Chapter 4

Internal Description

4.1 Source File Structure

The source tree of AmiTCP/IP is divided to few main parts:

1. Directory **sys** contains BSD Unix system header files, which are ported to the Amiga environment.

2. Directory **net** contains modules for the general networking services. They are independent of any particular protocol family.

3. Directory **netinet** contains the modules for the internet (inet) protocol family. IP, ICMP, TCP and UDP are internet protocols.

4. Directory **kern** contains modules, which provide the BSD kernel environment for the networking routines.

5. Directory **api** contains the BSD socket compatible shared library API code.

The files are briefly described in the following paragraphs. Note that some of the header files are actually located to the **NETINCLUDE:**[1] directory.

4.1.1 System Include Files

General Headers

sys/cdefs.h Definitions to harmonize various C language dialects.

sys/param.h General machine independent parameter definitions.

sys/time.h Definition of structure timeval.

[1]See section 3.1.3 on page 32.

sys/types.h Common C type definitions and file descriptor set macros for `select()`.

BSD Socket API

sys/errno.h Error code definitions for system functions.

sys/ioctl.h Definitions for socket IO control.

sys/socket.h Definitions related to sockets: types, address families, options and prototypes.

sys/uio.h IO structure definition for sendmsg() and recvmsg()[2].

BSD Kernel Data Structures

sys/domain.h Domain structure and global domain pointer declarations.

sys/kernel.h Global kernel variable definitions (used throughout the kernel).

sys/malloc.h Defines the `bsd_malloc()` and `bsd_free()` functions.

sys/mbuf.h Mbuf structure definition and inline macros.

sys/protosw.h Protocol switch structure definition with related definitions (options, actions, etc.).

sys/queue.h General queue data structures.

sys/socketvar.h Socket structure definition and related utility macros.

sys/synch.h Declarations for tsleep and spl functions.

sys/syslog.h Logging related definitions.

sys/systm.h System wide definitions and prototypes.

4.1.2 Protocol Independent Network Routines

Network Interface

net/if.c General network interface routines.

net/if.h Defines the interface for network adapter drivers.

net/if_arp.h General protocol independent ARP structures.

net/if_dl.h Defines the Link Level sockaddr structure.

net/if_loop.c Loopback device routines.

net/if_sana.c Interface module for the SANA-II network adapter drivers.

net/if_sana.h Defines the interface for the SANA-II network adapter drivers.

[2]These functions are not implemented in this release of the AmiTCP/IP

net/if_types.h Interface type numbers. Obsolete.

net/netisr.h and

net/netisr.c Defines the network input queue scheduling routines.

net/sana2arp.c ARP routines for Sana-II devices.

net/sana2arp.h SANA-II private ARP headers.

net/sana2copybuff.c Buffer management routines called from SANA-II device driver.

net/sana2errno.h SANA-II error lists and printing header.

net/sana2request.c Module to handle SANA-II IO Requests.

net/sana2request.h Inline routines to manage different SANA-II IO requests.

net/sana2tags.c Common customization parameters for the `sana_softc` network interface.

net/sana2tags.h Defines the tags to customize `sana_softc` network interface.

netlib:sana2perror.c Printing routines for SANA-II error messages.

Routing

net/radix.c Routines for searching, adding and removing data items in the radix binary tree.

net/radix.h Defines the radix tree data structures.

net/route.c General, protocol independent routing functions.

net/route.h Defines data structures for routing entries, tables and statistics.

net/rtsock.c The socket interface to the routing information.

Raw Sockets

net/raw_cb.c Routines to manage the raw protocol control blocks.

net/raw_cb.h The raw protocol control block definition.

net/raw_usrreq.c The raw socket interface to the low level protocols and network adapters.

4.1.3 Internet Protocol Modules

Inet Domain

netinet/in.c Generic Internet addressing routines.

netinet/in.h Protocol numbers, port conventions, inet address definitions.

netinet/in_cksum.c Calculates checksum for internet protocol headers.

netinet/in_pcb.c Generic internet protocol routines, binding, addressing.

netinet/in_pcb.h Declares generic internet protocol control block.

netinet/in_proto.c Defines internet protocol control blocks.

netinet/in_systm.h Some network byte order type definitions.

netinet/in_var.h Define an interface address structure for internet.

IP, ICMP

netinet/icmp_var.h ICMP statistics.

netinet/ip.h IP packet header, packet options, timestamp.

netinet/ip_icmp.c Routines to generate, receive and reflect ICMP packets.

netinet/ip_icmp.h ICMP packet structure.

netinet/ip_input.c IP input, packet reassemble and packet forwarding.

netinet/ip_output.c IP output, packet fragmenting.

netinet/ip_var.h Define IP statistics, external IP packet header, reassemble queues structures.

netinet/raw_ip.c Provide a raw interface to the IP protocol.

TCP

netinet/tcp.h Define the TCP packet structure.

netinet/tcp_debug.c TCP debugging routines.

netinet/tcp_debug.h Header for TCP debugging.

netinet/tcp_fsm.h TCP Finite State Machine states.

netinet/tcp_input.c TCP input routines.

netinet/tcp_output.c TCP output routines.

netinet/tcp_seq.h TCP sequence numbering.

netinet/tcp_subr.c Various TCP subroutines for initializing, connection maintenance etc.

netinet/tcp_timer.c TCP timeout routines.

netinet/tcp_timer.h TCP timing constants.

netinet/tcp_usrreq.c Process TCP user requests (send, timeout), attach, disconnect.

netinet/tcp_var.h Define TCP control block and TCP statistics.

netinet/tcpip.h Define the overlaid TCPIP packet header structure.

UDP

netinet/udp.h Define the UDP packet structure.

netinet/udp_usrreq.c Routines to UDP output, input and notification.

netinet/udp_var.h Define UDPIP packet overlay, UDP statistics.

4.1.4 BSD Kernel Service Modules

BSD Kernel Support

kern/amiga_includes.h Include file which includes all amiga specific include files.

kern/amiga_main.c Main module of the AmiTCP/IP.

kern/amiga_subr.h Miscellaneous function definitions (usually short inline functions).

kern/amiga_time.c Timer module for the timeout functions.

kern/amiga_time.h Amiga timer.device related functions.

kern/kern_malloc.c Malloc & free related functions.

kern/kern_synch.c tsleep and spl function definitions.

kern/uipc_mbuf.c Mbuf functions.

Socket Level Functions

kern/uipc_domain.c Domain functions, especially pfslowtimo() and pffasttimo().

kern/uipc_socket.c Higher level so-level functions (`socreate()`, `sobind()` etc.).

kern/uipc_socket2.c Lower level so-level functions (`soisconnecting()`, `soqremque()`, `sowakeup()` etc.).

NETTRACE Maintenance Process Support

kern/amiga_config.c Configuration and ARexx command support routines.

kern/amiga_config.h Defines the structure for configuration variables.

kern/amiga_cstat.c Query support routines.

kern/amiga_log.c Functions for initialization of log task and code for task itself.

kern/amiga_log.h Header file for logging functions.

kern/amiga_netdb.c Network database parsing and support functions.

kern/amiga_netdb.h Declares the network database structures and types.

kern/amiga_rexx.c Arexx-interface.

kern/amiga_rexx.h Arexx-interface definitions.

kern/config_var.awk Awk script to generate both code and documentation from the file **kern/variables.src**.

kern/subr_prf.c Interfaces for (s)printf(), panic() and log()

kern/variables.src Definition and documentation for configuration variables

4.1.5 BSD Socket API

Shared Library Interface

api/allocdatabuffer.c Client data buffer allocation functions.

api/allocdatabuffer.h Client data buffer allocation definitions.

api/amiga_api.c Contains API initialization and deinitialization functions and data. Opening and Closing of the AmiTCP/IP socket library bases.

api/amiga_api.h Description of the SocketBase structure (for internal AmiTCP/IP use only), and some inline functions related to the API functionality.

api/amiga_libcallentry.h Included in every module which defines AmiTCP/IP socket library calls.

api/amiga_libtables.c Exec library base list and AmiTCP/IP socket library function tables for `MakeLibrary()`.

api/amiga_raf.h Compiler dependent macros for Register Argument Functions.

api/apicalls.h Includes either **api/apicalls_gnuc.h** or **api/apicalls_sasc.h**.

api/apicalls_gnuc.h Inline functions for internal API calls for GNUC.

api/apicalls_sasc.h Inline functions for internal API calls for SASC.

API Functions

api/amiga_generic.c General Unix system calls related to file descriptors ported for AmiTCP/IP socket library. There are also some Amiga specific extensions to the BSD Unix system calls in this module.

api/amiga_libcalls.c Inet library functions (link library in Unix system). These inet functions are provided as a part of the AmiTCP/IP socket library.

api/amiga_syscalls.c Standard BSD style socket functions ported to the AmiTCP/IP socket library.

api/gethostnamadr.c Functions that calls resolver in order to obtain host information from nameserver. Calls netdatabase functions if information can not be resolved.

api/gethtbynamadr.h Prototypes for two functions in **api/getxbyy.h**.

api/getxbyy.c Host (without nameserver), network, service ja protocol query functions (also a link library in unix system).

api/hostbuf.h Structure definition for host queries and few minor netdb stuff.

Resolver Functions

api/arpa_nameser.h Information for nameserver query.

api/res_comp.c Routines to translate domain names between conventional ascii and the compressed format used in queries.

api/res_debug.c Functions that output debugging information[3]

api/res_init.h Resolver initializer function. Very simple in AmiTCP/IP implementation.

api/res_mkquery.c Function that forms a domain name query in buffer.

api/res_query.c Functions to generate query sequence and append domains to incomplete hostnames.

api/res_send.c Function to send and receive query to and from a nameserver, respectively.

api/resolv.h Resolver datatypes, defines, variables and prototypes.

4.1.6 Miscellaneous Files

Makefiles

GCCOPTS Compiler options for the GCC.

GNUmakefile Makefile used with GNU make in HP-UX workstations, where the source tree is maintained.

SCOPTIONS Compiler options for the SAS/C 6.x

Smakefile Makefile for SAS/C `smake` 6.x to compile the AmiTCP/IP.

