file, newfile->name);
    else if (samename)
        err = GUSIFSCatMove(&file, newfile->parID);
    else
        err = GUSIFSMoveRename(&file, &newfile);

cleanuptarget:
    <Clean up existing target 1231>

    return GUSISetMacError(err);
}

```

We can only move a file around a volume.

```

<Ensure that file to rename exists and new name resides on same volume 1228>≡ (1227)
if (!file.Exists())
    return GUSISetMacError(file.Error());
if (file->vRefNum != newfile->vRefNum)
    return GUSISetPosixError(EXDEV);

```

If one of the two files is locked, or if the two names are distinct only by case, we call ourselves recursively, wrapped in the special case code. These conditions are rare, so some inefficiency is not all that tragic.

```

<Handle special renaming cases by recursing 1229>≡ (1227)
    if (samename && samedir)
        return 0;
    if (file.CatInfo()->Locked()) {
        GUSIFSRstFLock(&file);
        int res = rename(file, newname);
        GUSIFSSetFLock(res ? &file : &newfile);
        return res;
    }
    if (newexists && file.CatInfo()->Locked()) {
        GUSIFSRstFLock(&newfile);
        int res = rename(file, newname);
        if (res)
            GUSIFSSetFLock(&newfile);
        return res;
    }
    if (samedir && !samename && EqualString(file->name, newfile->name, false, true))
        newexists = false;

```

If the target name refers to an existing file, we stash it away until we're sure the rename succeeded.

```

<Move existing target out of the way 1230>≡ (1227)
    GUSIFileSpec target;
    if (newexists) {
        GUSITempFileSpec target(newfile->vRefNum, newfile->parID);
        if (err = GUSIFSRename(&newfile, target->name))
            return GUSISetMacError(err);
    }

```

```

<Clean up existing target 1231>≡ (1227)
    if (newexists)
        if (!err)
            GUSIFSDDelete(&target);
        else
            GUSIFSRename(&target, newfile->name);

```

stat is a rather intimidating function which needs to pull together information from various sources.

```

<Overridden member functions for GUSIMacFileDevice 1213>+≡ (1206) <1226 1236>
    virtual int stat(GUSIFileToken & file, struct stat * buf);

```

```

(Member functions for class GUSIMacFileDevice 1209) +≡ (1205) <1227 1237>
int GUSIMacFileDevice::stat(GUSIFileToken & file, struct stat * buf)
{
    if (file.IsDevice()) {
        (return an alibi stat buffer 1234)
    }
    const GUSICatInfo & cb = *file.CatInfo();
    GUSIIOPBWrapper<ParamBlockRec> pb;
    Str63 vName;

    pb->volumeParam.ioNamePtr = vName;
    pb->volumeParam.ioVRefNum = file->vRefNum;
    pb->volumeParam.ioVolIndex = 0;

    if (GUSIFSGGetVInfo(&pb))
        return GUSISetPosixError(ENOENT);

    buf->st_dev = pb->ioParam.ioVRefNum;
    buf->st_ino = cb.DirInfo().ioDrDirID;
    buf->st_nlink = 1;
    buf->st_uid = 0;
    buf->st_gid = 0;
    buf->st_rdev = 0;
    buf->st_atime = cb.FileInfo().ioFlMdBdat;
    buf->st_mtime = cb.FileInfo().ioFlMdBdat;
    buf->st_ctime = cb.FileInfo().ioFlCrDat;
    buf->st_blksize = pb->volumeParam.ioValBlkSiz;

    if (!cb.IsFile()) {
        (Determine the link count for a directory 1235)
        buf->st_mode = S_IFDIR | 0666;
        if (!gGUSIExecHook || gGUSIExecHook(&file))
            buf->st_mode |= 0111;

        buf->st_size = cb.DirInfo().ioDrNmFls;
    } else if (cb.IsAlias()) {
        buf->st_mode = S_IFLNK | 0777;
        buf->st_size = cb.FileInfo().ioFlRLgLen; /* Data fork is ignored */
    } else if (cb.FInfo().fdType == 'OCK') {
        buf->st_mode = S_IFSOCK | 0666;
        buf->st_size = cb.FileInfo().ioFlRLgLen; /* Data fork is ignored */
    } else {
        buf->st_mode = S_IFREG | 0666;
        if (cb.FileInfo().ioFlAttrib & 0x01)
            buf->st_mode &= ~0222;

        if (gGUSIExecHook) {
            if (gGUSIExecHook(&file))
                buf->st_mode |= 0111;
        } else
            switch (cb.FInfo().fdType) {
                case 'APPL':
                case 'appe':
                    buf->st_mode |= 0111;
            }
    }
}

```



```

    }

    buf->st_size    =    cb.FileInfo().ioFlLgLen;        /* Resource fork is ignored */
}

buf->st_blocks    =    (buf->st_size + 511) >> 9;

return 0;
}

```

Any existing device should return an OK result, so give just a set of defaults.

```

⟨return an alibi stat buffer 1234⟩≡ (1233)
buf->st_dev        =    'dev ' ;
buf->st_ino        =    '????' ;
buf->st_mode       =    S_IFCHR | 0666 ;
buf->st_nlink      =    1;
buf->st_uid        =    0;
buf->st_gid        =    0;
buf->st_rdev       =    0;
buf->st_size       =    1;
buf->st_atime      =    time(NULL);
buf->st_mtime      =    time(NULL);
buf->st_ctime      =    time(NULL);
buf->st_blksize    =    0;
buf->st_blocks     =    1;

return 0;

```

Depending on our preference settings, we employ a faster or a slower method in determining the link count. The *accurate* method sets the link count to 2 plus the number of subdirectories, while the fast method establishes an upper boundary on that number as 2 plus the number of subitems.

```

⟨Determine the link count for a directory 1235⟩≡ (1233)
++buf->st_nlink;
if (GUSIConfiguration::Instance()->fAccurateStat) {
    GUSIFileSpec    spec;

    spec.SetVRef(pb->ioParam.ioVRefNum);
    spec.SetParID(cb.DirInfo().ioDrDirID);
    for (int i = 0; i++ < cb.DirInfo().ioDrNmFls; i) {
        spec    =    spec[i];
        if (!spec.Error() && spec.CatInfo() && !spec.CatInfo()->IsFile())
            ++buf->st_nlink;
    }
} else {
    buf->st_nlink    +=    cb.DirInfo().ioDrNmFls;
}

```

```

⟨Overridden member functions for GUSIMacFileDevice 1213⟩+≡ (1206) ◀1232 1238▶
virtual int chmod(GUSIFileToken & file, mode_t mode);

```

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) <1233 1239>

```
int GUSIMacFileDevice::chmod(GUSIFFileToken & file, mode_t mode)
{
    if (file.IsDevice())
        return GUSISetPosixError(EINVAL);
    if (!file.Exists())
        return GUSISetPosixError(ENOENT);
    if (file.IsFile())
        if (mode & S_IWUSR) {
            if (file.CatInfo()->Locked())
                GUSIFSRstFLock(&file);
        } else {
            if (!file.CatInfo()->Locked())
                GUSIFSSetFLock(&file);
        }

    return 0;
}
```

<Overridden member functions for GUSIMacFileDevice 1213>+≡ (1206) <1236 1240>

```
virtual int utime(GUSIFFileToken & file, const utimbuf * times);
```

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) <1237 1241>

```
int GUSIMacFileDevice::utime(GUSIFFileToken & file, const utimbuf * times)
{
    if (file.IsDevice())
        return GUSISetPosixError(EINVAL);
    if (!file.Exists())
        return GUSISetPosixError(ENOENT);
    GUSIIOPBWrapper<GUSICatInfo> info;
    info.fPB = *file.CatInfo();
    info->FileInfo().ioDirID = file->parID;
    if (times)
        info->FileInfo().ioFlMdDat = times->modtime;
    return GUSISetMacError(GUSIFSSetCatInfo(&info));
}
```

<Overridden member functions for GUSIMacFileDevice 1213>+≡ (1206) <1238 1242>

```
virtual int access(GUSIFFileToken & file, int mode);
```

```

{Member functions for class GUSIMacFileDevice 1209}+≡ (1205) ◀1239 1243▶
int GUSIMacFileDevice::access(GUSIFileToken & file, int mode)
{
    if (file.IsDevice())
        return GUSISetPosixError(EINVAL);

    const GUSICatInfo * info = file.CatInfo();

    if (!info)
    {
        switch (file.Error()) {
            case afpAccessDenied:
                return GUSISetPosixError(EACCES);
            case ioErr:
                return GUSISetPosixError(EIO);
            case bdNamErr:
            case fnfErr:
            case nsvErr:
            case paramErr:
            case dirNFErr:
            case afpObjectTypeErr:
                /* ENOENT */
            default:
                return GUSISetPosixError(ENOENT); /* Don't know what the error is. */
        }
    }

    //
    // Under our simplifying assumptions, we don't check AppleShare permissions,
    // so if we managed to do a GetCatInfo, R_OK and F_OK are already assumed
    // to be true
    //
    if (mode & W_OK) {
        //
        // Check if volume is locked
        //
        GUSIIOPBWrapper<HParamBlockRec> vol;

        vol->volumeParam.ioNamePtr = nil;
        vol->volumeParam.ioVolIndex = 0;
        vol->volumeParam.ioVRefNum = file->vRefNum;

        if (GUSIFSHGetVInfo(&vol))
            return GUSISetPosixError(EINVAL); // Should never happen

        if (vol->volumeParam.ioVAttrb & 0x8080)
            return GUSISetPosixError(EROFS); // Yup, volume is locked

        //
        // Check if file is locked
        //
        if (info->Locked())
            return GUSISetPosixError(EACCES); // Yup, file is locked
    }

    if (mode & X_OK)

```

```

        if (gGUSIExecHook) {
            if (!gGUSIExecHook(&file))
                return GUSISetPosixError(EACCES);
        } else
            switch (info->FInfo().fdType) {
                case 'APPL':
                case 'appe':
                    break;
                default:
                    return GUSISetPosixError(EACCES);
            }

        return 0;
    }

    <Overridden member functions for GUSIMacFileDevice 1213>+≡      (1206) <1240 1244>
        virtual int mkdir(GUSIFileToken & file);

    <Member functions for class GUSIMacFileDevice 1209>+≡          (1205) <1241 1245>
        int GUSIMacFileDevice::mkdir(GUSIFileToken & file)
        {
            return GUSISetMacError(GUSIFSDirCreate(&file));
        }

    <Overridden member functions for GUSIMacFileDevice 1213>+≡      (1206) <1242 1246>
        virtual int rmdir(GUSIFileToken & file);

    <Member functions for class GUSIMacFileDevice 1209>+≡          (1205) <1243 1247>
        int GUSIMacFileDevice::rmdir(GUSIFileToken & file)
        {
            return GUSISetMacError(GUSIFSDelete(&file));
        }

    <Overridden member functions for GUSIMacFileDevice 1213>+≡      (1206) <1244 1248>
        virtual GUSIDirectory * opendir(GUSIFileToken & file);

    <Member functions for class GUSIMacFileDevice 1209>+≡          (1205) <1245 1249>
        GUSIDirectory * GUSIMacFileDevice::opendir(GUSIFileToken & file)
        {
            return new GUSIMacDirectory(static_cast<FSSpec &>(++file));
        }

    symlink has to reproduce the Finder's alias creation process, which is quite complex.

    <Overridden member functions for GUSIMacFileDevice 1213>+≡      (1206) <1246 1259>
        virtual int symlink(GUSIFileToken & to, const char * newlink);

```

VRef2Icon finds the driver icon for a volume.

```

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) ◀1247 1252▶
static OSErr VRef2Icon(short vRef, Handle * icon)
{
    OSErr    err;
    short    cRef;

    {
        ◊Get volume information into vRef and cRef 1250◊
    }
    {
        ◊Get volume icon 1251◊
    }

    return noErr;
}

◊Get volume information into vRef and cRef 1250◊≡ (1249)
    GUSIIOPBWrapper<HParamBlockRec> hpb;
    hpb->volumeParam.ioVRefNum = vRef;
    hpb->volumeParam.ioNamePtr = nil;
    hpb->volumeParam.ioVolIndex = 0;
    if (err = GUSIFSHGetVInfo(&hpb))
        return err;
    vRef = hpb->volumeParam.ioVDrvInfo;
    cRef = hpb->volumeParam.ioVDRefNum;

◊Get volume icon 1251◊≡ (1249)
    GUSIIOPBWrapper<ParamBlockRec> pb;
    pb->cntrlParam.ioVRefNum = vRef;
    pb->cntrlParam.ioCRefNum = cRef;
    pb->cntrlParam.csCode = 21;

    if (err = pb.Control())
        return err;

    PtrToHand(*(Ptr *) pb->cntrlParam.csParam, icon, 256);

```

AliasTypeExpert finds the right file type, creator, icon location and icon ID for the alias. This is complicated by many special cases.

```

(1205) <1249 1256>
(1209) +≡
Member functions for class GUSIMacFileDevice
typedef OSType AliasTypeMap[2];

static AliasTypeMap sMap[] = {
    <Special folder types and their alias types 1254>
    { 0, 0 }
};

static void AliasTypeExpert(
    const GUSIFileSpec & file,
    OSType * fCreator,
    OSType * fType,
    GUSIFileSpec * iconFile,
    short * iconID)
{
    const GUSICatInfo * info = file.CatInfo();

    *fCreator = 'MACS';
    *iconFile = file;
    *iconID = kCustomIconResource;

    if (!info->IsFile()) {
        <Determine fType for folder and volume aliases 1253>
        if (file->parID == fsRtParID) {
            iconFile->SetParID(fsRtDirID);
            iconFile->SetName("\pIcon\n");
        } else {
            *iconFile += "\pIcon\n";
        }
    } else {
        *fType = info->FInfo().fdType;
        *fCreator = info->FInfo().fdCreator;
    }

    return;
error:
    *fType = 0;
    *fCreator = 0;
}

```

One of the complex issues is the various AppleShare related aliases, but those should not change too much.

⟨Determine fType for folder and volume aliases 1253⟩≡ (1252)
 GetVolParmsInfoBuffer volParms;

```

if (!GUSIFSGetVolParms(file->vRefNum, &volParms) && volParms.vMServerAdr)
  if (file->parID == fsRtParID)
    *fType = 'srvr';
  else if (!info->HasRdPerm())
    *fType = 'fadr';
  else
    *fType = 'faet';
else if (file->parID == fsRtParID)
  *fType = 'hdsk';
else if (info->DirIsMounted())
  *fType = 'famn';
else if (info->DirIsExported())
  *fType = 'fash';
else if (info->DirIsShared())
  *fType = 'faet';
else
  *fType = 'fdrp';

```

⟨Check for special folder 1255⟩

This is the current list of special folders. I suppose it should be revised occasionally. In fact, it is probably already out of date.

⟨Special folder types and their alias types 1254⟩≡ (1252)

```

{ 'amnu', 'faam' },
{ 'ctrl', 'fact' },
{ 'extn', 'faex' },
{ 'pref', 'fapf' },
{ 'prnt', 'fapn' },
{ 'empt', 'trsh' },
{ 'trsh', 'trsh' },
{ 'strt', 'fast' },
{ 'macs', 'fasy' },

```

⟨Check for special folder 1255⟩≡ (1253)

```

if (file->parID != fsRtParID && info->DirInfo().ioDrDirID < 9) {
  short vRef;
  long dirID;

  for (AliasTypeMap * mapp = sMap; **mapp; ++mapp)
    if (FindFolder(file->vRefNum, (*mapp)[0], false, &vRef, &dirID))
      continue;
    else if (dirID == info->DirInfo().ioDrDirID) {
      *fType = (*mapp)[1];

      break;
    }
}

```

AddIconsToFile adds the appropriate icons to the alias file as the custom icon.
The mechanical part of the work is performed by CopyIconFamily

```

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) <1252 1257>
static OSType sIconTypes[] = {
    'ICN#',
    'ics#',
    'icl4',
    'ics4',
    'icl8',
    'ics8',
    0
};

static bool CopyIconFamily(
    short srcResFile, short srcID, short dstResFile, short dstID)
{
    Handle icon;
    bool success = false;

    for (OSType * types = sIconTypes; *types; ++types) {
        UseResFile(srcResFile);
        if (icon = Get1Resource(*types, srcID)) {
            UseResFile(dstResFile);
            DetachResource(icon);
            AddResource(icon, *types, dstID, "\p");

            success = success || !ResError();
        }
    }

    return success;
}

static bool AddIconsToFile(
    const GUSIFileSpec & origFile, short aliasFile,
    const GUSIFileSpec & iconFile, short iconID,
    bool plainDisk)
{
    bool hasCustomIcons = false;
    short iFile = FSpOpenResFile(&iconFile, fsRdPerm);

    if (iFile != -1) {
        hasCustomIcons =
            CopyIconFamily(iFile, iconID, aliasFile, kCustomIconResource);
        CloseResFile(iFile);
    }
    if (!hasCustomIcons && plainDisk) {
        Handle icon;
        if (!VRef2Icon(origFile->vRefNum, &icon)) {
            AddResource(icon, 'ICN#', kCustomIconResource, "\p");

            hasCustomIcons = !ResError();
        }
    }
}

```



```

        return hasCustomIcons;
    }

    (Member functions for class GUSIMacFileDevice 1209)+≡ (1205) <1256 1260>
    int GUSIMacFileDevice::symlink(GUSIFFileToken & to, const char * newlink)
    {
        if (!to.Exists())
            return GUSISetPosixError(EIO);
        if (to.IsDevice())
            return GUSISetPosixError(EINVAL);

        OSType        fileType;
        OSType        fileCreator;
        short          iconID;
        short          aliasFile;
        AliasHandle    alias;
        bool           customIcon;
        GUSIFFileSpec  iconFile;
        FInfo          info;

        (Create and open aliasFile 1258)

        if (NewAlias(nil, &to, &alias))
            goto closeFile;

        AddResource((Handle) alias, 'alis', 0, to->name);
        if (ResError())
            goto deleteAlias;

        customIcon = AddIconsToFile(to, aliasFile, iconFile, iconID,
            fileType == 'hdsk' && fileCreator == 'MACS');

        CloseResFile(aliasFile);

        GUSIFGetFInfo(&newnm, &info);
        info.fdFlags |= kIsAlias | (customIcon ? kHasCustomIcon : 0);
        info.fdFlags &= ~kHasBeenInitd;
        GUSIFSetFInfo(&newnm, &info);
        newnm.TouchFolder();

        return 0;

    deleteAlias:
        DisposeHandle((Handle) alias);
    closeFile:
        CloseResFile(aliasFile);
    deleteFile:
        GUSIFDelete(&newnm);

        return GUSISetPosixError(EIO);
    }

```

```

<Create and open aliasFile 1258>≡ (1257)
    GUSIFileSpec    newnm(newlink, true);

    if (newnm.Error())
        return GUSISetPosixError(EIO);
    if (newnm.Exists())
        return GUSISetPosixError(EEXIST);

    AliasTypeExpert(to, &fileCreator, &fileType, &iconFile, &iconID);
    FSpCreateResFile(&newnm, fileCreator, fileType, smSystemScript);

    if (ResError())
        return GUSISetPosixError(EIO);

    aliasFile = FSpOpenResFile(&newnm, fsRdWrPerm);
    if (aliasFile == -1)
        goto deleteFile;

<Overridden member functions for GUSIMacFileDevice 1213>+≡ (1206) <1248 1261>
    virtual int readlink(GUSIFileToken & link, char * buf, int bufsize);

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) <1257 1262>
    int GUSIMacFileDevice::readlink(GUSIFileToken & link, char * buf, int bufsize)
    {
        if (link.IsDevice())
            return GUSISetPosixError(EINVAL);

        char * resPath = link.AliasPath();

        if (!(resPath))
            if (link.Error() == resFNotFound)
                return GUSISetPosixError(EINVAL);
            else
                return GUSISetMacError(link.Error());

        int len = strlen(resPath);
        strncpy(buf, resPath, bufsize);

        if (len >= bufsize)
            return GUSISetPosixError(ENAMETOOLONG);
        else
            return len;
    }

<Overridden member functions for GUSIMacFileDevice 1213>+≡ (1206) <1259 1264>
    virtual int fgetfileinfo(GUSIFileToken & file, OSType * creator, OSType * type);
    virtual int fsetfileinfo(GUSIFileToken & file, OSType creator, OSType type);

```

```

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) <1260 1263>
int GUSIMacFileDevice::fgetfileinfo(GUSIFFileToken & file, OSType * creator, OSType * type)
{
    FInfo    info;

    if (file.IsDevice())
        return GUSISetPosixError(EINVAL);
    if (file.Error() || !file.Exists() || GUSIFSGetFInfo(&file, &info))
        return GUSISetPosixError(EIO);
    if (creator)
        *creator    = info.fdCreator;
    if (type)
        *type       = info.fdType;

    return 0;
}

```

```

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) <1262 1265>
int GUSIMacFileDevice::fsetfileinfo(GUSIFFileToken & file, OSType creator, OSType type)
{
    FInfo    info;

    if (file.IsDevice())
        return GUSISetPosixError(EINVAL);
    if (file.Error() || !file.Exists() || GUSIFSGetFInfo(&file, &info))
        return GUSISetPosixError(EIO);

    info.fdType    = type;
    info.fdCreator = creator;
    info.fdFlags   &= ~kHasBeenInitied;

    if (GUSIFSSetFInfo(&file, &info))
        return GUSISetPosixError(EIO);
    return 0;
}

```

faccess is a somewhat curious case in that GUSIMacFileDevice accepts responsibility for handling it, but then does not, in fact, handle it.

```

<Overridden member functions for GUSIMacFileDevice 1213>+≡ (1206) <1261
virtual int faccess(GUSIFFileToken & file, unsigned * cmd, void * arg);

```

```

<Member functions for class GUSIMacFileDevice 1209>+≡ (1205) <1263
int GUSIMacFileDevice::faccess(GUSIFFileToken &, unsigned *, void *)
{
    return GUSISetPosixError(EINVAL);
}

```

31.5 Implementation of GUSIMacFileSocket

The implementation of GUSIMacFileSocket consists of a synchronous high level part which mostly deals with GUSIRingBuffers and an asynchronous low level part. Since file sockets have some different properties than other sockets, it is helpful to start with the high level routines.

31.5.1 High level interface for GUSIMacFileSocket

The constructor has to initialize a rather large number of data fields.

```
(Privatissima of GUSIMacFileSocket 1266)≡ (1207) 1269»
    short    fFileRef;
    bool     fAppend;
    bool     fWriteBehind;

(Member functions for class GUSIMacFileSocket 1267)≡ (1205) 1268»
GUSIMacFileSocket::GUSIMacFileSocket(short fileRef, bool append, int mode)
    : fFileRef(fileRef), fAppend(append), fWriteBehind(false)
{
    switch (mode) {
    case O_RDONLY:
        fWriteShutdown = true;
        fWritePB.ioParam.ioResult = wrPermErr;
        break;
    case O_WRONLY:
        fWriteBehind = true;
        // Fall through
    case O_RDWR:
        fReadShutdown = true;
        break;
    }
    (Initialize fields of GUSIMacFileSocket 1270)
}

(Member functions for class GUSIMacFileSocket 1267)+≡ (1205) «1267 1272»
GUSIMacFileSocket::~~GUSIMacFileSocket()
{
    fsync();
    if (fFileRef)
        FSClose(fFileRef);
    GUSIMacFileDevice::Instance()->CleanupTemporaries(false);
}
```

Under some circumstances, read uses a readahead buffer. Since this optimization can actually pessimize applications, we're very conservative by only turning it on for readonly files and by turning it off as soon as lseek is called. SyncRead puts the socket into a predictable state afterwards.

fPosition contains the next position where operations apply, not counting read-ahead data, but counting write-behind. fReadAhead indicates that a read-ahead should be started on the first read operation. fBeyondEOF indicates that fPosition resulted from a seek beyond the current end of file position.

