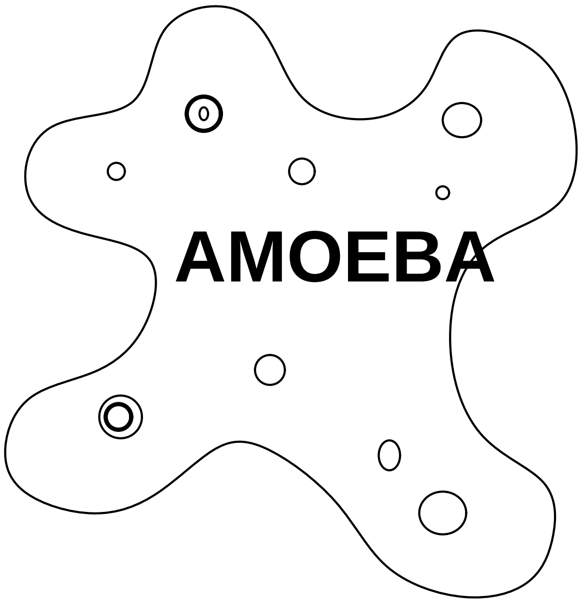
The Amoeba Reference Manual

Programming Guide



Contents

1 About This Manual

Intended Audience

This manual is intended for use by people writing programs for Amoeba. It is assumed that the reader is familiar with the introductory material in the User Guide.

Scope

This manual explains the basic concepts of Amoeba from a programmer's perspective, the paradigms, the programming tools, writing clients and servers and using the various module interfaces. The manual also provides a description of the various library modules and routines available under Amoeba.

Advice on the Use of the Manual

The introductory material describing the programming paradigms of Amoeba should be thoroughly understood before attempting the development of important software. The tutorial examples are designed to give quick hands-on experience, which should demonstrate the important concepts. It is well worth typing them in and running them.

Throughout the manual there are references of the form *prv*(L), *ack*(U) and *bullet*(A). These refer to manual pages at the back of each guide. The following table explains what each classification refers to and in which guide the manual page can be found.

Letter	Type	Location
(A)	Administrative Program	System Administration Guide
(H)	Include file	Programming Guide
(L)	Library routine	Programming Guide
(T)	Test Program	System Administration Guide
(U)	User Utility	User Guide

There is a full index at the end of the manual providing a permuted cross-reference to all the material. It gives references to both the manual pages and the introductory material which should direct you to enough information to clarify any difficulties in understanding.

Additional material may be required for a complete understanding of the programming environment. In particular the ANSI C and POSIX 1003.1 standards and some knowledge of UNIX are helpful.

Warnings

As stated in the release notes, there have been several major changes between Amoeba 4 and Amoeba 5. More are expected in Amoeba 6. It is important that the guidelines for software development be followed to avoid upgrade difficulties. In particular, it is vital that all remote procedure calls be encapsulated in stub routines to isolate the rest of the code from the possible syntactic and semantic changes to this interface.

2 Paradigms and Implementations

2.1. The Client-Server Model

The programming model of Amoeba is based on having programs, possibly running on different processors, working together to perform a task. This is inherent in the nature of distributed systems. The programs communicate with each other using various methods, but typically remote procedure call (RPC). RPCs may go over a network (typically Ethernet) or to another process on the same computer. The location of the sender and receiver of a message is invisible to the sender and receiver. Various functions can be implemented in programs that offer a service. For example, the file system provides a file creation and storage service for other programs. It can accept requests to create, read and write files. Such programs which accept requests to perform operations for another program are called *servers*. Programs which send requests to servers are known as *clients.* Some programs may at various times in their execution switch between being a client and a server and so the categorization cannot always be strictly applied to a particular program. For example, a file server may become a client of the time−of−day server when it needs to obtain the time for a time-stamp on a file.

Many standard servers are provided with Amoeba and so applications can usually be written as clients of one or more of the standard services. However it is also possible to write specialized servers to manage a data base or some other kind of object.

Amoeba is an *object-based* system. (N.B. Class inheritance is not *enforced*. It is permitted. Therefore Amoeba is **not** an *object-oriented* system but it is relatively simple to build an object-oriented system on top of Amoeba.) In general, each server manages a single type of object. For example, a file server manages files and a disk server manages disks. It is possible to write servers that manage several types of objects simultaneously but this can easily lead to inelegant and difficult to maintain programs. Extensions are planned to give proper support for this to avoid the inelegancies (see below).

2.2. Capabilities

When an object is created, the server that does the creation also creates a *capability* for the object and gives it to the client. Access to objects is exclusively via capabilities. In principle programs should never directly examine the contents of capabilities. Servers sometimes need certain data from a capability, such as the object number or the rights. This information should be extracted using the specially provided routines (see *prv*(L)). Similarly, generation of capabilities for new objects should be done with these routines and the function *uniqport*(L). Any programs not conforming to this may cease to function correctly in later releases of Amoeba since in a subsequent release the sizes of the fields in a capability may change.

The present structure of a capability is shown in figure 2.1. It is 128 bits long and contains four fields. The first field is the *server port*, and is used to identify the (server) process that manages the object. It is in effect a 48-bit random number chosen by the server.

The second field is the *object number*, which is used by the server to identify which of its objects is being addressed. Together, the server port and object number uniquely identify the object on which the operation is to be performed.

The third field is the *rights* field, which contains a bit map telling which operations the

48	24	8	48
Server port	Object number	Rts	Check field

Fig. 2.1. A capability. The numbers give the current sizes in bits.

holder of the capability may perform. If all the bits are 1s, all operations are allowed. However, if some of the bits are 0s, the holder of the capability may not perform the corresponding operations.

To prevent users from just turning all the 0 bits in the rights field into 1 bits, a cryptographic protection scheme is used. When a server is asked to create an object, it picks an available slot in its internal tables and puts the information about the object in there along with a newly generated 48-bit random number. The index into the table is put into the object number field of the capability, the rights bits are all set to 1, and the *check field* is generated by XOR-ing the rights field with the random number and running the result through a (publicly known) *one-way function*. The encryption process is built into the routines *prv_encode* and *prv_decode* (see *prv*(L)). (N.B. For efficiency, the one-way function is not applied when all the rights are ‘‘on’’ since there is no gain in security since all rights are available to the holder of the capability anyway.)

The server can construct a new capability with a subset of the rights by turning off some of the rights bits before XOR-ing the rights field with the random number. Each server should provide a rights-restriction service for clients.

When a capability arrives at a server, the server uses the object field to index into its tables to locate the information about the object. It performs an XOR of the original random number in its table with the rights field of the capability. This number is then run through the one-way function. If the output of the one-way function agrees with the contents of the check field, the capability is deemed valid, and the requested operation is performed if its rights bit is set to 1. Due to the fact that the one-way function cannot be inverted, it is extremely improbable that a user can ‘‘decrypt’’ a capability to get the original random number in order to generate a false capability with more rights.

2.3. Remote Procedure Call

The RPC mechanism of Amoeba is based on four primitives: *trans*, *getreq*, *putrep* and *timeout* (see *rpc*(L)). These functions form the basis of inter-process communication. The RPC interface is built on top the Fast Local Internet Protocol (FLIP). This network protocol provides automatic shortest path routing of messages and provides automatic gatewaying between connected networks. See the bibliography in the *Release Notes* for more details about FLIP.

The *getreq* and *putrep* functions are used by servers. Once a server has initialized itself it performs the *getreq* operation with the port that it wishes to listen to. The *getreq* function blocks the server until a client sends a request using *trans* (see below). When a request arrives, the server checks that the capability is valid and has sufficient rights. Then it performs the requested operation. Finally it sends a reply back to the client using *putrep*.

N.B. *getreq* and *putrep* must always come in balanced pairs. If a thread attempts to do a *putrep* when it has not done a *getreq* or does two *getreq* calls without an intervening *putrep* (or *grp_forward*) then this constitutes a programming error and the program will terminate with an uncatchable exception.

The *trans* function is used by clients to send requests to servers. The port of the server to which the RPC must go is embedded in the first parameter to *trans*. The *trans* call blocks until the server sends a reply. The Amoeba kernel on which the *trans* is executed attempts to locate the server by broadcasting a message with that port in it (unless it is on the same host). If another kernel has a server waiting for a request on that port it responds and the RPC is sent to the server which then handles the request and returns a reply. The kernel keeps a cache of known locations of ports to improve performance for subsequent RPCs. However it is recommended that *trans* not be called directly in client programs. Rather it should be embedded in a procedure call that handles the marshaling of data and the sending of the message to the server. This shall be described in more detail below.

Note that giving a null port (i.e., all bits 0) to any of the RPC routines will result in an RPC error. The null port is not a valid address for a server. The routine *uniqport*(L) which generates ports will not produce the null port.

The *timeout* function is used to set the amount of time spent searching for the server when doing a transaction. This is known as the *locate timeout*. If a server is unavailable (possibly due to network partitioning or a system crash) then programs attempting to communicate with that server will hang for the length of the timeout. The default timeout is 5 seconds. Such a delay will be very irritating to users so it is important in *interactive* programs that timeouts be set to a value short enough to avoid irritating the user but long enough to have a chance to find the server. Two seconds is the recommended minimum for interactive programs. Any shorter than this may report failure to find the server even though it is available. For non-interactive programs such as compilers, a longer timeout is usually acceptable and increases reliability.

2.3.1. Upcoming Features

Note that *timeout* does not limit the amount of time that a server may spend executing a request. This timeout is known as the *service timeout*. It is not yet implemented but may well be in a future release. There are also plans to add the ability for *getreq* to listen to multiple ports. This will be useful in the wide-area network gateways and any other server that wishes to handle more than one type of object.

Because of the improvements planned for this interface in the next major release of Amoeba, it is important that calls to the RPC interface routines be embedded in stub routines. A *stub routine* provides an interface specific to a particular server that hides the details of the data marshaling and message passing. For example, the routine *b_read* (see *bullet*(L)) provides the file read interface to the Bullet Server. This routine bundles up the relevant information for a read command and sends it in the form required by the Bullet Server. This has the effect of hiding the packaging of the data required by the server and providing an simple interface which minimizes the chance of error. It is possible to automatically generate the stubs and data unmarshaling in the server using *ail*(U). This is recommended since it will allow simple regeneration of servers for new releases of Amoeba. There is more information about how to use *AIL* in the programming tools section of this manual and in the section on writing clients and servers.

2.4. Group Communication

In addition to the RPC interface for sending point-to-point messages, it is possible to send a message to a group of processes. This is sometimes known as *multicast* or *broadcast*. This is a powerful tool with many uses. For example it is used for efficient implementation of distributed shared-memory in Orca and for replication of data such as files. The Amoeba group communication primitives are described in *grp*(L). They are implemented on top of a reliable multicast protocol and take advantage of any hardware support for multicast provided by the network. (If there is none they revert to a series of point-to-point messages to implement the multicast.) The group communication takes advantage of the auto-routing capabilities of the FLIP network protocol.

2.5. Processes and Threads

Under many systems (for example, UNIX), programs consist of a *text segment* which contains the program instructions, a *data segment* which contains initialized global data, a *bss segment* which contains uninitialized global data and a *stack segment*. These four segments are typically all in the same address space and together with the program counter and stack pointer, when running, comprise a *process*. The process has a single *thread* of execution. Under Amoeba each process can have multiple threads of execution all running within the same address space. Each thread has its own program counter, stack pointer and stack segment but all share the text, data and bss segments.

Since a certain amount of global data was necessary per thread, an extra data type was created. Each thread keeps a pointer to data allocated at run-time which all the functions in that thread can gain access through a pointer maintained by the library. This data is known as the *glocal* data (because it is local to a thread but global to all the functions in that thread).