Revision Support

bsdsocket.library_rev.rev Contains revision number of the current version (number of the latest build). This file is updated with the **BumpRev** utility[4].

bsdsocket.library_rev.h Automatically generated by **BumpRev**. Contains build date, version and revision strings.

SAS/C 6.x GST Support

[3] If `RES_DEBUG` defined at compile time

[4] The **BumpRev** is supplied by Commodore.

all_includes.c C file for GST generation. Just includes the **all_includes.h** file.

all_includes.h Includes all header files needed by the AmiTCP/IP, which are suitable for inclusion in a GST. Note that any headers with inline functions *can not* be included in a GST.

Other Files

conf/conf.h Static configuration information in form of preprocessor defines. Every C module *must* include this as the first include file.

conf/rcs.h RCS header inclusion macro for GCC. In SAS/C this macro is defined in the **SCOPTIONS** file.

protos/*/*.h Prototypes for functions in various modules.

4.2 AmiTCP/IP Initialization

All more or less distinct modules of the AmiTCP/IP must be initialized before they can be used for the real work. Initialization is done in module **kern/amiga_main.c**, which defines functions `init_all()` and `deinit_all()`. The `main()` function calls the `init_all()` after it has done all needed local initializations. If `init_all()` fails the `deinit_all()` is called to clean up.

4.2.1 `init_all()`

This function calls the initialization routines of all modules which have such. This must be done carefully in the correct partial order.

There are few general heuristics which can be applied on initialization (with respect to the ordering requirements):

1. Initialize the modules for which the initialization *cannot fail*. This includes semaphore initializations and such.

2. Initialize the modules which are most likely to fail, e.g. large memory allocations, modules which require recent versions of some libraries etc.

3. Initialize modules left over by rules 1 and 2.

Semaphore initializations are first, because later steps may use them to protect against race conditions.

The initialization process of the AmiTCP/IP is fully reversible. If the initialization fails in any step the `deinit_all()` function can be used to collect the garbage.

`init_all()` does the initializations in following order, returning nonzero if all steps succeed. Numbers in the parenthesis are the section and page numbers for more information, respectively:

1. `malloc_init()` initializes the `malloc_semaphore`. This is first since later steps may want to use `bsd_malloc()` to allocate memory (5.2.2, 101).
2. `spl_init()` initializes the priority level subsystem, which is used throughout the code to protect critical sections (5.3.1, 102).
3. `sleep_init()` initializes the `sleep_semaphore` and the sleep queues (5.3.2, 103).
4. `readconfig()` reads in and parses the command line arguments and the configuration file. This is the first phase which is expected to fail. This is done at the beginning in order to allow other initialization functions to use the configuration information given. Note that since the logging subsystem is not initialized yet, `readconfig()` cannot log any errors encountered (4.11, 97).
5. `log_init()` initializes the logging subsystem. From now on other initialization functions can log needed error messages (4.8, 95).
6. `mbinit()` allocates memory to be used by the protocols. This is a candidate to fail if the memory is nearly exhausted (4.5, 83).
7. `timer_init()` initializes the timeout module. This requires version 36 or greater of the operating system. Returns the signal mask to wait for the timeout messages (5.1, 99).
8. `api_init()` initializes and creates the master socket base structure (4.10.2, 96).
9. `res_init()` initializes resolver structure and semaphore.
10. `sana_init()` initializes the SANA-II network interface module. This returns the signal mask to be used for waiting the network related messages (4.7, 89).
11. `domaininit()` initializes all configured protocols. This is left at the end of the initialization, since this requires the other parts of the system to be initialized.
12. `readnetdb()` initializes and reads in the network data base information from **ENV:AmInet/netdb** (4.12, 98).
13. `api_show()` makes the shared library interface visible on the Exec library list (4.10.2, 96).

4.2.2 `deinit_all()`

`deinit_all()` is the reverse of `init_all()`; it deinitializes required modules in reverse order with respect to the initialization.

If the initialization process for a module does not allocate any resources, then there is no deinitialization function for that module. The deinitialization functions called are (in this order):

1. `api_hide()` removes the library from the Exec library list, so no-one can open the library any more (4.10.2, 96).

2. `sana_deinit()` deinitializes the network driver module (4.7, 89).

3. `api_deinit()` deinitializes the API (4.10.2, 96).

4. `timer_deinit()` deinitializes the timer module (5.1, 99).

5. `mbdeinit()` frees all memory used by mbufs (4.5, 83).

6. `log_deinit()` deinitializes the logging subsystem (4.8, 95).

4.3 The Main Module

The `main()` function is defined in the file **kern/amiga_main.c**. On startup it initializes all modules by calling `init_all()`. After a successful initialization it first starts the AmiTCP/IP timeouts by calling `timer_send()`. Then it enters the event loop.

AmigaOS function `Wait()` returns when one of the signals specified in the signal mask given as argument is received. The mask currently specifies signals for SANA-II drivers, timeouts and the break signal. When some set of these signals is received the `Wait()` returns and its return value, which indicates the received signals, is checked. The functions `sana_poll()` and `timer_poll()` are each called in turn if their signal was received. They each return boolean telling if they would like to be called again before waiting again. The loop in which these functions are called is terminated either when no one of the poll functions wants to continue or when the break signal is received.

The poll functions do handle only one message at a time, so that any of them should not starve.

When the break signal is received the AmiTCP/IP terminates if no library bases opened by API users are open any more. On termination all reserved resources are freed by calling `deinit_all()`.

4.4 Protocol Entities

One design goal was to keep the protocol entities intact. This is achieved through implementing all external dependencies of the protocol entities. Fortunately the protocol entities in BSD are highly independent of other UNIX kernel services. For example, all dynamic memory management is done through the *memory buffer*[5] abstraction, which means that we only had to provide the mbuf interface and the problem of memory management was solved.

[5] or mbuf for short

All protocol entities in a protocol family are defined in terms of a *protocol switch* structure. This structure is fully defined in the header file **sys/protosw.h** and [Leffler et al 1991b]. The protocol switch structure is defined as follows:

```
struct protosw {
    short   pr_type;            /* socket type used for */
    struct  domain *pr_domain;  /* domain protocol a member of */
    short   pr_protocol;        /* protocol number */
    short   pr_flags;           /* protocol flags */
/* protocol-protocol hooks */
    void    (*pr_input)();      /* input to protocol (from below) */
    int     (*pr_output)();     /* output to protocol (from above) */
    void    (*pr_ctlinput)();   /* control input (from below) */
    int     (*pr_ctloutput)();  /* control output (from above) */
/* user-protocol hook */
    int     (*pr_usrreq)();     /* user request hook */
/* utility hooks */
    void    (*pr_init)();       /* initialization hook */
    void    (*pr_fasttimo)();   /* fast timeout (200ms) */
    void    (*pr_slowtimo)();   /* slow timeout (500ms) */
    void    (*pr_drain)();      /* flush any excess space possible */
};
```

Note that the actual prototypes for the function pointers are omitted, see the file **sys/protosw.h** for full definition.

When protocol is started the `pr_init()` is called first to allow the protocol to initialize all needed internal structures. Then the input process will call `pr_fasttimo()` and `pr_slowtimo()` entries periodically if defined[6]. The `pr_drain()` entry asks the protocol to free all non-critical memory buffers in a low-memory situation.

Protocols call each other through the protocol–protocol interface. To pass a packet up in the hierarchy a protocol calls the `pr_input()`-entry of the protocol above it. `pr_output()`-entry is called when a protocol wishes to pass a packet down in the protocol hierarchy (towards network).

Protocols send control information to each other through the `pr_ctlinput()` and `pr_ctloutput()` entries.

All requests coming from the API are dispatched through the `pr_usrreq()` entry.

4.5 Memory Management

As stated earlier, the memory management of the protocol stack is done with *memory buffers*. An mbuf is a structure containing little amount of storage (usually 128 bytes).

[6] Member function is defined when the value of its address is not `NULL`.

Some bytes of this storage are used for the header, but most of it is used to store user data. These small buffers are linked together to get storage for larger data.

Mbufs are highly efficient in a network protocol environment where it must be able to attach and strip protocol headers with minimum overhead and most importantly, *without copying the data* as doing so. When data is stored in an mbuf chain, attaching a header is achieved by simply linking the mbuf containing the header to the head of the chain. Removing a header is also done simply by removing the first mbuf or by incrementing the data pointer inside the mbuf.

In general there are two types of mbufs. Ones with an packet header and ones without. An Mbuf with packet header is used as the first mbuf of every packet. This header contains extra information needed per packet. See the header file **sys/mbuf.h** for the mbuf header definition.

Mbufs can be chained in two dimensions. First they may be linked to form the storage for the whole message. Second these messages may be linked together so that the boundaries of messages are maintained. This second feature is mainly used by message oriented protocols such as UDP.

To gain efficiency an mbuf may have a reference to external memory page (a *cluster*), where a big message is copied instead of splitting it apart to many mbufs. The main advantage of this feature is avoidance of copying the data when sending it with TCP, since the clusters are shared between copies[7].

4.5.1 Mbuf Functions

Mbufs are accessed through set of functions which can be grouped as follows:

Maintenance

`int mb_check_conf(void *dp, LONG newvalue)`

Check configurable variable whose address is `dp`. Return TRUE if the `newvalue` is acceptable value for that variable. See section 4.11 for information about the configuration.

`BOOL mbinit(void)`

Initialize the whole mbuf-subsystem. Allocate a chunk of mbufs and clusters (using `m_alloc()` and `m_clalloc`). Must be called before any other mbuf-related function (except that `mb_check_conf()` can be called anytime).

`void mbdeinit(void)`

Free all resources used by mbuf subsystem. Must be called as the last mbuf-related function in the program.

[7] TCP must keep a copy of the sent data for possible retransmissions.

`BOOL m_alloc(int howmany, int canwait)`

Allocate `howmany` mbufs and place them on the mbuf free list. If `canwait` is true the caller can wait if memory is not readily available.

`BOOL m_clalloc(int ncl, int canwait);`

Allocate `ncl` mbuf clusters and place them on the cluster free list.

`struct mbuf *m_retry(int i, int t)`

Ask protocols to free space when short of memory and re-attempt to allocate an mbuf.