```
(Privatissima of GUSIMacFileSocket 1266)+≡ (1207) «1266 1271»
    bool     fReadAhead;
    bool     fBeyondEOF;
    long     fPosition;

(Initialize fields of GUSIMacFileSocket 1270)≡ (1267) 1278»
    fPosition = 0;
    fReadAhead = !fReadShutdown;
    fBeyondEOF = false;
```

<Privatissima of GUSIMacFileSocket 1266>+≡ (1207) ◀1269 1275▶
void SyncRead();

<Member functions for class GUSIMacFileSocket 1267>+≡ (1205) ◀1268 1276▶
void GUSIMacFileSocket::SyncRead()
{
 fReadAhead = false;
 fReadShutdown = true;
 <Wait for pending read operations on GUSIMacFileSocket to complete 1273>
 <Discard data in input buffer of GUSIMacFileSocket 1274>
}

As soon as fReadShutdown is set, no new reads will be started, so just wait for the current ones to complete.

<Wait for pending read operations on GUSIMacFileSocket to complete 1273>≡ (1272 1282)
if (fReadPB.ioParam.ioResult == 1) {
 AddContext();
 while (fReadPB.ioParam.ioResult == 1)
 GUSIContext::Yield(true);
 RemoveContext();
}

If there was data in the buffer, we need to move the file position back by the amount of data.

<Discard data in input buffer of GUSIMacFileSocket 1274>≡ (1272)
fPosition += fInputBuffer.Valid();
size_t consume = 0x7F000000;
fInputBuffer.Consume(nil, consume);

As opposed to SyncRead, SyncWrite doesn't change the long-term strategy. By temporarily turning off fWriteBehind, it signals that the write strategy should write the remaining data even if it does not align to cache blocks.

<Privatissima of GUSIMacFileSocket 1266>+≡ (1207) ◀1271 1277▶
void SyncWrite();

<Member functions for class GUSIMacFileSocket 1267>+≡ (1205) ◀1272 1280▶
void GUSIMacFileSocket::SyncWrite()
{
 fOutputBuffer.Lock();
 bool saveWriteBehind = fWriteBehind;
 fWriteBehind = false;
 fOutputBuffer.Release();
 AddContext();
 while (!fWriteShutdown && fOutputBuffer.Valid())
 GUSIContext::Yield(true);
 RemoveContext();
 fOutputBuffer.Lock();
 fWriteBehind = saveWriteBehind;
 fOutputBuffer.Release();
}

It is often advantageous to keep I/O requests a multiple of the allocation block size.

<Privatissima of GUSIMacFileSocket 1266>+≡ (1207) ◀1275 1309▶
size_t fBlockSize;

```

<Initialize fields of GUSIMacFileSocket 1270>+≡ (1267) <1270 1312>
    fBlockSize = 4096;

    GUSIIOPBWrapper<FCBPBRec> fcb;
    Str63 name;

    fcb->ioNamePtr = name;
    fcb->ioRefNum = fFileRef;
    fcb->ioFCBIndx = 0;
    if (!GUSIFSGetFCBInfo(&fcb)) {
        GUSIIOPBWrapper<ParamBlockRec> info;
        info->volumeParam.ioNamePtr = name;
        info->volumeParam.ioVRefNum = fcb->ioVRefNum;
        info->volumeParam.ioVolIndex= 0;

        if (!GUSIFSGetVInfo(&info))
            fBlockSize = info->volumeParam.ioValBlkSiz;
    }

<Overridden member functions for GUSIMacFileSocket 1279>≡ (1207) 1284>
    virtual ssize_t read(const GUSIScatterer & buffer);

<Member functions for class GUSIMacFileSocket 1267>+≡ (1205) <1276 1285>
    ssize_t GUSIMacFileSocket::read(const GUSIScatterer & buffer)
    {
        size_t rest = buffer.Length();
        size_t offset = 0;

        GUSI_MESSAGE(("pos: %d len: %d\n", fPosition, rest));

        SyncWrite();
        if (fBeyondEOF)
            if (GUSISetPosixError(GetAsyncError()))
                return -1;
            else
                return 0;

        <Read from read-ahead buffer of GUSIMacFileSocket while possible 1281>
        <Read directly from file into buffer if necessary 1282>
        size_t done = buffer.Length() - rest;
        <Start read-ahead if necessary 1283>

        return int(done);
    }

```

As long as there has been no fReadShutdown, we use the read-ahead buffer.

```
<Read from read-ahead buffer of GUSIMacFileSocket while possible 1281>≡ (1280)
while (!fReadAhead && (!fReadShutdown || fInputBuffer.Valid()) && rest) {
    if (fInputBuffer.Valid()) {
        size_t len = rest;
        fInputBuffer.Lock();
        fInputBuffer.Consume(buffer, len, offset);
        fPosition += len;
        fInputBuffer.Release();
        rest -= len;
    } else {
        if (GUSISetPosixError(GetAsyncError()))
            return -1;
        if (!fBlocking)
            if (!offset) // Nothing read yet
                return GUSISetPosixError(EWOULDBLOCK);
            else
                return offset;
        AddContext();
        while (!fReadShutdown && !fInputBuffer.Valid())
            GUSIContext::Yield(true);
        RemoveContext();
    }
}
```

If there is still data to be read, read it now.

<Read directly from file into buffer if necessary 1282>≡ (1280)

```

if (!rest)
    return buffer.Length();
do {
    size_t len = rest;
    fReadPB.ioParam.ioCompletion = sWakeupProc;
    fReadPB.ioParam.ioBuffer =
        buffer.Count()==1 ?
            static_cast<Ptr>(buffer.Buffer()+offset
                : static_cast<Ptr>(fInputBuffer.ProduceBuffer(len));
    fReadPB.ioParam.ioReqCount = len;
    GUSI_MESSAGE(("Sync read (0x%x bytes).\n", len));
    PBReadAsync(&fReadPB);
    <Wait for pending read operations on GUSIMacFileSocket to complete 1273>
    GUSI_MESSAGE(("Read done (0x%x/%d).\n", fReadPB.ioParam.ioActCount, fReadPB.ioParam.ioResult));
    switch (fReadPB.ioParam.ioResult) {
    case eofErr:
    case noErr:
        rest -= fReadPB.ioParam.ioActCount;
        fPosition += fReadPB.ioParam.ioActCount;
        if (buffer.Count() == 1) {
            offset += fReadPB.ioParam.ioActCount;
        } else {
            fInputBuffer.ValidBuffer(fReadPB.ioParam.ioBuffer, fReadPB.ioParam.ioActCount);
            fInputBuffer.Consume(buffer, *reinterpret_cast<size_t *>(&fReadPB.ioParam.ioActCount),
        }
        if (rest && len == fReadPB.ioParam.ioActCount)
            continue;
        break;
    default:
        return GUSISetMacError(fReadPB.ioParam.ioResult);
    }
} while (0);

```

We wait for the first read (and handle it synchronously) before deciding that file access is sequential enough for read-ahead.

<Start read-ahead if necessary 1283>≡ (1280)

```

if (fReadAhead) {
    fReadAhead = false;
    fReadPB.ioParam.ioCompletion = sReadProc;
    GUSIMFRead(this);
}

```

To speed up writes, they go into a write-behind buffer and are flushed out asynchronously. This causes errors to be reported with some delay.

<Overridden member functions for GUSIMacFileSocket 1279>+≡ (1207) <1279 1288>

```

virtual ssize_t write(const GUSIGatherer & buffer);

```



```

<Member functions for class GUSIMacFileSocket 1267>+≡ (1205) <1280 1289>
ssize_t GUSIMacFileSocket::write(const GUSIGatherer & buffer)
{
    if (GUSISetPosixError(GetAsyncError()))
        return -1;
    if (fWriteShutdown) // Background error
        return GUSISetMacError(fWritePB.ioParam.ioResult);

    if (fAppend)
        lseek(0, SEEK_END);
    else if (fBeyondEOF) {
        <Insert null bytes beyond EOF 1286>
    }

    size_t rest = buffer.Length();
    size_t offset = 0;
    if (!fBlocking && fOutputBuffer.Free() < rest)
        return GUSISetPosixError(EWOULDBLOCK);
    while (rest) {
        size_t len = rest;
        <Wait for free buffer space on GUSIMacFileSocket 1287>
        fOutputBuffer.Lock();
        fOutputBuffer.Produce(buffer, len, offset);
        fPosition += len;
        fOutputBuffer.Release();
        rest -= len;
    }

    return buffer.Length();
}

```

If a seek operation went beyond EOF and we try to do a write the space in-between has to be filled with null bytes.

```

<Insert null bytes beyond EOF 1286>≡ (1285)
long targetPos = fPosition;

for (lseek(0, SEEK_END); fPosition < targetPos; ) {
    <Wait for free buffer space on GUSIMacFileSocket 1287>
    size_t size = targetPos - fPosition;
    Ptr p = static_cast<Ptr>(fOutputBuffer.ProduceBuffer(size));
    memset(p, 0, size);
    fPosition += size;
    fOutputBuffer.ValidBuffer(p, size);
}

<Wait for free buffer space on GUSIMacFileSocket 1287>≡ (1285 1286)
if (!fWriteShutdown && !fOutputBuffer.Free()) {
    AddContext();
    while (!fWriteShutdown && !fOutputBuffer.Free())
        GUSIContext::Yield(true);
    RemoveContext();
}
if (fWriteShutdown)
    return GUSISetMacError(fWritePB.ioParam.ioResult);

```

⟨Overridden member functions for GUSIMacFileSocket 1279⟩+≡ (1207) <1284 1290>
 virtual bool select(bool * canRead, bool * canWrite, bool * exception);

⟨Member functions for class GUSIMacFileSocket 1267⟩+≡ (1205) <1285 1291>
 bool GUSIMacFileSocket::select(bool * canRead, bool * canWrite, bool *)
 {
 bool cond = false;
 if (canRead)
 if (*canRead = fBeyondEOF || fAsyncError || fReadAhead || fReadShutdown || fInputBuffer.Val
 cond = true;
 if (canWrite)
 if (*canWrite = fBeyondEOF || fAsyncError || fWriteShutdown || fOutputBuffer.Free()
 cond = true;

 return cond;
 }

⟨Overridden member functions for GUSIMacFileSocket 1279⟩+≡ (1207) <1288 1292>
 virtual int fsync();

⟨Member functions for class GUSIMacFileSocket 1267⟩+≡ (1205) <1289 1293>
 int GUSIMacFileSocket::fsync()
 {
 SyncWrite();
 if (fWriteShutdown)
 return GUSISetMacError(fWritePB.ioParam.ioResult);
 else
 return 0;
 }

fcntl() handles the blocking support.

⟨Overridden member functions for GUSIMacFileSocket 1279⟩+≡ (1207) <1290 1294>
 virtual int fcntl(int cmd, va_list arg);

⟨Member functions for class GUSIMacFileSocket 1267⟩+≡ (1205) <1291 1295>
 int GUSIMacFileSocket::fcntl(int cmd, va_list arg)
 {
 int result;

 if (GUSISMBlocking::DoFcntl(&result, cmd, arg))
 return result;

 GUSI_ASSERT_CLIENT(false, ("fcntl: illegal request %d\n", cmd));

 return GUSISetPosixError(EOPNOTSUPP);
 }

ioctl() deals with blocking support and with FIONREAD.

⟨Overridden member functions for GUSIMacFileSocket 1279⟩+≡ (1207) <1292 1296>
 virtual int ioctl(unsigned int request, va_list arg);

<Member functions for class GUSIMacFileSocket 1267>+≡ (1205) <1293 1297>

```
int GUSIMacFileSocket::ioctl(unsigned int request, va_list arg)
{
    int result;

    if (GUSISMBlocking::DoIoctl(&result, request, arg)
        || GUSISMBuffer::DoIoctl(&result, request, arg)
    )
        return result;

    GUSI_ASSERT_CLIENT(false, ("ioctl: illegal request %d\n", request));

    return GUSISetPosixError(EOPNOTSUPP);
}
```

getsockopt and setsockopt are available for setting buffer sizes and getting asynchronous errors.

<Overridden member functions for GUSIMacFileSocket 1279>+≡ (1207) <1294 1298>
virtual int getsockopt(int level, int optname, void *optval, socklen_t * optlen);

<Member functions for class GUSIMacFileSocket 1267>+≡ (1205) <1295 1299>
int GUSIMacFileSocket::getsockopt(int level, int optname, void *optval, socklen_t * optlen)
{
 int result;

 if (GUSISMBuffer::DoGetSockOpt(&result, level, optname, optval, optlen)
 || GUSISMBuffer::DoGetSockOpt(&result, level, optname, optval, optlen)
 || GUSISMAyncError::DoGetSockOpt(&result, level, optname, optval, optlen)
)
 return result;

 GUSI_ASSERT_CLIENT(false, ("getsockopt: illegal request %d\n", optname));

 return GUSISetPosixError(EOPNOTSUPP);
}

<Overridden member functions for GUSIMacFileSocket 1279>+≡ (1207) <1296 1300>
virtual int setsockopt(int level, int optname, void *optval, socklen_t optlen);

<Member functions for class GUSIMacFileSocket 1267>+≡ (1205) <1297 1301>
int GUSIMacFileSocket::setsockopt(int level, int optname, void *optval, socklen_t optlen)
{
 int result;

 if (GUSISMBuffer::DoSetSockOpt(&result, level, optname, optval, optlen)
 || GUSISMBuffer::DoSetSockOpt(&result, level, optname, optval, optlen)
)
 return result;

 GUSI_ASSERT_CLIENT(false, ("setsockopt: illegal request %d\n", optname));

 return GUSISetPosixError(EOPNOTSUPP);
}

<Overridden member functions for GUSIMacFileSocket 1279>+≡ (1207) <1298 1302>
virtual off_t lseek(off_t offset, int whence);

```

<Member functions for class GUSIMacFileSocket 1267>+≡ (1205) <1299 1303>
off_t GUSIMacFileSocket::lseek(off_t offset, int whence)
{
    if (!fReadShutdown)
        SyncRead();
    else if (!fWriteShutdown)
        SyncWrite();

    long    eof;
    long    pos;

    switch (whence) {
    case SEEK_CUR:
        pos = fPosition;
        break;
    case SEEK_END:
        if (GUSISetMacError(GetEOF(fFileRef, &eof)))
            return -1;
        pos = eof;
        break;
    case SEEK_SET:
        pos = 0;
        break;
    default:
        return GUSISetPosixError(EINVAL);
    }
    GUSI_MESSAGE(("pos: %d  off: %d eof: %d whence: %d\n", pos, offset, eof, whence));
    pos += offset;

    if (fPosition == pos)
        return fPosition;

    if (whence != SEEK_END && GUSISetMacError(GetEOF(fFileRef, &eof)))
        return -1;
    if (pos > eof)
        fBeyondEOF = true;
    else if (pos < 0)
        return GUSISetPosixError(EINVAL);
    else if (GUSISetMacError(SetFPos(fFileRef, fsFromStart, pos)))
        return -1;
    else
        fBeyondEOF = false;

    return fPosition = pos;
}

<Overridden member functions for GUSIMacFileSocket 1279>+≡ (1207) <1300 1304>
virtual int ftruncate(off_t offset);

```

```

<Member functions for class GUSIMacFileSocket 1267>+≡ (1205) ◀1301 1305▶
int GUSIMacFileSocket::ftruncate(off_t offset)
{
    if (lseek(offset, SEEK_SET) == -1)
        return -1;
    if (fBeyondEOF)
        return lseek(0, SEEK_END) == -1 ? -1 : 0;
    else
        return GUSISetMacError(SetEOF(fFileRef, fPosition));
}

```

fstat is implemented via stat (early implementations had it the other way around, but this failed with directories).

```

<Overridden member functions for GUSIMacFileSocket 1279>+≡ (1207) ◀1302 1306▶
virtual int fstat(struct stat * buf);

<Member functions for class GUSIMacFileSocket 1267>+≡ (1205) ◀1303 1307▶
int GUSIMacFileSocket::fstat(struct stat * buf)
{
    fsync();

    GUSIFileSpec    spec(fFileRef);

    if (!spec.Exists())
        return GUSISetPosixError(ENOENT);

    GUSIFileToken    token(spec, GUSIFileToken::kWillStat);

    return token.Device()->stat(token, buf);
}

```

File sockets implement simple calls.

```

<Overridden member functions for GUSIMacFileSocket 1279>+≡ (1207) ◀1304
virtual bool Supports(ConfigOption config);

<Member functions for class GUSIMacFileSocket 1267>+≡ (1205) ◀1305 1311▶
bool GUSIMacFileSocket::Supports(ConfigOption config)
{
    return config == kSimpleCalls;
}

```

31.5.2 Interrupt level routines for GUSIMacFileSocket

```

<Prototypes for GUSIMacFileSocket interrupt level routines 1308>≡ (1205)
class GUSIMacFileSocket;

static void GUSIMFRead(GUSIMacFileSocket * sock);
static void GUSIMFReadDone(ParamBlockRec * pb);
static void GUSIMFWrite(GUSIMacFileSocket * sock);
static void GUSIMFWriteDone(ParamBlockRec * pb);
static void GUSIMFWakeup(ParamBlockRec * pb);

```

```

<Privatissima of GUSIMacFileSocket 1266>+≡ (1207) <1277
    ParamBlockRec      fReadPB;
    ParamBlockRec      fWritePB;
    friend void         GUSIMFRead(GUSIMacFileSocket * sock);
    friend void         GUSIMFReadDone(ParamBlockRec * pb);
    friend void         GUSIMFWrite(GUSIMacFileSocket * sock);
    friend void         GUSIMFWriteDone(ParamBlockRec * pb);
    friend void         GUSIMFWakeup(ParamBlockRec * pb);
    static IOCompletionUPP sReadProc;
    static IOCompletionUPP sWriteProc;
    static IOCompletionUPP sWakeupProc;

```

On 68K machines, completion procedures get called with the argument in the A0 register, which really cramps our style. Therefore, we wrap each procedure.

```

<Interrupt routine wrappers for GUSIMacFileSocket 1310>≡ (1205)
    GUSI_COMPLETION_PROC_A0(GUSIMFReadDone, ParamBlockRec)
    GUSI_COMPLETION_PROC_A0(GUSIMFWriteDone, ParamBlockRec)
    GUSI_COMPLETION_PROC_A0(GUSIMFWakeup, ParamBlockRec)

```

The UPPs for the completion procedures are set up the first time a socket is constructed.

```

<Member functions for class GUSIMacFileSocket 1267>+≡ (1205) <1307
    IOCompletionUPP GUSIMacFileSocket::sReadProc = 0;
    IOCompletionUPP GUSIMacFileSocket::sWriteProc = 0;
    IOCompletionUPP GUSIMacFileSocket::sWakeupProc = 0;

```

```

<Initialize fields of GUSIMacFileSocket 1270>+≡ (1267) <1278 1313>
    if (!fReadShutdown && !sReadProc)
        sReadProc = NewIOCompletionProc(GUSIMFReadDoneEntry);
    if (!fWriteShutdown && !sWriteProc)
        sWriteProc = NewIOCompletionProc(GUSIMFWriteDoneEntry);
    if (!sWakeupProc)
        sWakeupProc = NewIOCompletionProc(GUSIMFWakeupEntry);

```

The write and read parameter blocks are highly specialized and never really change during the existence of a socket.

```

<Initialize fields of GUSIMacFileSocket 1270>+≡ (1267) <1312
    fReadPB.ioParam.ioResult = 0;
    fReadPB.ioParam.ioRefNum = fFileRef;
    fReadPB.ioParam.ioPosMode = fsAtMark;
    if (!fWriteShutdown) {
        fWritePB.ioParam.ioCompletion = sWriteProc;
        fWritePB.ioParam.ioResult = 0;
        fWritePB.ioParam.ioRefNum = fFileRef;
        fWritePB.ioParam.ioPosMode = fsAtMark;

        GUSIMFWrite(this);
    }

```

GUSIMFWriteDone() does all its work between fWritePB and fOutputBuffer. If a write fails, the whole write buffer is cleared.

```

<Interrupt level routines for GUSIMacFileSocket 1314>≡ (1205) 1315>
static void GUSIMFWriteDone(ParamBlockRec * pb)
{
    GUSI_SMESSAGE("Write done.\n");
    GUSIMacFileSocket * sock =
        reinterpret_cast<GUSIMacFileSocket *>((char *)pb-offsetof(GUSIMacFileSocket, fWritePB));
    <Perform sanity cross-check between sock and pb 1321>
    if (sock->fOutputBuffer.Locked())
        sock->fOutputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIMFWriteDone), pb);
    else {
        sock->fOutputBuffer.ClearDefer();
        sock->fOutputBuffer.FreeBuffer(
            sock->fWritePB.ioParam.ioBuffer, sock->fWritePB.ioParam.ioReqCount);
        if (sock->SetAsyncMacError(sock->fWritePB.ioParam.ioResult)) {
            for (long valid; valid = sock->fOutputBuffer.Valid(); )
                sock->fOutputBuffer.FreeBuffer(nil, valid);
            sock->fWriteShutdown = true;
        }
        GUSIMFWrite(sock);
        sock->Wakeup();
    }
}

```

GUSIMFWrite() starts a file write call if there is data to write and otherwise sets itself up as the deferred procedure of the output buffer (and thus is guaranteed to be called the next time data is deposited in the buffer again). If all data has been delivered and a shutdown is requested, terminate.

```

<Interrupt level routines for GUSIMacFileSocket 1314>+≡ (1205) <1314 1317>
static void GUSIMFWrite(GUSIMacFileSocket * sock)
{
    size_t valid = sock->fOutputBuffer.Valid();
    if (sock->fWriteShutdown && !valid)
        sock->fOutputBuffer.ClearDefer();
    else if (!valid)
        sock->fOutputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIMFWrite), sock);
    else {
        sock->fOutputBuffer.ClearDefer();
        valid = min(valid, sock->fOutputBuffer.Size() >> 1);
        sock->fWritePB.ioParam.ioBuffer =
            static_cast<Ptr>(sock->fOutputBuffer.ConsumeBuffer(valid));
        <Align write size with fBlockSize and file position 1316>
        sock->fWritePB.ioParam.ioReqCount = valid;
        sock->fWritePB.ioParam.ioActCount = 0;

        GUSI_MESSAGE(("Async write (0x%x bytes).\n", valid));
        PBWriteAsync(&sock->fWritePB);
    }
}

```

Whenever possible, we try to maintain alignment with allocation block boundaries. If write to a write-only file is aligned, we request that it not be cached, which improves performance.

⟨Align write size with fBlockSize and file position 1316⟩≡ (1315)

```
long pos      = sock->fPosition - sock->fOutputBuffer.Valid();
size_t align  = pos % sock->fBlockSize;
if (align)
    valid = min(valid, sock->fBlockSize-align);
else if (valid > sock->fBlockSize)
    valid -= valid % sock->fBlockSize;
sock->fWritePB.ioParam.ioPosMode = fsAtMark +
    (sock->fWriteBehind && valid >= sock->fBlockSize) ? 0x0020 : 0;
```

GUSIMFReadDone() does all its work between fReadPB and fInputBuffer. If a read fails, the whole write buffer is cleared.

⟨Interrupt level routines for GUSIMacFileSocket 1314⟩+≡ (1205) <1315 1318>

```
static void GUSIMFReadDone(ParamBlockRec * pb)
{
    GUSI_MESSAGE(("Read done (0x%x/%d).\n", pb->ioParam.ioActCount, pb->ioParam.ioResult));
    GUSIMacFileSocket * sock =
        reinterpret_cast<GUSIMacFileSocket *>((char *)pb-offsetof(GUSIMacFileSocket, fReadPB));
    ⟨Perform sanity cross-check between sock and pb 1321⟩
    if (sock->fInputBuffer.Locked())
        sock->fInputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIMFReadDone), pb);
    else {
        sock->fInputBuffer.ClearDefer();
        sock->fInputBuffer.ValidBuffer(
            sock->fReadPB.ioParam.ioBuffer, sock->fReadPB.ioParam.ioActCount);
        if (sock->SetAsyncMacError(sock->fReadPB.ioParam.ioResult))
            sock->fReadShutdown = true;
        else if (!sock->fReadShutdown)
            GUSIMFRead(sock);
        sock->Wakeup();
    }
}
```


GUSIMFRead() starts a file read call.

```

<Interrupt level routines for GUSIMacFileSocket 1314>+≡ (1205) <1317 1320>
static void GUSIMFRead(GUSIMacFileSocket * sock)
{
    size_t free = sock->fInputBuffer.Free();
    if (free < (sock->fInputBuffer.Size() >> 2)) {
        if (!sock->fReadShutdown)
            sock->fInputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIMFRead), sock);
        else
            sock->fInputBuffer.ClearDefer();
    } else {
        sock->fInputBuffer.ClearDefer();
        free = min(free, sock->fInputBuffer.Size() >> 1);
        <Align read size with fBlockSize and file position 1319>

        sock->fReadPB.ioParam.ioBuffer =
            static_cast<Ptr>(sock->fInputBuffer.ProduceBuffer(free));
        sock->fReadPB.ioParam.ioReqCount = free;
        sock->fReadPB.ioParam.ioActCount = 0;

        GUSI_MESSAGE(("Async read 0x%x 0x%x (0x%x bytes).\n", sock, &sock->fReadPB, free));
        PBReadAsync(&sock->fReadPB);
    }
}

```

Whenever possible, we try to maintain alignment with allocation block boundaries.

```

<Align read size with fBlockSize and file position 1319>≡ (1318)
long pos = sock->fPosition + sock->fInputBuffer.Valid();
size_t align = pos % sock->fBlockSize;
if (align)
    free = min(free, sock->fBlockSize-align);
else if (free > sock->fBlockSize)
    free -= free % sock->fBlockSize;

```

GUSIMFWakeup() simply wakes up its socket.

```

<Interrupt level routines for GUSIMacFileSocket 1314>+≡ (1205) <1318
static void GUSIMFWakeup(ParamBlockRec * pb)
{
    GUSIMacFileSocket * sock =
        reinterpret_cast<GUSIMacFileSocket *>((char *)pb-offsetof(GUSIMacFileSocket, fReadPB));
    <Perform sanity cross-check between sock and pb 1321>
    sock->Wakeup();
}

```

To atone for the risky practice of reinterpret_cast<>, we at least check for plausibility by comparing the two fields storing the file reference number.