2.5.1. Multithreaded Programming

The multithreaded environment is implemented with a set of routines described in *thread*(L). It provides routines for creating threads, destroying them and support for glocal data.

Careful thought needs to be given to programming in a multithreaded environment. Synchronization between threads is non-trivial since it is possible that a library function may perform a blocking operation and cause rescheduling at an undesirable moment. The normal way to organize synchronization between threads is by using the mutex and/or semaphore packages (see *mutex*(L) and *semaphore*(L)). The routine *threadswitch* also provides a mechanism for a thread to give up control voluntarily.

2.6. Scheduling

At any instant many threads and processes may be runnable so a scheduler is needed to determine which to run next. The scheduler uses a priority scheme as follows:

> Enqueued interrupt handlers in the kernel have the highest priority, then runnable kernel threads and finally, if none of the above are runnable, user processes.

Processes are scheduled round-robin on a time-slice (typically 10 milliseconds) or when a process blocks. Thus between processes there is preemptive scheduling. However, between individual threads within a process there is per default no preemption. Thus, if a process is rescheduled due to a time-slice then when it is restarted the same thread of that process will be resumed. Rescheduling between threads only occurs when a thread executes a blocking primitive. This includes the transaction primitives *trans* and *getreq* (see *rpc*(L)) and the mutex primitives (see *mutex*(L)). There is also a special function *threadswitch* (see *thread_scheduling*(L)) which allows a process to reschedule itself without executing a function which blocks. The kernel is regarded as a process, so there is no preemption between kernel threads. They are only rescheduled when they block.

It is also possible to allow a user process to use preemptive scheduling between its threads. This is done on a per process basis so that not all processes have to use preemptive scheduling. Threads can also be assigned priorities and there is time-slicing between threads in this case. See *thread_scheduling*(L) for details.

2.7. Signals

Signals are a method of sending unsolicited information to a thread or process. Lightweight signals are sent to threads. The manual page *signals*(L) explains how they are implemented and used. Heavyweight signals are used to stun a process (not a thread). This is described in manual page for *pro_stun* (see *process*(L)). It is important to note that if a thread is in a transaction the signal handler of the thread will not be called until the transaction completes. The signal will be propagated to the server which may either ignore it or stop what it is doing and reply at once. Thus, when a user interrupts a process with CTRL−C, it may not die immediately, but hang until the server replies to the current transaction.

There is a special kind of stun which allows a *trans* to be broken off without waiting for the server to reply. This is also described in *process*(L).

2.8. POSIX Emulation

The POSIX emulation under Amoeba currently provides a subset of the POSIX 1003.1 standard. In subsequent releases more of the POSIX functionality will be added, according to what is implementable. At present, the basic calls such as *open*, *close*, *read* and *write* are implemented along with the directory package, the time of day routines, signal handling and many miscellaneous functions.

Several POSIX concepts do not make any sense in Amoeba. One is the idea of a user id or group id. Security is based on capabilities. Identification of users by something other than capabilities only complicates matters. An emulation of user ids is provided but the group id is always 1. (See *posix*(L) for more details.)

The Bullet File Server creates files atomically. One of the side effects of this is that it is not possible to read a file as it is being created. Commands such as *tail −f* are therefore difficult to implement.

To manage shared file descriptors each user needs a *session server* (see *session*(U)). This also deals with the null device, */dev/tty*, pipes and the forking of processes. Forking is in fact slow and inefficient under Amoeba. This is because when a new process is started it will usually be started on a different processor from the parent process. The idea of copying the text of a totally irrelevant program to another processor is inefficient in the context of a distributed system. Therefore there is a routine called *newproc*(L) which can be used to start a new process more efficiently than with *fork* and *exec*.

See *posix*(L) for further details of POSIX conformance.

2.8.1. Porting UNIX Programs

In general, porting most programs from UNIX should not present problems if they are POSIX conformant, restrict themselves to the calls defined in P1003.1 and do not use UNIX specific networking. Programs that use sockets or the select system call may prove difficult to port. It some cases the functionality of sockets may be able to be reproduced with the support routines for the TCP/IP server.

2.9. Languages

Currently several programming languages are provided with Amoeba. The Amsterdam Compiler Kit (ACK) provides STD C, Pascal, BASIC, FORTRAN 77 and Modula-2. The standard run-time libraries are available for these languages (see *libmod2*(L) and *libpc*(L)). However, the Amoeba library routines have only been provided for C at present. Note that the ACK loader is capable of linking the routines from the C libraries with Modula-2 programs. See *libmod2*(L) for details.

The stub generator *ail*(U) only provides the possibility of generating stubs in C.

A special language for parallel programming has been developed. It is called *Orca*. It can be obtained from the Vrije Universiteit.

The GNU C compiler is available for all architectures under Amoeba.

For further details on programming languages see chapter 5, *Programming Languages*.

2.9.1. Programming in C

The ACK C compiler distributed with Amoeba is a STD C compiler. The Amoeba source code was written in K&R C but function prototypes and new−style declarations are gradually replacing the old-style code. The STD C compiler can be used to compile the system. The library routines and include files required by STD C are provided.

To compile a program under Amoeba it is sufficient to call *cc*(U). This will automatically link programs with the correct run-time start-off and the relevant libraries. The two libraries that are of importance are *libamoeba.a* and *libajax.a*. The latter is only required when using functions that interact with the session server. (That is, when a program will use shared file descriptors.) If linking programs without using *cc* then *libajax.a* should be linked before *libamoeba.a*, which should always be the last library linked.

Note that just calling *cc* will result in compiling a program for the default architecture. It is better to specify the target architecture if there is any doubt or if cross-compiling for another architecture (see the **−m** option of *ack*(U) for details).

If developing a software system it is recommended that *amake*(U) be used instead of *make*(U). *Amake* is more reliable under Amoeba (it does not use the unreliable time-stamps on directory entries) and provides much more flexibility. There are many examples of Amakefiles in the Amoeba distribution.

2.9.2. Standard Types

There are many standard types defined using *typedef*. The most important ones are defined in the include file *amoeba.h*. These include the definitions of *capability*, *port*, *header* and *errstat*. For the sake of portability it is important that these types be used uniformly throughout applications. In subsequent releases the actual underlying types will almost certainly change. For example, *bufsize* is presently an unsigned 16-bit integer. It is expected that this will be upgraded to a signed 32-bit integer in a future release.

Since many types are used it is worthwhile for the programmer to become acquainted with them.

In addition to the type definitions there are several macros defined which relate to the types. It is worthwhile becoming acquainted with these as well. Two macros are of special importance. The macro `PORTCMP` compares two *ports.* It returns 1 if they are identical and 0 otherwise. Similarly the macro `NULLPORT` returns 1 if the given port is null and zero otherwise. An example of their use is given below.

```
#include "amoeba.h"

port p1;
port p2;

if (PORTCMP(&p1, &p2))
    printf("p1 and p2 are equal\n");

if (NULLPORT(&p1))
    printf("p1 is the null port!\n");
```

2.9.3. Standard Commands and Errors

Like many other operating systems, Amoeba provides a standard set of error messages. These standard error codes are defined in the include file *stderr.h*. (See also *err*(L).) Servers may define extra non–standard error codes which their clients must be made aware of, but these numbers may not conflict with the standard set of errors. Otherwise it is not possible for the error message routines to print meaningful strings corresponding to error codes. Furthermore, there are several macros for manipulating an error status. At present certain unsigned fields in headers are required to carry error codes which are signed quantities. However to write code that will run on both the current and future releases with simple recompilation, two macros have been provided. The first is called `ERR_STATUS.` It takes a single argument (an unsigned 16-bit integer) and determines whether or not it is an error code. A related macro `ERR_CONVERT` converts an unsigned 16-bit integer to the type *errstat*. In practice almost no user programs will confront these problems if they confine themselves to calling stub routines whose external interface only deals with the type *errstat*. Only the writers of stubs will face these compatibility problems and by using the Amoeba Interface Language (AIL) the stubs can be automatically generated and thus eliminate nearly all compatibility issues.

Command codes sent to servers have also been standardized. There are standard command codes defined in *stdcmd.h* which must be implemented by all servers where applicable.

(Those that are not applicable should be ignored and a good status returned.) Furthermore there are reserved command codes for standard servers defined in the include file *cmdreg.h*. At the end of this file is the special symbol `UNREGISTERED_FIRST_COM` which defines where local server's command codes should begin. The reason for this division of command codes is to allow the possibility of inheriting interfaces in an object-oriented fashion. AIL is able to take advantage of this when building server interfaces. (See the Programming tools section for details of AIL class inheritance.) Similarly, if a server needs to use non-standard error codes, they should also be numbered from `UNREGISTERED_FIRST_ERR`.

3 Writing Servers and Clients

Servers are used to perform specialized functions and form the basis of centralized object management. There are many standard servers delivered with Amoeba. These include the Bullet File Server, the Soap Directory Server, the Virtual Disk Server and the Run Server (which does load balancing at process creation-time), among many others. To develop basic applications and port utilities from other operating systems the standard servers are usually adequate. However for some applications this is not enough. This section gives an indication of how to write your own servers and clients and points out some of the pitfalls to avoid.

Servers normally have a fixed set of commands that they accept. They sit in a loop accepting commands, executing them and sending a reply. To simplify the client's interaction with a server it calls a procedure, known as a *stub*. This packages the command and data in the manner required by the server and sends it to the server using a remote procedure call (RPC) and then awaits a reply. The procedure also unpacks any data in the reply and hands it back to the client. The stub allows details of the server interface and the possible byte-order differences between client and server's processors to be hidden from the programmer. Below is a description of how to write servers, their stubs and the clients that use them.

The following description of how to write clients and servers begins by showing what the C code actually looks like for a client / server interface. The reason for this is purely educational. In fact it is seldom necessary to write the server loop and client stub code yourself. AIL (the Amoeba Interface Language) can be used to generate this code automatically. The implementation of the commands must be written by the programmer of course. One advantage of this is that if the RPC interface ever changes, then recompiling the client / server interface with AIL will probably be sufficient to port the server to the new RPC interface. Another advantage is that it is in principle easier to write a specification of the interface in AIL than to write the code for every interface stub.

Before writing servers and clients it is important to understand the system of F-boxes described in many of the papers about Amoeba. These have been implemented in software. The idea of the F-box is that it prevents someone starting a server which intercepts RPCs intended for some other (important) server by deliberately listening to the same port.

The F-box mechanism works as follows. When a *getreq* call is done it listens to the *get-port* for the server. This is a port known only to the server. The server's kernel passes this port through the routine *priv2pub*(L) to encrypt the port. The result is known as the *put-port* and the kernel accepts requests sent to the *put-port*. *Priv2pub* is a one-way function so it is practically impossible to deduce the *get-port* given the *put-port*.

Before doing a *getreq* the server needs to publish the *put-port* through which it can be found by clients. Therefore it also passes the *get-port* through *priv2pub* and publishes the result in the directory server in a place accessible to its clients. In principle it is not possible to listen for requests on the public port unless the *get-port* is known.

This actually provides very little security if it is possible for someone to boot modified Amoeba kernels on your network. They can add a primitive for listening to *put-ports* or modify a kernel binary to avoid the conversion of *get-port* to *put-port* when *getreq* is called. The same deficiency occurs if the F-box is implemented in hardware and it is possible to connect to the network without an intervening F-box.

3.1. Servers

Most servers have the same basic structure, although exceptions do exist. Each has a main program which initializes any global variables, publishes the port of the server so that clients can use it and starts one or more instances of the server loop. The server loop is typically an infinite loop with a *getreq* call at the top, a switch on the command to be executed and a *putrep* call at the bottom. If the server is multithreaded then resource locking may also take place within the infinite loop as well. It is a good idea to be able to see the matching pairs of lock/unlock and *getreq* and *putrep*. Unmatched pairs can be difficult to debug so it is better to keep them in the same function wherever possible.