`void m_reclaim(void)`

Ask protocols to free space when short of memory.

Allocation and Deallocation

`struct mbuf *m_get(int canwait, int type)`

Allocate an mbuf of type `type`. If no mbufs are available we can wait for them if `canwait` is `M_WAIT`. Initialize mbuf to contain internal data.

`void MGET(struct mbuf *m, int canwait, int type)`

This is an macro form of above.

`struct mbuf *m_gethdr(int canwait, int type)`

Allocate an mbuf of type `type`. If no mbufs are available we can wait for them if `canwait` is `M_WAIT`. Initialize mbuf to contain a packet header and internal data.

`void MGETHDR(struct mbuf *m, int canwait, int type)`

This is a macro form of above.

`struct mbuf *m_getclr(int canwait, int type)`

Allocate an mbuf of type `type`. If no mbufs are available we can wait for them if `canwait` is `M_WAIT`. If allocation succeeds the data buffer is zeroed before returning.

`MCLALLOC(struct mcluster *p, int canwait)`

A macro to allocate an mbuf cluster. The result is placed in `p`.

`MCLGET(struct mcluster *p, int canwait)`

A macro to add a cluster to a normal mbuf. `M_EXT` flag of the mbuf is set on success.

`struct mbuf *m_free(struct mbuf *m)`

Free an mbuf. Next mbuf in the chain is returned, if any.

`void MFREE(struct mbuf *m, struct mbuf *n)`

This is an macro form of above. Successor mbuf is returned in `n`.

`MCLFREE(struct mcluster *p)`

A macro to free an mbuf cluster.

`void m_freem(struct mbuf *m)`

Free the whole chain of mbufs starting from `m`.

Utility Functions

`struct mbuf *m_copym(struct mbuf *m, int off0, int len, int canwait)`

Make a copy of an mbuf chain starting `off0` bytes from the beginning of `m`, continuing for `len` bytes. If len is M_COPYALL, copy to end of mbuf.

`void m_copydata(struct mbuf *m, int off, int len, caddr_t cp)`

Copy data from an mbuf chain starting `off` bytes from the beginning, continuing for `len` bytes, into buffer `cp`.

`struct mbuf *m_prepend(struct mbuf *m, int len, int canwait)`

Prepend a chain of mbufs (`m`) with new mbuf with `len` bytes allocated from the first mbuf aligned on a long word boundary.

`void M_PREPEND(struct mbuf *m, int plen, int canwait)`

Macro version of above optimized for the most general cases.

`void M_COPY_PKTHDR(struct mbuf *to, struct mbuf *from)`

Macro for copying an mbuf packet header from `from` to `to`. `from` must have flag `M_PKTHDR` set, and `to` must be empty.

`void M_ALIGN(struct mbuf *m, int len)`

Macro to set the `m` data pointer of a newly–allocated mbuf (with `m_get()/MGET()`) to place an object of the size `len` at the end of the mbuf, long word aligned.

`void MH_ALIGN(struct mbuf *m, int len)`

As above, but for mbufs allocated with `m_gethdr()/MGETHDR()` or initialized by `M_COPY_PKTHDR()`.

`int M_LEADINGSPACE(struct mbuf *m)`

Compute the amount of space available before the current start of data in an mbuf.

`int M_TRAILINGSPACE(struct mbuf *m)`

Compute the amount of space available after the end of data in an mbuf.

`void MCHTYPE(struct mbuf *m, type t)`

Change mbuf to a new `type`.

`void m_cat(struct mbuf *m, struct mbuf *n)`

Concatenate mbuf chain `n` to `m`. Both chains must be of the same type.

`m_adj(struct mbuf *mp, int req_len)`

Trim `req_len` bytes from the head of the mbuf chain `mp` if `req_len` is positive, else trim − `req_len` bytes from the tail of the mbuf chain.

`struct mbuf *m_pullup(struct mbuf *n, int len)`

Rearrange an mbuf chain so that `len` bytes from the beginning of the mbuf chain `n` are contiguous and in the data area of the mbuf so that the data can be used as a structure.

Utility Macros

`type t mtod(struct mbuf *m, type t)`

Convert mbuf pointer to a pointer to the start of the data area of the mbuf casted to type `t`.

`struct mbuf *dtom(type *x)`

Convert data pointer within an mbuf to mbuf pointer.

4.6 Concurrency Control

The protocol implementation in the BSD net/2 is driven by network and timer interrupts and user processes calling the system functions. As the whole protocol stack is moved inside normal AmigaOS process, some modifications are in place.

The processor priority levels are the main concurrency control tool of the BSD kernel. The levels defined are `SPL0` (user level), `SPLSOFTCLOCK`, `SPLNET` and `SPLIMP` (the most privileged level). Execution at a higher level disables the execution at all lower levels. In AmiTCP/IP the concurrency control is implemented either with semaphores (when debugging) or with prevention of the task switches (`Forbid()`/`Permit()`), see section 5.3.1 on page 102 for the implementation notes.

The protocol input and timeouts are driven by a single process that manages the whole protocol stack. The process sends appropriate IO requests to the timer device and the SANA-II device drivers in question. Actions are then taken as response to the returned requests. Before any protocol routines are called the priority level is raised either to `SPLSOFTCLOCK` or `SPLNET`. After the function returns the priority level is lowered back to `SPL0` and the request is sent back to the device driver.

On the API side the concurrent execution of system calls is mostly prohibited, because in UNIX the system calls are atomic in the sense that there is never more than one system call in execution. In AmigaOS the shared library functions must be re-entrant so the protection must be provided by the library functions themselves.

The priority at which the main process runs must be above the default value of 0 to provide enough time to process the networking protocols. On the other side there is no sense to drive the main process at greater priority than the SANA-II device drivers.

4.7 Network Device Drivers

AmiTCP/IP uses standard SANA-II drivers as its external network device drivers. A little glue is needed to attach a SANA-II driver into BSD net/2 code.

Network Interface

The BSD net/2 networking code provides a clean interface to the network device drivers. The *network interface* provides a consistent interface for all protocols that may be present in the BSD Unix kernel. Each hardware device is associated with an unique network interface which may be used by one or more protocol families.

The network interface is flexible enough to attach different SANA-II network device drivers into the AmiTCP/IP networking system. Common part of all network interfaces is described in [Leffler et al 1991b]:

```
struct ifnet {
        char    *if_name;               /* name, e.g. "en" or "lo" */
        short   if_unit;                /* sub-unit for lower level driver */
        short   if_mtu;                 /* maximum transmission unit */
        short   if_flags;               /* up/down, broadcast, etc. */
        short   if_timer;               /* time 'til if_watchdog called */
        int     if_metric;              /* routing metric (external only) */
        struct  ifaddr *if_addrlist;    /* linked list of addresses per if */
        struct  ifqueue {
                struct  mbuf *ifq_head;
                struct  mbuf *ifq_tail;
                int     ifq_len;
                int     ifq_maxlen;
                int     ifq_drops;
        } if_snd;                       /* output queue */
/* procedure handles */
        int     (*if_init)();           /* init routine */
```

```
        int     (*if_output)();         /* output routine (enqueue) */
        int     (*if_start)();          /* initiate output routine */
        int     (*if_done)();           /* output complete routine */
        int     (*if_ioctl)();          /* ioctl routine */
        int     (*if_reset)();          /* bus reset routine */
        int     (*if_watchdog)();       /* timer routine */
/* generic interface statistics */
        int     if_ipackets;            /* packets received on interface */
        int     if_ierrors;             /* input errors on interface */
        int     if_opackets;            /* packets sent on interface */
        int     if_oerrors;             /* output errors on interface */
        int     if_collisions;          /* collisions on csma interfaces */
/* end statistics */
        struct  ifnet *if_next;
};
```

Network interface for SANA-II devices are handled in the module **net/if_sana.c**. This is the only module aware of SANA-II devices inside the AmiTCP/IP network process. It hides most SANA-II specific details from the rest of the code.

Module Initialization

`ULONG sana_init(void)`

This initialization routine is called at startup time before any interfaces have been added to system. It creates the common message port used for all SANA-II network interfaces. It also attaches the loopback device into system. It returns the signal mask of the message port, if its creation was successful.

`void sana_deinit(void)`

This routine frees all resources allocated by the SANA-II interface module. It aborts all pending IO requests, frees them, closes network device drivers and frees the corresponding network interfaces. Finally it deletes the message port.

4.7.1 SANA-II Soft Network Interface

A message passing system based on the normal Exec IO requests is used to transfer packets between the AmiTCP/IP and SANA-II devices. The IO can be either synchronous or asynchronous. The SANA-II interface module has a message port, which receives all fulfilled or aborted asynchronous IO messages. A dispatch method (currently a dispatch function pointer) have been added to all asynchronously sent IO requests. Dispatching function handles the received message in an appropriate way.

A message may contain received packet, some buffers allocated for the sent message or an event mask. Dispatcher functions feed the received data to the protocol input queues.

If needed, the protocol input routines are run. Dispatchers also free the memory allocated for the sent packets, or relays events to the higher level protocols.

Because the interface for the SANA-II device driver must handle many different protocols and network adapters, it has some private data hidden from the rest of the system. The `struct sana_softc` network interface is defined in file **net/if_sana.h**:

```
/*
 * SANA-II Interface descriptor
 *      NOTE: most of the code outside will believe this to be simply
 *      a "struct ifnet". The other information is, on the other hand,
 *      our own business.
 */
struct sana_softc {
  struct ifnet    ss_if;              /* network-visible interface */
  struct in_addr  ss_ipaddr;          /* copy of ip address */
  ULONG           ss_hwtype;          /* wiretype */
  UBYTE           ss_hwaddr[MAXADDRSANA]; /* General hardware address */
  struct Device  *ss_dev;             /* pointer to device */
  struct Unit    *ss_unit;            /* pointer to unit */
  VOID           *ss_bufmgnt;         /* magic cookie for buffer mngement */
  UWORD           ss_reqno;           /* # of requests to allocate */
  struct IOIPReq *ss_reqs;            /* allocated requests */
  struct MinList  ss_freereq;         /* free requests */
#if     INET
  struct {
    UWORD reqno;              /* for listening ip packets */
    UWORD sent;
    ULONG type;
  } ss_ip;
  struct {                      /* for ARP */
    UWORD reqno;
    UWORD sent;
    ULONG type;                 /* ARP packet type */
    ULONG hrd;                  /* ARP header type */
    struct arptable *table;     /* ARP/IP table */
  } ss_arp;
#endif  /* INET */
  UWORD           ss_rawreqno;        /* for raw packets */
  UWORD           ss_rawsent;
  struct sana_softc *ss_next;
  char            ss_name[IFNAMSIZ];  /* name lives here */
};
```

There is an external interface to this structure via `SIOCSSANATAGS` and `SIOCGSANATAGS` `ioctls`.