```

<Perform sanity cross-check between sock and pb 1321>≡ (1314 1317 1320)
GUSI_CASSERT_INTERNAL(sock->fFileRef == pb->ioParam.ioRefNum);

```

31.6 Implementation of GUSIMacDirectory

```
(Privatissima of GUSIMacDirectory 1322)≡ (1208)
    GUSIFileSpec    fDir;
    long            fPos;
    dirent          fEnt;

(Member functions for class GUSIMacDirectory 1323)≡ (1205) 1325»
    GUSIMacDirectory::GUSIMacDirectory(const FSSpec & spec)
        : fDir(spec)
    {
        fPos = 1;
    }

(Overridden member functions for GUSIMacDirectory 1324)≡ (1208) 1326»
    virtual dirent * readdir();

(Member functions for class GUSIMacDirectory 1323)+≡ (1205) «1323 1327»
    dirent * GUSIMacDirectory::readdir()
    {
        fDir[fPos];
        if (fDir.Error())
            return GUSISetMacError(fDir.Error()), static_cast<dirent *>(nil);

        ++fPos;
        const GUSICatInfo * info(fDir.CatInfo());
        fEnt.d_ino = info->DirInfo().ioDrDirID;
        memcpy(fEnt.d_name, (char *)fDir->name+1, fDir->name[0]);
        fEnt.d_name[fDir->name[0]] = 0;

        return &fEnt;
    }

(Overridden member functions for GUSIMacDirectory 1324)+≡ (1208) «1324 1328»
    virtual long telldir();

(Member functions for class GUSIMacDirectory 1323)+≡ (1205) «1325 1329»
    long GUSIMacDirectory::telldir()
    {
        return fPos;
    }

(Overridden member functions for GUSIMacDirectory 1324)+≡ (1208) «1326 1330»
    virtual void seekdir(long pos);

(Member functions for class GUSIMacDirectory 1323)+≡ (1205) «1327 1331»
    void GUSIMacDirectory::seekdir(long pos)
    {
        fPos = pos;
    }

(Overridden member functions for GUSIMacDirectory 1324)+≡ (1208) «1328
```

<Member functions for class GUSIMacDirectory 1323>+≡

(1205) <1329

```
void GUSIMacDirectory::rewinddir()
```

```
{
```

```
    fPos = 1;
```

```
}
```


Chapter 32

The GUSI PPC Socket Class

The GUSIPPCSocket class implements communication via the Program-Program Communications Toolbox (For a change, the PPC does not stand for “PowerPC” here).

We give PPC sockets their own official looking header.

```
<sys/ppc.h 1332>≡
#include <PPCToolbox.h>

struct sockaddr_ppc {
    sa_family_t    sppc_family;    /* AF_PPC */
    LocationNameRec sppc_location;
    PPCPortRec     sppc_port;
};
```

The GUSIPPCFactory singleton creates GUSIPPCSockets.

```
<GUSIPPC.h 1333>≡
#ifndef _GUSIPPC_
#define _GUSIPPC_

#ifdef GUSI_INTERNAL

#include "GUSISocket.h"
#include "GUSIFactory.h"
#include <sys/ppc.h>

<Definition of class GUSIPPCFactory 1335>

<Inline member functions for class GUSIPPCFactory 1337>

#endif /* GUSI_INTERNAL */

#endif /* _GUSIPPC_ */
```

```

<GUSIPPC.cp 1334>≡
#include "GUSIInternal.h"
#include "GUSIPPC.h"
#include "GUSIBasics.h"
#include "GUSIBuffer.h"
#include "GUSISocketMixins.h"

#include <errno.h>

#include <algorithm>

<Definition of class GUSIPPCSocket 1339>
<Member functions for class GUSIPPCFactory 1336>
<Member functions for class GUSIPPCSocket 1341>
<Interrupt level routines for GUSIPPCSocket 1343>

```

32.1 Definition of GUSIPPCFactory

GUSIPPCFactory is a singleton subclass of GUSISocketFactory.

```

<Definition of class GUSIPPCFactory 1335>≡ (1333)
class GUSIPPCFactory : public GUSISocketFactory {
public:
    static GUSISocketFactory * Instance();
    virtual GUSISocket * socket(int domain, int type, int protocol);
private:
    GUSIPPCFactory() {}
    static GUSISocketFactory * sInstance;
};

```

32.2 Implementation of GUSIPPCFactory

```

<Member functions for class GUSIPPCFactory 1336>≡ (1334) 1338»
GUSISocketFactory * GUSIPPCFactory::sInstance = nil;

<Inline member functions for class GUSIPPCFactory 1337>≡ (1333)
inline GUSISocketFactory * GUSIPPCFactory::Instance()
{
    if (!sInstance)
        sInstance = new GUSIPPCFactory;
    return sInstance;
}

<Member functions for class GUSIPPCFactory 1336>+≡ (1334) <1336
GUSISocket * GUSIPPCFactory::socket(int, int, int)
{
    return new GUSIPPCSocket;
}

```

32.3 Definition of GUSIPPCSocket

A GUSIPPCSocket is implemented with the usual read and write buffers.

```
(Definition of class GUSIPPCSocket 1339)≡ (1334)
class GUSIPPCSocket :
    public GUSISocket,
    protected GUSISMBlocking,
    protected GUSISMState,
    protected GUSISIMInputBuffer,
    protected GUSISIMOutputBuffer
{
public:
    GUSIPPCSocket();
    virtual ~GUSIPPCSocket();
    (Overridden member functions for GUSIPPCSocket 1361)
protected:
    (Privatissima of GUSIPPCSocket 1340)
};
```

32.4 Implementation of GUSIPPCSocket

The implementation of GUSIPPCSocket is similar to many other socket classes. Let's start, once more, with the interrupt level routines.

32.4.1 Interrupt level routines for GUSIPPCSocket

Both GUSIPPCSendDone() and GUSIPPCRecvDone() are always called with the same TCPIopb in a GUSIPPCSocket so they can easily find out the address of the socket itself. GUSIPPCSend and GUSIPPCRecv set up send and receive calls.

```
(Privatissima of GUSIPPCSocket 1340)≡ (1339) 1347▷
    OSErr fAsyncError;
    PPCParamBlockRec fSendPB;
    PPCParamBlockRec fRecvPB;
    friend void GUSIPPCSend(GUSIPPCSocket * sock);
    friend void GUSIPPCRecv(GUSIPPCSocket * sock);
    friend void GUSIPPCSendDone(PPCParamBlockPtr pb);
    friend void GUSIPPCRecvDone(PPCParamBlockPtr pb);
    static PPCCompUPP sSendProc;
    static PPCCompUPP sRecvProc;
```

The UPPs for the completion procedures are set up the first time a socket is constructed.

```
(Member functions for class GUSIPPCSocket 1341)≡ (1334) 1348▷
    PPCCompUPP GUSIPPCSocket::sSendProc = 0;
    PPCCompUPP GUSIPPCSocket::sRecvProc = 0;

(Initialize fields of GUSIPPCSocket 1342)≡ (1360 1378) 1349▷
    if (!sSendProc)
        sSendProc = NewPPCCompProc(GUSIPPCSendDone);
    if (!sRecvProc)
        sRecvProc = NewPPCCompProc(GUSIPPCRecvDone);
```

GUSIPPCSendDone() does all its work between fSendPB and fOutputBuffer. If a send fails, the whole send buffer is cleared.

```

<Interrupt level routines for GUSIPPCSocket 1343>≡ (1334) 1344»
void GUSIPPCSendDone(PPCParamBlockPtr pb)
{
    GUSIPPCSocket * sock =
        reinterpret_cast<GUSIPPCSocket *>((char *)pb-offsetof(GUSIPPCSocket, fSendPB));
    if (sock->fOutputBuffer.Locked())
        sock->fOutputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIPPCSendDone), pb);
    else {
        PPCWritePBRec & writeParam      =  pb->writeParam;

        sock->fOutputBuffer.ClearDefer();
        sock->fOutputBuffer.FreeBuffer(writeParam.bufferPtr, writeParam.actualLength);
        if (writeParam.ioResult) {
            for (long valid; valid = sock->fOutputBuffer.Valid(); )
                sock->fOutputBuffer.FreeBuffer(nil, valid);
            sock->fWriteShutdown = true;
        }
        GUSIPPCSend(sock);
        sock->Wakeup();
    }
}

```

GUSIPPCSend() starts a PPC write call if there is data to send and otherwise sets itself up as the deferred procedure of the output buffer (and thus is guaranteed to be called the next time data is deposited in the buffer again). If all data has been delivered and a shutdown is requested, send one.

```

<Interrupt level routines for GUSIPPCSocket 1343>+≡ (1334) <1343 1345>
void GUSIPPCSend(GUSIPPCSocket * sock)
{
    size_t valid = sock->fOutputBuffer.Valid();

    sock->fOutputBuffer.ClearDefer();
    if (!valid) {
        if (!sock->fWriteShutdown)
            sock->fOutputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIPPCSend), sock);
        else if (sock->fState == GUSIPPCSocket::Connected)
            sock->fState = GUSIPPCSocket::Closing;
    } else {
        PPCWritePBRec & writeParam      =  sock->fSendPB.writeParam;
        valid = min(valid, min((size_t)65535, sock->fOutputBuffer.Size() >> 1));

        writeParam.bufferPtr      =
            reinterpret_cast<char *>(sock->fOutputBuffer.ConsumeBuffer(valid));
        writeParam.bufferLength =  (Size) valid;
        writeParam.more          =  false;
        writeParam.userData      =  0;
        writeParam.blockCreator =  'GUI';
        writeParam.blockType    =  'strm';

        PPCWriteAsync(&writeParam);
    }
}

```


GUSIPPCRecvDone() does all its work between fRecvPB and fInputBuffer.

```

<Interrupt level routines for GUSIPPCSocket 1343>+≡ (1334) <1344 1346>
void GUSIPPCRecvDone(PPCParamBlockPtr pb)
{
    GUSIPPCSocket * sock =
        reinterpret_cast<GUSIPPCSocket *>((char *)pb-offsetof(GUSIPPCSocket, fRecvPB));
    if (sock->fInputBuffer.Locked())
        sock->fInputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIPPCRecvDone), pb);
    else {
        PPCReadPBRec & readParam = pb->readParam;
        sock->fInputBuffer.ClearDefer();
        switch (readParam.ioResult) {
            case noErr:
                sock->fInputBuffer.ValidBuffer(readParam.bufferPtr, readParam.actualLength);
                GUSIPPCRecv(sock);
                break;
            default:
                sock->fReadShutdown = true;
                break;
        }
        sock->Wakeup();
    }
}

```

GUSIPPCRecv() starts a PPC receive call if there is room left and otherwise sets itself up as the deferred procedure of the output buffer (and thus is guaranteed to be called the next time there is free space in the buffer again).

```

<Interrupt level routines for GUSIPPCSocket 1343>+≡ (1334) <1345 1352>
void GUSIPPCRecv(GUSIPPCSocket * sock)
{
    size_t free = sock->fInputBuffer.Free();
    if (!free)
        sock->fInputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIPPCRecv), sock);
    else {
        PPCReadPBRec & readParam = sock->fRecvPB.readParam;

        sock->fInputBuffer.ClearDefer();
        free = min(free, min((size_t)65535, sock->fInputBuffer.Size() >> 1));

        readParam.ioCompletion = sock->sRecvProc;
        readParam.bufferPtr =
            reinterpret_cast<char *>(sock->fInputBuffer.ProduceBuffer(free));
        readParam.bufferLength = free;

        PPCReadAsync(&readParam);
    }
}

```

For listens it is necessary to build a backlog if the interrupt level gets opens faster than the high level routines can accept them.

<Privatissima of GUSIPPCSocket 1340>+≡ (1339) «1340 1350»

```
struct Listener {
    PPCSessRefNum    fSession;
    LocationNameRec  fLocation;
    PPCPortRec       fPort;
    bool             fBusy;
};
Listener * fListeners;
bool       fRestartListen;
char       fNumListeners;
char       fCurListener;
char       fNextListener;
friend void GUSIPPCListenDone(PPCParamBlockPtr pb);
friend void GUSIPPCListen(GUSIPPCSocket * sock);
static PPCCompUPP sListenProc;
```

<Member functions for class GUSIPPCSocket 1341>+≡ (1334) «1341 1351»

```
PPCCompUPP GUSIPPCSocket::sListenProc = 0;
```

<Initialize fields of GUSIPPCSocket 1342>+≡ (1360 1378) «1342 1356»

```
fListeners      = nil;
fRestartListen  = true;
fNumListeners   = 0;
fCurListener   = 0;
fNextListener   = 0;
if (!sListenProc)
    sListenProc = NewPPCCompProc(GUSIPPCListenDone);
```

SetupListener() prepares a Listener.

<Privatissima of GUSIPPCSocket 1340>+≡ (1339) «1347 1354»

```
void SetupListener(Listener & listener);
```

<Member functions for class GUSIPPCSocket 1341>+≡ (1334) «1348 1355»

```
void GUSIPPCSocket::SetupListener(Listener & listener)
{
    listener.fBusy = false;
}
```

GUSIPPCListenDone() saves the connection parameters and starts the next passive open, if possible. Blocking on fInputBuffer is somewhat bizarre; we're not actually using the buffer, just its lock. The only times this lock is used is when the socket is shutting down.

```

<Interrupt level routines for GUSIPPCSocket 1343>+≡ (1334) ◀1346 1353▶
void GUSIPPCListenDone(PPCParamBlockPtr pb)
{
    GUSIPPCSocket * sock =
        (GUSIPPCSocket *)((char *)pb-offsetof(GUSIPPCSocket, fRecvPB));
    if (!(sock->fAsyncError = pb->informParam.ioResult)) {
        GUSIPPCSocket::Listener & listener = sock->fListeners[sock->fCurListener];
        listener.fSession = pb->informParam.sessRefNum;
        listener.fBusy = true;
        sock->fCurListener = (sock->fCurListener+1) % sock->fNumListeners;
    }
    sock->Wakeup();
    if (!sock->fAsyncError)
        if (sock->fInputBuffer.Locked())
            sock->fInputBuffer.Defer(GUSIRingBuffer::Deferred(GUSIPPCListenDone), pb);
        else {
            sock->fInputBuffer.ClearDefer();
            GUSIPPCListen(sock);
        }
}

```

GUSIPPCListen() initiates a passive open.

```

<Interrupt level routines for GUSIPPCSocket 1343>+≡ (1334) ◀1352 1357▶
void GUSIPPCListen(GUSIPPCSocket * sock)
{
    if (sock->fReadShutdown)
        return;
    if (sock->fRestartListen = sock->fListeners[sock->fCurListener].fBusy)
        return;
    PPCInformPBRec & informParam = sock->fRecvPB.informParam;
    GUSIPPCSocket::Listener & listener = sock->fListeners[sock->fCurListener];
    informParam.autoAccept = true;
    informParam.portName = &listener.fPort;
    informParam.locationName = &listener.fLocation;
    informParam.userName = nil;

    PPCInformAsync(&informParam);
}

```

Some functions simply need to wake up the socket again. These are always called on fSendPB, because fRecvPB might be in use by a PPCInform call.

```

<Privatissima of GUSIPPCSocket 1340>+≡ (1339) ◀1350 1358▶
friend void GUSIPPCDone(PPCParamBlockPtr pb);
static PPCCompUPP sDoneProc;

```

```

<Member functions for class GUSIPPCSocket 1341>+≡ (1334) ◀1351 1360▶
PPCCompUPP GUSIPPCSocket::sDoneProc = 0;

```

```

<Initialize fields of GUSIPPCSocket 1342>+≡ (1360 1378) ◀1349
if (!sDoneProc)
    sDoneProc = NewPPCCompProc(GUSIPPCDone);

```

```

<Interrupt level routines for GUSIPPCSocket 1343>+≡ (1334) <1353
void GUSIPPCDone(PPCParamBlockPtr pb)
{
    GUSIPPCSocket * sock =
        (GUSIPPCSocket *)((char *)pb-offsetof(GUSIPPCSocket, fSendPB));
    sock->fAsyncError = pb->endParam.ioResult;
    sock->Wakeup();
}

```

32.4.2 High level interface for GUSIPPCSocket

```

<Privatissima of GUSIPPCSocket 1340>+≡ (1339) <1354 1359>
LocationNameRec    fLocation;
PPCPortRec         fPort;
LocationNameRec    fPeerLoc;
PPCPortRec         fPeerPort;

```

The port reference number is shared between a listener and all of its clients, so we cannot simply close it at any time and have to store it in a shared data structure.

```

<Privatissima of GUSIPPCSocket 1340>+≡ (1339) <1358 1377>
struct PortRef {
    short          fRefCount;
    PPCPortRefNum  fPort;

    PortRef(PPCPortRefNum port) : fRefCount(1), fPort(port) {}
};
PortRef *          fPortRef;

```

The constructors have to initialize a rather large number of data fields.

```

<Member functions for class GUSIPPCSocket 1341>+≡ (1334) <1355 1362>
GUSIPPCSocket::GUSIPPCSocket()
{
    <Initialize fields of GUSIPPCSocket 1342>
}

```

bind() is absolutely required for all forms of PPC sockets.

```

<Overridden member functions for GUSIPPCSocket 1361>≡ (1339) 1366>
virtual int bind(void * addr, socklen_t namelen);

```

```

(⟨Member functions for class GUSIPPCSocket 1341⟩+≡ (1334) ◀1360 1367▶
int GUSIPPCSocket::bind(void * addr, socklen_t namelen)
{
    struct sockaddr_ppc *name = (struct sockaddr_ppc *)addr;
    ⟨Sanity checks for GUSIPPCSocket::bind() 1363⟩
    fPort = name->sppc_port;
    fLocation = name->sppc_location;

    PPCOpenPBRec & openParam = fSendPB.openParam;
    openParam.ioCompletion = sDoneProc;
    openParam.serviceType = ppcServiceRealTime;
    openParam.resFlag = 0;
    openParam.portName = &fPort;
    openParam.locationName = &fLocation;
    openParam.networkVisible = true;
    PPCOpenAsync(&openParam);
    AddContext();
    while (fSendPB.openParam.ioResult == 1)
        GUSIContext::Yield(true);
    RemoveContext();

    switch (fSendPB.openParam.ioResult) {
    case noErr:
        fPortRef = new PortRef(openParam.portRefNum);
        fState = Unconnected;
        return 0;
    case noGlobalsErr:
        return GUSISetPosixError(ENOMEM);
    case portNameExistsErr:
    case nbpDuplicate:
        return GUSISetPosixError(EADDRINUSE);
    case nameTypeErr:
    case badReqErr:
    case badPortNameErr:
    case badLocNameErr:
    default:
        return GUSISetPosixError(EINVAL);
    }
}

```

The address to be passed must be up to a minimal standard of decency. For instance, the host address must be either the real IP number of our host or one of the two legitimate pseudo-addresses for "localhost".

(Sanity checks for GUSIPPCSocket::bind() 1363)≡ (1362)

```

if (!GUSI_ASSERT_CLIENT(
    namelen >= sizeof(struct sockaddr_ppc),
    ("bind: address len %d < %d\n", namelen, sizeof(struct sockaddr_ppc)))
)
    return GUSISetPosixError(EINVAL);
if (!GUSI_ASSERT_CLIENT(
    name->sppc_family == AF_PPC,
    ("bind: family %d != %d\n", name->sppc_family, AF_PPC))
)
    return GUSISetPosixError(EAFNOSUPPORT);
if (!GUSI_ASSERT_CLIENT(fState==Unbound, "bind: Socket already bound\n"))
    return GUSISetPosixError(EINVAL);

```

getsockname() and getpeername() return the stored values.

(Overridden member functions for GUSIMTInetSocket 1364)≡

```

virtual int getsockname(void * addr, socklen_t * namelen);
virtual int getpeername(void * addr, socklen_t * namelen);

```

(Member functions for class GUSIMTInetSocket 1365)≡

```

int GUSIMTInetSocket::getsockname(void *name, socklen_t *namelen)
{
    if (!GUSI_ASSERT_CLIENT(*namelen >= 0, "getsockname: passed negative length\n"))
        return GUSISetPosixError(EINVAL);

    sockaddr_ppc addr;
    addr.sppc_len      = sizeof(sockaddr_ppc);
    addr.sppc_family   = AF_PPC;
    addr.sppc_port     = fPort;
    addr.sppc_location = fLocation;

    memcpy(name, &addr, *namelen = min(*namelen, socklen_t(sizeof(sockaddr_ppc))));
    return 0;
}
int GUSIMTInetSocket::getpeername(void *name, socklen_t *namelen)
{
    if (!GUSI_ASSERT_CLIENT(*namelen >= 0, "getpeername: passed negative length\n"))
        return GUSISetPosixError(EINVAL);

    sockaddr_ppc addr;
    addr.sppc_len      = sizeof(sockaddr_ppc);
    addr.sppc_family   = AF_PPC;
    addr.sppc_port     = fPeerPort;
    addr.sppc_location = fPeerLoc;

    memcpy(name, &addr, *namelen = min(*namelen, socklen_t(sizeof(sockaddr_ppc))));
    return 0;
}

```

connect() opens a connection actively.

(Overridden member functions for GUSIPPCSocket 1361)+≡ (1339) <1361 1369>

```

virtual int connect(void * address, socklen_t addrlen);

```

```

<Member functions for class GUSIPPCSocket 1341>+≡ (1334) <1362 1370>
int GUSIPPCSocket::connect(void * address, socklen_t addrlen)
{
    sockaddr_ppc * addr    = (sockaddr_ppc *) address;

    <Sanity checks for GUSIPPCSocket::connect() 1368>

    fPeerPort    = addr->sppc_port;
    fPeerLoc     = addr->sppc_location;

    PPCStartPBRec & startParam = fSendPB.startParam;
    startParam.portRefNum    = fPortRef->fPort;
    startParam.resFlag      = 0;
    startParam.portName     = &fPeerPort;
    startParam.locationName = &fPeerLoc;
    startParam.userData     = 0;
    startParam.userRefNum   = 0;

    Boolean guest;
    Str32  uname;
    uname[0] = 0;

    switch (StartSecureSession(&startParam, uname, true, true, &guest, (StringPtr) "\p")) {
    case userCanceledErr:
        return GUSISetPosixError(EINTR);
    default:
        return GUSISetPosixError(EINVAL);
    case noErr:
        fState = Connected;

        PPCWritePBRec & writeParam = fSendPB.writeParam;
        writeParam.ioCompletion = sSendProc;

        PPCReadPBRec & readParam = fRecvPB.readParam;
        readParam.ioCompletion = sRecvProc;
        readParam.sessRefNum    = writeParam.sessRefNum;

        GUSIPPCSend(this);
        GUSIPPCRecv(this);

        return 0;
    }
}

```

```

<Sanity checks for GUSIPPCSocket::connect() 1368>≡ (1367)
if (!GUSI_CASSERT_CLIENT(addrlen >= int(sizeof(sockaddr_ppc))))
    return GUSISetPosixError(EINVAL);
if (!GUSI_CASSERT_CLIENT(addr->sppc_family == AF_PPC))
    return GUSISetPosixError(EAFNOSUPPORT);
switch (fState) {
case Unbound:
    return GUSISetPosixError(EINVAL);           // Must be bound before connecting
case Unconnected:
    break;                                     // Go ahead
default:
    return GUSISetPosixError(EISCONN);         // Already connected in some form
}

```

Most of the dirty work of listen() is already handled in GUSIPPCListen.

```

<Overridden member functions for GUSIPPCSocket 1361>+≡ (1339) <1366 1373>
virtual int listen(int queueLength);

```

```

<Member functions for class GUSIPPCSocket 1341>+≡ (1334) <1367 1374>
int GUSIPPCSocket::listen(int queueLength)
{
    <Sanity checks for GUSIPPCSocket::listen() 1371>
    <Adjust queueLength according to BSD definition 1372>

    fInputBuffer.SwitchBuffer(0);
    fOutputBuffer.SwitchBuffer(0);
    fState = Listening;
    fListeners = new Listener[fNumListeners = queueLength];
    while (queueLength--)
        SetupListener(fListeners[queueLength]);

    GUSIPPCListen(this);

    return 0;
}

```

```

<Sanity checks for GUSIPPCSocket::listen() 1371>≡ (1370)
if (!GUSI_CASSERT_CLIENT(fState <= Unconnected))
    return GUSISetPosixError(EISCONN);
if (!GUSI_CASSERT_CLIENT(fState != Unbound))
    return GUSISetPosixError(EINVAL);

```

For some weird reason, BSD multiplies queue lengths with a fudge factor.

```

<Adjust queueLength according to BSD definition 1372>≡ (1370)
if (queueLength < 1)
    queueLength = 1;
else if (queueLength > 4)
    queueLength = 8;
else
    queueLength = ((queueLength * 3) >> 1) + 1;

```

accept() also is able to delegate most of the hard work to GUSIPPCListen.

```

<Overridden member functions for GUSIPPCSocket 1361>+≡ (1339) <1369 1379>
virtual GUSISocket * accept(void *from, socklen_t *fromlen);

```



```

<Member functions for class GUSIPPCSocket 1341>+≡ (1334) <1370 1378>
GUSISocket * GUSIPPCSocket::accept(void *from, socklen_t *fromlen)
{
    GUSIPPCSocket * sock;

    <Sanity checks for GUSIPPCSocket::accept() 1375>
    <Wait for a connection to arrive for the GUSIPPCSocket 1376>

    sock = new GUSIPPCSocket(this, fListeners[fNextListener]);

    SetupListener(fListeners[fNextListener]);
    fNextListener = (fNextListener+1) % fNumListeners;

    if (fRestartListen)
        GUSIPPCListen(this);

    if (sock && from)
        sock->getpeername(from, fromlen);

    return sock;
}

<Sanity checks for GUSIPPCSocket::accept() 1375>≡ (1374)
if (!GUSI_CASSERT_CLIENT(fState == Listening)) {
    GUSISetPosixError(ENOTCONN);
    return nil;
}

```

Listener slots are filled one by one, so we simply check whether the next listener block has been filled yet.