If a server is expected to have more than one client then it is important to make it multithreaded. This allows the server to take advantage of any possibility for parallelism in handling requests. As soon as one thread gets a request, another will be ready to run with the next RPC that arrives. It also means that there is a high probability that the server is listening to its port and so locates of the server will not fail. Each thread in the server that handles clients will normally run the same code. Of course, there may well be several other threads inaccessible to clients performing different tasks within the server.

The best way to understand the structure of a server is by looking at one. Below is an example of a server written in C. The example is of a trivial server which when sent two long integers reverses their order in the message and sends them back to the client. This is a ridiculous thing for a server to do since it is trivially done in the application itself but it does demonstrate the important aspects of writing servers. Note that some unimportant details have been left out to avoid obscuring the structure.

We begin with the include file *this_server.h* which is particular to this server. It defines the request buffer size `REQ_BUFSZ`, the command codes for the commands that the server accepts, the relevant rights bits for the server and the number of threads that the server should run. They are in an include file since some of this information is also needed by the client or for tuning.

Note that it is not acceptable to use just any command codes. You must begin at the value `UNREGISTERED_FIRST_COM` defined in the file *cmdreg.h* as explained in the chapter *Paradigms and Implementations*.

```
/* the include file this_server.h */

#define NTHREADS        5
#define STACKSZ         0x4000

#define SERVER_CAP      "/home/this_server/default"
#define REQ_BUFSZ       (2*sizeof(long))

/* commands accepted by the server */
#define DO_SWAP         UNREGISTERED_FIRST_COM
#define DO_NOTHING      (DO_SWAP+1)

/* rights bits in capabilities */
#define RGT_DONOTHING   ((rights_bits)0x01)
#define RGT_DOSWAP      ((rights_bits)0x02)
```

Before we look at how the server loop works it is worth knowing how to invoke multiple threads all running the server loop. This is done using the thread interface routines (see *thread*(L)). The use of *priv2pub* is also demonstrated.

```
#include "amoeba.h"
#include "stderr.h"
#include "module/rnd.h"
#include "module/name.h"
#include "this_server.h"

void server_loop();
capability get_cap;
port check_field;

main()
{
    capability put_cap;
    errstat err;
    int i;

    /* initialize the server’s get capability */
    uniqport(&get_cap.cap_port);
    uniqport(&check_field);
    if (prv_encode(&get_cap.cap_priv, (objnum) 0,
                       (rights_bits) 0xFF, &check_field) != 0) {
          printf("Could not initialize capability\n");
          exit(1);
    }

    priv2pub(&get_cap.cap_port, &put_cap.cap_port);
    put_cap.cap_priv = get_cap.cap_priv;
    /* publish put capability in directory server */
    if ((err = name_append(SERVER_CAP, &put_cap)) != STD_OK) {
          printf("Cannot append to %s: %s\n", SERVER_CAP, err_why(err));
          exit(1);
    }

    /* start the NTHREADS server_loop threads */
    for (i = 0; i < NTHREADS-1; i++)
          if (!thread_newthread(server_loop, STACKSZ, (char *) 0, 0))
              printf("Cannot start server thread\n");

    server_loop();
    /*NOTREACHED*/
}
```

Note that if *main* exits then the process will die. Therefore it is important that *main* either goes into an infinite loop (in this case by also executing the server loop) or goes to sleep (for example, on a mutex).

In the *server_loop* routine it is assumed that the initialization of global variables has taken place in the main program which started it. In particular the port listened to by the server should have been initialized and published under the name defined by `SERVER_CAP` in the include file *this_server.h*. The directory where this capability is to be found must be writable by the person starting the server. Otherwise the new capability cannot be registered.

```
#include "amoeba.h"
#include "cmdreg.h"
#include "stderr.h"
#include "this_server.h"

extern capability get_cap;
extern errstat do_swap(), do_nothing();

/*ARGSUSED*/
void
server_loop(param, size)
char * param;
int size;
{
    header hdr;
    char reqbuf[REQ_BUFSZ], repbuf[REQ_BUFSZ];
    bufsize n;

    for (;;) {
         hdr.h_port = get_cap.cap_port;
         n = getreq(&hdr, reqbuf, REQ_BUFSZ);
         if (ERR_STATUS(n)) {
             printf("getreq failed: %s\n", err_why((errstat) n));
             exit(1);
         }
         switch (hdr.h_command) {
         case DO_SWAP:
             hdr.h_status = do_swap(&hdr, reqbuf, n, repbuf);
             break;

         case DO_NOTHING:
             hdr.h_status = do_nothing(&hdr, reqbuf, n, repbuf);
             break;

         default:
             hdr.h_status = STD_COMBAD;
             hdr.h_size = 0;
             break;
         }
         putrep(&hdr, repbuf, hdr.h_size);
    }
}
```

The routine *err_why* (see *error*(L)) produces a human readable error message for standard error codes. For unknown error codes it merely returns the string

```
      amoeba error nn
```

where *nn* is the error code returned. Note that *getreq* takes a *get-port* as parameter in the *hdr* variable. However when it returns the *hdr* contains the *put-port* and other fields sent by the client so it is important to reinitialize *hdr* before each call to *getreq*.

The routine *do_swap* will check the capability in the header, take the *n* bytes of data in the request buffer and perform the command specified by `DO_SWAP`. It fills the reply buffer to be returned to the client and returns the status of the command. It returns the status for clarity. If every entry in the switch sets the return status of the command it is easy to check that all pathways return a status. Other arrangements are also possible.

Now we shall consider the structure of the routine *do_swap*. The structure of the data that it expects in the request buffer determines what data the client must send to it. Since the server should also provide the client programs with stub routines to communicate with the server, the routine do_swap will largely determine the client stubs. However, we first need a way of validating capabilities. Below is an example routine. Note that it is important to distinguish between a bad capability and a capability with insufficient rights when generating an error code.

```
#include "amoeba.h"
#include "cmdreg.h"
#include "stderr.h"
#include "module/prv.h"

extern port check_field;

errstat
check_cap(priv, required_rights)
private * priv;
rights_bits required_rights;
{
    rights_bits rights;

    if (prv_number(priv) != 0 ||
                   prv_decode(priv, &rights, &check_field) != 0)
          return STD_CAPBAD;

    if ((rights & required_rights) != required_rights)
          return STD_DENIED;

    return STD_OK;
}
```

In this example the server has only one capability of interest, namely the capability for the server itself. There are no object capabilities. Note well that the capability is checked against the original *check_field* and not against the *get_cap*. If you have more than one object then the original check field for each object should be stored with the per-object information and not the *get* or *put* capability. The *check_field* can be used to generate the capability, but the converse may not be true. The *prv*(L) routines should always be used the to check capabilities to protect programs from future changes to data structures.

```
#include "amoeba.h"
#include "cmdreg.h"
#include "stderr.h"
#include "module/buffers.h"
#include "this_server.h"

errstat check_cap();

errstat
do_swap(hdr, req, reqsz, rep)
header * hdr;
char * req;
bufsize reqsz;
char * rep;
{
    char * p;
    char * q;
    long val1, val2;
    errstat err;

    hdr->h_size = 0; /* in case of error */
    if ((err = check_cap(&hdr->h_priv, RGT_DOSWAP)) != STD_OK)
         return err;

    q = req;
    p = rep;

    /* get the two longs from the request buffer */
    q = buf_get_long(q, req + reqsz, &val1);
    q = buf_get_long(q, req + reqsz, &val2);
    /* if q is 0 then the request buffer did not contain two longs! */
    if (q == 0)
         return STD_ARGBAD;

    /* now put them into the reply buffer in reverse order */
    p = buf_put_long(p, rep + REQ_BUFSZ, val2);
    p = buf_put_long(p, rep + REQ_BUFSZ, val1);
    /* if p is 0 then the buffer of the server is too small */
    if (p == 0)
         return STD_SYSERR;

    hdr->h_size = p - rep;
    return STD_OK;
}
```

The routine *do_swap* uses the *buf_get* and *buf_put* routines (see *buffer*(L)) to receive and send data. These ensure that no byte-order problems occur between client and server. The stub routines should also use the corresponding routines. The *buf_get* and *buf_put* routines are very pleasant to use since it is not necessary to check for errors until the last operation is completed. They check for buffer overflow and thus prevent overrunning array bounds.

They return the next free position in the buffer if they succeed and the null pointer if they fail. If given a null pointer as a buffer argument then they immediately return a null pointer. Thus if any of the calls returns a null pointer then all subsequent calls will also do so and the error test need only be done once at the end of a series of *get* or *put* calls.

In this example the server expects two longs in the request buffer. It swaps them and inserts the result into the reply buffer using *buf_put_long* and returns the total size of the reply buffer to the *server_loop* which sends the reply.

3.2. Clients

It is now time to consider what is needed on the client side. It is possible to write a client program which knows all about the data formats that the server expects. However it is almost certainly simpler to provide the client with single routine to call which hides the details of marshaling the data and sending it to the server. One reason for doing this is that the implementation of the RPC is then hidden from the client program and any of the planned changes to that interface need only be modified in a few stub routines and the programs should then continue to function after recompilation. If AIL is used then recompilation of the sources should be sufficient.

A stub routine for `DO_SWAP` might look like the following.

```
#include "amoeba.h"
#include "cmdreg.h"
#include "stderr.h"
#include "module/buffers.h"
#include "this_server.h"

errstat
swap_longs(svr, val1, val2)
capability * svr;
long * val1;
long * val2;
{
    header hdr;
    bufsize t;
    char buf[REQ_BUFSZ];
    char * p;

    hdr.h_port = svr->cap_port;
    hdr.h_priv = svr->cap_priv;
    hdr.h_command = DO_SWAP;
    p = buf_put_long(buf, buf + REQ_BUFSZ, *val1);
    p = buf_put_long(p, buf + REQ_BUFSZ, *val2);
    if (p == 0) /* REQ_BUFSZ is too small! */
          return STD_SYSERR;
    t = trans(&hdr, buf, REQ_BUFSZ, &hdr, buf, REQ_BUFSZ);
    if (ERR_STATUS(t))
          return ERR_CONVERT(t);
    if (ERR_STATUS(hdr.h_status))
          return ERR_CONVERT(hdr.h_status);
    p = buf_get_long(buf, buf + t, val1);
    p = buf_get_long(p, buf + t, val2);
    if (p == 0) /* server returned garbage */
          return STD_SYSERR;
    return STD_OK;
}
```

In general it is important that stub routines return an error status. This makes it much simpler for the client to deduce the cause of errors. Returning a null pointer or something similar on failure complicates things and a single *errno*-like variable is complicated and cumbersome in a multithreaded process.

Note once again the use of the *buf_put* and *buf_get* routines for marshaling data. There are routines for various data types. It is important that the server and the stub marshal the data in the same way or the results may be unfortunate.

In this example we used the same buffer to send data as to receive it. Similarly we used the same *header* struct to send and receive information. This is perfectly acceptable practice but the following should be noted. The fields *h_command* and *h_status* occupy the same position in the header struct. (The field name *h_status* is actually a #define.) Therefore if a stub has a loop of RPCs rather than a single RPC then the *h_command* field must be reinitialized each time around the loop.

There are some fields in the *header* struct which are intended for small amounts of out of band data. These could have been used to send and receive small pieces of data. In this example we have two longs and they do not both fit into the header. All byte swapping of header fields is done automatically by the kernel and if only one or two small integers need to be sent then it is better to use the header and give the parameter `NILBUF` and buffer size zero (0) since this will lead to greater efficiency in the marshaling of data.