`struct ifnet *iface_find(char *name)`

This function initializes a network interface for given interface. It is called on a non-existent interface from `ifunit()`. It tries to open the appropriate SANA-II device driver, and if successful it initializes a descriptor and calls `if_attach()` with it.

Its argument is the device driver name concatenated with a slash and unit number. This name is used by AmiTCP/IP to open appropriate unit of the SANA-II driver.

The initialization routine provides the specified SANA-II network device unit with the `CopyToBuf()` and `CopyFromBuf()` function tags[8]. Those tags are used to copy data to and from the internal buffers of the network device. The taglist given to the device driver is defined in file **net/sana2copybuff.c**.

After successful open `iface_find()` initializes the appropriate members of the new `sana_softc` structure. It stores the device and unit pointers, magic cookie for buffer management, hardware type, MTU and address length. It also searches for the hardware type specific taglist, and sets the rest of interface parameters according the taglist.

Next the interface initialization routine attaches the new network interface into `ifnet` list with `if_attach()`. Then it initializes interface with `if_init()`.

Interface Routines

`BOOL sana_poll(void)`

The AmiTCP/IP processes received messages by calling this function. It effectively hides the actual implementation from the rest of the system. This routine is called when AmiTCP/IP receives the signal allocated by `sana_init()`, and it returns `TRUE`, if it should be called again before `Wait()`.

`sana_poll()` retrieves messages from the message port, dispatches them and then runs the input queues.

```
int sana_output(struct ifnet *ifp, struct mbuf *m0,
                struct sockaddr *dst, struct rtentry *rt)
```

This function is used as the `if_output()` method. It tries to get a free IO request from the `ss_freereq` list. If no free request is available it drops the packet. It attaches the packet `m0` to request, sets dispatching function to `free_written_packet()` and sends the request to the device driver.

The raw packets to the SANA-II interface use the following variation of socket address. The addressing family of the raw packets must be `AF_UNSPEC`. Currently only the ARP uses the raw SANA-II packets.

[8] For discussion for these functions, see [SANA-II 1992 add]

```
/*
 * A socket address for a generic SANA-II host
 */
struct sockaddr_sana2 {
  u_char ss2_len;
  u_char ss2_family;
  u_long ss2_type;
  u_char ss2_host[MAXADDRSANA];
};
```

`void sana_ioctl(struct ifnet *ifp, int cmd, caddr_t data)`

This function is used as the `if_ioctl()` method. SANA-II devices and their respective network interfaces are configured via raw sockets by the `IoctlSocket()` requests. When the `sana_ioctl()` gets the `SIOCSIFADDR` request, it changes the IP address of the interface. The `SIOCGIFFLAGS ioctl` is used to set the parameter flags of the interface[9].

Special SANA-II configuration is done with `SIOCSSANATAGS ioctl`. It passes a tag list to the `parse_sana_param_tags()` function.

`void if_down(struct ifnet *ifp)`

This function pulls into `if_ioctl()` from **net/if.h**. `if_down()` marks the interface down and informs all affected network protocols about the matter. If the interface handles SANA-II device, it calls `sana_down()`, which handles the dirty work to put the interface down.

`static void sana_run(struct sana_softc *ssc, int n, struct ifaddr *ifa)`

`sana_run()` configures the SANA-II interface and allocates the IO requests to use with the SANA-II device driver. Because the SANA-II device can be configured only once (see `S2_CONFIGINTERFACE` in [SANA-II 1992 add]) the initialization routine does not configure it. Among other things the hardware address of the network adapter is set in configuration process. This function is called by the `SIOGSIFADDR` ioctl, which also sets the protocol address of the interface.

`static void sana_up(struct sana_softc *ssc)`

`sana_up()` marks interface up and enables the interface to listen the packets from the network. It sends read request with an appropriate packet type number and dispatch function to the device.

`static BOOL sana_down(struct sana_softc *ssc)`

`sana_down()` aborts all pending requests sent to a SANA-II device driver.

[9] See file **sys/ioctl.h** for full list of all `ioctls`

`static void sana_ip_read(struct sana_softc *ssc, struct IOIPReq *req)` and

`static void sana_arp_read(struct sana_softc *ssc, struct IOIPReq *req)`

These dispatch functions are used to feed data from a network device to the protocol input queues. The network interface has not the input routines as members. Dispatching functions allocate mbufs for the next packet and send IO requests again to the network device.

Statistics

BSD Network interface contains a lot of excessive statistical data. Most of it is made redundant by the statistics gathered by the SANA-II driver. Because network statistics are not retrieved by looking at **/dev/kmem**, there is no need to gather BSD compatible statistics. A public ARexx port, named `AMITCP`, is set up for statistics retrieval and we can use appropriate SANA-II commands to get needed data when asked.

4.7.2 ARP

The requirements for the ARP implementation for a SANA-II interface differ radically from the original implementation BSD. The original code was written exclusively for the Ethernet, which has a global addressing scheme and a fixed address length.

The SANA-II ARP (in the module **net/sana2arp.c**) holds a separate address mapping cache for each interface. The number of entries in the cache may be configured at the run time. The hardware address length varies from one interface to another. The mapping caches are hash tables with linked lists, so there is no limitations in the bucket size. ARP Table locking is done with a signal semaphore instead of `spln()` functions.

The ARP table is externally accessed only by `IoctlSocket()` calls. With the new `SIOCGARPT ioctl` the whole ARP cache may be read at once. There is no need to awkwardly read **/dev/kmem**.

4.8 Logging

As everything is not predictable, programs like to inform the user about certain situations to help the user and/or maintainer to get programs work better. This is the motivation for `log()` and `panic()` functions. The fact that the file I/O routines can't be called from interrupts must be taken into account, since a program may want to inform the user even while executing at an interrupt level.

Among the very few functions of the AmigaOS which are callable from interrupts are `GetMsg()` and `PutMsg()`. These are used to implement the logging subsystem.

When AmiTCP/IP has something to tell to the user, it first checks if there is any free messages available. There is only a limited number of these messages to use, since they are preallocated during initialization[10]. If there is no message available, a counter indicating that the message could not be delivered is incremented. If a free message is available, the text given by caller is printed into the buffer of the message. Then the message is sent to the private port of the **N**ETTRACE task.

The **N**ETTRACE task is created early in the initialization of the AmiTCP/IP. This task waits for incoming messages and when one arrives, it prints the message to the log window and/or log file. Then message is freed by sending it back to AmiTCP/IP for reuse. If loss of log messages is detected, `log()` is called to tell that to the user.

The functions are defined as follows:

`int log(unsigned long level, const char *fmt, ...)`

> This logs a message with format specified in `exec.library/RawDoFmt()`, similar to the `printf()`. It is preceded with "`<N>`" where `N` is a level as defined in file **sys/syslog.h**.

`void panic(const char *fmt, ...)`

> When this function is called, we are in *BIG* trouble. If task, which called this, is not AmiTCP/IP, we just halt this task and send a message to AmiTCP/IP to halt immediately and free all resources. Format is as in `exec.library/RawDoFmt()`.
>
> If AmiTCP/IP runs into a `panic()` it first patches all API functions to return an error code to the caller. Then, if the `panic()` was not in the context of the AmiTCP/IP it signals the AmiTCP/IP and halts. As the AmiTCP/IP receives the signal, it send a message to the log and signals all application programs waiting for network to take attention. As this is done, it opens an *User Requester* to inform the user. After the user responds, AmiTCP/IP waits for all library openers to close libraries and finally unloads itself from the memory.

`int printf(const char *fmt, ...)`

> Like the normal C–library printf (with format of `exec.library/RawDoFmt()`) except that the printing is done using the logging mechanism.

`int sprintf(char *buf, const char *fmt, ...)`

> As in a normal C–library. Format is as in `exec.library/RawDoFmt()`.

All functions (except `panic()`) return number of printed (or logged) characters.

[10]This is because memory cannot be allocated from interrupt code.

Initialization Routines

`BOOL log_init(void)`

This function initializes **NETTRACE** subsystem by opening `intuition.library` for opening UserRequest in the case of `panic()`. The the log messages are initialized to use by preallocating memory for them.

Then **NETTRACE** is started and AmiTCP/IP waits for a signal from it. If **NETTRACE** success in it's initialization, then it sends a message back, which is then replied. If initialization fails, a variable is set to specific value and `CTRL-F` is sent to the AmiTCP/IP. If all this succeeds, the log messages are initialized and sent to `logReplyPort` which works as a queue for the free messages.

`void log_deinit(void)`

This works as reverse to initialization process. If **NETTRACE** is still running, a message is sent to it telling it to terminate. Then AmiTCP/IP waits until the message is replied. Then the memory reserved by the messages can be freed. Finally the `intuition.library` is closed.

4.9 ARexx Interface

The ARexx port of the AmiTCP/IP is maintained by the **NETTRACE** task. The messages are parsed with `parseline()` (defined in **kern/amiga_config.c**).

Initialization Routines

`ULONG rexx_init(void)`

This initialization routine is called at the startup time of the **NETTRACE** process. It opens the **utility.library** and the **rexxsyslib.library** to be used by the ARexx code and creates a public **ARexx** message port. The signal mask of the ARexx port is returned upon a successful initialization.

`void rexx_deinit(void)`

Free all resources allocated by the **ARexx** interface module. First the ARexx port is removed from the system's list of message ports so that no-one is able find the port any more to send new messages. Then all pending messages are returned with error code set. Finally the ARexx port is deleted and libraries opened by `rexx_init()` are closed.