```

<Wait for a connection to arrive for the GUSIPPCSocket 1376>≡ (1374)
if (!fListeners[fNextListener].fBusy && !fReadShutdown) {
    if (!fBlocking) {
        GUSISetPosixError(EWOULDBLOCK);
        return nil;
    }
    bool signal = false;
    AddContext();
    while (!fListeners[fNextListener].fBusy && !fReadShutdown) {
        if (signal = GUSIContext::Yield(true))
            break;
    }
    RemoveContext();
    if (!fListeners[fNextListener].fBusy && signal)
        return GUSISetPosixError(EINTR), static_cast<GUSISocket *>(0);
}
if (!fListeners[fNextListener].fBusy && fReadShutdown)
    return GUSISetPosixError(ESHUTDOWN), static_cast<GUSISocket *>(0);

accept() uses a special constructor of GUSIPPCSocket which constructs a socket
directly from a Listener.

```

```

<Privatissima of GUSIPPCSocket 1340>+≡ (1339) <1359>
GUSIPPCSocket(GUSIPPCSocket * orig, Listener & listener);

```

<Member functions for class GUSIPPCSocket 1341>+≡ (1334) <1374 1380>

```
GUSIPPCSocket::GUSIPPCSocket(GUSIPPCSocket * orig, Listener & listener)
```

```
{
```

```
    <Initialize fields of GUSIPPCSocket 1342>
```

```
    fLocation      = orig->fLocation;
```

```
    fPort          = orig->fPort;
```

```
    fPeerLoc       = listener.fLocation;
```

```
    fPeerPort      = listener.fPort;
```

```
    fPortRef       = orig->fPortRef;
```

```
    ++fPortRef->fRefCount;
```

```
    fState         = Connected;
```

```
    PPCWritePBBRec & writeParam = fSendPB.writeParam;
```

```
    writeParam.ioCompletion = sSendProc;
```

```
    writeParam.sessRefNum   = listener.fSession;
```

```
    PPCReadPBBRec & readParam = fRecvPB.readParam;
```

```
    readParam.ioCompletion = sRecvProc;
```

```
    readParam.sessRefNum   = listener.fSession;
```

```
    GUSIPPCSend(this);
```

```
    GUSIPPCRecv(this);
```

```
}
```

recvfrom() reads from fInputBuffer.

<Overridden member functions for GUSIPPCSocket 1361>+≡ (1339) <1373 1383>

```
virtual ssize_t recvfrom(const GUSIScatterer & buffer, int, void * from, socklen_t * fromlen);
```

<Member functions for class GUSIPPCSocket 1341>+≡ (1334) <1378 1384>

```
ssize_t GUSIPPCSocket::recvfrom(const GUSIScatterer & buffer, int flags, void * from, socklen_t * f
```

```
{
```

```
    if (from)
```

```
        getpeername(from, fromlen);
```

```
    <Sanity checks for GUSIPPCSocket::recvfrom() 1381>
```

```
    <Wait for valid data on GUSIPPCSocket 1382>
```

```
    size_t len = buffer.Length();
```

```
    if (flags & MSG_PEEK)
```

```
        fInputBuffer.Peek(buffer, len);
```

```
    else
```

```
        fInputBuffer.Consume(buffer, len);
```

```
    return (int)len;
```

```
}
```

```

<Sanity checks for GUSIPPCSocket::recvfrom() 1381>≡ (1380)
    if (fReadShutdown && !fInputBuffer.Valid())
        return 0;
    switch (fState) {
    case Unbound:
    case Unconnected:
    case Listening:
        return GUSISetPosixError(ENOTCONN);
    case Closing:
    case Connecting:
    case Connected:
        break;
    }

```

The socket needs to be in Connected or Closing state and the input buffer needs to be nonempty before a read can succeed.

```

<Wait for valid data on GUSIPPCSocket 1382>≡ (1380)
    if (!fReadShutdown && (fState == Connecting || fState == Connected || fState == Closing) && !fInputBuffer.Valid())
        if (!fBlocking)
            return GUSISetPosixError(EWOULDBLOCK);
    bool signal = false;
    AddContext();
    while (!fReadShutdown && (fState == Connecting || fState == Connected || fState == Closing) && !fInputBuffer.Valid())
        if (signal = GUSIContext::Yield(true))
            break;
    RemoveContext();
    if (signal && !fInputBuffer.Valid())
        return GUSISetPosixError(EINTR);
}

```

sendto() writes to fOutputBuffer. As opposed to reads, writes have to be executed fully. This leads to a problem when a nonblocking write wants to write more data than the total length of the buffer. In this case, GUSI disregards the nonblocking flag.

```

<Overridden member functions for GUSIPPCSocket 1361>+≡ (1339) <1379 1387>
    virtual ssize_t sendto(const GUSIGatherer & buffer, int flags, const void * to, socklen_t);

```

```

<Member functions for class GUSIPPCSocket 1341>+≡ (1334) <1380 1388>
    ssize_t GUSIPPCSocket::sendto(const GUSIGatherer & buffer, int flags, const void * to, socklen_t)
    {
        <Sanity checks for GUSIPPCSocket::sendto() 1385>

        size_t rest = buffer.Length();
        size_t offset = 0;
        while (rest) {
            size_t len = rest;
            bool signal = false;
            <Wait for free buffer space on GUSIPPCSocket 1386>
            fOutputBuffer.Produce(buffer, len, offset);
            rest -= len;
        }

        return offset;
    }

```

```

<Sanity checks for GUSIPPCSocket::sendto() 1385>≡ (1384)
    if (fWriteShutdown)
        return GUSISetPosixError(ESHUTDOWN);

    if (!GUSI_SASSERT_CLIENT(!to, "Can't sendto() on a stream socket"))
        return GUSISetPosixError(EOPNOTSUPP);
    switch (fState) {
    case Unbound:
    case Unconnected:
    case Listening:
        return GUSISetPosixError(ENOTCONN);
    case Closing:
    case Connecting:
    case Connected:
        break;
    }

    if (!fBlocking && !fOutputBuffer.Free())
        return GUSISetPosixError(EAGAIN);

<Wait for free buffer space on GUSIPPCSocket 1386>≡ (1384)
    if (!fBlocking && !fOutputBuffer.Free())
        break;
    if (!fWriteShutdown && (fState == Connecting || fState == Connected) && !fOutputBuffer.Free()) {
        AddContext();
        while (!fWriteShutdown && (fState == Connecting || fState == Connected) && !fOutputBuffer.Free())
            signal = GUSIContext::Yield(true);
        RemoveContext();
    }
    if (signal || (fWriteShutdown && !fOutputBuffer.Free()))
        if (offset)
            break;
        else
            return GUSISetPosixError(signal ? EINTR : ESHUTDOWN);

    select() checks for various conditions on the socket.

<Overridden member functions for GUSIPPCSocket 1361>+≡ (1339) <1383 1389>
    virtual bool    select(bool * canRead, bool * canWrite, bool * exception);

```

⟨Member functions for class GUSIPPCSocket 1341⟩+≡ (1334) ◀1384 1390▶

```
bool GUSIPPCSocket::select(bool * canRead, bool * canWrite, bool *)
```

```
{
    bool cond = false;
    if (canRead)
        if (*canRead = fReadShutdown ||
            (fState == Listening
             ? fListeners[fNextListener].fBusy
             : fInputBuffer.Valid() > 0
            )
        )
            cond = true;
    if (canWrite)
        if (*canWrite = fWriteShutdown || fOutputBuffer.Free())
            cond = true;

    return cond;
}
```

⟨Overridden member functions for GUSIPPCSocket 1361⟩+≡ (1339) ◀1387

```
virtual int shutdown(int how);
```

⟨Member functions for class GUSIPPCSocket 1341⟩+≡ (1334) ◀1388 1391▶

```
int GUSIPPCSocket::shutdown(int how)
```

```
{
    if (!GUSI_SASSERT_CLIENT(how >= 0 && how < 3, "shutdown: 0,1, or 2\n"))
        return GUSISetPosixError(EINVAL);

    GUSISMState::Shutdown(how);

    return 0;
}
```

(Member functions for class GUSIPPCSocket 1341)+≡

(1334) ◀1390

```
GUSIPPCSocket::~GUSIPPCSocket()
{
    switch (fState) {
    case Listening:
        shutdown(2);
        (Shut down listening GUSIPPCSocket 1392)
        break;
    case Connected:
        shutdown(2);
        AddContext();
        while (fState == Connected)
            GUSIContext::Yield(true);
        RemoveContext();
        // Fall through
    case Closing:
        fSendPB.endParam.ioCompletion = sDoneProc;
        PPCEndAsync(&fSendPB.endParam);
        AddContext();
        while (fSendPB.endParam.ioResult == 1)
            GUSIContext::Yield(true);
        RemoveContext();
        break;
    }
    if (fPortRef && !--fPortRef->fRefCount) {
        fSendPB.closeParam.ioCompletion = sDoneProc;
        fSendPB.closeParam.portRefNum = fPortRef->fPort;
        PCCCloseAsync(&fSendPB.closeParam);
        AddContext();
        while (fSendPB.closeParam.ioResult == 1)
            GUSIContext::Yield(true);
        RemoveContext();
        delete fPortRef;
    }
}
```

(Shut down listening GUSIPPCSocket 1392)≡

(1391)

```
fInputBuffer.Lock();
for (int i = 0; i < fNumListeners; i++)
    if (fListeners[i].fBusy) {
        fSendPB.endParam.ioCompletion = sDoneProc;
        fSendPB.endParam.sessRefNum = fListeners[i].fSession;
        PPCEndAsync(&fSendPB.endParam);
        AddContext();
        while (fSendPB.endParam.ioResult == 1)
            GUSIContext::Yield(true);
        RemoveContext();
    }
}
```

Part IV

Library Specific Code

Chapter 33

The Interface to the MSL

This section interfaces GUSI to the Metrowerks Standard Library (MSL) by reimplementing various internal MSL routines. Consequently, some of the code used here is borrowed from the MSL code itself. The routines here are in three different categories:

- Overrides of MSL functions (all internal, as it happens).
- Implementations of ANSI library specific public GUSI functions like `fdopen()`.
- Implementations of ANSI library specific internal GUSI functions.

```
<GUSIMSL.h 1393>≡
#ifdef _GUSIMSL_
#define _GUSIMSL_

#endif /* _GUSIMSL_ */

<GUSIMSL.cp 1394>≡
#include "GUSIInternal.h"
#include "GUSIMSL.h"
#include "GUSIDescriptor.h"

#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

extern "C" {
<Prototypes for internal MSL functions 1399>
<Prototypes for MSL override functions 1395>
}

<Implementation of MSL override functions 1396>
<Implementation of ANSI library specific public GUSI functions 1416>
<Implementation of ANSI library specific internal GUSI functions 1418>
```

33.1 Implementation of MSL override functions

All opens from the ANSI C and C++ libraries eventually call `__open_file`, so that's one of our most important override candidates. We translate the file mode and call `open()`.

```
<Prototypes for MSL override functions 1395>≡ (1394) 1398⤵
    int __open_file(const char * name, __file_modes mode, __file_handle * handle);

<Implementation of MSL override functions 1396>≡ (1394) 1400⤵
    int __open_file(const char * name, __file_modes mode, __file_handle * handle)
    {
        int fd;
        int posixmode;

        <Translate mode to posixmode 1397>

        fd = open(name, posixmode);

        if (fd == -1)
            return __io_error;

        *handle = (unsigned) fd;

        return __no_io_error;
    }
```

Translation of the constants is pretty obvious.

```
(Translate mode to posixmode 1397)≡ (1396)
switch (mode.open_mode) {
case __must_exist:
    posixmode = 0;
    break;
case __create_if_necessary:
    posixmode = O_CREAT;
    break;
case __create_or_truncate:
    posixmode = O_CREAT | O_TRUNC;
    break;
}

switch (mode.io_mode) {
case __read:
    posixmode |= O_RDONLY;
    break;
case __write:
    posixmode |= O_WRONLY;
    break;
case __read_write:
    posixmode |= O_RDWR;
    break;
case __write | __append:
    posixmode |= O_WRONLY | O_APPEND;
    break;
case __read_write | __append:
    posixmode |= O_RDWR | O_APPEND;
    break;
}
```

`__open_temp_file` is used to open files that get deleted after they are closed. We simply use the underlying GUSI support for the POSIXish `open()` / `unlink()` idiom.

```
(Prototypes for MSL override functions 1395)+≡ (1394) ◀1395 1401▶
int __open_temp_file(__file_handle * handle);

(Prototypes for internal MSL functions 1399)≡ (1394) 1415▶
void __temp_file_name(char * name_str, void * fsspec);
extern __file_modes __temp_file_mode;
```

(Implementation of MSL override functions 1396)+≡ (1394) <1396 1402>

```
int __open_temp_file(__file_handle * handle)
{
    char    temp_name[L_tmpnam];
    FSSpec  spec;
    int     ioresult;

    __temp_file_name(temp_name, &spec);

    ioresult = __open_file(temp_name, __temp_file_mode, handle);

    if (ioresult == __no_io_error)
    {
        unlink(temp_name);
    }

    return(ioresult);
}
```

I/O operations in MSL lead to calls to `__read_file`, `__write_file`, `__position_file`, and `__close_file`, so we'll reimplement those.

(Prototypes for MSL override functions 1395)+≡ (1394) <1398 1403>

```
int __read_file(__file_handle handle, unsigned char * buffer, size_t * count, __idle_proc idle_proc)
```

(Implementation of MSL override functions 1396)+≡

(1394) <1400 1404>

```
int __read_file(__file_handle handle, unsigned char * buffer, size_t * count, __idle_proc)
{
    int result = read((int) handle, (char *) buffer, (int) *count);

    if (result < 0) {
        *count = 0;

        return __io_error;
    } else {
        *count = result;

        return result ? __no_io_error : __io_EOF;
    }
}
```

We treat the console variations almost identically to their file counterparts, except that we have to try to guess the correct handle.

(Prototypes for MSL override functions 1395)+≡ (1394) <1401 1405>

```
int __read_console(__file_handle handle, unsigned char * buffer, size_t * count, __idle_proc idle_proc)
```

(Implementation of MSL override functions 1396)+≡

(1394) <1402 1406>

```
int __read_console(__file_handle handle, unsigned char * buffer, size_t * count, __idle_proc idle_proc)
{
    return __read_file(handle, buffer, count, idle_proc);
}
```

(Prototypes for MSL override functions 1395)+≡

(1394) <1403 1407>

```
int __write_file(__file_handle handle, unsigned char * buffer, size_t * count, __idle_proc idle_proc)
```

```

<Implementation of MSL override functions 1396>+≡ (1394) <1404 1408>
int __write_file(__file_handle handle, unsigned char * buffer, size_t * count, __idle_proc)
{
    int result = write((int) handle, (char *) buffer, (int) *count);

    if (result < 0) {
        *count = 0;

        return __io_error;
    } else {
        *count = result;

        return __no_io_error;
    }
}

```

```

<Prototypes for MSL override functions 1395>+≡ (1394) <1405 1409>
int __write_console(__file_handle handle, unsigned char * buffer, size_t * count, __idle_proc idle_proc);

```

```

<Implementation of MSL override functions 1396>+≡ (1394) <1406 1410>
static __file_handle GuessHandle(__file_handle handle, unsigned char * buffer)
{
    if (handle)
        return handle;
    if (buffer > stderr->buffer && (buffer - stderr->buffer) < stderr->buffer_size)
        return 2;
    return 1;
}

int __write_console(__file_handle handle, unsigned char * buffer, size_t * count, __idle_proc idle_proc)
{
    return __write_file(GuessHandle(handle, buffer), buffer, count, idle_proc);
}

```

The only one of these presenting us with some subtlety is `__position_file`. Unless we catch the case of positioning at a place where the position already is, MSL will be unable to `fdopen()` any sockets.

```

<Prototypes for MSL override functions 1395>+≡ (1394) <1407 1411>
int __position_file(__file_handle handle, unsigned long * position, int mode, __idle_proc idle_proc);

```

```

<Implementation of MSL override functions 1396>+≡ (1394) <1408 1412>
int __position_file(__file_handle handle, unsigned long * position, int mode, __idle_proc)
{
    long result = lseek((int) handle, *position, mode);

    if (result < 0)
        if (errno == ESPIPE && !*position && mode != SEEK_END)
            *position = 0;
        else
            return __io_error;
    else
        *position = result;

    return __no_io_error;
}

```

<Prototypes for MSL override functions 1395>+≡ (1394) <1409 1413>
int __close_file(__file_handle handle);

<Implementation of MSL override functions 1396>+≡ (1394) <1410 1414>
int __close_file(__file_handle handle)
{
return close((int) handle) < 0 ? __io_error : __no_io_error;
}

__close_console is an undecidable function unless you build with MPW MSL libraries.

<Prototypes for MSL override functions 1395>+≡ (1394) <1411
int __close_console(__file_handle handle);

<Implementation of MSL override functions 1396>+≡ (1394) <1412
int __close_console(__file_handle handle)
{
return __close_file(handle);
}

33.2 Implementation of ANSI library specific public GUSI functions

fdopen() is an operation inherently tied to the ANSI library used.

<Prototypes for internal MSL functions 1399>+≡ (1394) <1399 1417>
FILE * __find_unopened_file();
// __handle_reopen is declared in <stdio.h>

<Implementation of ANSI library specific public GUSI functions 1416>≡ (1394)
FILE * fdopen(int fildes, const char *type)
{
FILE *str;

if (!(str = __find_unopened_file()))
return(0);

return(__handle_reopen(fildes, type, str));
}

33.3 Implementation of ANSI library specific internal GUSI functions

While there is a function __close_all, it turns out that calling it is problematic, at least under CodeWarrior Pro 4, because it may fail catastrophically upon being called a second time.

<Prototypes for internal MSL functions 1399>+≡ (1394) <1415
void __flush_all(void);

<Implementation of ANSI library specific internal GUSI functions 1418>≡ (1394) 1419>
void GUSIStdioClose() { }
void GUSIStdioFlush() { __flush_all(); }

(Implementation of ANSI library specific internal GUSI functions 1418)+≡ (1394) <1418

```
static void MSLSetupStdio(int fd, FILE * f)
{
    f->handle          = fd;
    f->position_proc    = __position_file;
    f->read_proc        = __read_file;
    f->write_proc       = __write_file;
    f->close_proc       = __close_file;
}
```

```
void GUSISetupConsoleStdio()
{
    MSLSetupStdio(0, stdin);
    MSLSetupStdio(1, stdout);
    MSLSetupStdio(2, stderr);
}
```


Chapter 34

The Interface to the MPW Stdio library

This section interfaces GUSI to the variant of the Stdio library used for SC and MrC.

<GUSIMPWStdio.h 1420>≡

```
#ifndef _GUSIMPWStdio_
#define _GUSIMPWStdio_
```

```
#endif /* _GUSIMPWStdio_ */
```

<GUSIMPWStdio.cp 1421>≡

```
#include "GUSIInternal.h"
#include "GUSIDescriptor.h"
```

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
```

<Implementation of MPW ANSI library specific public GUSI functions 1422>

<Implementation of internal GUSI functions for MPW Stdio 1428>

34.1 Implementation of MPW ANSI library specific public GUSI functions

We have to reimplement a substantial portion of the MPW stdio library because of shared library problems and incompatible macro definitions for `open()` flags.

We'll start with a couple of internals. `findfile()` searches for a free `FILE` structure.

(Implementation of MPW ANSI library specific public GUSI functions 1422)≡ (1421) 1423▷

```
static FILE * findfile()
{
    FILE *      stream;

    for (stream = _iob; stream < _iob+_NFILE; ++stream)
        if (!(stream->_flag & (_IOREAD | _IOWRT | _IORW)))
            return stream;

    return NULL;
}
```

`fdreopen()` initializes a `FILE` for use, which is actually quite simple.

(Implementation of MPW ANSI library specific public GUSI functions 1422)+≡ (1421) ◀1422 1424▷

```
static FILE *fdreopen(int fd, short flags, FILE* stream)
{
    stream->_cnt      = 0;
    stream->_ptr      = NULL;
    stream->_base     = NULL;
    stream->_end      = NULL;
    stream->_size     = NULL;
    stream->_flag     = flags;
    stream->_file     = fd;

    return stream;
}
```

mode2flags parses the mode string and translates it to open() and stdio flags.

(Implementation of MPW ANSI library specific public GUSI functions 1422)+≡ (1421) <1423 1425>

```
static int mode2flags(const char * mode, int * openflags, int * stdioflags)
{
    bool    read_write  = mode[1] == '+' || mode[2] == '+';

    *openflags  = 0;
    *stdioflags = 0;

    switch (mode[0]) {
    case 'r':
        *openflags |= read_write ? O_RDWR : O_RDONLY;
        *stdioflags |= read_write ? _IORW : _IOREAD;
        break;
    case 'w':
        *openflags |= (read_write ? O_RDWR : O_WRONLY) | O_CREAT | O_TRUNC;
        *stdioflags |= read_write ? _IORW : _IOWRT;
        break;
    case 'a':
        *openflags |= (read_write ? O_RDWR : O_WRONLY) | O_CREAT | O_APPEND;
        *stdioflags |= read_write ? _IORW : _IOWRT;
        break;
    default:
        return -1;
    }

    return 0;
}
```

fopen, freopen, and fdopen can then be composed from the above.

(Implementation of MPW ANSI library specific public GUSI functions 1422)+≡ (1421) <1424 1426>

```
FILE *fopen(const char *filename, const char *mode)
{
    FILE * stream;
    int    flags;
    int    ioflags;
    int    fd;

    if ((stream = findfile())
        && mode2flags(mode, &flags, &ioflags) >= 0
        && (fd = open(filename, flags)) >= 0
    )
        return fdreopen(fd, ioflags, stream);
    else
        return NULL;
}
```

(Implementation of MPW ANSI library specific public GUSI functions 1422)+≡ (1421) ◀1425 1427▶

```
FILE *freopen(const char *filename, const char *mode, FILE *stream)
{
    int    flags;
    int    ioflags;
    int    fd;

    flags = errno;
    fclose(stream);
    errno = flags;

    if (mode2flags(mode, &flags, &ioflags) >= 0
        && (fd = open(filename, flags)) >= 0
    )
        return fdreopen(fd, ioflags, stream);
    else
        return NULL;
}
```

(Implementation of MPW ANSI library specific public GUSI functions 1422)+≡ (1421) ◀1426

```
FILE *fdopen(int fd, const char *mode)
{
    FILE * stream;
    int    flags;
    int    ioflags;

    if ((stream = findfile())
        && mode2flags(mode, &flags, &ioflags) >= 0
    )
        return fdreopen(fd, ioflags, stream);
    else
        return NULL;
}
```

34.2 Implementation of internal GUSI functions for MPW Stdio

(Implementation of internal GUSI functions for MPW Stdio 1428)+≡

(1421) 1429▶

```
void GUSIStdioClose()
{
    for (FILE * f = _iob; f<_iob+_NFILE; ++f)
        if (f->_flag & (_IOREAD|_IOWRT))
            fclose(f);
}

void GUSIStdioFlush() { fflush(NULL); }
```

MPW Stdio already does everything to get started, no need for us to get in the act. With MrC, however, we have to pay attention that ioctl does not get stripped, otherwise the library will call a bad version internally.

```
<Implementation of internal GUSI functions for MPW Stdio 1428>+≡ (1421) <1428
static void * sDontStrip;

void GUSISetupConsoleStdio()
{
    sDontStrip = ioctl;
}
```


Chapter 35

The Interface to the Sflo library

This section interfaces GUSI to the Sflo library.

```
<GUSISflo.h 1430>≡
    #ifndef _GUSISflo_
    #define _GUSISflo_

    #endif /* _GUSISflo_ */

<GUSISflo.cp 1431>≡
    #ifdef __MWERKS__
    #define __cstdrio__ 1
    #else
    #define __STDIO__ 1
    #endif

    #include "GUSIInternal.h"
    #include <sfhdr.h>
    #include "GUSIDescriptor.h"

<Implementation of internal GUSI functions for Sflo 1432>
```

35.1 Implementation of internal GUSI functions for Sflo

GUSISfloFlushClose was adapted from the Sflo internal _sfall function.

(Implementation of internal GUSI functions for Sflo 1432)≡ (1431) 1433▷

```
static int GUSISfloFlushClose(bool close)
{
    reg Sfpool_t    *p, *next;
    reg Sflo_t* f;
    reg int         n, rv;
    reg int         nsync, count, loop;
#define MAXLOOP 3

    for(loop = 0; loop < MAXLOOP; ++loop)
    {
        rv = nsync = count = 0;
        for(p = &_Sfpool; p; p = next)
        {
            /* find the next legitimate pool */
            for(next = p->next; next; next = next->next)
                if(next->n_sf > 0)
                    break;

            /* walk the streams for _Sfpool only */
            for(n = 0; n < ((p == &_Sfpool) ? p->n_sf : 1); ++n)
            {
                count += 1;
                f = p->sf[n];

                if(f->flags&SF_STRING )
                    goto did_sync;
                if(SFFROZEN(f))
                    continue;
                if( (close ? sfclose(f) : sfsync(f)) < 0)
                    rv = -1;

                did_sync:
                nsync += 1;
            }
        }

        if(nsync == count)
            break;
    }

    return rv;
}

void GUSISstdioClose() { GUSISfloFlushClose(true); }
void GUSISstdioFlush() { GUSISfloFlushClose(false); }
```

Sflo already does everything to get started, no need for us to get in the act.

(Implementation of internal GUSI functions for Sflo 1432)+≡ (1431) ◀1432

```
void GUSISetupConsoleStdio()
{
}
```


Chapter 36

MPW Support

In MPW tools, we have to direct some of the I/O operations to the standard library functions, which we otherwise try to avoid as much as possible. Getting at those library calls is a bit tricky: For 68K, we rename entries in the MPW glue library, while for PowerPC, we look up the symbols dynamically.