The macros `ERR_STATUS` and `ERR_CONVERT` are also important. In the current version of Amoeba the error status returned by *trans* or in *hdr.h_status* is an unsigned 16-bit integer. It is intended that this be changed to a signed 32-bit integer in a future release. However, error codes are signed. To avoid massive changes later, stubs return a signed 32-bit integer and the grizzly details of the current implementation are thus hidden from the clients and servers. However it is necessary to make conversions that deal correctly with sign extension. Therefore the macro `ERR_STATUS` is provided to determine whether what was returned is indeed an error and `ERR_CONVERT` is provided to correctly convert the unsigned 16-bit integer to a signed 32-bit quantity.

Now let us consider what the main program of the client looks like. One of the things it needs to do is to get the capability for the server it needs to talk to. In general this is not done by the stub routine because there may be more than one instance of a particular server and the main program should be able to choose the server. It gets the server's capability by doing a *name_lookup* of the name that the server should have used to publish its capability. In this case it is the name defined by `SERVER_CAP` in *this_server.h*.

Having converted the command line arguments to integers, the next step is to call the stub routine. This is a normal procedure call, but the stub then takes the server's *put* capability and the two integers and puts them into a message which it then sends out. The message it gets back has the two numbers swapped around and it sets them into the original variables, but now in the reverse order.

Thus the client might be implemented as follows:

```
#include "amoeba.h"
#include "stderr.h"
#include "stdlib.h"
#include "this_server.h"
#include "module/name.h"

main(argc, argv)
int argc;
char * argv[];
{
    errstat err;
    capability svr;
    long num1, num2;

    if (argc != 3) {
          printf("Usage: %s num1 num2\n", argv[0]);
          exit(1);
    }

    /* get the capability for the server */
    if ((err = name_lookup(SERVER_CAP, &svr)) != STD_OK) {
          printf("%s: lookup of %s failed: %s\n",
                   argv[0], SERVER_CAP, err_why(err));
          exit(2);
    }

    num1 = strtol(argv[1], (char **) 0, 10);
    num2 = strtol(argv[2], (char **) 0, 10);

    /* send the command to the server */
    if ((err = swap_longs(&svr, &num1, &num2)) != STD_OK) {
          printf("%s: swap_longs failed: %s\n", argv[0], err_why(err));
          exit(3);
    }

    printf("new number order is %d %d\n", num1, num2);
    exit(0);
}
```

3.3. Using AIL

Although it is possible to write the server loop and client stubs by hand it is also possible to generate the stubs and server loop using the Amoeba Interface Language (AIL). All the details of marshaling and unmarshaling data are handled by AIL. The precise details of how to use AIL are described in *ail*(U) and the chapter *Programming Tools* in this volume of the manual.

An important thing to watch out for is that AIL-generated servers or clients do *not* necessarily cooperate with hand-written servers or clients. The reason for this is that AIL may decide to marshal some of the parameters of an operation into the header. Another optimization it currently performs is the marshaling of parameters to and from request and reply buffers. Rather than marshaling the parameters in a byte-order independent way (using the *buf_put* routines, described earlier) it sends them over in the original byte-order. Additionally, one of the header fields is initialized to allow the server to see whether it is necessary to byte-swap parameters contained in the buffer.

In the next two sections we will reimplement the server described in the preceding sections. Only this time we will show how much of the labour required can be transferred to AIL.

3.3.1. Writing a Server with AIL

In this section we will show how the example swap server can be implemented using AIL. The most important part of the AIL specification is the interface itself. An interface is defined by means of *class* definitions. For each command included in the class, the class definition specifies the *prototype* of the corresponding client stub. For example, the interface for the integer swap server could be specified as follows:

```
/* class file swap.cls */

#include "cmdreg.h"
#include "this_server.h"

class swaplong [DO_SWAP .. DO_NOTHING] {

        do_swap [DO_SWAP] (*,
                in out long first,
                in out long second
        );

        do_nothing [DO_NOTHING] (*,
                in out long first,
                in out long second
        );

};
```

It defines the class *swaplong* which consists of commands in the range from `DO_SWAP` up to `DO_NOTHING`. The client stub for the command `DO_SWAP` has three parameters: the ‘‘*’’, representing the capability for the object on which the operation is performed, and two long parameters that are passed by reference (the *in out* clauses are responsible for that). Note that although the server in this case is not really object-based, the ‘‘*’’ is still required in order to address the server.

The class presented here is very simple; in general it may also define class-specific types and constants. Moreover, a class can inherit procedures, types and constants from other classes, thus making it very easy to write a server supporting a subset of the standard server commands (see *std*(L)). In fact all servers should do this. This is outside the scope of this introduction however. For details see the AIL manuals.

To make use of the AIL class definition, the command *ail*(U) must be told what to do with it. To generate the server main-loop which was hand-coded earlier, the following file should be given as argument to *ail*:

```
/* file swapsvr.gen */

#include "swap.cls"

generate swaplong {
    client_interface;
    server;                 /* Generate server main loop */
};
```

The C source for the server main-loop will be produced in the file *ml_swaplong.c*. This file will include a header file *swaplong.h* which is generated as well.

The main-loop generated is called *ml_swaplong*, and it has a single parameter supplying the *get-port* the server should be listening to. The code for the function *server_loop* suddenly becomes very simple:

```
#include "amoeba.h"
#include "cmdreg.h"
#include "stderr.h"
#include "this_server.h"

errstat ml_swaplong();
capability get_cap;

void
server_loop()
{
    errstat err;

    err = ml_swaplong(&get_cap.cap_port);   /* should never return */
    printf("getreq failed: %s\n", err_why(err));
    exit(1);
}
```

As soon as a request has been accepted by *ml_swaplong* it will unmarshal the parameters of the command and call a user-supplied function taking care of the actual implementation of the command. After that, it will marshal the result values (as specified by the client interface), return the reply to the client, and wait for a new request.

The implementation functions for our swap server are called *impl_do_swap* and *impl_do_nothing*. The code for *impl_do_swap* now becomes:

```
errstat
impl_do_swap(hdr, first, second)
header *hdr;
long   *first, *second;
{
    errstat check_cap();
    long temp;
    errstat err;

    if ((err = check_cap(&hdr->h_priv, RGT_DOSWAP)) != STD_OK)
        return err;

    temp = *second;
    *second = *first;
    *first = temp;

    return STD_OK;
}
```

Observe that, compared with function *do_swap* in the hand-coded example, the code is now almost trivial, because the programmer is not bothered with the marshaling of input and output data.

Combined with a completely analogous definition of *impl_do_nothing* and the functions *main* and *check_cap* taken from the original implementation, our AIL-based server is now complete.

3.3.2. Writing a Client with AIL

Now we turn our attention to the client side. We have two options here. When a client program only consists of a command-line interface to a command supported by the server, AIL is able to generate the program all by itself. The other possibility is to let AIL only generate the client stubs, writing the rest of the program by hand. Although the latter option is a bit more work, it is also more flexible.

We will first show an AIL specification generating the entire client program *do_swap*:

```
#include "swap.cls"

generate swaplong {
    client_interface;
    client_stubs(do_swap);
    command(do_swap);
};
```

The clause *client_interface* causes a header file *swaplong.h* containing C prototypes for the client stubs to be generated. The client stub for the operation `DO_SWAP` is requested by the *client_stubs* clause. It will be produced in the file *do_swap.c*. The rest of the program, which takes care of parsing the arguments, calling the stub and printing the results is requested by the *command* clause. It will be produced in the file *cmd_do_swap.c*.

It must be noted that the command produced is a direct reflection of the *do_swap* operation as defined in the class definition. More specifically, it requires the user to specify the name of the swap server capability as the first argument. The resulting program has the following interface:

```
$ swap /home/this_server/default 0 1
first=1
second=0
```

The other possibility is to use AIL only to generate the client stub *do_swap*. The AIL specification becomes:

```
#include "swap.cls"

generate swaplong {
    client_interface;
    client_stubs(do_swap);
};
```

The main program is then the same as in the hand-written case, except that the client stub is called *do_swap* rather than *swap_longs*.

4 Amoeba Under Unix

The FLIP network protocol can be installed in many popular versions of UNIX so that processes running under UNIX can communicate using the Amoeba RPC. Many programs that run under Amoeba will also allow access to Amoeba from UNIX. However not all programs that run under native Amoeba will work under UNIX since UNIX does not support multithreaded processes in a the same way as Amoeba. Many threads packages are available that use non-preemptive scheduling between threads so compatibility with Amoeba should be possible but later releases may present problems. There is a special library for linking programs that use the Amoeba RPC but which are to run under UNIX. It is called *libamunix.a*. To see how to use it, examine the *Amakefile*s that are distributed with Amoeba for generating programs to run under UNIX.

Because of limitations in the UNIX networking code, the Amoeba protocols inside a UNIX kernel are usually slower than under native Amoeba. This can lead to the fast client – slow server problem. In this case a server running on a slow system receives a request, executes it and sends a reply. However, before it can issue another *getreq* the client is trying to send another request. Since the server is temporarily unavailable, the client issues one or more locate broadcasts which can grow into broadcast storms. Under Amoeba this can be avoided by having multiple threads doing a *getreq* on the same port so that a locate always succeeds. The problem of *busy* servers will be solved in a subsequent release.

Two commands which are very useful under UNIX are *tob*(U) and *fromb*(U) which allow files to be sent between UNIX and the Bullet file server.

Amoeba has a special server for converting between its network protocol and TCP/IP (see *ipsvr*(A)). With the aid of this and the program *ttn*(U) (an implementation of telnet) it is possible to do remote login to UNIX hosts. Electronic mail can also be transferred between Amoeba and public TCP/IP networks using the TCP/IP server.

5 Programming Languages

This section describes the various programming languages available under Amoeba.

The Amsterdam Compiler Kit (ACK) provides much of the language support for Amoeba. Several languages are currently provided: STD C, Pascal, BASIC, FORTRAN 77 and Modula-2. The target architectures currently supported by the compiler under Amoeba are the MC680x0 family and the Intel 80386.

Details of how to invoke these compilers are provided in the User Guide. See *ack*(U), *cc*(U), *f77*(U), *m2*(U) and *pascal*(U) for further details. In addition an archiver, name list and sizing program are provided. See *aal*(U), *anm*(U) and *size*(U) for details.

The internal details of ACK, including the link editor and information about descriptions files for front and back-ends is rather esoteric and irrelevant to most programmers. However for completeness there is a description of the structure of the compiler kit and the manual pages for the linker and description files. Also included is a specification of the conformance of the STD C, Pascal and Modula 2 compilers.

In addition to ACK the GNU C Compiler is available under Amoeba for all supported architectures. It is freely available but does not form part of the distribution. For details of its use see the separate GNU documentation for the compiler. The command *f2c*(U) can be used as a preprocessor for the GNU C compiler to compile Fortran 77.

5.1. The Amsterdam Compiler Kit (ACK)

Introduction

The Amsterdam Compiler Kit (ACK) is an integrated collection of programs designed to simplify the task of producing portable (cross) compilers and interpreters. For each language to be compiled, a program (called the *front end*) must be written to translate the source program into a common intermediate code called `EM` (short for Encoding Machine). The `EM` code can be optimized and then either directly interpreted or translated into the assembly language of the desired target machine using a program called the *back end*.

The ideal situation would be an integrated system containing a family of (cross) compilers, each compiler accepting a standard source language and producing code for a wide variety of target machines. Furthermore, the compilers should be compatible, so programs written in one language can call procedures written in another language. This has largely been achieved.

ACK consists of a number of parts that can be combined to form compilers (and interpreters) with various properties. It is based on the idea of a *Universal Computer Oriented Language* (UNCOL) that was first suggested in 1960, but which never really caught on then. The problem which UNCOL attempts to solve is how to make a compiler for each of *N* languages on *M* different machines without having to write *N* x *M* programs.