Reply Routine

`BOOL rexx_poll(void)`

Checks if any **ARexx** messages has arrived and handles them one at a time. The `parseline()` function is used to parse and execute the given command. Returns `TRUE` if there might still be messages to handle, otherwise the return value is `FALSE`.

4.10 Application Interface Concepts

4.10.1 SocketBase – an Extension of the Task Structure

In Unix systems, where the network code is integrated into the kernel, a process structure holds fields for per-process information of network related data. In AmiTCP/IP, where socket API is implemented as a shared library, each opener gets a newly created *library base* that holds data used by the AmiTCP/IP system. Each library base function makes sure that the caller is from the right Amiga task and refuses to operate if wrong task is attempting to use it (see section 5.5 on page 106 for detailed information).

4.10.2 The System Call Semaphore and Task Priorities

Currently, when program enters to some of socket library functions, it attempts to get semaphore to hold other callers executing library code simultaneously. This is done so, since in Unix system, where this code originally runs, doesn't *pre-empt* process that is executing system call. In BSDSS, where "Unix system calls" run in user mode, system call emulation glue uses a *mutex* to prevent simultaneous use of that part of the server code. Although spl functions are used in NET/2 code to prevent simultaneous access of critical sections, there may still be some sections that leave out protection if system call semaphore is removed. Unnecessary system call semaphore usage is going to be removed in later releases. Hopefully it, and the overhead it generates, becomes obsolete.

The priority of the application process is raised to the same with the AmiTCP/IP, while the application executes AmiTCP/IP code. This is to prevent situations where a process with a low priority gets blocked for a long time while holding e.g. the system call semaphore, since otherwise all networking programs would be blocked with it. This applies to all semaphores used internally by the AmiTCP/IP, not just the system call semaphore.

Initialization Routines

Making application interface visible and operative contains a few steps: call to `api_init()` creates the master socket library base and initializes semaphores and lists API needs. Library base is not in Exec library base list yet. Routine `api_show()` checks first if bsdsocket.library is already in Exec list and if not, calls Exec `AddLibrary()` to make it visible.

AmiTCP/IP can remove socket library from the Exec list at any time, i.e. make it not visible, by calling `api_hide()`. No new socket bases can be opened after this call. Socket bases opened before `api_hide()` operate normally. If AmiTCP/IP calls `api_show()` again, new libraries can be opened.

`api_setfunctions()` takes all socket library bases out of operation. It sets all function vectors in every socket base to return an error. This function is called if AmiTCP/IP `panic()`ed and all libraries are expected to be closed.

When application interface is to be removed from system, `api_deinit()` is called. It waits for all opened libraries to close and then calls expunge function to deallocate the master socket base from the memory.

4.11 Configuration Variables

The configuration variable definitions are stored into a structure named `cfg_variable`. It is defined in **kern/amiga_netdb.h** as follows:

```
/* Variable types */
/* Note: Query calls value, Set calls notify functions */
enum var_type
{
 VAR_FUNC = 1,          /* value is function pointer */
 VAR_LONG,              /* value is pointer to LONG */
 VAR_STRP,              /* value is pointer to string */
 VAR_FLAG,              /* LONG value is set once */
 VAR_INET,              /* struct sockaddr_in */
 VAR_ENUM               /* value is pointer to long, whose value is set
                           according to a enumeration string in notify */
};

typedef LONG
  (*var_f)(struct CSource *args, UBYTE **errstrp, struct CSource *res);
typedef int (*notify_f)(void *pt, LONG new);

/* Configureable variable structure */
struct cfg_variable {
  enum var_type type;           /* type of value */
  WORD  flags;                  /* see below */
  const UBYTE *index;           /* optional index keyword list */
  void  *value;                 /* pointer to value... */
  notify_f notify;              /* notification function */
};

#define boolean_enum (notify_f)"NO=FALSE=OFF=0,YES=TRUE=ON=1"
```

```
/* Variable flags */
#define VF_TABLE    (1<<0) /* with an index... */
#define VF_READ     (1<<1) /* readable */
#define VF_WRITE    (1<<2) /* writeable */
#define VF_CONF     (1<<3) /* writeable only during configuration */
#define VF_RW       (VF_WRITE|VF_READ)
#define VF_RCONF    (VF_CONF|VF_READ)
#define VF_FREE     (1<<8) /* free when replaced? */
```

The configuration file (by default **AmiTCP:db/AmiTCP.config**) is read with the function `readconfig()`. This function also parses the command line arguments.

4.12 Network Database

The network database is initialized by the `init_netdb()` function. This function allocates the `NetDataBase` structure and parses the file **AmiTCP:db/netdb**. The `NetDataBase` structure is defined as follows:

```
struct NetDataBase {
  struct SignalSemaphore ndb_Lock;
  struct MinList         ndb_Hosts;
  struct MinList         ndb_Networks;
  struct MinList         ndb_Services;
  struct MinList         ndb_Protocols;
  struct MinList         ndb_NameServers;
  struct MinList         ndb_Domains;
};
```

This structure contains a lock and lists for the different network database entries. The lock semaphore is obtained in the shared mode for reading, and in the exclusive mode for writing. See section 2.5.1 for the information about the different entries.

Chapter 5

Implementation Notes

This chapter describes some points of the implementation of the AmiTCP/IP. The code that is not changed from the BSD net/2 -release is not reviewed. [Leffler et al 1989] describes the design and implementation of the BSD Unix, including the networking system. [Leffler et al 1991b] is also very helpful.

Most of the knowledge gathered during this project is gained by reading the source code itself. This chapter does not try to make that totally unnecessary.

5.1 Time outs

The Unix `timeout()` function implements the time out needs of the Unix kernel. When kernel code calls `timeout()`, the function given as argument will be called after the specified time out has elapsed. In AmigaOS time outs are provided by the timer device. AmiTCP/IP sends *time out requests* to the device and gets them back when the time specified has elapsed.

The functions called by the time out service are:

`if_slowtimo()`

: This is the time out function of the network interfaces and is defined in **net/if.c**.

`arptimer()`

: Handles the time outs of ARP protocol (**net/sana2arp.c**).

`pfslowtimo()`

: This is the main slow time out function for all protocols. It calls the `pr_slowtimo()` function of each configured protocol. The interval of this timer is 500 ms (**kern/uipc_domain.c**).

`pffasttimo()`

: Is the corresponding function for fast (200 ms) time outs.

The request structure used by AmiTCP/IP has few fields in addition to the normal `struct timerequest`[1]. The structure is defined in **kern/amiga_time.h** as follows:

```
struct timeoutRequest {
  struct timerequest timeout_request;   /* timer.device sees only this */
  struct timeval     timeout_timeval;   /* timeout interval */
  TimerCallback_t    timeout_function;  /* timeout function to be called */
};
```

In this implementation the time out functions themselves do *not* call any time out services, but the functions are called by `timer_poll()`, which is called by `main()` when there might be a message to handle (e.g. when the timer signal is received). `timer_poll()` checks the reply port to see if there really is a message to handle. After the time out function is serviced `timer_poll()` sends the request back to the timer device and returns.

Time out service is implemented by files **kern/amiga_time.c** and **kern/amiga_time.h**. The functions defined are:

`ULONG timer_init(void)`

> Initializes the time out subsystem by allocating the IO requests and opening the timer device. Note that AmiTCP/IP uses functions that are new to version 36 of the timer device, so the code refuses to success with Kickstart 1.3 (or lower).
>
> Returns the signal mask to wait for if successful.

`void timer_deinit(void)`

> Frees all resources used by time out subsystem.

`void timer_send(void)`

> Sends time out requests allocated by `timer_init()` to the timer device.

`struct timeoutRequest *createTimeoutRequest(TimerCallback_t fun, ULONG seconds, ULONG micros)`

> Creates a new time out request. This can be called only after successful `timer_init()`.

`void deleteTimeoutRequest(struct timeoutRequest *tr)`

> Deletes requests created by `createTimeoutRequest()`.

`BOOL timer_poll(VOID)`

> Checks if there are any timer requests in the reply port and if there is handles them by calling the `handleTimeoutRequest()`. Then it sends the request back to the timer device.

[1] See standard Amiga include file **devices/timer.h**

`void handleTimeoutRequest(struct timeoutRequest *tr)`

Inline function which simply calls the function specified in the time out request.

`void sendTimeoutRequest(struct timeoutRequest *tr)`

Inline function which sends the request to the timer device.

See the files **kern/amiga_main.c** and **kern/amiga_time.c** for example of the usage.

5.2 Memory Management

5.2.1 Mbufs

Mbufs are ported just as they are. Memory is allocated in small chunks for both mbufs and clusters. The size of the cluster and the number of mbufs to allocate in one chunk are configurable variables, see section 2.4 for summary of the configurable variables in general.

Note that since data must be copied at the SANA-II interface there is no need to use trailer protocols (whose main gain is avoidance of that copy) and so the mbuf clusters need not begin at page boundaries[2]. This fact lead to the implementation of the clusters where the size of the cluster may be arbitrary (now user configurable) and the reference count of the cluster is stored in a little header before the actual data.

One note on allocating memory for the mbufs: Since the mbuf must be perfectly aligned (i.e. 128 byte mbuf must be 128–aligned), we need to allocate one extra mbuf to be able to align the mbufs in arbitrary memory chunk returned by Exec `AllocMem()`.

The '`canwait`' argument of the mbuf functions is *ignored* by now, more memory will be allocated if limit of maximum memory usage is not hit. This is all right as long as the mbuf allocation functions are not called from interrupts. The only functions in the AmiTCP/IP which may get called from the interrupt code are the SANA-II callback functions `m_copy_from_mbuf()` and `m_copy_to_mbuf()` defined in file **net/sana2copybuff.c**.

The function descriptions for the mbufs are on section 4.5. See files **sys/mbuf.h** and **kern/uipc_mbuf.c** for the actual implementation.

5.2.2 `malloc()` & `free()`

Do not call the `malloc()` *and* `free()` *functions directly!* Since AmiTCP/IP is multi–threaded program these functions are not safe, since they use static data. File **sys/malloc.h** defines two inline functions to be used instead. They are defined as follows:

[2] Well, there is no virtual memory in Amiga either.