MPW support is installed implicitly through `GUSISetupConsoleDescriptors`

```
<GUSIMPW.h 1434>≡  
#ifndef _GUSIMPW_  
#define _GUSIMPW_  
  
#endif /* _GUSIMPW_ */
```

```

<GUSIMPW.cp 1435>≡
#include "GUSIInternal.h"
#include "GUSIMPW.h"
#include "GUSIDevice.h"
#include "GUSIDescriptor.h"
#include "GUSIMacFile.h"
#include "GUSIBasics.h"
#include "GUSIDdiag.h"
#include "GUSITimer.h"
#include "GUSIConfig.h"

#include <fcntl.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <errno.h>

#include <CursorCtl.h>

#include <CodeFragments.h>

<MPW library functions 1438>
<Declaration of class GUSIMPWSocket 1437>
<Declaration of class GUSIMPWDevice 1436>
<Member functions for class GUSIMPWSocket 1453>
<Member functions for class GUSIMPWDevice 1441>
<MPW implementation of GUSIGetEnv 1472>
<MPW spin hook 1451>
<MPW implementation of GUSISetupConsoleDescriptors 1452>

```

36.1 Definition of GUSIMPWDevice

A GUSIMPWDevice is a singleton class supporting the standard open and faccess operations and a domain specific stdopen operation to support opening already-open descriptors.

```

<Declaration of class GUSIMPWDevice 1436>≡ (1435)
class GUSIMPWDevice : public GUSIDevice {
public:
    static GUSIMPWDevice * Instance();

    <Overridden member functions for class GUSIMPWDevice 1443>

    GUSISocket * stdopen(int fd, int flags);
private:
    GUSIMPWDevice() {}

    static GUSIMPWDevice * sInstance;
};

```

36.2 Definition of GUSIMPWSocket

A GUSIMPWSocket is a proxy class for an MPW file descriptor.

```
<Declaration of class GUSIMPWSocket 1437>≡ (1435)
class GUSIMPWSocket : public GUSISocket {
public:
    ~GUSIMPWSocket();

    <Overridden member functions for class GUSIMPWSocket 1454>
protected:
    friend class GUSIMPWDevice;

    GUSIMPWSocket(int fd);
private:
    int    fFD;
};
```

36.3 Interfacing to MPW library routines

As mentioned above, our interface strategy differs for 68K and PPC code. We try to handle this with a macro.

```
<MPW library functions 1438>≡ (1435)
extern "C" {
    #if !defined(powerc) && !defined(__powerc)
    #define DECL_stdlib(name, pname, ret, args) ret name args;
    #ifdef __MWERKS__
    #pragma pointers_in_D0
    #endif
    <Declarations for MPW functions 1440>
    #ifdef __MWERKS__
    #pragma pointers_in_A0
    #endif
    static bool ConnectToMPWLibrary() { return true; }
    #else
    #define DECL_stdlib(name, pname, ret, args) ret (*name) args;
    <Declarations for MPW functions 1440>
    <Connecting to the StdCLib code fragment 1439>
    #endif
}
```

On PowerPC, we have to connect to the dynamic library (which, in principle, can fail).

```

(Connecting to the StdCLib code fragment 1439)≡ (1438)
static bool sConnected;

static void DoConnectToMPWLibrary()
{
    CFragConnectionID StdCLib;
    CFragSymbolClass symClass;
    Ptr whoCares;
    Str255 error;

    if (GetSharedLibrary(
        StringPtr("\pStdCLib"), kPowerPCCFragArch, kLoadCFrag, &StdCLib, &whoCares, error)
    )
        return;

#undef DECL_stdlib
#define DECL_stdlib(name, pname, ret, args) \
    if (FindSymbol(StdCLib, pname, (Ptr *) &name, &symClass)) \
        goto failed;
    (Declarations for MPW functions 1440)

    sConnected = true;

    return;

failed:
#undef DECL_stdlib
#define DECL_stdlib(name, pname, ret, args) name = 0;

    (Declarations for MPW functions 1440)
}

static bool ConnectToMPWLibrary()
{
    if (!sConnected)
        DoConnectToMPWLibrary();
    return sConnected;
}

```

Now we only have to declare the list once, and can reuse it numerous times.

```

(Declarations for MPW functions 1440)≡ (1438 1439)
DECL_stdlib(MPW_open, "\popen", int, (const char * name, int flags))
DECL_stdlib(MPW_close, "\pclose", int, (int s))
DECL_stdlib(MPW_read, "\pread", int, (int s, char *buffer, unsigned buflen))
DECL_stdlib(MPW_write, "\pwrite", int, (int s, char *buffer, unsigned buflen))
DECL_stdlib(MPW_fcntl, "\pfcntl", int, (int s, unsigned int cmd, int arg))
DECL_stdlib(MPW_ioctl, "\pioclt", int, (int d, unsigned int request, long *argp))
DECL_stdlib(MPW_lseek, "\plseek", long, (int fd, long offset, int whence))
DECL_stdlib(MPW_faccess, "\pfaccess", int, (char *fileName, unsigned int cmd, long * arg))
DECL_stdlib(MPW_getenv, "\pgetenv", char *, (const char *env))

```

36.4 Implementation of GUSIMPWDevice

GUSIMPWDevice is a singleton class.

```
(Member functions for class GUSIMPWDevice 1441)≡ (1435) 1442▷  
GUSIMPWDevice * GUSIMPWDevice::sInstance;
```

```
(Member functions for class GUSIMPWDevice 1441)+≡ (1435) ◁1441 1444▷  
GUSIMPWDevice * GUSIMPWDevice::Instance()  
{  
    if (!sInstance)  
        sInstance = new GUSIMPWDevice();  
    return sInstance;  
}
```

GUSIMPWDevice is prepared to handle an open() on a limited set of names.

```
(Overridden member functions for class GUSIMPWDevice 1443)≡ (1436) 1447▷  
virtual bool Want(GUSIFFileToken & file);
```

```
(Member functions for class GUSIMPWDevice 1441)+≡ (1435) ◁1442 1445▷  
bool GUSIMPWDevice::Want(GUSIFFileToken & file)  
{  
    switch (file.WhichRequest()) {  
    case GUSIFFileToken::kWillOpen:  
        return file.IsDevice() && (file.StrStdStream(file.Path()) > -1);  
    default:  
        return false;  
    }  
}
```

open translates the file flags, opens the file, and passes the resulting descriptor to stdopen.

The values of some of the flags that MPW uses differ a bit from the ones used in our headers.

```
(Member functions for class GUSIMPWDevice 1441)+≡ (1435) ◁1444 1446▷  
#define MPW_O_RDONLY 0 /* Bits 0 and 1 are used internally */  
#define MPW_O_WRONLY 1 /* Values 0..2 are historical */  
#define MPW_O_RDWR 2 /* NOTE: it goes 0, 1, 2, *!* 8, 16, 32, ... */  
#define MPW_O_APPEND (1<< 3) /* append (writes guaranteed at the end) */  
#define MPW_O_RSRC (1<< 4) /* Open the resource fork */  
#define MPW_O_ALIAS (1<< 5) /* Open alias file */  
#define MPW_O_CREAT (1<< 8) /* Open with file create */  
#define MPW_O_TRUNC (1<< 9) /* Open with truncation */  
#define MPW_O_EXCL (1<<10) /* w/ O_CREAT: Exclusive "create-only" */  
#define MPW_O_BINARY (1<<11) /* Open as a binary stream */  
#define MPW_O_NRESOLVE (1<<14) /* Don't resolve any aliases */
```

TranslateOpenFlags translates the header flags into the MPW flags.

(Member functions for class GUSIMPWDevice 1441)+≡ (1435) <1445 1448>
static int TranslateOpenFlags(int mode)

```
{
    int mpwMode;

    switch (mode & 3) {
    case O_RDWR:
        mpwMode = MPW_O_RDWR;
        break;
    case O_RDONLY:
        mpwMode = MPW_O_RDONLY;
        break;
    case O_WRONLY:
        mpwMode = MPW_O_WRONLY;
        break;
    }
    if (mode & O_APPEND)
        mpwMode |= MPW_O_APPEND;
    if (mode & O_CREAT)
        mpwMode |= MPW_O_CREAT;
    if (mode & O_EXCL)
        mpwMode |= MPW_O_EXCL;
    if (mode & O_TRUNC)
        mpwMode |= MPW_O_TRUNC;

    return mpwMode;
}
```

(Overridden member functions for class GUSIMPWDevice 1443)+≡ (1436) <1443
virtual GUSISocket * open(GUSIFileToken &, int flags);

(Member functions for class GUSIMPWDevice 1441)+≡ (1435) <1446 1449>
GUSISocket * GUSIMPWDevice::open(GUSIFileToken & file, int flags)
{
 if (!ConnectToMPWLibrary())
 return GUSISetPosixError(ENOEXEC), static_cast<GUSISocket *>(nil);

 int fd = MPW_open(file.Path(), TranslateOpenFlags(flags));

 if (fd == -1)
 return static_cast<GUSISocket *>(nil);
 else
 return stdopen(fd, flags);
}

stdopen handles the GUSI side of the opening.

```
<Member functions for class GUSIMPWDevice 1441>+≡ (1435) <1448
extern int StandAlone;

GUSISocket * GUSIMPWDevice::stdopen(int fd, int flags)
{
    if (!ConnectToMPWLibrary())
        return GUSISetPosixError(ENOEXEC), static_cast<GUSISocket *>(nil);

    <Open and return a MacFileSocket if possible 1450>

    GUSISocket * sock = new GUSIMPWSocket(fd);

    return sock ? sock : (GUSISetPosixError(ENOMEM), static_cast<GUSISocket *>(nil));
}
```

Our support of MacOS files is far superiour to our MPW console support, so whenever we find that in fact we're talking to a real file, we switch to using a MacFileSocket instead. This whole parocedure does not apply to SIOW applications: The initial sockets 0, 1, and 2 get closed right away and calling anything else on them would be counterproductive.

```
<Open and return a MacFileSocket if possible 1450>≡ (1449)
short fRef;

if (!StandAlone
    && MPW_ioctl(fd, FIOINTERACTIVE, nil) == -1
    && MPW_ioctl(fd, FIOREFNUM, (long *) &fRef) != -1
) {
    MPW_close(fd);
    return GUSIMacFileDevice::Instance()->open(fRef, flags);
}

<MPW spin hook 1451>≡ (1435)
void GUSIMPWSpin(bool wait)
{
    static GUSITimer sSpinDue(false);

    GUSIConfiguration::Instance()->AutoInitGraf();

    if (!sSpinDue.Primed()) {
        RotateCursor(32);
        sSpinDue.Sleep(125, true);
    } else if (wait)
        GUSIHandleNextEvent(600);
}
```

As opposed to an application, an MPW tool connects to the three standard descriptors on startup.

```

<MPW implementation of GUSISetupConsoleDescriptors 1452>≡ (1435)
void GUSISetupConsoleDescriptors()
{
    GUSIMPWDevice *      mpw      = GUSIMPWDevice::Instance();
    GUSIDescriptorTable * table = GUSIDescriptorTable::Instance();

    GUSIDeviceRegistry::Instance()->AddDevice(mpw);

    if (!(*table)[0]) {
        table->InstallSocket(mpw->stdopen(0, O_RDONLY));
        table->InstallSocket(mpw->stdopen(1, O_WRONLY));
        table->InstallSocket(mpw->stdopen(2, O_WRONLY));
    }

    GUSISetHook(GUSI_EventHook+mouseDown, (GUSIHook)-1);
    GUSISetHook(GUSI_SpinHook, (GUSIHook)GUSIMPWSpin);
}

```

36.5 Implementation of GUSIMPWSocket

A GUSIMPWSocket acts as a wrapper for a file descriptor from the MPW library.

```

<Member functions for class GUSIMPWSocket 1453>≡ (1435) 1455»
GUSIMPWSocket::GUSIMPWSocket(int fd)
    : fFD(fd)
{
}
GUSIMPWSocket::~~GUSIMPWSocket()
{
    MPW_close(fFD);
}

```

Some member functions are fairly trivial wrappers.

```

<Overridden member functions for class GUSIMPWSocket 1454>≡ (1437) 1456»
ssize_t read(const GUSIScatterer & buffer);

```

```

<Member functions for class GUSIMPWSocket 1453>+≡ (1435) «1453 1457»
ssize_t GUSIMPWSocket::read(const GUSIScatterer & buffer)
{
    GUSIStdioFlush();
    GUSIConfiguration::Instance()->AutoSpin();
    return buffer.SetLength(MPW_read(fFD, (char *) buffer.Buffer(), (unsigned)buffer.Length()));
}

```

```

<Overridden member functions for class GUSIMPWSocket 1454>+≡ (1437) «1454 1458»
ssize_t write(const GUSIGatherer & buffer);

```

```

<Member functions for class GUSIMPWSocket 1453>+≡ (1435) «1455 1459»
ssize_t GUSIMPWSocket::write(const GUSIGatherer & buffer)
{
    GUSIConfiguration::Instance()->AutoSpin();
    return MPW_write(fFD, (char *) buffer.Buffer(), (unsigned)buffer.Length());
}

```


<Overridden member functions for class GUSIMPWSocket 1454>+≡ (1437) ◀1456 1460▶
 virtual off_t lseek(off_t offset, int whence);

<Member functions for class GUSIMPWSocket 1453>+≡ (1435) ◀1457 1461▶
 off_t GUSIMPWSocket::lseek(off_t offset, int whence)
 {
 return MPW_lseek(fFD, offset, (long)whence);
 }

fcntl and ioctl have to get their final arguments from va_lists.

<Overridden member functions for class GUSIMPWSocket 1454>+≡ (1437) ◀1458 1462▶
 virtual int fcntl(int cmd, va_list arg);

<Member functions for class GUSIMPWSocket 1453>+≡ (1435) ◀1459 1463▶
 int GUSIMPWSocket::fcntl(int cmd, va_list arg)
 {
 return MPW_fcntl(fFD, cmd, va_arg(arg, int));
 }

<Overridden member functions for class GUSIMPWSocket 1454>+≡ (1437) ◀1460 1464▶
 virtual int ioctl(unsigned int request, va_list arg);

<Member functions for class GUSIMPWSocket 1453>+≡ (1435) ◀1461 1465▶
 int GUSIMPWSocket::ioctl(unsigned int request, va_list arg)
 {
 return MPW_ioctl(fFD, request, va_arg(arg, long *));
 }

ftruncate translates into an ioctl request.

<Overridden member functions for class GUSIMPWSocket 1454>+≡ (1437) ◀1462 1466▶
 virtual int ftruncate(off_t offset);

<Member functions for class GUSIMPWSocket 1453>+≡ (1435) ◀1463 1467▶
 int GUSIMPWSocket::ftruncate(off_t offset)
 {
 return MPW_ioctl(fFD, FIOSETEOF, (long *) offset);
 }

Since we know we're running on a pseudodevice, we can pass on that fact.

<Overridden member functions for class GUSIMPWSocket 1454>+≡ (1437) ◀1464 1468▶
 virtual int fstat(struct stat * buf);

<Member functions for class GUSIMPWSocket 1453>+≡ (1435) ◀1465 1469▶
 int GUSIMPWSocket::fstat(struct stat * buf)
 {
 GUSISocket::fstat(buf);
 buf->st_mode = S_IFCHR | 0666;

 return 0;
 }

And we also know we're a TTY.

<Overridden member functions for class GUSIMPWSocket 1454>+≡ (1437) ◀1466 1470▶
 virtual int isatty();

```

<Member functions for class GUSIMPWSocket 1453>+≡ (1435) <1467 1471>
int GUSIMPWSocket::isatty()
{
    return 1;
}

```

We have no choice but to be optimistic and claim that we are always ready for reading and writing.

```

<Overridden member functions for class GUSIMPWSocket 1454>+≡ (1437) <1468
virtual bool select(bool * canRead, bool * canWrite, bool * exception);

```

```

<Member functions for class GUSIMPWSocket 1453>+≡ (1435) <1469
bool GUSIMPWSocket::select(bool * canRead, bool * canWrite, bool *)
{
    bool cond = false;
    if (canRead)
        cond = *canRead = true;
    if (canWrite)
        cond = *canWrite = true;

    return cond;
}

```

```

<MPW implementation of GUSIGetEnv 1472>≡ (1435)
char *GUSIGetEnv(const char * name)
{
    if (!ConnectToMPWLibrary())
        return static_cast<char *>(nil);
    return MPW_getenv(name);
}

```

Chapter 37

SIOUX Support

To combine GUSI with SIOUX, terminal I/O needs to interface with the SIOUX event handling.

SIOUX support is installed implicitly through GUSISetupConsoleDescriptors

```
<GUSISIOUX.h 1473>≡
    #ifndef _GUSISIOUX_
    #define _GUSISIOUX_

    #endif /* _GUSISIOUX_ */

<GUSISIOUX.cp 1474>≡
    #include "GUSIInternal.h"
    #include "GUSISIOUX.h"
    #include "GUSIDevice.h"
    #include "GUSIDescriptor.h"
    #include "GUSIBasics.h"
    #include "GUSIDiag.h"

    #include <LowMem.h>

    #include <fcntl.h>
    #include <sys/stat.h>
    #include <sys/ioctl.h>
    #include <errno.h>
    #include <console.h>

    <Declaration of class GUSISIOUXSocket 1476>
    <Declaration of class GUSISIOUXDevice 1475>
    <Member functions for class GUSISIOUXSocket 1484>
    <Member functions for class GUSISIOUXDevice 1477>
    <SIOUX implementation of GUSISetupConsoleDescriptors 1483>
```

37.1 Definition of GUSISIOUXDevice

A GUSISIOUXDevice is a singleton class supporting the standard open operation.

```
<Declaration of class GUSISIOUXDevice 1475>≡ (1474)
class GUSISIOUXDevice : public GUSIDevice {
public:
    static GUSISIOUXDevice *    Instance();

    <Overridden member functions for class GUSISIOUXDevice 1479>
private:
    GUSISIOUXDevice()           {}

    static GUSISIOUXDevice *    sInstance;
};
```

37.2 Definition of GUSISIOUXSocket

A GUSISIOUXSocket is another singleton class: There is only one SIOUX console per application.

```
<Declaration of class GUSISIOUXSocket 1476>≡ (1474)
class GUSISIOUXSocket : public GUSISocket {
public:
    ~GUSISIOUXSocket();

    <Overridden member functions for class GUSISIOUXSocket 1486>

    static GUSISIOUXSocket *    Instance();
private:
    static GUSISIOUXSocket *    sInstance;

    GUSISIOUXSocket();
};
```

37.3 Implementation of GUSISIOUXDevice

GUSISIOUXDevice is a singleton class.

```
<Member functions for class GUSISIOUXDevice 1477>≡ (1474) 1478▷
    GUSISIOUXDevice * GUSISIOUXDevice::sInstance;

<Member functions for class GUSISIOUXDevice 1477>+≡ (1474) ◁1477 1480▷
    GUSISIOUXDevice * GUSISIOUXDevice::Instance()
    {
        if (!sInstance)
            sInstance = new GUSISIOUXDevice();
        return sInstance;
    }
```

GUSISIOUXDevice is prepared to handle an open() on a limited set of names.

```
<Overridden member functions for class GUSISIOUXDevice 1479>≡ (1475) 1481▷
    virtual bool Want(GUSIFileToken & file);
```

```

⟨Member functions for class GUSISIOUXDevice 1477⟩+≡ (1474) ◀1478 1482▶
bool GUSISIOUXDevice::Want(GUSIFileToken & file)
{
    switch (file.WhichRequest()) {
    case GUSIFileToken::kWillOpen:
        return !file.IsFile() && (file.StrStdStream(file.Path()) > -1);
    default:
        return false;
    }
}

```

open returns the sole instance of GUSISIOUXSocket.

```

⟨Overridden member functions for class GUSISIOUXDevice 1479⟩+≡ (1475) ◀1479
virtual GUSISocket * open(GUSIFileToken &, int flags);

```

```

⟨Member functions for class GUSISIOUXDevice 1477⟩+≡ (1474) ◀1480
GUSISocket * GUSISIOUXDevice::open(GUSIFileToken &, int)
{
    return GUSISIOUXSocket::Instance();
}

```

A SIOUX application connects to the three standard descriptors on startup.

```

⟨SIOUX implementation of GUSISetupConsoleDescriptors 1483⟩≡ (1474)
void GUSISetupConsoleDescriptors()
{
    GUSIDescriptorTable * table = GUSIDescriptorTable::Instance();
    GUSISIOUXSocket * SIOUX = GUSISIOUXSocket::Instance();

    table->InstallSocket(SIOUX);
    table->InstallSocket(SIOUX);
    table->InstallSocket(SIOUX);
}

```

37.4 Implementation of GUSISIOUXSocket

A GUSISIOUXSocket is a dataless wrapper for the SIOUX library. To protect the sole instance from ever being deleted again, we artificially raise its reference count.

```

⟨Member functions for class GUSISIOUXSocket 1484⟩≡ (1474) 1485▶
GUSISIOUXSocket * GUSISIOUXSocket::sInstance;

GUSISIOUXSocket * GUSISIOUXSocket::Instance()
{
    if (!sInstance)
        if (sInstance = new GUSISIOUXSocket)
            sInstance->AddReference();

    return sInstance;
}

```

On construction, we make sure to install the SIOUX event handler for all relevant elements.

```
(Member functions for class GUSISIOUXSocket 1484)+≡ (1474) <1484 1487>
// This declaration lies about the return type
extern "C" void SIOUXHandleOneEvent(EventRecord *userevent);
```

```
GUSISIOUXSocket::GUSISIOUXSocket()
{
    InstallConsole(0);
    GUSISetHook(GUSI_EventHook+nullEvent, (GUSIHook)SIOUXHandleOneEvent);
    GUSISetHook(GUSI_EventHook+mouseDown, (GUSIHook)SIOUXHandleOneEvent);
    GUSISetHook(GUSI_EventHook+mouseUp, (GUSIHook)SIOUXHandleOneEvent);
    GUSISetHook(GUSI_EventHook+updateEvt, (GUSIHook)SIOUXHandleOneEvent);
    GUSISetHook(GUSI_EventHook+diskEvt, (GUSIHook)SIOUXHandleOneEvent);
    GUSISetHook(GUSI_EventHook+activateEvt, (GUSIHook)SIOUXHandleOneEvent);
    GUSISetHook(GUSI_EventHook+osEvt, (GUSIHook)SIOUXHandleOneEvent);
}
GUSISIOUXSocket::~GUSISIOUXSocket()
{
    RemoveConsole();
}
```

Some member functions are fairly trivial wrappers.

```
(Overridden member functions for class GUSISIOUXSocket 1486)+≡ (1476) 1488>
ssize_t read(const GUSIScatterer & buffer);
```

```
(Member functions for class GUSISIOUXSocket 1484)+≡ (1474) <1485 1489>
ssize_t GUSISIOUXSocket::read(const GUSIScatterer & buffer)
{
    GUSIStdioFlush();
    return buffer.SetLength(
        ReadCharsFromConsole((char *) buffer.Buffer(), (int)buffer.Length()));
}
```

```
(Overridden member functions for class GUSISIOUXSocket 1486)+≡ (1476) <1486 1490>
ssize_t write(const GUSIGatherer & buffer);
```

```
(Member functions for class GUSISIOUXSocket 1484)+≡ (1474) <1487 1491>
ssize_t GUSISIOUXSocket::write(const GUSIGatherer & buffer)
{
    return WriteCharsToConsole((char *) buffer.Buffer(), (int)buffer.Length());
}
```

```
(Overridden member functions for class GUSISIOUXSocket 1486)+≡ (1476) <1488 1492>
virtual int ioctl(unsigned int request, va_list arg);
```

```
(Member functions for class GUSISIOUXSocket 1484)+≡ (1474) <1489 1493>
int GUSISIOUXSocket::ioctl(unsigned int request, va_list)
{
    switch (request) {
        case FIOINTERACTIVE:
            return 0;
        default:
            return GUSISetPosixError(EOPNOTSUPP);
    }
}
```

Since we know we're running on a pseudodevice, we can pass on that fact.

```
<Overridden member functions for class GUSISIOUXSocket 1486>+≡ (1476) <1490 1494>  
virtual int fstat(struct stat * buf);
```

```
<Member functions for class GUSISIOUXSocket 1484>+≡ (1474) <1491 1495>  
int GUSISIOUXSocket::fstat(struct stat * buf)  
{  
    GUSISocket::fstat(buf);  
    buf->st_mode = S_IFCHR | 0666;  
  
    return 0;  
}
```

And we also know we're a TTY.

```
<Overridden member functions for class GUSISIOUXSocket 1486>+≡ (1476) <1492 1496>  
virtual int isatty();
```

```
<Member functions for class GUSISIOUXSocket 1484>+≡ (1474) <1493 1497>  
int GUSISIOUXSocket::isatty()  
{  
    return 1;  
}
```

select requires a walk of the event queue.

```
<Overridden member functions for class GUSISIOUXSocket 1486>+≡ (1476) <1494  
bool select(bool * canRead, bool * canWrite, bool *);
```

```
<Member functions for class GUSISIOUXSocket 1484>+≡ (1474) <1495  
static bool input_pending()  
{  
    QHdrPtr eventQueue = LMGetEventQueue();  
    EvQElPtr element = (EvQElPtr)eventQueue->qHead;  
  
    // now, count the number of pending keyDown events.  
    while (element != nil) {  
        if (element->evtQWhat == keyDown || element->evtQWhat == autoKey)  
            return true;  
        element = (EvQElPtr)element->qLink;  
    }  
  
    return false;  
}
```

```
bool GUSISIOUXSocket::select(bool * canRead, bool * canWrite, bool *)  
{  
    bool cond = false;  
    if (canRead)  
        if (*canRead = input_pending())  
            cond = true;  
    if (canWrite)  
        cond = *canWrite = true;  
  
    return cond;  
}
```


Chapter 38

SIOU Support

SIOU support is based on MPW support, adding a few event hooks so update and activate events get handled during blocking calls. SIOU support is installed implicitly through GUSIDefaultSetupConsole.

```
<GUSISIOU.h 1498>≡
    #ifndef _GUSISIOU_
    #define _GUSISIOU_

    #endif /* _GUSISIOU_ */

<GUSISIOU.cp 1499>≡
    #include "GUSIInternal.h"
    #include "GUSISIOU.h"
    #include "GUSIBasics.h"
    #include "GUSIDescriptor.h"

    #include <Events.h>
    #include <Windows.h>

    extern "C" {
    <Prototypes for internal SIOU functions 1500>
    }

    <Event handlers for SIOU 1501>
    <SIOU implementation of GUSIDefaultSetupConsole 1506>
```

The activate and suspend/resume handlers are very similar and both ultimately call `_DoActivate`. For reasons that I don't quite remember anymore, we are a bit more conservative about treating the current port than SIOU itself is.

```
<Prototypes for internal SIOU functions 1500>≡ (1499) 1502▷
    void _DoActivate(WindowPtr win, int activate);
```

(Event handlers for SLOW 1501)≡

(1499) 1503»

```
static void GUSIDoActivate(WindowPtr win, bool activate)
{
    GrafPtr port;
    GetPort(&port);
    if (win)
        SetPort(win);
    _DoActivate(win, activate);
    SetPort(port);
}

static void GUSISLOWActivate(EventRecord * ev)
{
    GUSIDoActivate(reinterpret_cast<WindowPtr>(ev->message), ev->modifiers & activeFlag);
}
```

As a slight extra complication, not all OS events are suspend/resume events.