The UNCOL approach is to write *N* ‘‘front ends,’’ each of which translates one source language to a common intermediate language, UNCOL , and *M* "back ends," each of which translates programs in UNCOL to a specific machine language.

Under these conditions, only *N* + *M* programs must be written to provide all *N* languages on all *M* machines, instead of *N* x *M* programs.

Previous attempts to develop a suitable intermediate language have not become popular because they were over ambitious and attempted to support all languages and target architectures. ACK’s approach is more modest: it caters only to algebraic languages and machines whose memory consists of 8-bit bytes, each with its own address. Typical languages that could be handled include Ada, ALGOL 60, ALGOL 68, BASIC, C, FORTRAN, Modula, Pascal, PL/I, PL/M, and RATFOR, whereas COBOL, LISP, and SNOBOL would be less efficient. Examples of machines that could be included are the Intel 8080 and 80x86, Motorola 6800, 6809, and 680x0, Zilog Z80 and Z8000, DEC PDP-11 and VAX, and IBM 370 but not the Burroughs 6700, CDC Cyber, or Univac 1108 (because they are not byte-oriented). With these restrictions, the old UNCOL idea has been used as the basis of a practical compiler-building system.

An Overview of the Amsterdam Compiler Kit

The tool kit consists of eight components:

1. The preprocessor.
2. The front ends.
3. The peephole optimizer.
4. The global optimizer.
5. The back end.
6. The target machine optimizer.
7. The universal assembler/linker.
8. The utility package.

A fully optimizing compiler, depicted in figure 5.1, has seven cascaded phases. Conceptually, each component reads an input file and writes a output file to be used as input to the next component. In practice, some components may use temporary files to allow multiple passes over the input or internal intermediate files.

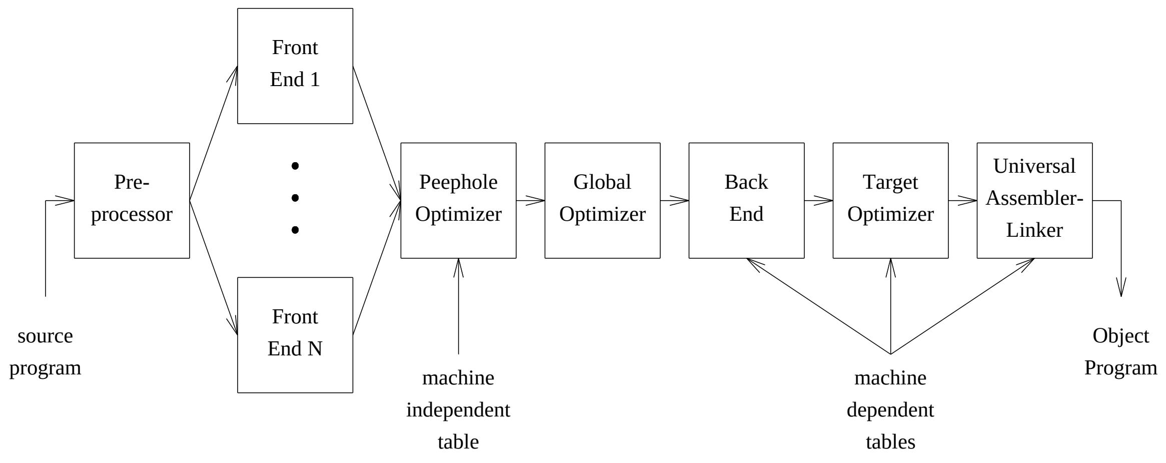


Fig. 5.1. Structure of the Amsterdam Compiler Kit.

In the following paragraphs we will briefly describe each component. A program to be compiled is first fed into the (language independent) preprocessor, which provides a simple macro facility, and similar textual facilities. The preprocessor's output is a legal program in one of the programming languages supported, whereas the input is a program possibly augmented with macros, etc.

This output goes into the appropriate front end, whose job it is to produce intermediate code. This intermediate code (our UNCOL) is the machine language for a simple stack machine called `EM`. A typical front end might build a parse tree from the input, and then use the parse tree to generate `EM` code, which is similar to reverse-Polish. In order to perform this work, the front end has to maintain tables of declared variables, labels, etc., determine where to place the data structures in memory, and so on.

The `EM` code generated by the front end is fed into the peephole optimizer, which scans it with a window of a few instructions, replacing certain inefficient code sequences by better ones. Such a search is important because `EM` contains instructions to handle numerous important special cases efficiently (for example, incrementing a variable by 1). The strategy is to relieve the front ends of the burden of hunting for special cases because there are many front ends and only one peephole optimizer. By handling the special cases in the peephole optimizer, the front ends become simpler, easier to write and easier to maintain.

Following the peephole optimizer is a global optimizer, which unlike the peephole optimizer, examines the program as a whole. It builds a data flow graph to make possible a variety of global optimizations, among them, moving invariant code out of loops, avoiding redundant computations, live/dead analysis and eliminating tail recursion. Note that the output of the global optimizer is still EM code.

Next comes the back end, which differs from the front ends in a fundamental way. Each front end is a separate program, whereas the back end is a single program that is driven by a machine dependent driving table. The driving table for a specific machine tells how the EM code is mapped onto the machine's assembly language. Although a simple driving table might just macro expand each EM instruction into a sequence of target machine instructions, a much more sophisticated translation strategy is normally used, as described later. For speed, the back end does not actually read in the driving table at run-time. Instead, the tables are compiled along with the back end in advance, resulting in one binary program per machine.

The output of the back end is a program in the assembly language of some particular machine. The next component in the pipeline reads this program and performs peephole optimization on it. The optimizations performed here involve idiosyncrasies of the target machine that cannot be performed in the machine-independent EM-to-EM peephole optimizer. Typically these optimizations take advantage of special instructions or special addressing modes.

The optimized target machine assembly code then goes into the final component in the pipeline, the universal assembler/linker. This program assembles the input to object format, extracting routines from libraries and including them as needed.

The final component of the tool kit is the utility package, which contains various test programs, interpreters for EM code, EM libraries, conversion programs, and other aids for the implementer and user.

5.1.1. ACK Description File Reference Manual

Introduction

The program *ack*(U) internally maintains a table of possible transformations and a table of string variables. The transformation table contains one entry for each possible transformation of a file. Which transformations are used depends on the suffix of the source file. Each transformation table entry tells which input suffices are allowed and what suffix/name the output file has. When the output file does not already satisfy the request of the user (indicated with the flag **–c.suffix**), the table is scanned starting with the next transformation in the table for another transformation that has as input suffix the output suffix of the previous transformation. A few special transformations are recognized, among them is the combiner, which is a program combining several files into one. When no stop suffix was specified (flag **–c.suffix**) *ack* stops after executing the combiner with as arguments the – possibly transformed – input files and libraries. *Ack* will only perform the transformations in the order in which they are presented in the table.

The string variables are used while creating the argument list and program call name for a particular transformation.

Which Descriptions Are Used

Ack always uses two description files: one to define the front-end transformations and one for the machine dependent back-end transformations. Each description has a name. First the way of determining the name of the descriptions needed is described.

When the string environment variable ACKFE is set *ack* uses that to determine the front-end table name, otherwise it uses **fe**.

The way the back end table name is determined is more convoluted.
First, when the last file name in the program call name is not one of *ack* or the front-end call-names, this file name is used as the back end description name. Second, when the **–m** is present the **–m** is chopped off this flag and the rest is used as the back end description name. Third, when both failed the shell environment variable ACKM is used. Last, when ACKM was not present, the default back end is used, determined by the definition of ACKM in h/local.h. The presence and value of the definition of ACKM is determined at the compile time of *ack*.

Now we have the names, but that is only the first step. *Ack* stores a few descriptions at compile time. These descriptions are simply files read in at compile time. At the moment of writing this document, the descriptions included are: pdp, fe, i86, m68k2, vax2 and int. The name of a description is first searched for internally, then in *lib/descr/*name, then in *lib/*name*/descr*, and finally in the current directory of the user.

Using The Description File

Before starting on a narrative of the description file, the introduction of a few terms is necessary. All these terms are used to describe the scanning of zero terminated strings, thereby producing another string or sequence of strings.

Backslashing

All characters preceded by \ are modified to prevent recognition at further scanning. This modification is undone before a string is passed to the outside world as argument or message. When reading the description files the sequences \\, \# and \<newline> have a special meaning. \\ translates to a single \, \# translates to a single # that is not recognized as the start of comment, but can be used in recognition and finally, \<newline> translates to nothing at all, thereby allowing continuation lines.

Variable replacement

The scan recognizes the sequences {{, {NAME} and {NAME?text} where NAME can be any combination if characters excluding ? and } and text may be anything excluding }. (\} is allowed of course.) The first sequence produces an unescaped single {. The second produces the contents of the NAME, definitions are done by *ack* and in description files. When the NAME is not defined an error message is produced on the diagnostic output. The last sequence produces the contents of NAME if it is defined and text otherwise.

Expression replacement

Syntax: (*suffix sequence*:*suffix sequence*=*text*)

Example: (.c.p.e:.e=tail_em)

If the two suffix sequences have a common member – .e in this case – the text is produced. When no common member is present the empty string is produced. Thus the example given is a constant expression. Normally, one of the suffix sequences is produced by variable replacement. *Ack* sets three variables while performing the diverse transformations: HEAD, TAIL and RTS. All three variables depend on the properties *rts* and *need* from the transformations used. Whenever a transformation is used for the first time, the text following the *need* is appended to both the HEAD and TAIL variable. The value of the variable RTS is determined by the first transformation used with a *rts* property.

Two run-time flags have effect on the value of one or more of these variables. The flag **–.suffix** has the same effect on these three variables as if a file with that **suffix** was included in the argument list and had to be translated. The flag **–r.suffix** only has that effect on the TAIL variable. The program call names *acc* and *cc* have the effect of an automatic **–.c** flag. *Apc* and *pc* have the effect of an automatic **–.p** flag.

Line splitting

The string is transformed into a sequence of strings by replacing the blank space by string separators (nulls).

IO replacement

The > in the string is replaced by the output file name. The < in the string is replaced by the input file name. When multiple input files are present the string is duplicated for each input file name.

Each description is a sequence of variable definitions followed by a sequence of transformation definitions. Variable definitions use a line each, transformations definitions consist of a sequence of lines. Empty lines are discarded, as are lines with nothing but comment. Comment is started by a # character, and continues to the end of the line. Three special two-characters sequences exist: \#, \\ and \<newline>. Their effect is described under 'backslashing' above. Each – nonempty – line starts with a keyword, possibly preceded by blank space. The keyword can be followed by a further specification. The two are separated by blank space.

Variable definitions use the keyword *var* and look like this:

```
var NAME=text
```

The name can be any identifier, the text may contain any character. Blank space before the equal sign is not part of the NAME. Blank space after the equal is considered as part of the text. The text is scanned for variable replacement before it is associated with the variable name.