`static inline void * bsd_malloc(unsigned long size, int type, int flags)`

This function calls `malloc()` to allocate memory of size `size` and type `type`. The types are defined in **sys/malloc.h** and are used for bookkeeping purposes. The `flags` may have either value `M_WAITOK` or `M_NOWAIT`. The flags are not used by this implementation, however, but are defined for portability.

`static inline void bsd_free(void *addr, int type)`

Frees memory allocated by `bsd_malloc()`.

The `malloc()` and `free()` are made safe by `malloc_semaphore`, which protects mallocs and frees from collisions. It is obtained before the actual calls to `malloc()` or `free()` and released after them.

In addition to these functions **sys/malloc.h** defines macro versions for the most usual usages.

Initialization for these functions is done by:

`BOOL malloc_init(void)`

Initializes the `malloc_semaphore`. This must be called early in the initialization process, since `bsd_malloc()` nor `bsd_free()` cannot be called before `malloc_semaphore` is initialized.

5.3 Concurrency Control

5.3.1 Processor Priority Levels

In Unix systems the critical sections are mainly protected by raising the processor priority level (i.e. preventing interrupts upto a certain level). This crude way might hurt a real time operating system as the AmigaOS, so it can not be implemented as such. Besides, the AmiTCP/IP runs as a normal user level process which has no needed privilege to alter the interrupt levels of the processor.

The implementation in AmiTCP/IP is twofold; a semaphore is used in the debugging mode and task switch prevention is used in the production version. Using semaphore makes debugging easy as single stepping and tracing is possible while keeping the system alive. The semaphore adds certain overhead which is not acceptable in the production version, so the prevention of the task switches is used.

When the preprocessor symbol `DEBUG` is defined the `spl_semaphore` is used. When this semaphore is free the process is at level `SPL0` (user level) and when the semaphore is allocated the process is at "interrupt" level (`SPLSOFTCLOCK`, `SPLNET` or `SPLIMP`), effectively disallowing anyone else to enter critical section. When the symbol `DEBUG` is not defined the functions and macros are defined differently. They manipulate directly the Task Disable Nest Count (`TDNextCnt` field of the `SysBase`). This field is normally used by Exec functions `Forbid()` and `Permit()` which increment and decrement the value of the field, respectively. Since an assembler macro is provided by Commodore for the `Forbid()` function, the semantics of the field cannot change in the future[3]. The semantics of the `TDNestCnt` is little abused by the AmiTCP/IP, however. The value of the spl level in question is directly assigned as the value of the field. This is not visible outside of the AmiTCP/IP, since basically functions which use this field either directly or indirectly (via `Forbid()`/`Permit()`) need to return the value of the field as it was when the function was called. This is also the semantics of the usage of the spl functions.

The `spl_semaphore` is initialized by the function `BOOL spl_init(void)` which is called among the very first functions in the initialization process.

The function `spl_n(int)` (defined in **sys/synch.h**) is used to alter the priority levels. In the non-debugging mode there are two other functions, too: `spl_const()`, which is used with a constant (non-zero) argument, and `spl_0()` which is used to switch to the level 0. In addition, a macro has been defined for each separate level for portability. The macros are as follows:

`spl0()` switches to the normal execution level.

`splsoftclock()` is the level on which timer events are executed.

`splnet()` is the level of the network interrupts in UNIX.

`splimp()` is the highest of the priority levels used in the networking code. For example, mbuf functions are executed at this level.

These macros return the previous level which may then be set back with `splx(int)`-macro, which sets the level to the level given as argument.

5.3.2 Sleeping Facilities

Sleeping facility is implemented by **kern/kern_synch.c**. Processes sleep on a *channel*, which is the key used to identify sleepers. Usually this is some address which is unique to the calling process. Socket base structure for the sleeping task is linked in the sleep queue before the actual sleep is started. This is how the waking task can find the sleeper to wake up when something happens on the channel the process is sleeping on.

The sleep queue is implemented as a hash table, where the channel value is mapped to an index of a sleep queue with the hash function[4] `SLEEP_HASH()` (defined in **kern/kern_synch.c**).

The actual sleep is implemented by sending a time out request for the time out duration to the timer device. The sleep completes on the time out, a wakeup, a break signal or users specified signal.

Note that all critical resources (e.g. semaphores) *must* be freed before sleeping, since otherwise the whole networking code hangs. `tsleep()` does this for you.

[3] If it would, then all the code using that official macro would break.

[4] Actually a macro.

The functions which implement the sleep system are:

`BOOL sleep_init(void)`

This initializes the `sleep_semaphore` and the sleep queues. This must be called before any other functions of this module.

```
int tsleep(struct SocketBase *p, caddr_t chan,
    char *wmesg, const struct timeval *time_out)
```

This function is the function usually called by the processes[5] and implements the sleep by using the other functions of this module. The caller goes to sleep for at most the time specified in the `struct timeval` argument. `chan` is the channel to sleep on. `wmesg` is a string which is marked in the socket base (`p`) as the reason to sleep. Currently no-one ever reads it, though.

`void wakeup(caddr_t chan)`

Wakes up any sleepers on channel `chan`. Searches the sleep queue for entries with key `chan` and wakes them up by first clearing the key (`p_wchan` field of the socket base structure) and then signalling the process with the signal of the time out message. The usage of the sleep queues and the `p_` fields in the socket base structures are protected with `sleep_semaphore`, which must be obtained before even reading the sleep queues.

Note that since a task in AmigaOS *may* get signals anytime, the sleeper checks the `p_wchan` field on reception of the signal and if it is nonzero it goes to sleep again.

```
void tsleep_send_timeout(struct SocketBase *p,
    const struct timeval *time_out)
```

First ensures that the message previously sent to the timer device is back. Then sends timer device a time out request for duration specified in `time_out` if it is not `NULL`. The request sent is allocated when the library is opened.

```
void tsleep_abort_timeout(struct SocketBase *p,
    const struct timeval *time_out)
```

Aborts the time out sent by the `tsleep_send_timeout()`. This function must be used when the time out must be cancelled (when the sleeper is waken up).

This function just sets the timer reply port (`timerPort` field of the socket base) to the mode in which reception of the message does not cause any action.

`void tsleep_enter(struct SocketBase *p, caddr_t chan, char *wmesg)`

Puts the caller on to a sleep queue.

[5] Only `WaitSelect()` system call uses sleeping facilities without this function.

`int tsleep_main(struct SocketBase *p, ULONG wakemask)`

Waits for either time out, wakeup, break or user defined action to happen. The `sigIntrMask` field of the socket base structure defines which signals will cause a break. Return value of `EINTR` is returned if any of the signals specified in that mask are received. In addition the signals are set back with `SetSignal()` for the user program to be able to detect them. The `wakemask` argument specifies a mask for signals which should cause a return from the sleep. In such case the return value `ERESTART` is returned.

On exit the process is guaranteed not to be in the sleep queues any more, but the time out remains active if it is not the reason for return. Return value on wakeup is 0 and on time out `EWOULDBLOCK` is returned.

5.4 Socket Library Creation Procedure

Since a new socket base is created each time a different task opens the AmiTCP/IP socket library, the procedure is a bit more complicated than on libraries where the same library base is returned (See [RKM Libraries 1992]). There is, for example, two socket library bases in use. All code discussed here is located in **api/amiga_api.c**.

5.4.1 Master Library Base

This is the library base that is made shown by `api_show()` (see section 4.10.2). It is placed in Exec's library list. This is of type `struct Library` and contains information that an outsider can read by scanning through the Exec library list. Information available is version and revision numbers and count of tasks that have (application) library base open.

Master library base has only functions `ELL_Open()` and `ELL_Expunge()`. When applications opens the socket library, the Exec calls `ELL_Open()`. This function creates new application socket bases and increments the reference count of open application library bases. If the calling task has a socket base open already, a new socket base is *not* created but the reference count of task's socket base is incremented and the base pointer is returned to the caller. This feature has many useful possibilities, for example in intermediate libraries which need to manipulate the sockets of the calling task.

`ELL_Expunge()` does (not) do one task. When it is called, it checks if there is any libraries still open or if AmiTCP/IP lets this function execute further (in fact, currently this second check is sufficient since only AmiTCP/IP can close the library and it doesn't do it until all bases has been closed. The next `Remove()` is there for future reference too). Then, the memory of the master library base is deallocated and NULL is returned (no AmigaDOS seglist to free). The `SIGBREAKF_CTRL_C` signal that is sent with the global varianble `SB_Expunged` set to `TRUE` notifies `api_deinit()` function about the fact that *now* all libraries are closed.

5.4.2 Application Library Bases

These are the library bases that are returned to the openers of the socket library. In this base the `Open()` function is obsolete since all `OpenLibrary()` calls go through the master socket base. Exec and AmiTCP/IP generated `Expunge()` calls go also through the master socket base.

`UL_Close()` is the close function for all application library bases. First it decrements the reference count of this base and returns `NULL` if there are still references left (again, `NULL` informs Exec that there is no AmigaDOS seglist needed to be removed).

If there are no more references to this library base, following steps are taken to remove it from the memory: All socket descriptors still open are closed. The base is removed from the AmiTCP/IP list of open application socket bases. The timer request is deallocated. Then the library base is removed from the memory and open count of application socket bases is decremented in the master library base. Finally, `ELL_Expunge()` is called if the open count reached zero and the `LIBF_DELEXP` flag is set (by a previous `ELL_Expunge()` call).