(Prototypes for internal SLOW functions 1500)+≡

(1499) «1500 1504»

```
void _DoIdle();
```

(Event handlers for SLOW 1501)+≡

(1499) «1501 1505»

```
static void GUSISLOWSusRes(EventRecord * ev)
{
    switch ((ev->message >> 24) & 0xFF) {
    case suspendResumeMessage:
        GUSIDoActivate(FrontWindow(), ev->message & resumeFlag);
        break;
    case mouseMovedMessage:
        _DoIdle();
        break;
    }
}
```

The update handler similarly calls _DoUpdate.

(Prototypes for internal SLOW functions 1500)+≡

(1499) «1502

```
void _DoUpdate(WindowPtr win);
```

(Event handlers for SLOW 1501)+≡

(1499) «1503

```
static void GUSISLOWUpdate(EventRecord * ev)
{
    _DoUpdate((WindowPtr) ev->message);
}
```

This is one of the rare occasions where overriding `GUSIDefaultConsole` makes sense. This way, we can reuse the MPW version of `GUSISetupConsoleDescriptors`.

```
<SIOW implementation of GUSIDefaultSetupConsole 1506>≡ (1499)
static void GUSISetupSIOW()
{
    GUSISetHook(GUSI_EventHook+updateEvt, (GUSIHook)GUSISIOWUpdate);
    GUSISetHook(GUSI_EventHook+activateEvt, (GUSIHook)GUSISIOWActivate);
    GUSISetHook(GUSI_EventHook+osEvt, (GUSIHook)GUSISIOWSusRes);
}

void GUSIDefaultSetupConsole()
{
    GUSISetupConsoleDescriptors();
    GUSISetupSIOW();
    GUSISetupConsoleStdio();
}
```


Chapter 39

Supporting threads made outside of GUSI

As convenient as the pthreads interface is, some applications may link to other libraries which create thread manager threads directly, such as the PowerPlant thread classes.

Unfortunately, there is no really elegant way to welcome these lost sheep into the pthread flock, since the thread manager offers no way to retrieve thread switching and termination procedures. We therefore have to resort to a violent technique used already successfully for MPW support: CFM, we override the default entry point and use the CFM manager to find the real implementation.

For non-CFM, we unfortunately don't have such an effective technique, since the thread manager is called through traps (and no, I'm not going to patch any traps for this). You will therefore have to recompile your foreign libraries with a precompiled header that includes GUSIForeignThreads.h

```
<GUSIForeignThreads.h 1507>≡
#ifdef _GUSIForeignThreads_
#define _GUSIForeignThreads_

#define NewThread(threadStyle, threadEntry, threadParam, stackSize, options, threadResult, threadMade) \
    GUSINewThread((threadStyle), (threadEntry), (threadParam), (stackSize), (options), (threadResult), (threadMade))
#define SetThreadSwitcher(thread, threadSwitcher, switchProcParam, inOrOut) \
    GUSISetThreadSwitcher((thread), (threadSwitcher), (switchProcParam), (inOrOut))
#define SetThreadTerminator(thread, threadTerminator, terminationProcParam) \
    GUSISetThreadTerminator((thread), (threadTerminator), (terminationProcParam))
#endif /* _GUSIForeignThreads_ */

<GUSIForeignThreads.cp 1508>≡
#include "GUSIInternal.h"
#include "GUSIContext.h"

#include <ConditionalMacros.h>
#include <CodeFragments.h>

<Declaration of thread manager functions 1509>
<Override implementations of thread manager functions 1512>
<Declaration of class GUSIThreadManagerForeignProxy 1513>
<Override implementation of GUSIThreadManagerProxy::MakeInstance 1515>
<Member functions for class GUSIThreadManagerForeignProxy 1514>
```

39.1 Interfacing to the Thread Manager routines

As mentioned above, our interface strategy differs for non-CFM and CFM code. We try to handle this with a macro. As opposed to the MPW interface, we have to handle CFM68K.

```
<Declaration of thread manager functions 1509>≡ (1508)
extern "C" {
    #if !TARGET_RT_MAC_CFM
    #define DECL_thread(name, pname, ret, args) ret name args;
    #ifdef __MWERKS__
    #pragma pointers_in_D0
    #endif
    <Declarations for Threads functions 1511>
    #ifdef __MWERKS__
    #pragma pointers_in_A0
    #endif
    static bool ConnectToThreadLibrary()    { return true; }
    #define sGUSIThreadFailure 0
    #else
    #define DECL_thread(name, pname, ret, args) static ret (*name) args;
    <Declarations for Threads functions 1511>
    <Connecting to the ThreadsLib code fragment 1510>
    #endif
}
```

On PowerPC, we have to connect to the dynamic library (which, in principle, can fail).

```

<Connecting to the ThreadsLib code fragment 1510>≡ (1509)
static OSErr sGUSIThreadFailure = 1;

static void DoConnectToThreadLibrary()
{
    CFragConnectionID  ThreadsLib;
    CFragSymbolClass   symClass;
    Ptr                whoCares;
    Str255              error;

    if (sGUSIThreadFailure = GetSharedLibrary(
        StringPtr("\pThreadsLib"), GetCurrentArchitecture(), kLoadCFrag, &ThreadsLib, &whoCares, error)
    )
        return;

#undef DECL_thread
#define DECL_thread(name, pname, ret, args) \
    if (sGUSIThreadFailure = FindSymbol(ThreadsLib, pname, (Ptr *) &name, &symClass)) \
        goto failed;
    <Declarations for Threads functions 1511>

    return;

failed:
#undef DECL_thread
#define DECL_thread(name, pname, ret, args) name = 0;
    <Declarations for Threads functions 1511>
}

static bool ConnectToThreadLibrary()
{
    if (sGUSIThreadFailure == 1)
        DoConnectToThreadLibrary();
    return !sGUSIThreadFailure;
}

```

Now we only have to declare the list once, and can reuse it numerous times.

```

<Declarations for Threads functions 1511>≡ (1509 1510)
DECL_thread(GUSISTdNewThread, "\pNewThread", OSErr, (ThreadStyle threadStyle, ThreadEntryProcPtr threadEntry,
DECL_thread(GUSISTdSetThreadSwitcher, "\pSetThreadSwitcher", OSErr, (ThreadID thread, ThreadSwitchProcPtr threadSwitcher,
DECL_thread(GUSISTdSetThreadTerminator, "\pSetThreadTerminator", OSErr, (ThreadID thread, ThreadTerminationProcPtr threadTerminator,

```

39.2 Redirecting thread manager calls to their GUSI equivalents

People naively using thread manager calls will find themselves redirected to GUSI routines instead.

```
<Override implementations of thread manager functions 1512>≡ (1508)
    pascal OSErr NewThread(ThreadStyle threadStyle, ThreadEntryProcPtr threadEntry, void *threadParam,
    {
        return GUSINewThread(threadStyle, threadEntry, threadParam, stackSize, options, threadResult, t
    }

    pascal OSErr SetThreadSwitcher(ThreadID thread, ThreadSwitchProcPtr threadSwitcher, void *switchProcParam,
    {
        return GUSISetThreadSwitcher(thread, threadSwitcher, switchProcParam, inOrOut);
    }

    pascal OSErr SetThreadTerminator(ThreadID thread, ThreadTerminationProcPtr threadTerminator, void *threadParam,
    {
        return GUSISetThreadTerminator(thread, threadTerminator, terminationProcParam);
    }
```

39.3 Installing our GUSI thread manager hooks

GUSIThreadManagerForeignProxy is now quite straightforward.

```
<Declaration of class GUSIThreadManagerForeignProxy 1513>≡ (1508)
    class GUSIThreadManagerForeignProxy : public GUSIThreadManagerProxy {
    public:
        virtual OSErr NewThread(ThreadStyle threadStyle, ThreadEntryProcPtr threadEntry, void *threadParam,
        virtual OSErr SetThreadSwitcher(ThreadID thread, ThreadSwitchProcPtr threadSwitcher, void *switchProcParam,
        virtual OSErr SetThreadTerminator(ThreadID thread, ThreadTerminationProcPtr threadTerminator, void *threadParam,
    };

```


Note that the calls below resolve to a statically linked procedure on non-CFM, but call through an indirect procedure pointer on CFM.

```

⟨Member functions for class GUSIThreadManagerForeignProxy 1514⟩≡ (1508)
OSErr GUSIThreadManagerForeignProxy::NewThread(ThreadStyle threadStyle, ThreadEntryProcPtr threadEntry, void *
{
    if (!ConnectToThreadLibrary())
        return sGUSIThreadFailure;
    return GUSIStdNewThread(threadStyle, threadEntry, threadParam, stackSize, options, threadResult, threadMad

}

OSErr GUSIThreadManagerForeignProxy::SetThreadSwitcher(ThreadID thread, ThreadSwitchProcPtr threadSwitcher, vo
{
    if (!ConnectToThreadLibrary())
        return sGUSIThreadFailure;
    return GUSIStdSetThreadSwitcher(thread, threadSwitcher, switchProcParam, inOrOut);
}

OSErr GUSIThreadManagerForeignProxy::SetThreadTerminator(ThreadID thread, ThreadTerminationProcPtr threadTermi
{
    if (!ConnectToThreadLibrary())
        return sGUSIThreadFailure;
    return GUSIStdSetThreadTerminator(thread, threadTerminator, terminationProcParam);
}

```

Finally, to make sure that these hooks are called, we sneak in and replace GUSIThreadManagerProxy::MakeInstance.

```

⟨Override implementation of GUSIThreadManagerProxy::MakeInstance 1515⟩≡ (1508)
GUSIThreadManagerProxy * GUSIThreadManagerProxy::MakeInstance()
{
    return new GUSIThreadManagerForeignProxy;
}

```


Part V

The Build System

Chapter 40

Build rules for GUSI

```
<Makefile.mk 1516>≡
<Paths to be configured 1517>
<Compiler options for GUSI 1518>
<Files for GUSI 1520>
<Directory locations for GUSI 1526>
<Pattern rules for GUSI 1527>
<Top level rules for GUSI 1519>
<Library build rules for GUSI 1529>
<Additional dependences for GUSI 1530>
<Documentation build rules for GUSI 1531>
<Object build rules for GUSI 1528>
<Bulk build rules for GUSI 1532>
```

For some parts of GUSI, you may want to use STLport and/or SFIO. If so, specify their paths here.

```
<Paths to be configured 1517>≡ (1516)
SFIOInc      = -i :::sfio98:src:lib:sfio:
STLportInc   = -i :::STLport-3.12.3:stl:
```

Our build targets are the Metrowerks C/C++ compiler and the standard MPW compilers SCpp and MrC.

```
<Compiler options for GUSI 1518>≡ (1516)
MWCWarn      = -w noimplicitconv,nostructclass,nopossible
MWCInc        = -nodefaults -i- -i ::include -i "{{CWANSIIncludes}}" -i "{{CIncludes}}" -i ::DCon:Headers: ${S
MPWInc        = -i ::include -i "{{CIncludes}}" -i ::DCon:Headers: ${SFIOInc} ${STLportInc}
COpt          = -d SystemSevenOrLater -d OLDROUTINENAMES=0 -bool on -sym on
Opt68K        = -model far -mc68020 -mbg on
OptPPC        = -traceback
MWCOpt        = ${COpt} ${MWCInc} ${MWCWarn} -opt full
MPWOpt        = ${COpt} ${MPWInc} -ER -includes unix -w 2,6,35
C68K          = MWC68K ${MWCOpt} ${Opt68K}
CPPC          = MWCPPC ${MWCOpt} ${OptPPC}
CSC           = SCpp ${MPWOpt} ${Opt68K}
CMRC          = MrCpp ${MPWOpt} ${OptPPC} -ansifor
ROptions      = -i :
Lib68K        = MWLink68K -xm library -sym on
LibPPC        = MWLinkPPC -xm library -sym on
LibSC         = Lib -sym on -d
LibMrC        = PPCLink -xm l -sym on -d
```

<Top level rules for GUSI 1519>≡ (1516)

```
all      :   lib
.PHONY   :   source lib
source   :   tangled $(TANGLED)
    If "{{EzDepend}}" != ""
        ::scripts:EzDepend -f Makefile.mk :tangled:.cp -i ::include
    End
    Echo > source
lib       :   lib-ppc lib-68k lib-sc lib-mrc
lib-68k   :   source $(GUSILIBS68K)
lib-ppc   :   source $(GUSILIBSPPC)
lib-sc    :   source $(GUSILIBSSC)
lib-mrc   :   source $(GUSILIBSMRC)
liber     :   woven "Liber GUSI.ps"
```

The GUSI distribution consists of a variety of file types.

<Files for GUSI 1520>≡ (1516)

```
<Noweb files for GUSI 1521>
<Tangled files for GUSI 1522>
<Woven files for GUSI 1523>
<Object files for GUSI 1524>
<Library files for GUSI 1525>
```

The original source code resides in noweb .nw files.

<Noweb files for GUSI 1521>≡ (1520)

```
COREWEBS      =   \
    GUSIDiag.nw      GUSIBasics.nw      GUSIContext.nw      GUSISpecific.nw      \
    GUSISocket.nw    GUSIBuffer.nw      GUSIPipe.nw      GUSIFactory.nw      \
    GUSIMTinet.nw    GUSISocketMixins.nw GUSIMTTcp.nw      GUSIFileSpec.nw      \
    GUSIDevice.nw    GUSIMacFile.nw      GUSIConfig.nw      GUSIDescriptor.nw    \
    GUSIPOSIX.nw     GUSIPThread.nw      GUSIContextQueue.nw GUSINull.nw          \
    GUSIIinet.nw     GUSINetDB.nw      GUSIMTNetDB.nw     GUSIFSWrappers.nw    \
    GUSIMTudp.nw     GUSIOTinet.nw      GUSIOTNetDB.nw     GUSIOpenTransport.nw \
    GUSIDCon.nw      GUSIPPC.nw      GUSITimer.nw      GUSISignal.nw
MSLWEBS       =   \
    GUSIMSL.nw
SFIOWEBS      =   \
    GUSISfio.nw
STDIOWEBS     =   \
    GUSIMPWStdio.nw
MPWWEBS       =   \
    GUSIMPW.nw
SIOUXWEBS     =   \
    GUSISIOUX.nw
SIOWEBS       =   \
    GUSISIOW.nw
FTWEBS        =   \
    GUSIForeignThreads.nw
SRCWEBS       =   $(COREWEBS) $(MSLWEBS) $(SFIOWEBS) $(STDIOWEBS) $(MPWWEBS) $(SIOUXWEBS) $(SIOWEBS)
BUILDWEBS     =   Makefile.nw
WEBS          =   $(SRCWEBS) $(BUILDWEBS)
```

These are tangled (or untangled) into C++ and DMake files.

```

⟨Tangled files for GUSI 1522⟩≡ (1520)
TANGLED      = $(SRCWEBS:s/.nw/.nws/) pthread.h sched.h GUSIInternal.h ::include:sys:ppc.h
BUILDS       = $(BUILDWEBBS:s/.nw/.mk/)
CORESRC      = $(COREWEBS:s/.nw/.cp/)
MSLSRC       = $(MSLWEBS:s/.nw/.cp/)
SFIOSRC      = $(SFIOWEBS:s/.nw/.cp/)
STDIOSRC     = $(STDIOWEBS:s/.nw/.cp/)
MPWSRC       = $(MPWWEBS:s/.nw/.cp/)
SIOUXSRC     = $(SIOUXWEBS:s/.nw/.cp/)
SIOWSRC      = $(SIOWWEBS:s/.nw/.cp/)
FTSRC        = $(FTWEBS:s/.nw/.cp/)
FORWARDSRC   = GUSIPOSIX.cp GUSISignal.cp
SOURCES      = $(CORESRC) $(MSLSRC) $(SFIOSRC) $(MPWSRC) $(SIOUXSRC) $(FTSRC)
MPWSOURCES   = $(CORESRC) $(SFIOSRC) $(MPWSRC) $(STDIOSRC) $(SIOWSRC) $(FTSRC)

```

Alternatively, the noweb sources are woven into T_EXsources.

```

⟨Woven files for GUSI 1523⟩≡ (1520)
WOVEN        = $(WEBS:s/.nw/.nw/)

```

The C++ files get compiled into object files.

```

⟨Object files for GUSI 1524⟩≡ (1520)
OBJ68K       = {$(SOURCES)}.68K.o
OBJPPC       = {$(SOURCES)}.PPC.o
OBJJSC       = {$(MPWSOURCES)}.SC.o
OBJMRC       = {$(MPWSOURCES)}.MrC.o

```

And finally, the object files get linked into libraries.

```

⟨Library files for GUSI 1525⟩≡ (1520)
GUSIL        = Core MSL Sflw MPW SIOUX ForeignThreads
GUSILMPW     = Core Sflw MPW SIOW ForeignThreads
GUSILIBS68K  = GUSI_{$(GUSIL)}.68K.Lib GUSI_Forward.68K.Lib
GUSILIBSPPC  = GUSI_{$(GUSIL)}.PPC.Lib
GUSILIBSSC   = GUSI_{$(GUSILMPW) Stdio}.SC.Lib
GUSILIBSMRC  = GUSI_{$(GUSILMPW)}.MrC.Lib

```

All files in the web subdirectory are original. All files in tangled and obj are derived from noweb. include contains both GUSI headers, which are built from noweb, and BSD 4.4 headers which are included as is.

```

⟨Directory locations for GUSI 1526⟩≡ (1516)
.SOURCE.nw   : ""
.SOURCE.h    : "::include"
.SOURCE.c    : ":tangled"
.SOURCE.cp   : ":tangled"
.SOURCE.nws  : ":tangled"
.SOURCE.nww  : ":woven"
.SOURCE.o    : ":obj"
.SOURCE.68K  : ":obj"
.SOURCE.PPC  : ":obj"
.SOURCE.SC   : ":obj"
.SOURCE.MrC  : ":obj"
.SOURCE.Lib  : "::lib"

```

Most targets can be defined by a pattern rules, combined with a directory rule.

(Pattern rules for GUSI 1527)≡

(1516)

```
% .nws      : % .nw
    pnotangle '-c++' -L -t -R$*.h  $< > tmp; ::scripts:update-source tmp $(@:s/nws/h/:s/:tangled/:s/:woven/:s/:lib/:s/:include:mpw:/)
    pnotangle '-c++' -L -t -R$*.cp $< > tmp; ::scripts:update-source tmp $(@:s/nws/cp/)
    Set EzDepend 1
    echo > $@
% .nww      : % .nw
    pnoweave -t4 -x -n -db GUSI.db $< > $@
% .68K.o     : %
    Set Echo 0
    Set Src68K "{{Src68K}}" $<"
% .PPC.o     : %
    Set Echo 0
    Set SrcPPC "{{SrcPPC}}" $<"
% .SC.o      : %
    $(CSC) $< -o $@
% .MrC.o     : %
    $(CMRC) $< -o $@
```

(Object build rules for GUSI 1528)≡

(1516)

```
obj :
    NewFolder    obj

"::lib" :
    NewFolder    "::lib"

tangled :
    NewFolder    tangled
woven :
    NewFolder    woven

"::include:mpw:" :
    NewFolder    "::include:mpw:"
    Duplicate -y "{{CIncludes}}" errno.h ::include:mpw:errno.h
```



```
GUSI_Core.68K.Lib : Objects68K
$(Lib68K) -o $@ :obj:${$(CORESRC)}.68K.o
GUSI_Core.PPC.Lib : ObjectsPPC
$(LibPPC) -o $@ :obj:${$(CORESRC)}.PPC.o
GUSI_Core.SC.Lib : ObjectsSC
$(LibSC) -o $@ :obj:${$(CORESRC)}.SC.o
GUSI_Core.MrC.Lib : ObjectsMrC
$(LibMrC) -o $@ :obj:${$(CORESRC)}.MrC.o
GUSI_MSL.68K.Lib : Objects68K
$(Lib68K) -o $@ :obj:${$(MSLSRC)}.68K.o
GUSI_MSL.PPC.Lib : ObjectsPPC
$(LibPPC) -o $@ :obj:${$(MSLSRC)}.PPC.o
GUSI_Sfio.68K.Lib : Objects68K
$(Lib68K) -o $@ :obj:${$(SFIOSRC)}.68K.o
GUSI_Sfio.PPC.Lib : ObjectsPPC
$(LibPPC) -o $@ :obj:${$(SFIOSRC)}.PPC.o
GUSI_Sfio.SC.Lib : ObjectsSC
$(LibSC) -o $@ :obj:${$(SFIOSRC)}.SC.o
GUSI_Sfio.MrC.Lib : ObjectsMrC
$(LibMrC) -o $@ :obj:${$(SFIOSRC)}.MrC.o
GUSI_Stdio.SC.Lib : ObjectsSC
$(LibSC) -o $@ :obj:${$(STDIOSRC)}.SC.o
GUSI_MPW.68K.Lib : Objects68K
$(Lib68K) -o $@ :obj:${$(MPWSRC)}.68K.o GUSIMPWGlue.68K.Lib
GUSI_MPW.PPC.Lib : ObjectsPPC
$(LibPPC) -o $@ :obj:${$(MPWSRC)}.PPC.o
GUSI_MPW.SC.Lib : ObjectsSC
$(LibSC) -o $@ :obj:${$(MPWSRC)}.SC.o GUSIMPWGlue.SC.Lib
GUSI_MPW.MrC.Lib : ObjectsMrC
$(LibMrC) -o $@ :obj:${$(MPWSRC)}.MrC.o
GUSI_SIOUX.68K.Lib : Objects68K
$(Lib68K) -o $@ :obj:${$(SIOUXSRC)}.68K.o
GUSI_SIOUX.PPC.Lib : ObjectsPPC
$(LibPPC) -o $@ :obj:${$(SIOUXSRC)}.PPC.o
GUSI_SIOUW.SC.Lib : ObjectsSC
$(LibSC) -o $@ :obj:${$(SIOUWSRC) $(MPWSRC)}.SC.o GUSIMPWGlue.SC.Lib
GUSI_SIOUW.MrC.Lib : ObjectsMrC
$(LibMrC) -o $@ :obj:${$(SIOUWSRC) $(MPWSRC)}.MrC.o
GUSI_ForeignThreads.68K.Lib : Objects68K
$(Lib68K) -o $@ :obj:${$(FTSRC)}.68K.o
GUSI_ForeignThreads.PPC.Lib : ObjectsPPC
$(LibPPC) -o $@ :obj:${$(FTSRC)}.PPC.o
GUSI_ForeignThreads.SC.Lib : ObjectsSC
$(LibSC) -o $@ :obj:${$(FTSRC)}.SC.o
GUSI_ForeignThreads.MrC.Lib : ObjectsMrC
$(LibMrC) -o $@ :obj:${$(FTSRC)}.MrC.o
GUSI_Forward.68K.Lib : Objects68K
$(Lib68K) -o $@ :obj:${$(FORWARDSRC)}.68K.o
```

The pthread.h header file is a standard C library header, but generated from a noweb file. GUSIInternal.h is too small to merit its own noweb file.