The start of a transformation definition is indicated by the keyword *name*. The last line of such a definition contains the keyword *end*. The lines in between associate properties to a transformation and may be presented in any order. The identifier after the *name* keyword determines the name of the transformation. This name is used for debugging and by the **–R** flag. The keywords are used to specify which input suffices are recognized by that transformation, the program to run, the arguments to be handed to that program and the name or suffix of the resulting output file. Two keywords are used to indicate which run-time start-offs and libraries are needed. The possible keywords are:

from
: followed by a sequence of suffices. Each file with one of these suffices is allowed as input file. Preprocessor transformations do not need the *from* keyword. All other transformations do.

to
: followed by the suffix of the output file name or in the case of a linker the output file name.

program
: followed by name of the load file of the program, a path name most likely starts with either a / or {EM}. This keyword must be present, the remainder of the line is subject to backslashing and variable replacement.

mapflag
: The mapflags are used to grab flags given to *ack* and pass them on to a specific transformation. This feature uses a few simple pattern matching and replacement facilities. Multiple occurrences of this keyword are allowed. This text following the keyword is subjected to backslashing. The keyword is followed by a match expression and a variable assignment separated by blank space. As soon as both description files are read, *ack* looks at all transformations in these files to find a match for the flags given to *ack*. The flags **–m**, **–o**, **–O**, **–r**, **–v**, **–g**, **––c**, **–t**, **–k**, **–R** and **––.** are specific to *ack* and not handed down to any transformation. The matching is performed in the order in which the entries appear in the definition. The scanning stops after first match is found. When a match is found, the variable assignment is executed. A * in the match expression matches any sequence of characters, a * in the right hand part of the

assignment is replaced by the characters matched by the * in the expression. The right hand part is also subject to variable replacement. The variable will probably be used in the program arguments. The **–l** flags are special, the order in which they are presented to *ack* must be preserved. The identifier LNAME is used in conjunction with the scanning of **–l** flags. The value assigned to LNAME is used to replace the flag. The example further on shows the use of all this.

args

The keyword is followed by the program call arguments. It is subject to backslashing, variable replacement, expression replacement, line splitting and IO replacement. The variables assigned to by *mapflags* will probably be used here. The flags not recognized by *ack* or any of the transformations are passed to the linker and inserted before all other arguments.

stdin

This keyword indicates that the transformation reads from standard input.

stdout

This keyword indicates that the transformation writes on standard output.

optimizer

The presence of this keyword indicates that this transformation is an optimizer. It can be followed by a number, indicating the "level" of the optimizer (see description of the **–O** option in the *ack*(U) manual page).

priority

This optional keyword is followed by a number. Positive priority means that the transformation is likely to be used, negative priority means that the transformation is unlikely to be used. Priorities can also be set with a *ack*(U) command-line option. Priorities come in handy when there are several implementations of a certain transformation. They can then be used to select a default one.

linker

This keyword indicates that this transformation is the linker.

combiner

This keyword indicates that this transformation is a combiner. A combiner is a program combining several files into one, but is not a linker. An example of a combiner is the global optimizer.

prep

This optional keyword is followed an option indicating its relation to the preprocessor. The possible options are:

always	the input files must be preprocessed
cond	the input files must be preprocessed when starting with #
is	this transformation is the preprocessor

rts

This optional keyword indicates that the rest of the line must be used to set the variable RTS, if it was not already set. Thus the variable RTS is set by the first transformation executed which such a property or as a result from *ack*'s program call name (acc, cc, apc or pc) or by the **–.suffix** flag.

need

This optional keyword indicates that the rest of the line must be concatenated to the NEEDS variable. This is done once for every transformation used or indicated by one of the program call names mentioned above or indicated by the **−.suffix** flag.

Conventions Used In Description Files

Ack reads two description files. A few of the variables defined in the machine specific file are used by the descriptions of the front-ends. Other variables, set by *ack*, are of use to all transformations.

Ack sets the variable EM to the home directory of the Amsterdam Compiler Kit. The variable SOURCE is set to the name of the argument that is currently being massaged, this is useful for debugging. The variable SUFFIX is set to the suffix of the argument that is currently being massaged.

The variable M indicates the directory in lib/{M}/tail_..... and NAME is the string to be defined by the preprocessor with −D{NAME}. The definitions of {w}, {s}, {l}, {d}, {f} and {p} indicate EM_WSIZE, EM_SSIZE, EM_LSIZE, EM_DSIZE, EM_FSIZE and EM_PSIZE respectively.

The variable INCLUDES is used as the last argument to *cpp*. It is used to add directories to the list of directories containing #include files.

The variables HEAD, TAIL and RTS are set by *ack* and used to compose the arguments for the linker.

Example

Description for front-end

```
name cpp                                    # the C-preprocessor
                                            # no from, it's governed by the P property
        to .i                               # result files have suffix i
        program {EM}/lib/cpp                # path name of loadfile
        mapflag −I* CPP_F={CPP_F?} −I*      # grab −I.. −U.. and
        mapflag −U* CPP_F={CPP_F?} −U*      # −D.. to use as arguments
        mapflag −D* CPP_F={CPP_F?} −D*      # in the variable CPP_F
        args {CPP_F?} {INCLUDES?} −D{NAME} −DEM_WSIZE={w} −DEM_PSIZE={p} \
            −DEM_SSIZE={s} −DEM_LSIZE={l} −DEM_FSIZE={f} −DEM_DSIZE={d} <
                                            # The arguments are: first the −[IUD]...
                                            #  then the include dir's for this machine
                                            #  then the NAME and size values finally
                                            #  followed by the input file name
        stdout                              # Output on stdout
        prep is                             # Is preprocessor
end
name cem                                    # the C-compiler proper
        from .c                             # used for files with suffix .c
        to .k                               # produces compact code files
        program {EM}/lib/em_cem             # path name of loadfile
        mapflag −p CEM_F={CEM_F?} −Xp       # pass −p as −Xp to cem
        mapflag −L CEM_F={CEM_F?} −l        # pass −L as −l to cem
        args −Vw{w}i{w}p{p}f{f}s{s}l{l}d{d}  {CEM_F?}
                                            # the arguments are the object sizes in
                                            # the −V... flag and possibly −l and −Xp
        stdin                               # input from stdin
        stdout                              # output on stdout
        prep always                         # use cpp
        rts .c                              # use the C run-time system
        need .c                             # use the C libraries
end
name decode                                 # make human readable files from compact code
        from .k.m                           # accept files with suffix .k or .m
        to .e                               # produce .e files
        program {EM}/lib/em_decode          # path name of loadfile
        args <                              # the input file name is the only argument
        stdout                              # the output comes on stdout
end
```

Example of a back end, in this case the EM assembler/loader.

```
var w=2                                 # wordsize 2
var p=2                                 # pointersize 2
var s=2                                 # short size 2
var l=4                                 # long size 4
var f=4                                 # float size 4
var d=8                                 # double size 8
var M=em22
var NAME=em22                           # for cpp (NAME=em22 results in #define em22 1)
var LIB=lib/{M}/tail_                   # part of file name for libraries
var RT=lib/{M}/head_                    # part of file name for run-time start-off
var SIZE_FLAG=−sm                       # default internal table size flag
var INCLUDES=−I{EM}/include             # use {EM}/include for #include files
name asld                               # Assembler/loader
        from .k.m.a                     # accepts compact code and archives
        to e.out                        # output file name
        program {EM}/lib/em_ass         # load file path name
        mapflag −l* LNAME={EM}/{LIB}*   # e.g. −ly becomes
                                        #{EM}/mach/int/lib/tail_y
        mapflag −+* ASS_F={ASS_F?} −+*  # recognize −+ and −−
        mapflag −−* ASS_F={ASS_F?} −−*
        mapflag −s* SIZE_FLAG=−s*       # overwrite old value of SIZE_FLAG
        args {SIZE_FLAG} \
            ({RTS}:.c={EM}/{RT}cc) ({RTS}:.p={EM}/{RT}pc) −o > < \
            (.p:{TAIL}={EM}/{LIB}pc) \
            (.c:{TAIL}={EM}/{LIB}cc.1s {EM}/{LIB}cc.2g) \
            (.c.p:{TAIL}={EM}/{LIB}mon)
                                        # −s[sml] must be first argument
                                        # the next line contains the choice for head_cc or
                                        # head_pc and the specification of in- and output.
                                        # the last three args lines choose libraries
        linker
end
```

The command *ack −mem22 −v −v −I../h −L −ly prog.c* would result in the following calls:

```
1)  /lib/cpp −I../h −I/usr/em/include −Dem22 −DEM_WSIZE=2 −DEM_PSIZE=2 \
        −DEM_SSIZE=2 −DEM_LSIZE=4 −DEM_FSIZE=4 −DEM_DSIZE=8 prog.c
2)  /usr/em/lib/em_cem −Vw2i2p2f4s2l4d8 −l
3)  /usr/em/lib/em_ass −sm /usr/em/lib/em22/head_cc −o e.out prog.k
    /usr/em/lib/em22/tail_y /usr/em/lib/em22/tail_cc.1s
    /usr/em/lib/em22/tail_cc.2g /usr/em/lib/em22/tail_mon
```

5.1.2. The Link Editor

Synopsis

`led [`*options*`]` *file* `...`

Description

Led is a link editor (linker) for object modules, created by one of the ACK assemblers. It combines several object programs into one, resolves external references, and searches archives. Usually, *led* is not invoked directly by the user, but via the generic compiler driver *ack*(U). In the simplest case several object *files* are given, and *led* combines them, producing an object module which can be either fed into a machine specific conversion program or become the input for a further *led* run. (In the latter case, if there are no unresolved references, the **−r** option must be given to preserve the relocation information.) The resulting object file of *led* is named **a.out**.

The argument routines are concatenated in the order specified. The entry point of the output is the beginning of the first routine.

If an argument is an archive, its table of contents is searched for names which are undefined at the point at which the argument is encountered in the argument list. This procedure is repeated as long as unresolved references are satisfied.

Options

Led understands several options. The flags **−c**, **−r**, **−s**, and **−u** should appear before the file names.

−a*dd:nnnn*
: The alignment of section *dd*, where *dd* is a decimal number, is set to *nnnn*. If *nnnn* starts with ‘0x’, it is hexadecimal, else if its starts with ‘0b’, it is binary, else if it starts with ‘0’, it is octal, else it is decimal.

−b*dd:nnnn*
: The base address in the machine of section *dd*, is set to *nnnn*. The previous remarks about *dd* and *nnnn* apply.

−o
: The *name* argument after **−o** is used as the name of the *led* output file, instead of *a.out*.

−r
: Generate relocation information in the output file so that it can be the subject of another *led* run. This flag suppresses the ‘Undefined:’ diagnostic.

−c
: Indicates that relocation information must be produced, but commons must be resolved. This may be useful for machines that need a last relocation step at load time. This flag disables the **−r** flag.

−s
: ‘Strip’ the output. That is, remove the name table and relocation information to save space (but impair the usefulness of the debuggers).

−u
: Take the following argument as a symbol and enter it as undefined in the name table. This is useful for loading wholly from a library, since initially the name table is empty and an unresolved reference is needed to force the loading of the first routine.

−v For each member of a library that is linked, give a message on standard error telling why *led* chose to link it (which unresolved reference it resolves). This option is useful if you have ‘multiply defined’ problems.

Files

/profile/module/ack/lib/back/any/em_led: led binary

a.out: output file

See Also

ack(U).

5.1.3. ACK STD C Compiler

This document specifies the implementation-defined behavior of the STD C front end of the Amsterdam Compiler Kit as required by ANS X3.159-1989. Although the C compiler available on Amoeba is fully derived from the Standard Conforming ACK STD C Compiler, some parts of the Amoeba libraries still require some work in order to reach full Standard C conformance. Details of the Amoeba library conformance are in *ansi_C*(L). A companion document, *posix*(L), provides information about functions not defined by the C Standard but by the POSIX standard, as well as additional information about functions defined (partially) by both standards.

As the Amoeba Standard C library is also used by cross compilers for Amoeba that are not necessarily STD C compatible, the section specifying implementation defined behavior of the Standard C library (marked **ANS A.6.3.14**) deserves attention in any case.

ACK STD C compliance statements

ANS A.6.3.1:

- Diagnostics are placed on the standard error output. They have the following specification:

  ‘‘<file>’’, line <nr>: [(<class>)] <diagnostic>

  There are three classes of diagnostics: *error*, *strict* and *warning*. When the class is *error*, the <class> is absent.
  The class *strict* is used for violations of the standard which are not severe enough to stop compilation. An example is the occurrence of non white-space after an ‘#else’ or ‘#endif’ preprocessing directive. The class *warning* is used for legal but dubious constructions. An example is overflow of constant expressions.