5.5 The SocketBase Structure

The `SocketBase` structure is defined in file **api/amiga_api.h** as follows:

```
struct SocketBase {
  struct Library        libNode;
/* "Global" Errno */
  WORD                  errnoSize;
/* -- now we are longword aligned -- */
  UBYTE *               errnoPtr; /* this points to errno */
  LONG                  defErrno;
/* Task pointer of owner task */
  struct Task *         thisTask;
/* task priority changes (WORDS so we keep structure longword aligned) */
  WORD                  myPri;      /* task's priority just after libcall */
  WORD                  libCallPri; /* task's priority during library call */
/* -- descriptor sets -- */
  WORD                  dTableSize;
  WORD                  nextDToSearch;
  struct socket **      dTable;
/* AmiTCP signal masks */
  ULONG                 sigIntrMask;
  ULONG                 sigIOMask;
  ULONG                 sigUrgMask;
/* -- these are used by tsleep()/wakeup() -- */
  char *                p_wmesg;
  queue_chain_t         p_sleep_link;
```

```
  caddr_t              p_wchan;       /* event process is awaiting */
  struct timerequest *  tsleep_timer;
  struct MsgPort *      timerPort;
/* -- pointer to select buffer during Select() -- */
  struct newselbuf *    p_sb;
/* -- per process fields used by various 'library' functions -- */
/* buffer for inet_ntoa */
  char inet_ntoa[20]; /* xxx.xxx.xxx.xxx\0 */
/* pointers for data buffers that MAY be used */
  struct DataBuffer  selitems;
  struct DataBuffer  hostents;
  struct DataBuffer  netents;
  struct DataBuffer  protoents;
  struct DataBuffer servents;
};
```

`libNode` is a normal library base structure and is used by the system. Since in this implementation each opener gets a task specific library base, AmiTCP/IP links all "user library bases" together using `Node` field of `libNode`.

When socket library function encounters an error, it saves the value of the error to the memory location addressed by `errnoPtr`. `errnoSize` specifies the size of the variable pointed by the `errnoPtr`. By default `errnoPtr` points to the `defErrno` but can be changed to point any memory location – usually to the global `errno` variable in the context of the user task.

In entry of each bsdsocket.library function call, value of `thisTask` is compared to the task pointer of calling task to make sure right task is calling the function. This variable is also used to find library base of some executing task.

When task is executing system calls in bsdsocket.library, its process priority is changed to the same as that of the AmiTCP/IP task in order not to hold semaphores and block the whole network system. If some higher priority process becomes active and a lower priority task is holding some vital semaphore of the AmiTCP/IP then the precess cannot continue to run. The manipulation of the process priorities uses `myPri` and `libCallPri` fields of the socket base.

`dTableSize` is the number of current maximum limit of socket descriptors. `dTable` is the descriptor table containing pointers to socket structures, i.e. sockets. `nextDToSearch` makes searching of free socket descriptor faster[6].

`sigIntrMask` is a task specific mask of the signals which should break the `Wait()` call in the `tsleep_main()`. Reception of such signals causes the system calls to return `-1` and the error code pointed by `errnoPtr` to be set to `EINTR`. `sigIOMask` field specifies the signals to send when asynchronous notification is requested. Signals specified in `sigUrgMask` are sent when out of band data is received. These masks implement the functionality of the `SIGIO` and `SIGURG` signals of the Unix systems, respectively. All these

[6]Semantics of allocating lowest free socket descriptor is preserved.

masks can be set with the `SetSocketSignals()` API call. The default mask for the `sigIntrMask` specifies the signal for the `ctrl-C`, other two are zero by default.

The next group of variables are used by `tsleep()` and `wakeup()`. `p_wmesg` points to a string telling the reason why task is sleeping. `p_sleep_link` is used to chain library bases in the sleep queues. Waiting channel key is hold in variable `p_wchan` and data handling time outs in variables `tsleep_timer` and `timerPort` (more about this in section 5.3.2 on page 103).

`WaitSelect()` inserts one `selitem` on each socket it wants event information of. `p_sb` points to a `newselbuf` that contains these items.

Some API functions in the original environment use static buffers to store their output. As a shared library cannot use static buffers (to be re-entrant), the buffers must be allocated dynamically. The `SocketBase` structure has space for the output buffer for the `Inet_Ntoa()` and pointers for other needed buffers. These buffers are allocated only when needed.

5.6 The Application Program Interface

Most of API code is original NET/2 code taken from BSDSS[7]. BSDSS mutexes are replaced by Amiga semaphores and `struct proc` references are changed to references to our socket library base[8].

Many functions used `copyin()` and `copyout()` to copy data around. Those functions copy data between system and user space (different virtual memory mappings) in original BSD Unix system. It is also possible that those functions will fail e.g. if user tries to reference illegal memory locations. In AmiTCP/IP system `copyin()` and `copyout()` functions are replaced with `bcopy()`, arguments are the same but the `bcopy()` never fails. Therefore some obsolete checks are removed from the code.

5.6.1 How API Functions Are Ported

Most functions in our API are ported from BSD Unix system calls. BSD Unix system call interface calls the actual function with three arguments. First is user process structure pointer. Second contains the given arguments and is locally named as `uap` structure. Third argument is a pointer to the return value. The function returns error value or 0 if no error occurred.

The `socket()` function is a good example of this:

```
socket(p, uap, retval)
        struct proc *p;
        register struct args {
```

[7]BSDSS networking code is almost completely the same.
[8]See 5.5.

```
                int     domain;
                int     type;
                int     protocol;
        } *uap;
        int *retval;
```

The system function interface maps directly to Amiga shared library. Since every task has socket library base of its own, Unix process pointer matches to library base pointer given in register `A6`[9]. `uap` arguments are passed in registers normally. Return value is returned in the register `D0` (as any standard C compiler does). So the `*retval` was changed to a local variable `retval` and removed, when not needed. The returned value is -1 on error, in which case the errno is also set to indicate the error (see file **sys/errno.h** for list of error codes), or retval, or 0 if no other return value is needed.

To emulate Unix system call interface, each function first obtains the `syscall_semaphore` (why, see section 4.10.2 on page 96) and while this task is holding the semaphore, no other task can continue to execute the library code[10]. This means that every system call function needs to release `syscall_semaphore` before returning. To accomplish this, each `return` inside the function is changed to `goto Return;` and at label `Return:` is code that releases the `syscall_semaphore`[11].

The default modifications were: changing file descriptor tables and pointers to socket tables and pointers, respectively, removing usage of struct fileops function pointers – replacing them with direct socket functions, changing `copyin()` and `copyout()` functions to `bcopy()`s – no more error checking here needed, and last, changing parameters on `tsleep()` calls.

5.6.2 API Functions Which Needed More Modifications

`IoctlSocket()` (Former `ioctl()`): Non-socket stuff removed, and ioctl code from `soo_ioctl()` inserted.

`WaitSelect()` (Former `select()`) used to count remaining time out time if `tsleep()` returned accidentally too early. AmiTCP/IP uses the timer device for its time outs and the time out request is aborted only when it is needed again, so there is no need to send new time out requests (and to calculate their time out durations). `tsleep()` is broken apart into pieces so that the time out request is sent only once.

`CloseSocket()` Decreases socket's reference count and calls `soclose()` to kill the socket if it becomes zero.

`socket()` **and** `accept()` Added initialization of `so_refcnt`. `fdAlloc()` doesn't bind fd to socket. It is done explicitly.

[9]All Amiga shared libraries expect them to be called relative to register `A6`

[10]Not all functions require obtaining `syscall_semaphore` so those can continue to run.

[11]syscall_semaphore is also freed when library function does `tsleep()`.

Resolver functions used to allocate huge amounts of stack. Now memory is allocated dynamically from the head using `bsd_malloc`.

5.7 Changes in Functions Below API Level

Functions that API functions call are mostly functions that use `struct socket` type arguments, possibly having some other arguments too. In most cases no modifications were needed. There was some modifications, like parameters for `tsleep()` call, which had to be changed throughout the code.

5.7.1 Other Changes

`selscan()` calls `soo_select()` directly, and uses socket pointer instead of file pointer.

`soo_select()` also uses socket pointer instead of file pointer.

`socreate()` allows all sockets to be privileged. This means that user can obtain raw sockets and use normally privileged port numbers.

`sosend()` uses `uioread()` instead of the original `uiomove()`. `sblock()` and `sbwait()` are called with library base pointer as the second argument.

`soreceive()` uses `uiowrite()` instead of the original `uiomove()`. See above about the modification of call `sblock()` and `sbwait()`.

`sorflush()` calls `sblock()` with base pointer argument as NULL.

`sosetopt()` **and** `sogetopt()` : type of `so_linger` and `so_timeo` fields in socket structure is changed from `short int` to `struct timeval`. Manipulation of these data is changed accordingly.

`sbwait()` takes socket base pointer as second argument. It is then passed to `tsleep()` (see section 5.3.2 on page 103).

`sblock()` **and** `sb_lock()`: `sblock()` is a macro that calls `sb_lock()`. Both take socket base pointer as second argument. `sblock()` forwards that pointer directly to `sb_lock()` which, again, passes it to the `tsleep()`.

5.8 Agnet.device

We used the **agnet.device**, an SANA-II test device, to test and develop the AmiTCP/IP network code. The usage and features of the **agnet.device** are described in the section 1.7.1, page 11.

SANA-II is an standard network device driver interface for the Amiga (see [SANA-II 1992 add]). It is an extension to the normal device interface, which is described in the [RKM Libs & Devs 1989]. Device drivers are accessed by their name from the system list. Drivers may be loaded dynamically from the disk, if they are not currently in the main memory. There may be several units sharing common driver code so each (network) device is specified by an unit number and the device driver name.

We wrote the **agnet.device** using the SLIP driver which Commodore has provided as the example code for SANA-II drivers. However, there was some problems with the supplied code. First, the SLIP driver code obviously follows an obsolete SANA-II draft. There was some modifications in the final standard e.g. in the event handling and multicast addressing.

The provided example code was also very fragile, it did not get compiled as such with the newer SAS C version 6. Code depended on some features of the SAS C 5.10. For example, it always expected to find device base pointer in address register A6.

5.8.1 IO Commands

There is a detailed description of SANA-II device commands and functions in [SANA-II 1992 add]. The following IO commands are implemented in **agnet.device**:

`CMD_CLEAR`
: This standard command should return `IOERR_NOCMD` when issued to SANA-II device.

`CMD_INVALID`
: This standard command should return `IOERR_NOCMD`.

`CMD_READ`
: Get the next packet available of the requested packet type. The data returned (via a call to the requester-provided `CopyToBuffer()` function) is the Data Link Layer packet data only. Raw packets are not supported.