(Additional dependences for GUSI 1530)≡ (1516)

```
pthread.h      :  GUSIPThread.nw
    nountangle '-c++' -t -Rpthread.h      $< > $@
sched.h       :  GUSIPThread.nw
    nountangle '-c++' -t -Rsched.h      $< > $@
GUSIPThread.cp :  GUSIContext.h
GUSIInternal.h :  GUSIBasics.nw
    notangle '-c++' -L -t -RGUSIInternal.h $< > $@
"::include:sys:ppc.h" :  GUSIPPC.nw
    nountangle '-c++' -t '-Rsys/ppc.h' $< > ::include:sys:ppc.h
```

(Documentation build rules for GUSI 1531)≡ (1516)

```
"Liber GUSI.ps" : GUSI.dvi
    dvips -o "Liber GUSI.ps" GUSI.dvi
```

```
GUSI.dvi : GUSI.tex
    biglatrix GUSI
    pnoindex GUSI
    biglatrix GUSI
```

```
GUSI.tex : $(WOVEN)
    setfile -m . $@
```

(Bulk build rules for GUSI 1532)≡ (1516)

```
Objects68K :  obj "::-lib" $(OBJ68K)
    Set Echo 1
    If "{{Src68K}}" != ""
        ${C68K} -t -fatext {{Src68K}} -o :Obj:
    End
    echo > Objects68K
```

```
ObjectsPPC :  obj "::-lib" $(OBJPPC)
    Set Echo 1
    If "{{SrcPPC}}" != ""
        ${CPPC} -t -fatext {{SrcPPC}} -o :Obj:
    End
    echo > ObjectsPPC
```

```
ObjectsSC :  obj "::-include:mpw:" "::-lib" $(OBJSC)
ObjectsMrC :  obj "::-lib" $(OBJMRC)
```

Part VI

Appendixes

Appendix A

Index

GUSIAlarm: [646](#)
GUSIBuiltinServiceDB: [753](#)
GUSI_COMPILER_HAS_NAMESPACE: [4](#)
GUSICatInfo: [1127](#)
GUSIConfigRsrc: [337](#)
GUSIConfiguration: [322](#)
GUSIConfiguration::AutoInitGraf: [326](#)
GUSIConfiguration::AutoSpin: [325](#)
GUSIConfiguration::ConfigureSuffices: [323](#)
GUSIConfiguration::FileSuffix: [323](#)
GUSIConfiguration::SetDefaultFType: [324](#)
GUSIContext: [34](#)
GUSIContextFactory: [35](#)
GUSIContextQueue: [381](#)
GUSIContextQueue::element: [383](#)
GUSIContextQueue::iterator: [382](#)
GUSIDConDevice: [401](#)
GUSIDConDevice::Want: [401](#)
GUSIDConDevice::open: [401](#)
GUSIDConDevice::sInstance: [401](#)
GUSIDConSocket: [402](#)
GUSIDefaultSetupConsole: [444](#)
GUSIDevice: [240](#)
GUSIDeviceRegistry: [241](#)
GUSIDeviceRegistry::AddDevice: [243](#)
GUSIDeviceRegistry::Instance: [242](#)
GUSIDeviceRegistry::RemoveDevice: [243](#)
GUSIDeviceRegistry::fFirstDevice: [300](#)
GUSIDeviceRegistry::iterator: [245](#)
GUSIDirectory: [288](#)
GUSIDoActivate: [1501](#)
GUSIErrorSaver: [13](#)
GUSIEventFn: [10](#)
GUSI_EventHook: [10](#)
GUSIExecFn: [10](#)

GUSI_ExecHook: [10](#)
GUSIFSCatMove: [1120](#)
GUSIFSCreate: [1109](#)
GUSIFSDelete: [1111](#)
GUSIFSDirCreate: [1113](#)
GUSIFSGetCatInfo: [1091](#)
GUSIFSGetFCBInfo: [1093](#)
GUSIFSGetFInfo: [1101](#)
GUSIFSGetVInfo: [1095](#)
GUSIFSHGetFInfo: [1101](#)
GUSIFSHGetVInfo: [1095](#)
GUSIFSHGetVolParms: [1106](#)
GUSIFSHSetFInfo: [1101](#)
GUSIFSMoveRename: [1122](#)
GUSIFSOpen: [1098](#)
GUSIFSOpenDF: [1103](#)
GUSIFSOpenDriver: [1098](#)
GUSIFSOpenRF: [1103](#)
GUSIFSRename: [1117](#)
GUSIFSRstFLock: [1115](#)
GUSIFSSetCatInfo: [1091](#)
GUSIFSSetFInfo: [1101](#)
GUSIFSSetFLock: [1115](#)
GUSIFileServiceDB: [756](#)
GUSIFileSpec: [1128](#)
GUSIFileToken: [239](#)
GUSIGatherer: [143](#)
GUSIGetHook: [9](#)
GUSIHook: [9](#)
GUSIIOPBWrapper: [37](#)
GUSIKillTimers: [431](#)
GUSIMFRead: [1309](#)
GUSIMFReadDone: [1309](#)
GUSIMFWakeup: [1309](#)
GUSIMFWrite: [1309](#)
GUSIMFWriteDone: [1309](#)
GUSIMPWDevice: [1436](#)
GUSIMPWSocket: [1437](#)
GUSIMTInetSocket: [764](#)
GUSIMTInetSocket::Driver: [767](#)
GUSIMTInetSocket::HostAddr: [767](#)
GUSIMTInetSocket::fLocation: [1358](#)
GUSIMTInetSocket::fPeerAddr: [765](#)
GUSIMTInetSocket::fPeerLoc: [1358](#)
GUSIMTInetSocket::fPeerPort: [1358](#)
GUSIMTInetSocket::fPort: [1358](#)
GUSIMTInetSocket::fSockAddr: [765](#)
GUSIMTInetSocket::fStream: [765](#)
GUSIMTNetDB: [897](#)
GUSIMTTRecv: [801](#)

GUSIMTTRecvDone: [801](#)
GUSIMTTSend: [801](#)
GUSIMTTSendDone: [801](#)
GUSIMTTtcpFactory: [795](#)
GUSIMTTtcpFactory::Instance: [795](#)
GUSIMTTtcpFactory::socket: [795](#)
GUSIMTTtcpSocket: [800](#)
GUSIMTTtcpSocket::CreateStream: [820](#)
GUSIMTTtcpSocket::GUSIMTTConnectDone: [813](#)
GUSIMTTtcpSocket::GUSIMTTListenDone: [817](#)
GUSIMTTtcpSocket::SetupListener: [822](#)
GUSIMTTtcpSocket::accept: [836](#)
GUSIMTTtcpSocket::fCurListener: [817](#)
GUSIMTTtcpSocket::fListenLock: [817](#)
GUSIMTTtcpSocket::fListeners: [817](#)
GUSIMTTtcpSocket::fNextListener: [817](#)
GUSIMTTtcpSocket::fNumListeners: [817](#)
GUSIMTTtcpSocket::fSelf: [800](#)
GUSIMTTtcpSocket::fSendPB: [801](#), [801](#)
GUSIMTTtcpSocket::sConnectProc: [813](#)
GUSIMTTtcpSocket::sListenProc: [817](#)
GUSIMTTtcpSocket::sNotifyProc: [809](#)
GUSIMTTtcpSocket::sRecvProc: [801](#)
GUSIMTTtcpSocket::sSendProc: [801](#)
GUSIMTTtcpSocket::sendto: [846](#)
GUSIMTURecv: [864](#)
GUSIMTURecvDone: [864](#)
GUSIMTUSend: [864](#)
GUSIMTUSendDone: [864](#)
GUSIMTUdpFactory: [858](#)
GUSIMTUdpFactory::Instance: [858](#)
GUSIMTUdpFactory::socket: [858](#)
GUSIMTUdpSocket: [863](#)
GUSIMTUdpSocket::CreateStream: [875](#)
GUSIMTUdpSocket::fSendPB: [864](#), [864](#)
GUSIMTUdpSocket::sRecvProc: [864](#)
GUSIMTUdpSocket::sSendProc: [864](#)
GUSIMTUdpSocket::sendto: [887](#)
GUSIMacDirectory: [1208](#)
GUSIMacFileDevice: [1206](#)
GUSIMacFileDevice::Want: [1206](#)
GUSIMacFileDevice::open: [1206](#)
GUSIMacFileDevice::sInstance: [1206](#)
GUSIMacFileSocket: [1207](#)
GUSIMacFileSocket::fReadPB: [1309](#)
GUSIMacFileSocket::fWritePB: [1309](#)
GUSIMacFileSocket::sReadProc: [1309](#)
GUSIMacFileSocket::sWakeupProc: [1309](#)
GUSIMacFileSocket::sWriteProc: [1309](#)
GUSIMapMacError: [12](#)

GUSINetDB: [725](#)
GUSINewThread: [36](#)
GUSINullDevice: [673](#)
GUSINullDevice::Want: [673](#)
GUSINullDevice::open: [673](#)
GUSINullDevice::sInstance: [673](#)
GUSINullSocket: [674](#)
GUSIOT: [922](#)
GUSIOTDatagramFactory: [917](#)
GUSIOTDatagramSocket: [925](#)
GUSIOTFactory: [915](#)
GUSIOTInetStrategy: [1027](#)
GUSIOTMInetOptions: [1029](#)
GUSIOTNetDB: [1069](#)
GUSIOTSocket: [923](#)
GUSIOTSocket::Lock: [939](#)
GUSIOTStrategy: [918](#)
GUSIOTStrategy::ConfigPath: [920](#)
GUSIOTStrategy::CreateConfiguration: [920](#)
GUSIOTStrategy::PackAddress: [921](#)
GUSIOTStrategy::UnpackAddress: [921](#)
GUSIOTStreamFactory: [916](#)
GUSIOTStreamSocket: [924](#)
GUSIOTTBind: [922](#)
GUSIOTTCall: [922](#)
GUSIOTTDiscon: [922](#)
GUSIOTTOptMgmt: [922](#)
GUSIOTTUError: [922](#)
GUSIOTTUnitData: [922](#)
GUSIOTTcpFactory: [1025](#)
GUSIOTTcpSocket: [1032](#)
GUSIOTTcpStrategy: [1028](#)
GUSIOTUdpFactory: [1026](#)
GUSIOTUdpSocket: [1031](#)
GUSIOTUdpStrategy: [1030](#)
GUSIPPCFactory: [1335](#)
GUSIPPCFactory::Instance: [1335](#)
GUSIPPCFactory::socketpair: [1335](#)
GUSIPPCRecv: [1340](#)
GUSIPPCRecvDone: [1340](#)
GUSIPPCSend: [1340](#)
GUSIPPCSendDone: [1340](#)
GUSIPPCSocket: [1339](#)
GUSIPPCSocket::GUSIPPCDone: [1354](#)
GUSIPPCSocket::GUSIPPCListenDone: [1347](#)
GUSIPPCSocket::SetupListener: [1350](#)
GUSIPPCSocket::accept: [1374](#)
GUSIPPCSocket::fCurListener: [1347](#)
GUSIPPCSocket::fListenLock: [1347](#)
GUSIPPCSocket::fListeners: [1347](#)

GUSIPPCSocket::fNextListener: [1347](#)
GUSIPPCSocket::fNumListeners: [1347](#)
GUSIPPCSocket::fPortRef: [1359](#)
GUSIPPCSocket::fSendPB: [1340](#), [1340](#)
GUSIPPCSocket::sDoneProc: [1354](#)
GUSIPPCSocket::sListenProc: [1347](#)
GUSIPPCSocket::sRecvProc: [1340](#)
GUSIPPCSocket::sSendProc: [1340](#)
GUSIPPCSocket::sendto: [1384](#)
GUSIPipeFactory: [692](#)
GUSIPipeFactory::Instance: [692](#)
GUSIPipeFactory::socketpair: [692](#)
GUSIPipeSocket: [697](#)
GUSIPProcess: [32](#)
GUSIPProcess::A5Saver: [33](#)
GUSIRingBuffer: [144](#)
GUSIRingBuffer::Consume: [146](#)
GUSIRingBuffer::ConsumeBuffer: [147](#)
GUSIRingBuffer::Defer: [148](#)
GUSIRingBuffer::Deferred: [148](#)
GUSIRingBuffer::Distance: [168](#)
GUSIRingBuffer::Free: [146](#)
GUSIRingBuffer::FreeBuffer: [147](#)
GUSIRingBuffer::GUSIRingBuffer: [145](#)
GUSIRingBuffer::~GUSIRingBuffer: [145](#)
GUSIRingBuffer::Invariant: [168](#)
GUSIRingBuffer::Lock: [148](#)
GUSIRingBuffer::Peeker: [150](#)
GUSIRingBuffer::Produce: [146](#)
GUSIRingBuffer::ProduceBuffer: [147](#)
GUSIRingBuffer::Release: [148](#)
GUSIRingBuffer::SwitchBuffer: [149](#)
GUSIRingBuffer::Valid: [146](#)
GUSIRingBuffer::ValidBuffer: [147](#)
GUSIRingBuffer::fBuffer: [165](#)
GUSIRingBuffer::fConsume: [165](#)
GUSIRingBuffer::fDeferred: [171](#)
GUSIRingBuffer::fDeferredArg: [171](#)
GUSIRingBuffer::fEnd: [165](#)
GUSIRingBuffer::fFree: [165](#)
GUSIRingBuffer::fLocked: [171](#)
GUSIRingBuffer::fNewBuffer: [174](#)
GUSIRingBuffer::fOldBuffer: [174](#)
GUSIRingBuffer::fProduce: [165](#)
GUSIRingBuffer::fSpare: [165](#)
GUSIRingBuffer::fValid: [165](#)
GUSISIOUXDevice: [1475](#)
GUSISIOUXSocket: [1476](#)
GUSISIOUActivate: [1501](#)
GUSISIOWSusRes: [1503](#)

GUSISMAsyncError: [655](#)
GUSISMBlocking: [651](#)
GUSISMBlocking::DoFcntl: [651](#)
GUSISMBlocking::DoIoctl: [651](#)
GUSISMBlocking::fBlocking: [651](#)
GUSISMInputBuffer: [653](#)
GUSISMInputBuffer::DoGetSockOpt: [653](#)
GUSISMInputBuffer::DoIoctl: [653](#)
GUSISMInputBuffer::SoSetSockOpt: [653](#)
GUSISMInputBuffer::fInputBuffer: [653](#)
GUSISMOutputBuffer: [654](#)
GUSISMState: [652](#)
GUSISMState::Shutdown: [652](#)
GUSISMState::fReadShutdown: [652](#)
GUSISMState::fState: [652](#)
GUSISMState::fWriteShutdown: [652](#)
GUSIScattGath: [139](#)
GUSIScattGath::Buffer: [141](#)
GUSIScattGath::Count: [141](#)
GUSIScattGath::GUSIScattGath: [140](#)
GUSIScattGath::~GUSIScattGath: [140](#)
GUSIScattGath::IOVec: [141](#)
GUSIScattGath::Length: [141](#)
GUSIScattGath::SetLength: [141](#)
GUSIScattGath::fBuf: [151](#)
GUSIScattGath::fCount: [151](#)
GUSIScattGath::fGather: [151](#)
GUSIScattGath::fIo: [151](#)
GUSIScattGath::fLen: [151](#)
GUSIScattGath::fScratch: [151](#)
GUSIScatterer: [142](#)
GUSIServiceDB: [730](#)
GUISetHook: [9](#)
GUISetHostError: [12](#)
GUISetMacError: [12](#)
GUISetMacHostError: [12](#)
GUISetPosixError: [12](#)
GUISetThreadSwitcher: [36](#)
GUISetThreadTerminator: [36](#)
GUISetupConfig: [332](#)
GUISetupConsole: [443](#)
GUISetupConsoleDescriptors: [445](#)
GUISetupDevices: [296](#)
GUISetupFactories: [216](#)
GUISigContext: [611](#)
GUISigFactory: [612](#)
GUISigProcess: [610](#)
GUISocket: [103](#)
GUISocket::AddReference: [104](#)
GUISocket::ConfigOption: [106](#)

GUSISocketDomainRegistry: [209](#)
GUSISocketDomainRegistry::AddFactory: [211](#)
GUSISocketDomainRegistry::Instance: [210](#)
GUSISocketDomainRegistry::RemoveFactory: [211](#)
GUSISocketDomainRegistry::factory: [220](#)
GUSISocketFactory: [208](#)
GUSISocket::RemoveReference: [104](#)
GUSISocket::Supports: [106](#)
GUSISocketTypeRegistry: [212](#)
GUSISocketTypeRegistry::AddFactory: [214](#)
GUSISocketTypeRegistry::Entry: [226](#)
GUSISocketTypeRegistry::Find: [230](#)
GUSISocketTypeRegistry::Initialize: [228](#)
GUSISocketTypeRegistry::RemoveFactory: [214](#)
GUSISocketTypeRegistry::factory: [226](#)
GUSISocketTypeRegistry::socket: [213](#)
GUSISocket::accept: [108](#)
GUSISocket::bind: [107](#)
GUSISocket::connect: [108](#)
GUSISocket::fcntl: [110](#)
GUSISocket::fstat: [111](#)
GUSISocket::fsync: [114](#)
GUSISocket::getpeername: [107](#)
GUSISocket::getsockname: [107](#)
GUSISocket::getsockopt: [110](#)
GUSISocket::ioctl: [110](#)
GUSISocket::isatty: [110](#)
GUSISocket::listen: [108](#)
GUSISocket::lseek: [111](#)
GUSISocket::post_select: [112](#)
GUSISocket::pre_select: [112](#)
GUSISocket::read: [109](#)
GUSISocket::recvmsg: [109](#)
GUSISocket::select: [112](#)
GUSISocket::sendmsg: [109](#)
GUSISocket::setsockopt: [110](#)
GUSISocket::shutdown: [113](#)
GUSISocket::write: [109](#)
GUSISpecific: [85](#)
GUSISpecificData: [87](#)
GUSISpecificTable: [86](#)
GUSISpinFn: [10](#)
GUSISpinHook: [10](#)
GUSITempFileSpec: [1150](#)
GUSIThreadManagerForeignProxy: [1513](#)
GUSIThreadManagerProxy: [38](#)
GUSITime: [419](#)
GUSITimerProc: [429](#)
GUSIUDPHeader: [869](#)
GUSI::ftruncate: [111](#)

GUSIhostent: [733](#)
GUSImsghdr: [127](#)
GUSIservent: [734](#)
GUSIwithInetSockets: [722](#)
GUSIwithOTInetSockets: [1024](#)
MidiWDS: [766](#)
MiniWDS: [766](#)
PTHREAD_COND_INITIALIZER: [549](#)
PTHREAD_MUTEX_INITIALIZER: [548](#)
PTHREAD_ONCE_INIT: [547](#)
abort: [644](#)
accept: [493](#)
access: [475](#)
alarm: [647](#)
bind: [490](#)
chdir: [481](#)
chmod: [473](#)
close: [457](#)
closedir: [480](#)
connect: [491](#)
creat: [468](#)
dup2: [462](#)
dup: [462](#)
endprotoent: [528](#)
endservent: [529](#)
_exit: [645](#)
faccess: [539](#)
fcntl: [461](#)
fgetfileinfo: [538](#)
fsetfileinfo: [538](#)
fstat: [463](#)
fsync: [456](#)
ftruncate: [516](#)
gGUSIExecHook: [11](#)
gGUSISpinHook: [11](#)
getcwd: [482](#)
getdtablesize: [487](#)
gethostbyaddr: [530](#)
gethostbyname: [531](#)
gethostid: [524](#)
gethostname: [525](#)
getpeername: [511](#)
getprotobyname: [532](#)
getprotobynumber: [533](#)
getprotoent: [534](#)
getservbyname: [535](#)
getservbyport: [536](#)
getservent: [537](#)
getsockname: [511](#)
getsockopt: [514](#)

gettimeofday: [484](#)
gmtime: [485](#)
h_errno: [27](#)
inet_addr: [522](#)
inet_aton: [521](#)
inet_ntoa: [523](#)
ioctl: [513](#)
isatty: [465](#)
listen: [492](#)
localtime: [485](#)
lseek: [464](#)
lstat: [472](#)
mktime: [486](#)
open: [467](#)
opendir: [477](#)
pipe: [455](#)
pthread_attr_destroy: [555](#)
pthread_attr_getdetachstate: [557](#)
pthread_attr_getstacksize: [559](#)
pthread_attr_init: [553](#)
pthread_attr_setdetachstate: [557](#)
pthread_attr_setstacksize: [559](#)
pthread_attr_t: [545](#)
pthread_condattr_destroy: [589](#)
pthread_condattr_init: [589](#)
pthread_cond_broadcast: [599](#)
pthread_cond_destroy: [591](#)
pthread_cond_init: [591](#)
pthread_cond_signal: [597](#)
pthread_cond_t: [549](#)
pthread_cond_timedwait: [595](#)
pthread_cond_wait: [593](#)
pthread_create: [563](#)
pthread_detach: [565](#)
pthread_equal: [603](#)
pthread_exit: [569](#)
pthread_getspecific: [575](#), [577](#)
pthread_join: [567](#)
pthread_key_create: [571](#)
pthread_key_delete: [573](#)
pthread_key_t: [546](#)
pthread_kill: [637](#)
pthread_mutexattr_destroy: [579](#)
pthread_mutexattr_init: [579](#)
pthread_mutex_destroy: [581](#)
pthread_mutex_init: [581](#)
pthread_mutex_lock: [583](#)
pthread_mutex_t: [548](#)
pthread_mutex_trylock: [585](#)
pthread_mutex_unlock: [587](#)

pthread_once: [605](#)
pthread_once_t: [547](#)
pthread_self: [601](#)
pthread_sigmask: [641](#)
pthread_t: [544](#)
raise: [636](#)
read: [459](#)
readdir: [478](#)
readlink: [520](#)
readv: [494](#)
recv: [495](#)
recvfrom: [496](#)
recvmsg: [497](#)
remove: [469](#)
rename: [471](#)
rewinddir: [479](#)
sched_yield: [607](#)
seekdir: [479](#)
select: [506](#)
select_forever: [504](#)
select_once: [502](#)
select_sleep: [503](#)
select_timed: [505](#)
send: [499](#)
sendmsg: [501](#)
sendto: [500](#)
setprotoent: [526](#)
setservent: [527](#)
setsockopt: [515](#)
shutdown: [512](#)
sigaction: [638](#)
sigaddset: [635](#)
sigdelset: [635](#)
sigemptyset: [635](#)
sigfillset: [635](#)
sigismember: [635](#)
signal: [639](#)
sigpending: [640](#)
sigprocmask: [641](#)
sigwait: [643](#)
sleep: [466](#)
socket: [488](#)
socketpair: [489](#)
stat: [472](#)
symlink: [519](#)
telldir: [479](#)
time: [483](#)
truncate: [517](#)
ualarm: [648](#)
unlink: [470](#)

usleep: 518
utime: 474
write: 460
writev: 498

Appendix B

Names of the Sections

<Additional dependences for GUSI 1530>
<Adjust queueLength according to BSD definition 834>
<Adjust queueLength according to BSD definition 1372>
<Advance fPeek by len 202>
<Alias resolution for a GUSIFileSpec 1139>
<Align read size with fBlockSize and file position 1319>
<Align write size with fBlockSize and file position 1316>
<Asynchronous notifier function for GUSIOTNetDB 1074>
<Asynchronous notifier function for Open Transport 938>
<Attempt Open into fileRef 1216>
<Automatic cursor spin 325>
<Automatic initialization of QuickDraw 326>
<Auxiliary data structures for class GUSISocket 127>
<Auxiliary functions for class GUSIFileSpec 1166>
<Avoid silly windows in GUSIRingBuffer 183>
<Buffer switching for GUSIRingBuffer 149>
<Bulk build rules for GUSI 1532>
<Calculate the size of a free block in GUSIRingBuffer and adjust len 181>
<Call OTAccept and return if successful 990>
<Call OTListen and queue a candidate 988>
<Call post_select for all file descriptors 509>
<Call pre_select for all file descriptors and determine canSleep 508>
<Check and store received UDP packet 873>
<Check for eligible signals 75>
<Check for special folder 1255>
<Check that all specified file descriptors are valid or return -1 507>
<Clean up existing target 1231>
<Compare received address to stored peer address 1016>
<Comparing two GUSIFileSpec objects 1148>
<Compiler options for GUSI 1518>
<Configuration options for GUSISocket 106>
<Connecting to the StdCLib code fragment 1439>
<Connecting to the ThreadsLib code fragment 1510>
<Connection establishment for GUSISocket 108>
<Constants for CmdPeriod 370>

<Constructing instances of GUSINetDB 726>
<Constructor and destructor for GUSIRingBuffer 145>
<Constructor and destructor for GUSIScattGath 140>
<Constructors for GUSIFileSpec 1129>
<Context links for GUSISocket 105>
<Converting a GUSIFileSpec to a FSSpec 1135>
<Copy fBuf unless it points to a scratch area 162>
<Copy internal descriptor sets to parameters 510>
<Copy uncontroversial fields of GUSIScattGath 161>
<Create UDP stream if necessary 881>
<Create and open aliasFile 1258>
<Creation of GUSIDeviceRegistry 242>
<Data members for GUSIMTInetSocket 765>
<Declaration of class GUSIMPWDevice 1436>
<Declaration of class GUSIMPWSocket 1437>
<Declaration of class GUSISIOUXDevice 1475>
<Declaration of class GUSISIOUXSocket 1476>
<Declaration of class GUSIThreadManagerForeignProxy 1513>
<Declaration of thread manager functions 1509>
<Declarations for MPW functions 1440>
<Declarations for Threads functions 1511>
<Declarations of C GUSIFSWrappers 1097>
<Declarations of C++ GUSIFSWrappers 1090>
<Declare and initialize macHost and unixHost 906>
<Declare and initialize otHost and unixHost 1081>
<Default directory handling in GUSIFileSpec 1134>
<Default implementation of GUSISetupConsole 443>
<Define iterator for GUSIContextQueue 382>
<Definition of GUSISetupConfig hook 332>
<Definition of GUSISetupDevices hook 296>
<Definition of GUSISetupFactories hook 216>
<Definition of GUSIwithInetSockets 721>
<Definition of GUSIwithMTInetSockets 790>
<Definition of GUSIwithMTTcpSockets 794>
<Definition of GUSIwithMTUdpSockets 857>
<Definition of GUSIwithOTInetSockets 1024>
<Definition of IO wrappers 37>
<Definition of class GUSIBuiltinServiceDB 753>
<Definition of class GUSICatInfo 1127>
<Definition of class GUSIConfiguration 322>
<Definition of class GUSIContext 34>
<Definition of class GUSIContextFactory 35>
<Definition of class GUSIContextQueue 381>
<Definition of class GUSIDConDevice 401>
<Definition of class GUSIDConSocket 402>
<Definition of class GUSIDDescriptorTable 437>
<Definition of class GUSIDevice 240>
<Definition of class GUSIDeviceRegistry 241>
<Definition of class GUSIDirectory 288>
<Definition of class GUSIFileServiceDB 756>