ANS A.6.3.2:

- The function ‘main’ can have two arguments. The first argument is an integer specifying the number of arguments on the command line. The second argument is a pointer to an array of pointers to the arguments (as strings).
- Interactive devices are terminals.

ANS A.6.3.3:

- The number of significant characters is an option. By default it is 64. There is a distinction between upper and lower case.

ANS A.6.3.4:

- The compiler assumes ASCII-characters in both the source and execution character set.
- There are no multi-byte characters.
- There are 8 bits in a character.
- Character constants with values that can not be represented in 8 bits are truncated.
- Character constants that are more than 1 character wide will have the first character specified in the least significant byte.
- The only supported locale is ‘‘C’’.
- A plain ‘char’ has the same range of values as ‘signed char’.

ANS A.6.3.5:

- The compiler assumes that it works on, and compiles for, a 2’s-complement binary-number system. Shorts will use 2 bytes and longs will use 4 bytes. The size of integers is machine dependent.
- Converting an integer to a shorter signed integer is implemented by ignoring the high-order byte(s) of the former. Converting a unsigned integer to a signed integer of the same type is only done in administration. This means that the bit-pattern remains unchanged.
- The result of bitwise operations on signed integers are what can be expected on a 2-complement machine.
- If either operand is negative, whether the result of the / operator is the largest integer less than or equal to the algebraic quotient or the smallest integer greater than or equal to the algebraic quotient is machine dependent, as is the sign of the result of the % operator.
- The right-shift of a negative value is negative.

ANS A.6.3.6:

- The representation of floating-point values is machine-dependent. When native floating-point is not present an IEEE-emulation is used. The compiler uses high-precision floating-point for constant folding.
- Truncation is always to the nearest floating-point number that can be represented.

ANS A.6.3.7:

- The type returned by the sizeof-operator (also known as size_t) is ’unsigned int’. This is done for backward compatibility reasons.
- Casting an integer to a pointer or vice versa has no effect in bit-pattern when the sizes are equal. Otherwise the value will be truncated or zero-extended (depending on the direction of the conversion and the relative sizes).
- When a pointer is as large as an integer, the type of a ’ptrdiff_t’ will be ’int’. Otherwise the type will be ’long’.

ANS A.6.3.8:

- Since the front end has only limited control over the registers, it can only make it more likely that variables that are declared as registers also end up in registers. The only things that can possibly be put into registers are : ‘int’, ‘long’, ‘float’, ‘double’, ‘long double’ and pointers.

ANS A.6.3.9:

- When a member of a union object is accessed using a member of a different type, the resulting value will usually be garbage. The compiler makes no effort to catch these errors.
- The alignment of types is a compile-time option. The alignment of a structure-member is the alignment of its type. Usually, the alignment is passed on to the compiler by the ‘ack’ program. When a user wants to do this manually, he/she should be prepared for trouble.
- A *plain* ‘int’ bit-field is taken as a ‘signed int’. This means that a field with a size 1 bit

can only store the values 0 and −1.

- The order of allocation of bit-fields is a compile-time option. By default, high-order bits are allocated first.
- An enum has the same size as a *plain* ‘int’.

ANS A.6.3.10:

- An access to a volatile declared variable is done by just mentioning the variable. For example, the statement `x;` where x is declared volatile, constitutes an access.

ANS A.6.3.11:

- There is no fixed limit on the number of declarators that may modify an arithmetic, structure or union type, although specifying too many may cause the compiler to run out of memory.

ANS A.6.3.12:

- The maximum number of cases in a switch-statement is in the order of 1e9, although the compiler may run out of memory somewhat earlier.

ANS A.6.3.13:

- Since both the preprocessor and the compiler assume ASCII-characters, a single character constant in a conditional-inclusion directive matches the same value in the execution character set.
- The preprocessor recognizes −I... command-line options. The directories thus specified are searched first. After that, depending on the command that the preprocessor is called with, machine and system-dependent directories are searched. After that, */profile/module/ack/include/posix* and */profile/module/ack/include* are visited.
- Quoted names are first looked for in the directory in which the file which does the include resides.
- The characters in a h- or q- character sequence are taken to be path names.
- Neither the compiler nor the preprocessor know any pragmas.
- Since the compiler runs on Amoeba, __DATE__ and __TIME__ will always be defined. When the time-of-day server does not respond within few seconds, time and date will be set to the local equivalent of Jan 1, 1970.

ANS A.6.3.14:

- NULL is defined as ((void *) 0). This in order to flag dubious constructions like `int x = NULL;`.
- The diagnostic printed by ‘assert’ is as follows:

  ```
  Assertion "<expr>" failed, file "<file>", line <line>
  ```

  where <expr> is the argument to the assert macro, printed as string. (the <file> and <line> should be clear)

- The sets for character test macros.

name:	set:
isalnum()	0-9A-Za-z
isalpha()	A-Za-z
iscntrl()	\000-\037\177
islower()	a-z
isupper()	A-Z
isprint()	<space>- (== \040-\176)

  As an addition, there is an *isascii()* macro, which tests whether a character is an ASCII character. Characters in the range from \000 to \177 are ASCII characters.

- The behavior of mathematical functions on domain error:

name:	returns:
asin()	0.0
acos()	0.0
atan2()	0.0
fmod()	0.0
log()	–HUGE_VAL
log10()	–HUGE_VAL
pow()	0.0
sqrt()	0.0

- Underflow range errors do not cause *errno* to be set.
- The function *fmod()* returns 0.0 and sets *errno* to EDOM when the second argument is 0.0.
- Under native Amoeba, the set of signals for the *signal()* function is listed in the file */profile/module/ack/include/posix/signal.h*. The default handling, semantics and behavior of signals in Amoeba are intended to be (or become) POSIX compatible. The signal SIGILL is handled like all other signals.
- A text-stream need not end in a new-line character.
- White space characters before a new-line appear when read in.
- No null characters will be appended to a binary stream.
- The file position indicator of an append-mode stream is initially positioned at the beginning of the file.
- A write on a text stream does not cause the associated file to be truncated beyond that point.
- The buffering intended by the standard is fully supported.
- A zero-length file actually exists.
- A file name can consist of any character, except for the '\0' and the '/'.
- When a file is opened for writing, all modifications will be made on a private, *uncommitted* (see *bullet*(A)) copy of the original. As soon as the file is closed, it is *committed*, installed in the directory using the name server (see *soap*(A)), after which any old version is destroyed. So while a file is being written to, it should be open only

once. Having a file open for reading multiple times is safe, however.

- When a *remove()* is done on an open file, reading and writing behave just as can be expected from a non-removed file. When the associated stream is closed, all written data will be retained, as a result of the file semantics described above.
- When the destination file exists prior to a call to *rename()*, it is removed first.
- The %p conversion in *fprintf()* has the same effect as %#x or %#lx, depending on the sizes of pointer and integer.
- The %p conversion in *fscanf()* has the same effect as %x or %lx, depending on the sizes of pointer and integer.
- A '–' character that is neither the first nor the last character in the scanlist for %[conversion is taken to be a range indicator. When the first character has a higher ASCII-value than the second, the '–' will just be put into the scanlist.
- The value of *errno* when *fgetpos()* or *ftell()* failed is that of *lseek()*. This means:

EBADF	– when the stream is not valid
ESPIPE	– when fildes is associated with a pipe or another non-file server
EINVAL	– when an invalid seek mode is specified, or when the resulting file pointer would be negative
EIO	– when the bullet file server does not respond.

- The messages generated by *perror()* depend on the value of *errno*. The mapping of errors to strings is done by *strerror()*.
- When the requested size is zero, *malloc()*, *calloc()* and *realloc()* return a null-pointer.
- When *abort()* is called, output buffers will be flushed. Temporary files (made with the *tmpfile()* function) will have disappeared when SIGABRT is not caught or ignored.
- The *exit()* function returns the low-order eight bits of its argument to the environment.
- The predefined environment names are controlled by the user. Setting environment variables is done through the *putenv()* function. This function accepts a pointer to char as its argument. Example: to set the environment variable TERM to a230 one writes

```
static char terminal[] = "TERM=a230"; putenv(terminal);
```

  The argument to *putenv()* is stored in an internal table, strings allocated with *malloc()* can not be freed until another call to *putenv()* (which sets the same environment variable) is made. The argument to *putenv()* must be writable, which means that officially, the argument cannot be a string constant. The function returns 1 if it fails, 0 otherwise.
- The argument to system is passed as argument to */bin/sh -c*.
- The strings returned by *strerror()* depend on *errno* in the following way:

errno	string
0	"Error 0"
EPERM	"Operation not permitted"
ENOENT	"No such file or directory"
ESRCH	"No such process"
EINTR	"Interrupted system call"

EIO	"Input/output error"
ENXIO	"No such device or address"
E2BIG	"Arg list too long"
ENOEXEC	"Exec format error"
EBADF	"Bad file descriptor"
ECHILD	"No child processes"
EAGAIN	"Resource temporarily unavailable"
ENOMEM	"Not enough core"
EACCES	"Permission denied"
EFAULT	"Bad address"
EBUSY	"Resource busy"
EEXIST	"File exists"
ENODEV	"No such device"
ENOTDIR	"Not a directory"
EISDIR	"Is a directory"
EINVAL	"Invalid argument"
ENFILE	"Too many open files in system"
EMFILE	"Too many open files"
ENOTTY	"Inappropriate ioctl operation"
EFBIG	"File too large"
ENOSPC	"No space left on device"
ESPIPE	"Invalid seek"
EROFS	"Read-only file system"
EMLINK	"Too many links"
EPIPE	"Broken pipe"
EDOM	"Domain error"
ERANGE	"Result too large"
EDEADLK	"Operation would block"
ENAMETOOLONG	"File name too long"
ENOTEMPTY	"Directory not empty"
ENOLCK	"No locks available"
ENOSYS	"Function not implemented"

Errors that lie within the range 0..ENOSYS, and are not presented in the table cause *strerror()* to return ‘‘Error <num>’’, where <num> is the error number in decimal. Everything else causes *strerror()* to return ‘‘unknown error’’

- The local time zone is per default MET (GMT + 1:00:00). This can be changed through the TZ environment variable.
- The function *clock()* always returns -1 on Amoeba.

References

[1] ANS X3.159-1989 *American National Standard for Information Systems - Programming Language C*

See Also

posix(L), ctime(L), bullet(A), soap(A).

5.1.4. ACK Pascal Compiler Compliance Statements

Introduction

This document refers to the (1982) BSI standard for Pascal [1]. Ack-Pascal complies with the requirements of level 1 of BS 6192: 1982, with the exceptions as listed in this document.

The standard requires an accompanying document describing the implementation-defined and implementation-dependent features, the reaction on errors and the extensions to standard Pascal. These four items will be treated in the rest of this document, each in a separate section. The other chapters describe the deviations from the standard and the list of options recognized by the compiler.

The Ack-Pascal compiler produces code for an EM machine as defined in [2]. It is up to the implementor of the EM machine to decide whether errors like integer overflow, undefined operand and range bound error are recognized or not.

There does not (yet) exist a hardware EM machine. Therefore, EM programs must be interpreted, or translated into instructions for a target machine. The Ack-Pascal compiler is currently available for use with the VAX, Motorola MC680x0, Intel 80x86, PDP-11, and Intel 8086 code-generators. For the 8086, and MC680x0, floating point emulation is used. This is made available with the **–fp** option, which must be passed to *ack*(U).

Implementation-defined features

For each implementation-defined feature mentioned in the BSI standard we give the section number, the quotation from that section and the definition. First we quote the definition of implementation-defined:

> Possibly differing between processors, but defined for any particular processor.

BS 6.1.7: Each string-character shall denote an implementation-defined value of the required char-type.