`CMD_RESET`
: This standard command should return `IOERR_NOCMD` when issued to SANA-II device.

`CMD_START`
: This standard command should return `IOERR_NOCMD` when issued to SANA-II device.

`CMD_STOP`
: This standard command should return `IOERR_NOCMD` when issued to SANA-II device.

`CMD_UPDATE`
: This standard command should return `IOERR_NOCMD` when issued to SANA-II device.

`CMD_WRITE`
: Send packet to the network. Raw packets are not supported. Sending packet with a broadcast hardware address is not supported.

`S2_BROADCAST`
: Broadcast a packet to the network. Raw packets are not supported.

`S2_CONFIGINTERFACE`
: Configure the interface. The address field will be set depending on the specified hardware type.

`S2_DEVICEQUERY`
: Report the statistical information about the device.

`S2_GETGLOBALSTATS`
: Report accumulated statistics as defined in `struct Sana2Devicestats`.

`S2_GETSTATIONADDRESS`
: Report the "hardware" address for the unit. Before the configuration, the current hardware address has all bits set. The default hardware address is not stored anywhere.

`S2_GETTYPESTATS`
: Report accumulated statistics of the tracked packets.

`S2_OFFLINE`
: Remove interface from service. Flush all queued IO requests.

`S2_ONEVENT`
: Return when specified event(s) occur(s).

`S2_ONLINE`
: Put the interface back in service. This command resets the unit statistics.

`S2_READORPHAN`
: If there is no pending `CMD_READ` request with the type of the received packet, the packet is given to first pending `S2_READORPHAN` request. The data returned (via a call to the requester-provided CopyToBuffer function) is the Data Link Layer packet data only. Raw packets are not supported.

`S2_TRACKTYPE`
: Start tracking of the specified packet type packets.

`S2_UNTRACKTYPE`
Stop tracking of the specified packet type packets.

Uninplemeted IO Commands

These SANA-II device commands are not supported.

`CMD_FLUSH`
This standard command returns `IOERR_NOCMD` when issued to the **agnet.device**.

`S2_ADDMULTICASTADDRESS`
This SANA-II command is not supported. It returns `IOERR_NOCMD`.

`S2_DELMULTICASTADDRESS`
This SANA-II command is not supported. It returns `IOERR_NOCMD`.

`S2_GETSPECIALSTATS`
This SANA-II command is not fully supported. It returns an empty `Sana2SpecialStat` structure.

This command should report accumulated driver specific statistics. This includes ethernet "retries".

`S2_MULTICAST`
This SANA-II command is not supported. It returns `IOERR_NOCMD`.

5.8.2 Initialization Procedure

agnet.device must be started as a DOS process by the **Run** command. The dynamic loading is not yet implemented. Its own startup module, **init.c**, opens needed libraries, initializes device base and calls the `main()` function. Main function opens **timer.device** and initializes the ARexx port; if initialization was successful, it adds the device base to the system list.

In the main loop the device task waits for three different events: user or `Expunge()` generated break signal (`SIGF_BREAK_F`), ARexx messages or user IO request messages.

5.8.3 The Device Interface Functions

The device interface contains 6 standard library calls in the device base. The device may be opened or closed, the IO requests may be initiated or aborted and the system may reclaim storage allocated by the device driver.

These library calls are not normally executed directly by the user code but instead higher level convenience functions in the Exec. The device base library calls are made in the context of the caller, so some synchronous IO commands may be executed like library calls without message passing overhead (quick IO).

The synopsis of the functions specifies the registers where the call parameters are passed (`REG(r`n`)`).

Opening an Unit

An IO device is opened by the Exec function call `OpenDevice()`. When Exec has found the named device driver in the system list it calls the special `DevOpen()` function from the device base. `DevOpen()` function has following synopsis:

```
ULONG ASM DevOpen(REG(a1) struct IOSana2Req *ios2,
                  REG(d0) ULONG unit, REG(d1) ULONG flags)
```

The device open function tries to allocate and initialize various resources for the specified unit if the unit does not already exist. The initialization routine `InitUnit()` is called; if it returns an unit structure, a private `buffermanagement` structure is filled from the user provided tag list. The user supplied IO request is filled with appropriate values.

The promiscuous or exclusive modes are not supported.

`struct AgnetDevUnit *InitUnit(ULONG);`

The initialization routine allocates an unit structure and then calls the configuration routine `ReadConfig()`. If the configuration file was read and interpreted without errors the lists and locks in the unit structure is initialized. The unit is then put online and unit structure is added to the device base.

`BOOL ReadConfig(struct AgnetDevUnit *adu)`

The configuration routine attempts to read in the configuration file for the given unit. It strips the comments out of the file and provides the file as a single line to the parsing routine `ParseConfig()`. If there is no configuration file, the parsing routine is called with an empty line.

Closing an Unit

The accessed unit is closed after use with the Exec `CloseDevice()` function. It runs the `DevClose()` function from the device base. The call has the following synopsis:

`BPTR ASM DevClose(REG(a1) struct IOSana2Req *ios2)`

The device close function calls the unit close function. If the device has been asked to `Expunge()` itself, the close function sends an appropriate signal to the device task. The device task then performs the postponed expunge function.

The `DevOpen()` and `DevClose()` calls are executed while `Forbid()`'den unless the code explicitly `Wait()`'s.

Initiating an IO Request

The `DevBeginIO` function from the device base is called to initiate an IO command. This call is made in the context of the requesting task (task calling Exec functions `DoIO()` or `SendIO()`). The `DevBeginIO()` call has the following synopsis:

`VOID ASM DevBeginIO(REG(a1) struct IOSana2Req *ios2)`

The IO request is sent to the device task for dispatching and execution. Currently all IO requests are executed in the context of the device task, i.e. no quick IO is supported.

Aborting an IO Request

Some IO requests may be aborted before they are completed by Exec function call `AbortIO()`. The aborting function has the following synopsis:

`VOID ASM DevAbortIO(REG(a1) struct IOSana2Req *ios2)`

Currently only the `CMD_READ`, `S2_READORPHAN`, `CMD_WRITE`, `S2_BROADCAST`, and `S2_ONEVENT` IO commands may be aborted. Other IO commands are executed in an atomic way and can not be aborted reliably.

Expunging the Device

The system may reclaim the storage allocated by the device driver by calling the `DevExpunge()` function. Memory reclaiming is normally done in a low memory situation or after a user requested memory flush. The expunging function has the following synopsis:

`VOID ASM DevExpunge(VOID)`

The `DevExpunge()` may not `Wait()` because it may be called from the Exec function `AllocMem()` protected by `Forbid()`/`Permit()` pair.

The `DevExpunge()` function call currently signals the device task, which performs the actual expunging by calling the `DoExpunge()`. Each unit structure is expunged by the function `ExpungeUnit()`.

5.8.4 Packet Delivery

The packet sent to the pseudo network may be delayed or mutated randomly. A special structure (`struct DelayRequest`) stores the packet type and data during the "transmit delay". The delay is implemented by sending this structure as a the timer IO request to the timer device. The packet transmit functions are as follows:

`VOID WritePacket(struct AgnetDevUnit *adu, struct IOSana2Req *ios2)`

The function checks that the unit is on line and checks for the legal data length. Then the routine adds the given IO request (`CMD_WRITE, S2_BROADCAST`) into send queue. If there is a free delay request the `SendPacket()` is called immediately.

`VOID SendPacket(struct DelayRequest *delayed)`

The `SendPacket()` function takes an empty `DelayRequest` structure as its argument. First, it gets a send IO request from the send queue. The request is immediately returned if the packet is "lost" during transmit. This is repeated until a packet is found which is not to be lost.

The packet data, type, length, transmission type and the address of the sender are then copied into the `DelayRequest` structure. If needed, bit errors are made into packet data.

If there is specified delay for the unit, the request is sent to the timer device. After the delay the timer device returns request into the device port and it is dispatched by the `ReceivePacket()` function. If there is no delay, the `ReceivePacket()` is called immediately.

`VOID ReadPacket(struct AgnetDevUnit *adu, struct IOSana2Req *ios2)`

The function checks that unit is online, then adds the read request into an appropriate queue.

`VOID ReceivePacket(struct DelayRequest *delayed)`

The `ReceivePacket()` function calls the `DoReceive()` function on each unit the packet is destined. It then handles the `DelayRequest` structure back to the `SendPacket()` function.

`VOID DoReceive(struct AgnetDevUnit *adu, struct DelayRequest *delayed)`

The `DoReceive()` function tries to find a receive IO message waiting for this particular type packets. If none is found, the first `S2_READORPHAN` IO request is selected.

The `CopyBack()` function sets the appropriate IO request return values and copies the packet data to the receive buffer.

5.8.5 Arexx Interface

The ARexx interface is implemented by using the **SimpleRexx** package provided in Commodore Native Developer Update 2.0 kit. The Arexx commands executed by the **agnet.device** are described in section 1.7.1, page 12.

`LONG ParseRexx(UBYTE *arg, UBYTE **errstr, UBYTE **result)`

The ARexx command string is copied into a buffer and passed to the parser function `ParseRexx()`.

The parser allocates a DOS `struct RDArgs` structure for the ARexx command. This structure holds the information for the DOS parsing functions. The first keyword in the string is read and tokenized by the DOS functions `ReadItem()` and `FindArg()`.

The rest of the line is parsed according this token. In the case of `Query` and `Unit` commands the unit number is read from the command line and the rest of the line is passed to the appropriate functions.

```
LONG ParseConfig(struct AgnetDevUnit *adu, struct RDArgs *rdargs,
                 STRPTR *errormessage)
```

The command line stored into the `RDArgs` structure is parsed by the DOS function `ReadArgs()`. The parsed configuration information are then gathered, its legality is checked and it is stored into the unit structure.

```
LONG ParseQuery(struct AgnetDevUnit *adu, struct RDArgs *rdargs,
                STRPTR *errstr, STRPTR *result)
```

Like the `ParseConfig()`, `ParseQuery()` parses the command line by the `ReadArgs()`. It then fills the reply buffer by the requested configuration parameter values and then makes an ARexx string out of the buffer. This string is then returned to the ARexx process.