<Definition of class GUSIFileSpec 1128>
<Definition of class GUSIFileToken 239>
<Definition of class GUSIGatherer 143>
<Definition of class GUSIMTInetSocket 764>
<Definition of class GUSIMTNetDB 897>
<Definition of class GUSIMTTcpFactory 795>
<Definition of class GUSIMTTcpSocket 800>
<Definition of class GUSIMTUDPFactory 858>
<Definition of class GUSIMTUDPsocket 863>
<Definition of class GUSIMacDirectory 1208>
<Definition of class GUSIMacFileDevice 1206>
<Definition of class GUSIMacFileSocket 1207>
<Definition of class GUSINetDB 725>
<Definition of class GUSINullDevice 673>
<Definition of class GUSINullSocket 674>
<Definition of class GUSIOTDatagramFactory 917>
<Definition of class GUSIOTDatagramSocket 925>
<Definition of class GUSIOTFactory 915>
<Definition of class GUSIOTInetStrategy 1027>
<Definition of class GUSIOTMInetOptions 1029>
<Definition of class GUSIOTNetDB 1069>
<Definition of class GUSIOTSocket 923>
<Definition of class GUSIOTStrategy 918>
<Definition of class GUSIOTStreamFactory 916>
<Definition of class GUSIOTStreamSocket 924>
<Definition of class GUSIOTTcpFactory 1025>
<Definition of class GUSIOTTcpSocket 1032>
<Definition of class GUSIOTTcpStrategy 1028>
<Definition of class GUSIOTUDPFactory 1026>
<Definition of class GUSIOTUDPsocket 1031>
<Definition of class GUSIOTUDPStrategy 1030>
<Definition of class GUSIPPCFactory 1335>
<Definition of class GUSIPPCSocket 1339>
<Definition of class GUSIPipeFactory 692>
<Definition of class GUSIPipeSocket 697>
<Definition of class GUSIProcess 32>
<Definition of class GUSIProcess::A5Saver 33>
<Definition of class GUSIRingBuffer 144>
<Definition of class GUSIRingBuffer::Peeker 150>
<Definition of class GUSISMAsyncError 655>
<Definition of class GUSISMBlocking 651>
<Definition of class GUSISMInputBuffer 653>
<Definition of class GUSISMOOutputBuffer 654>
<Definition of class GUSISMState 652>
<Definition of class GUSIScattGath 139>
<Definition of class GUSIScatterer 142>
<Definition of class GUSIServiceDB 730>
<Definition of class GUSISigContext 611>
<Definition of class GUSISigFactory 612>
<Definition of class GUSISigProcess 610>

<Definition of class GUSISocket 103>
<Definition of class GUSISocketDomainRegistry 209>
<Definition of class GUSISocketFactory 208>
<Definition of class GUSISocketTypeRegistry 212>
<Definition of class GUSISpecific 85>
<Definition of class GUSISpecificTable 86>
<Definition of class GUSIThreadManagerProxy 38>
<Definition of class GUSITime 419>
<Definition of class GUSITimer 420>
<Definition of class GUSIhostent 733>
<Definition of class GUSIservent 734>
<Definition of classes MiniWDS and MidiWDS 766>
<Definition of compiler features 4>
<Definition of error handling 12>
<Definition of event handling 14>
<Definition of hook handling 9>
<Definition of struct GUSIConfigRsrc 337>
<Definition of template GUSIOT 922>
<Definition of template GUSISpecificData 87>
<Definition of thread manager hooks 36>
<Delete b if enough free elements remain 388>
<Determine fType for folder and volume aliases 1253>
<Determine the link count for a directory 1235>
<Determine the starting directory of the path 1169>
<Direct interface for GUSIRingBuffer 146>
<Directory locations for GUSI 1526>
<Discard data in input buffer of GUSIMacFileSocket 1274>
<Disconnect the GUSIOTSocket, dammit 947>
<Do the TCPPassiveOpen and pick up port number if necessary 826>
<Documentation build rules for GUSI 1531>
<Ensure that file to rename exists and new name resides on same volume 1228>
<Error handling in GUSIFileSpec 1133>
<Event handlers for SIOW 1501>
<Execute the signal handler 630>
<File oriented operations for GUSISocket 111>
<Files for GUSI 1520>
<Find ASCII equivalent of virtual key 371>
<Find out if alias points at a file or a directory 1193>
<Forward ProduceBuffer to fNewBuffer 180>
<Friends of GUSIContext 62>
<Fudge queueLength 979>
<GUSIBasics.cp 3>
<GUSIBasics.h 2>
<GUSIBuffer.cp 138>
<GUSIBuffer.h 137>
<GUSIConfig.cp 321>
<GUSIConfig.h 320>
<GUSIContextQueue.cp 379>
<GUSIContextQueue.h 378>
<GUSIContext.cp 30>

<GUSIContext.h 29>
<GUSIDCon.cp 400>
<GUSIDCon.h 399>
<GUSIDDescriptor.cp 436>
<GUSIDDescriptor.h 435>
<GUSIDevice.cp 238>
<GUSIDevice.h 237>
<GUSIFSWrappers.cp 1089>
<GUSIFSWrappers.h 1088>
<GUSIFactory.cp 207>
<GUSIFactory.h 206>
<GUSIFileSpec.cp 1126>
<GUSIFileSpec.h 1125>
<GUSIForeignThreads.cp 1508>
<GUSIForeignThreads.h 1507>
<GUSIInet.cp 720>
<GUSIInet.h 719>
<GUSIInternal.h 1>
<GUSIMPWStdio.cp 1421>
<GUSIMPWStdio.h 1420>
<GUSIMPW.cp 1435>
<GUSIMPW.h 1434>
<GUSIMSL.cp 1394>
<GUSIMSL.h 1393>
<GUSIMTInet.cp 763>
<GUSIMTInet.h 762>
<GUSIMTNetDB.cp 896>
<GUSIMTNetDB.h 895>
<GUSIMTTcp.cp 793>
<GUSIMTTcp.h 792>
<GUSIMTUDP.cp 856>
<GUSIMTUDP.h 855>
<GUSIMacFile.cp 1205>
<GUSIMacFile.h 1204>
<GUSINetDB host database 727>
<GUSINetDB protocol database 729>
<GUSINetDB service database 728>
<GUSINetDB.cp 724>
<GUSINetDB.h 723>
<GUSINull.cp 672>
<GUSINull.h 671>
<GUSIOTInet.cp 1023>
<GUSIOTInet.h 1022>
<GUSIOTNetDB.cp 1067>
<GUSIOTNetDB.h 1066>
<GUSIOpenTransport.cp 914>
<GUSIOpenTransport.h 913>
<GUSIPOSIX.cp 454>
<GUSIPOSIX.h 453>
<GUSIPPC.cp 1334>

<GUSIPPC.h 1333>
<GUSIPThread.cp 543>
<GUSIPThread.h 542>
<GUSIPipe.cp 691>
<GUSIPipe.h 690>
<GUSISIOUX.cp 1474>
<GUSISIOUX.h 1473>
<GUSISIOW.cp 1499>
<GUSISIOW.h 1498>
<GUSIScattGath constructed with contiguous IO vector 155>
<GUSIScattGath constructed with empty IO vector 154>
<GUSIScattGath constructed with sparse IO vector 156>
<GUSISfio.cp 1431>
<GUSISfio.h 1430>
<GUSISignal.cp 609>
<GUSISignal.h 608>
<GUSISocketMixins.cp 650>
<GUSISocketMixins.h 649>
<GUSISocket.cp 102>
<GUSISocket.h 101>
<GUSISpecific.cp 84>
<GUSISpecific.h 83>
<GUSITimer.cp 418>
<GUSITimer.h 417>
<Gather fIo contents in fBuf 163>
<Get the AliasHandle for the alias file and detach it 1192>
<Get volume icon 1251>
<Get volume information into vRef and cRef 1250>
<Getting path names from a GUSIFileSpec 1137>
<Getting the GUSICatInfo for a GUSIFileSpec 1136>
<Handle O_EXCL and O_TRUNC 1219>
<Handle errors from Open 1218>
<Handle special renaming cases by recursing 1229>
<Hook getter code 18>
<Hook setter code 17>
<Hook variables 16>
<Hooks for ANSI library interfaces 441>
<If file didn't exist, try to create it and goto tryOpen if successful 1217>
<Implementation of ANSI library specific internal GUSI functions 1418>
<Implementation of ANSI library specific public GUSI functions 1416>
<Implementation of GUSIFSWrappers 1091>
<Implementation of GUSIwithInetSockets 722>
<Implementation of GUSIwithOTInetSockets 1065>
<Implementation of MPW ANSI library specific public GUSI functions 1422>
<Implementation of MSL override functions 1396>
<Implementation of Pthread attribute management 550>
<Implementation of Pthread creation and destruction 561>
<Implementation of Pthread thread specific data 571>
<Implementation of Pthread varia 579>
<Implementation of completion handling 40>

- <Implementation of context queues 385>
- <Implementation of error handling 25>
- <Implementation of event handling 28>
- <Implementation of hook handling 15>
- <Implementation of internal GUSI functions for MPW Stdio 1428>
- <Implementation of internal GUSI functions for Sfiio 1432>
- <Indirect interface for GUSIRingBuffer 147>
- <Initialize buffer of GUSIRingBuffer 166>
- <Initialize fields of GUSIMTTcpSocket 803>
- <Initialize fields of GUSIMTudpSocket 866>
- <Initialize fields of GUSIMacFileSocket 1270>
- <Initialize fields of GUSINetDB 742>
- <Initialize fields of GUSIOTDatagramSocket 1009>
- <Initialize fields of GUSIOTNetDB 1073>
- <Initialize fields of GUSIOTSocket 937>
- <Initialize fields of GUSIOTStreamSocket 981>
- <Initialize fields of GUSIPPCSocket 1342>
- <Initialize fields of GUSIRingBuffer 167>
- <Inline member functions for class GUSICatInfo 1151>
- <Inline member functions for class GUSIConfiguration 335>
- <Inline member functions for class GUSIContextQueue 390>
- <Inline member functions for class GUSIDConDevice 405>
- <Inline member functions for class GUSIDDescriptorTable 452>
- <Inline member functions for class GUSIDeviceRegistry 299>
- <Inline member functions for class GUSIFileSpec 1153>
- <Inline member functions for class GUSIMTTcpFactory 797>
- <Inline member functions for class GUSIMTudpFactory 860>
- <Inline member functions for class GUSINullDevice 677>
- <Inline member functions for class GUSIPPCFactory 1337>
- <Inline member functions for class GUSIPipeFactory 694>
- <Inline member functions for class GUSIPipeSocket 701>
- <Inline member functions for class GUSIRingBuffer 193>
- <Inline member functions for class GUSISMAsyncError 667>
- <Inline member functions for class GUSISMBlocking 656>
- <Inline member functions for class GUSISMInputBuffer 660>
- <Inline member functions for class GUSISMOOutputBuffer 664>
- <Inline member functions for class GUSISMState 658>
- <Inline member functions for class GUSIScattGath 152>
- <Inline member functions for class GUSIServiceDB 752>
- <Inline member functions for class GUSISocket 115>
- <Inline member functions for class GUSISocketDomainRegistry 219>
- <Inline member functions for class GUSISocketTypeRegistry 227>
- <Inline member functions for class GUSISpecific 89>
- <Inline member functions for class GUSISpecificTable 91>
- <Inline member functions for file GUSIContext 41>
- <Insert id at the appropriate place into basename 1203>
- <Insert null bytes beyond EOF 1286>
- <Internal iterator protocol of GUSIServiceDB 732>
- <Interrupt level routines for GUSIMTTcpSocket 805>
- <Interrupt level routines for GUSIMTudpSocket 868>

<Interrupt level routines for GUSIMacFileSocket 1314>
<Interrupt level routines for GUSIPPCSocket 1343>
<Interrupt routine wrappers for GUSIMacFileSocket 1310>
<Iterating over the GUSIServiceDB 731>
<Iterator on GUSIDeviceRegistry 245>
<Library build rules for GUSI 1529>
<Library files for GUSI 1525>
<Looking up a device in the GUSIDeviceRegistry 244>
<MPW function wrappers 538>
<MPW implementation of GUSIGetEnv 1472>
<MPW implementation of GUSISetupConsoleDescriptors 1452>
<MPW library functions 1438>
<MPW spin hook 1451>
<MacTCP DNR code 898>
<MacTCP driver management 767>
<Maintaining properties for GUSISocket 110>
<Makefile.mk 1516>
<Manipulating a GUSIFileSpec 1141>
<Member functions for class GUSIBuiltinServiceDB 754>
<Member functions for class GUSIConfiguration 334>
<Member functions for class GUSIDConDevice 403>
<Member functions for class GUSIDConSocket 408>
<Member functions for class GUSIDDescriptorTable 438>
<Member functions for class GUSIDevice 292>
<Member functions for class GUSIDeviceRegistry 298>
<Member functions for class GUSIFileServiceDB 757>
<Member functions for class GUSIFileSpec 1155>
<Member functions for class GUSIFileToken 289>
<Member functions for class GUSIMPWDevice 1441>
<Member functions for class GUSIMPWSocket 1453>
<Member functions for class GUSIMTInetSocket 769>
<Member functions for class GUSIMTInetSocket 1365>
<Member functions for class GUSIMTNetDB 900>
<Member functions for class GUSIMTTcpFactory 796>
<Member functions for class GUSIMTTcpSocket 802>
<Member functions for class GUSIMTUdpFactory 859>
<Member functions for class GUSIMTUdpSocket 865>
<Member functions for class GUSIMacDirectory 1323>
<Member functions for class GUSIMacFileDevice 1209>
<Member functions for class GUSIMacFileSocket 1267>
<Member functions for class GUSINetDB 736>
<Member functions for class GUSINullDevice 675>
<Member functions for class GUSINullSocket 681>
<Member functions for class GUSIOTDatagramFactory 931>
<Member functions for class GUSIOTDatagramSocket 1001>
<Member functions for class GUSIOTFactory 927>
<Member functions for class GUSIOTInetStrategy 1041>
<Member functions for class GUSIOTMInetOptions 1045>
<Member functions for class GUSIOTNetDB 1071>
<Member functions for class GUSIOTSocket 941>

<Member functions for class GUSIOTStrategy 933>
<Member functions for class GUSIOTStreamFactory 929>
<Member functions for class GUSIOTStreamSocket 970>
<Member functions for class GUSIOTTcpFactory 1034>
<Member functions for class GUSIOTTcpSocket 1053>
<Member functions for class GUSIOTTcpStrategy 1051>
<Member functions for class GUSIOTUdpFactory 1038>
<Member functions for class GUSIOTUdpSocket 1060>
<Member functions for class GUSIOTUdpStrategy 1058>
<Member functions for class GUSIPPCFactory 1336>
<Member functions for class GUSIPPCSocket 1341>
<Member functions for class GUSIPipeFactory 693>
<Member functions for class GUSIPipeSocket 700>
<Member functions for class GUSIRingBuffer 169>
<Member functions for class GUSIRingBuffer::Peeker 199>
<Member functions for class GUSISIOUXDevice 1477>
<Member functions for class GUSISIOUXSocket 1484>
<Member functions for class GUSIScattGath 153>
<Member functions for class GUSIServiceDB 750>
<Member functions for class GUSISigContext 616>
<Member functions for class GUSISigFactory 613>
<Member functions for class GUSISigProcess 623>
<Member functions for class GUSISocket 116>
<Member functions for class GUSISocketDomainRegistry 218>
<Member functions for class GUSISocketFactory 215>
<Member functions for class GUSISocketTypeRegistry 229>
<Member functions for class GUSISpecific 88>
<Member functions for class GUSISpecificTable 93>
<Member functions for class GUSIThreadManagerForeignProxy 1514>
<Member functions for class GUSITime 421>
<Member functions for class GUSITimer 429>
<Member functions for class GUSIhostent 760>
<Member functions for class GUSIservent 761>
<Miscellaneous operations for GUSISocket 113>
<Move and rename, moving the corner file out of the way if necessary 1124>
<Move existing target out of the way 1230>
<Multiplexing for GUSISocket 112>
<Name dropping for file GUSIContext 31>
<Name dropping for file GUSIContextQueue 380>
<Name dropping for file GUSIOTNetDB 1068>
<No TCP/IP support, determine host name from chooser 739>
<No default implementation, return EOPNOTSUPP 119>
<No more data in GUSIPipeSocket, check for EOF and consider waiting 709>
<Noweb files for GUSI 1521>
<Object build rules for GUSI 1528>
<Object files for GUSI 1524>
<ObsoleteBuffer if possible 185>
<Open MacTCP DNR or fail lookup 905>
<Open Open Transport DNR or fail lookup 1080>
<Open and return a MacFileSocket if possible 1450>

<Operations for GUSIDevice 247>
<Operations for GUSIDeviceRegistry 248>
<Overridden member functions for GUSIDConSocket 409>
<Overridden member functions for GUSIMTInetSocket 773>
<Overridden member functions for GUSIMTInetSocket 1364>
<Overridden member functions for GUSIMTNetDB 903>
<Overridden member functions for GUSIMTTcpSocket 828>
<Overridden member functions for GUSIMTUDPsocket 878>
<Overridden member functions for GUSIMacDirectory 1324>
<Overridden member functions for GUSIMacFileDevice 1213>
<Overridden member functions for GUSIMacFileSocket 1279>
<Overridden member functions for GUSINullSocket 682>
<Overridden member functions for GUSIOTDatagramSocket 1002>
<Overridden member functions for GUSIOTNetDB 1078>
<Overridden member functions for GUSIOTSocket 951>
<Overridden member functions for GUSIOTStreamSocket 971>
<Overridden member functions for GUSIOTTcpSocket 1052>
<Overridden member functions for GUSIOTUDPsocket 1059>
<Overridden member functions for GUSIPPCSocket 1361>
<Overridden member functions for GUSIPipeSocket 704>
<Overridden member functions for class GUSIMPWDevice 1443>
<Overridden member functions for class GUSIMPWSocket 1454>
<Overridden member functions for class GUSISIOUXDevice 1479>
<Overridden member functions for class GUSISIOUXSocket 1486>
<Override implementation of GUSIThreadManagerProxy::MakeInstance 1515>
<Override implementations of thread manager functions 1512>
<POSIX function wrappers 455>
<POSIX functions for signal handling 635>
<Paths to be configured 1517>
<Pattern rules for GUSI 1527>
<Peer management for GUSIPipeSocket 698>
<Perform pthreads style range check on signo 634>
<Perform range check on signo 633>
<Perform sanity cross-check between sock and pb 1321>
<Privatissima of GUSIConfiguration 333>
<Privatissima of GUSIContext 48>
<Privatissima of GUSIContextQueue 383>
<Privatissima of GUSIContextQueue::element 384>
<Privatissima of GUSIContextQueue::iterator 396>
<Privatissima of GUSIDescriptorTable 439>
<Privatissima of GUSIDeviceRegistry 297>
<Privatissima of GUSIFileSpec 1152>
<Privatissima of GUSIMTTcpSocket 801>
<Privatissima of GUSIMTUDPsocket 864>
<Privatissima of GUSIMacDirectory 1322>
<Privatissima of GUSIMacFileDevice 1223>
<Privatissima of GUSIMacFileSocket 1266>
<Privatissima of GUSINetDB 735>
<Privatissima of GUSIOTDatagramSocket 930>
<Privatissima of GUSIOTFactory 926>

- ⟨Privatissima of GUSIOTNetDB 1070⟩
- ⟨Privatissima of GUSIOTSocket 935⟩
- ⟨Privatissima of GUSIOTStrategy 920⟩
- ⟨Privatissima of GUSIOTStreamSocket 928⟩
- ⟨Privatissima of GUSIOTTcpFactory 1033⟩
- ⟨Privatissima of GUSIOTUdpFactory 1037⟩
- ⟨Privatissima of GUSIPPCSocket 1340⟩
- ⟨Privatissima of GUSIPipeSocket 699⟩
- ⟨Privatissima of GUSIPProcess 39⟩
- ⟨Privatissima of GUSIRingBuffer 165⟩
- ⟨Privatissima of GUSIRingBuffer::Peeker 198⟩
- ⟨Privatissima of GUSIScattGath 151⟩
- ⟨Privatissima of GUSISigContext 615⟩
- ⟨Privatissima of GUSISigProcess 622⟩
- ⟨Privatissima of GUSISocketDomainRegistry 217⟩
- ⟨Privatissima of GUSISocketTypeRegistry 226⟩
- ⟨Privatissima of GUSISpecificTable 90⟩
- ⟨Prototypes for GUSIMacFileSocket interrupt level routines 1308⟩
- ⟨Prototypes for MSL override functions 1395⟩
- ⟨Prototypes for internal MSL functions 1399⟩
- ⟨Prototypes for internal SIOW functions 1500⟩
- ⟨Pthreads data types 544⟩
- ⟨Pthreads function declarations 552⟩
- ⟨Public interface to GUSIScattGath 141⟩
- ⟨Read configuration resource into GUSIConfiguration 340⟩
- ⟨Read directly from file into buffer if necessary 1282⟩
- ⟨Read from read-ahead buffer of GUSIMacFileSocket while possible 1281⟩
- ⟨Read rest of datagram if buffer allocation was too stingy 1017⟩
- ⟨Reference counting for GUSISocket 104⟩
- ⟨Registration interface of GUSIDeviceRegistry 243⟩
- ⟨Registration interface of GUSISocketDomainRegistry 211⟩
- ⟨Registration interface of GUSISocketTypeRegistry 214⟩
- ⟨Requests for GUSIFileToken 246⟩
- ⟨Reset pointers if GUSIRingBuffer is empty 182⟩
- ⟨SIOUX implementation of GUSISetupConsoleDescriptors 1483⟩
- ⟨SIOW implementation of GUSIDefaultSetupConsole 1506⟩
- ⟨Sanity checks for GUSIMTInetSocket::bind() 775⟩
- ⟨Sanity checks for GUSIMTTcpSocket::accept() 837⟩
- ⟨Sanity checks for GUSIMTTcpSocket::connect() 830⟩
- ⟨Sanity checks for GUSIMTTcpSocket::listen() 833⟩
- ⟨Sanity checks for GUSIMTTcpSocket::recvfrom() 843⟩
- ⟨Sanity checks for GUSIMTTcpSocket::sendto() 847⟩
- ⟨Sanity checks for GUSIMTudpSocket::connect() 880⟩
- ⟨Sanity checks for GUSIMTudpSocket::recvfrom() 884⟩
- ⟨Sanity checks for GUSIMTudpSocket::sendto() 888⟩
- ⟨Sanity checks for GUSIOTInetStrategy::PackAddress 1042⟩
- ⟨Sanity checks for GUSIOTInetStrategy::UnpackAddress 1044⟩
- ⟨Sanity checks for GUSIPPCSocket::accept() 1375⟩
- ⟨Sanity checks for GUSIPPCSocket::bind() 1363⟩
- ⟨Sanity checks for GUSIPPCSocket::connect() 1368⟩

<Sanity checks for GUSIPPCSocket::listen() 1371>
<Sanity checks for GUSIPPCSocket::recvfrom() 1381>
<Sanity checks for GUSIPPCSocket::sendto() 1385>
<Scatter fBuf contents to fIo 164>
<Sched function declarations 606>
<Sending and receiving data for GUSISocket 109>
<Set defaults for GUSIConfiguration members 339>
<Set up fSendWDS 871>
<Shut down listening GUSIMTTcpSocket 854>
<Shut down listening GUSIPPCSocket 1392>
<Skip offset bytes from beginning of vec 192>
<Socket creation interface of GUSISocketDomainRegistry 210>
<Socket creation interface of GUSISocketTypeRegistry 213>
<Socket function wrappers 487>
<Socket name management for GUSISocket 107>
<Socket protocol matches ent->protocol 233>
<Socket type matches ent->type 232>
<Special folder types and their alias types 1254>
<Start read-ahead if necessary 1283>
<Strategic interfaces for GUSIOTStrategy 919>
<Suspend the current process if possible 74>
<Suspend the current thread if possible 76>
<Synchronization support for GUSIRingBuffer 148>
<Tangled files for GUSI 1522>
<This should not happen, fail assertion and return EOPNOTSUPP 293>
<This should not happen, fail assertion and return nil 294>
<Too much data in GUSIPipeSocket, consider writing in portions 713>
<Top level rules for GUSI 1519>
*<Translate descriptor s to GUSISocket * sock or return -1 458>*
<Translate flags into permission 1215>
<Translate host IP number to host name 740>
<Translate mode to posixmode 1397>
<Try PBMoveRename if available 1123>
<Try converting the path with FSMakeFSSpec and return 1168>
<Try decoding the path as an encoded FSSpec 1167>
<Try matching file suffix rules 347>
<Type and creator rules for newly created files 323>
<Upgrade application context to threading 47>
<Various flags 327>
<Wait for a connection to arrive for the GUSIMTTcpSocket 838>
<Wait for a connection to arrive for the GUSIPPCSocket 1376>
<Wait for free buffer space on GUSIMTTcpSocket 848>
<Wait for free buffer space on GUSIMTudpSocket 889>
<Wait for free buffer space on GUSIMacFileSocket 1287>
<Wait for free buffer space on GUSIPPCSocket 1386>
<Wait for pending read operations on GUSIMacFileSocket to complete 1273>
<Wait for valid data on GUSIMTTcpSocket 844>
<Wait for valid data on GUSIMTudpSocket 885>
<Wait for valid data on GUSIPPCSocket 1382>
<Wake up appropriate contexts to deliver signal 628>

⟨Walk directories upwards 1170⟩
⟨Woven files for GUSI 1523⟩
⟨pthread.h 540⟩
⟨return an alibi stat buffer 1234⟩
⟨return an appropriate error if GUSIOTSocket::accept failed 989⟩
⟨sched.h 541⟩
⟨sys/ppc.h 1332⟩