> All 7-bits ASCII characters except linefeed LF (10) are allowed.

BS 6.4.2.2: The values of type *real* shall be an implementation-defined subset of the real numbers denoted as specified in 6.1.5 by *signed real*.

> The format of reals is not defined in EM. Even the size of reals depends on the EM-implementation. The compiler can be instructed, by the **V** option, to use a different size for real values. The size of reals is preset by the calling program *ack*(U) to the proper size.

BS 6.4.2.2: The type char shall be the enumeration of a set of implementation-defined characters, some possibly without graphic representations.

> The 7-bits ASCII character set is used, where LF (10) denotes the end-of-line marker on text-files.

BS 6.4.2.2: The ordinal numbers of the character values shall be values of integer-type, that are implementation-defined, and that are determined by mapping the character values on to consecutive non-negative integer values starting at zero.

> The normal ASCII ordering is used: ord('0')=48, ord('A')=65, ord('a')=97, etc.

BS 6.6.5.2: The post-assertions imply corresponding activities on the external entities, if any, to which the file-variables are bound. These activities, and the point at which they are actually performed, shall be implementation-defined.

The reading and writing of objects on files is buffered. This means that when a program terminates abnormally, IO may be unfinished. Terminal IO is unbuffered. Files are closed whenever they are rewritten or reset, or on program termination.

BS 6.7.2.2: The predefined constant maxint shall be of integer-type and shall denote an implementation-defined value, that satisfies the following conditions:

(a) All integral values in the closed interval from −maxint to +maxint shall be values of the integer-type.

(b) Any monadic operation performed on an integer value in this interval shall be correctly performed according to the mathematical rules for integer arithmetic.

(c) Any dyadic integer operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic, provided that the result is also in this interval.

(d) Any relational operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic.

The representation of integers in EM is a *n**8-bit word using two's complement arithmetic. Where *n* is called wordsize. The range of available integers depends on the EM implementation: For 2-byte machines, the integers range from −32767 to +32767. For 4-byte machines, the integers range from −2147483647 to 2147483647. The number −maxint−1 may be used to indicate 'undefined'.

BS 6.7.2.2: The result of the real arithmetic operators and functions shall be approximations to the corresponding mathematical results. The accuracy of this approximation shall be implementation-defined.

Since EM does not specify floating point format, it is not possible to specify the accuracy. When the floating point emulation is used, and the default size of reals is 8 bytes, the accuracy is 11 bits for the exponent, and 53 bits for the mantissa. This gives an accuracy of about 16 digits, and exponents ranging from −309 to +307.

BS 6.9.3.1: The default TotalWidth values for integer, Boolean and real types shall be implementation-defined.

The defaults are:

integer	6 for 2-byte machines, 11 for 4-byte machines
Boolean	5
real	14

BS 6.9.3.4.1: ExpDigits, the number of digits written in an exponent part of a real, shall be implementation-defined.

ExpDigits is defined as 3. This is sufficient for all implementations currently available. When the representation would need more than 3 digits, then the string '***' replaces the exponent.

BS 6.9.3.4.1: The character written as part of the representation of a real to indicate the beginning of the exponent part shall be implementation-defined, either 'E' or 'e'.

The exponent part starts with ’e’.

BS 6.9.3.5: The case of the characters written as representation of the Boolean values shall be implementation-defined.

The representations of true and false are ’true’ and ’false’.

BS 6.9.5: The effect caused by the standard procedure page on a text file shall be implementation-defined.

The ASCII character form feed FF (12) is written.

BS 6.10: The binding of the variables denoted by the program-parameters to entities external to the program shall be implementation-defined if the variable is of a file-type.

The program parameters must be files and all, except input and output, must be declared as such in the program block.

The program parameters input and output, if specified, will correspond with the Amoeba streams ’standard input’ and ’standard output’.

The other program parameters will be mapped to the argument strings provided by the caller of this program. The argument strings are supposed to be path names of the files to be opened or created. The order of the program parameters determines the mapping: the first parameter is mapped onto the first argument string etc. Note that input and output are ignored in this mapping.

The mapping is recalculated each time a program parameter is opened for reading or writing by a call to the standard procedures reset or rewrite. This gives the programmer the opportunity to manipulate the list of string arguments using the external procedures argc, argv and argshift available in *libpc*(L).

BS 6.10: The effect of an explicit use of reset or rewrite on the standard textfiles input or output shall be implementation-defined.

The procedures reset and rewrite are no-ops if applied to input or output.

Implementation-dependent Features

For each implementation-dependent feature mentioned in the BSI standard, we give the section number, the quotation from that section and the way this feature is treated by the Ack-Pascal system. First we quote the definition of ‘‘implementation-dependent’’:

> Possibly differing between processors and not necessarily defined for any particular processor.

BS 6.7.2.1: The order of evaluation of the operands of a dyadic operator shall be implementation-dependent.

All operands are always evaluated, so the program part

```
if (p<>nil) and (p^.value<>0) then
```

is probably incorrect.

The left-hand operand of a dyadic operator is almost always evaluated before the right-hand side. Some peculiar evaluations exist for the following cases:

1. The modulo operation is performed by a library routine to check for negative values of

the right operand.

2. The expression

```
    set1 <= set2
```

where set1 and set2 are compatible set types is evaluated in the following steps:

– evaluate set2
– evaluate set1
– compute set2+set1
– test set2 and set2+set1 for equality

3 The expression

```
    set1 >= set2
```

where set1 and set2 are compatible set types is evaluated in the following steps:

– evaluate set1
– evaluate set2
– compute set1+set2
– test set1 and set1+set2 for equality

BS 6.7.3: The order of evaluation, accessing and binding of the actual-parameters for functions shall be implementation-dependent.

The order of evaluation is from right to left.

BS 6.8.2.2: The decision as to the order of accessing the variable and evaluating the expression in an assignment-statement, shall be implementation-dependent.

The expression is evaluated first.

BS 6.8.2.3: The order of evaluation and binding of the actual-parameters for procedures shall be implementation-dependent.

The same as for functions.

BS 6.9.5: The effect of inspecting a text file to which the page procedure was applied during generation is implementation-dependent.

The formfeed character written by page is treated like a normal character, with ordinal value 12.

BS 6.10: The binding of the variables denoted by the program-parameters to entities external to the program shall be implementation-dependent unless the variable is of a file-type.

Only variables of a file-type are allowed as program parameters.

Error handling

There are three classes of errors to be distinguished. In the first class are the error messages generated by the compiler. The second class consists of the occasional errors generated by the other programs involved in the compilation process. Errors of the third class are the errors as defined in the standard by:

> An error is a violation by a program of the requirements of this standard
> that a processor is permitted to leave undetected.

Compiler Errors

Errors are written on the standard error output. Each line has the form:

```
<file>, line <number>: <description>
```

Every time the compiler detects an error that does not have influence on the code produced by the compiler or on the syntax decisions, a warning messages is given. If only warnings are generated, compilation proceeds and probably results in a correctly compiled program.

Sometimes the compiler produces several errors for the same line. They are only shown up to a maximum of 5 errors per line. Warning are also shown up to a maximum of 5 per line.

Extensive treatment of these errors is outside the scope of this manual.

Run-time Errors

Errors detected at run time cause an error message to be generated on the diagnostic output stream (Amoeba file descriptor 2). The message consists of the name of the program followed by a message describing the error, possibly followed by the source line number. Unless the **–L** option is turned on, the compiler generates code to keep track of which source line causes which EM instructions to be generated. It depends on the EM implementation whether these LIN instructions are skipped or executed.

For each error mentioned in the standard we give the section number, the quotation from that section and the way it is processed by the Pascal compiler or run-time system.

For detected errors the corresponding message and trap number are given. Trap numbers are useful for exception-handling routines. Normally, each error causes the program to terminate. By using exception-handling routines one can ignore errors or perform alternate actions. Only some of the errors can be ignored by restarting the failing instruction. These errors are marked as non-fatal, all others as fatal. A list of errors with trap number between 0 and 63 (EM errors) can be found in [2]. Errors with trap number between 64 and 127 (Pascal errors) have the following meaning:

64:	more args expected
65:	error in exp
66:	error in ln
67:	error in sqrt
68:	assertion failed
69:	array bound error in pack
70:	array bound error in unpack
71:	only positive j in *i mod j*
72:	file not yet open
73:	dispose error
74:	function not assigned
75:	illegal field width

96:	file xxx: not writable
97:	file xxx: not readable
98:	file xxx: end of file
99:	file xxx: truncated
100:	file xxx: reset error
101:	file xxx: rewrite error
102:	file xxx: close error
103:	file xxx: read error
104:	file xxx: write error
105:	file xxx: digit expected
106:	file xxx: non-ASCII char read

BS 6.4.6: It shall be an error if a value of type T2 must be assignment-compatible with type T1, while T1 and T2 are compatible ordinal-types and the value of type T2 is not in the closed interval specified by T1.

The compiler distinguishes between array-index expressions and the other places where assignment-compatibility is required.

Array subscripting is done using the EM array instructions. These instructions have three arguments: the array base address, the index and the address of the array descriptor. An array descriptor describes one dimension by three values: the lower bound on the index, the number of elements minus one and the element-size. It depends on the EM implementation whether these bounds are checked. Since most implementations do not, an extra compiler flag is added to force these checks.

The other places where assignment-compatibility is required are:

– assignment
– value parameters
– procedures read and readln
– the final value of the for-statement

For these places the compiler generates an EM range check instruction, except when the **R** option is turned on, or when the range of values of T2 is enclosed in the range of T1. If the expression consists of a single variable and if that variable is of a subrange type, then the subrange type itself is taken as T2, not its host-type. Therefore, a range instruction is only generated if T1 is a subrange type and if the expression is a constant, an expression with two or more operands, or a single variable with a type not enclosed in T1. If a constant is assigned, then the EM optimizer removes the range check instruction, except when the value is out of bounds.

It depends on the EM implementation whether the range check instruction is executed or skipped.

BS 6.4.6: It shall be an error if a value of type T2 must be assignment-compatible with type T1, while T1 and T2 are compatible set-types and any member of the value of type T2 is not in the closed interval specified by the base-type of the type T1.

This error is not detected.

BS 6.5.3.3: It shall be an error if a component of a variant-part of a variant, where the selector of the variant-part is not a field, is accessed unless the variant is active for the entirety of each reference and access to each component of the variant.

This error is not detected.

BS 6.5.4: It shall be an error if the pointer-variable of an identified-variable either denotes a nil-value or is undefined.

The EM definition does not specify the binary representation of pointer values, so it is not possible to choose an otherwise illegal binary representation for the pointer value NIL. Rather arbitrarily the compiler uses the integer value zero to represent NIL. For all current implementations this does not cause problems.

The size of pointers depends on the implementation and is preset in the compiler by *ack*(U). The compiler can be instructed, by the V option, to use another size for pointer objects. NIL is represented here by the appropriate number of zero words.

It depends on the EM implementation whether dereferencing of a pointer with value NIL causes an error.

BS 6.5.4: It shall be an error to remove the identifying-value of an identified variable from its pointer-type when a reference to the variable exists.

When the identified variable is an element of the record-variable-list of a with_statement, a warning is given at compile-time. Otherwise, this error is not detected.

BS 6.5.5: It shall be an error to alter the value of a file-variable f when a reference to the buffer-variable f^ exists.

When f is altered when it is an element of the record-variable-list of a with-statement, a warning is given. When a buffer-variable is used as a variable-parameter, an error is given. This is done at compile-time.

BS 6.6.5.2: It shall be an error if the stated pre-assertion does not hold immediately prior to any use of the file handling procedures rewrite, put, reset and get.

For each of these four operations the pre-assertions can be reformulated as:

rewrite(f):	no pre-assertion.
put(f):	f is opened for writing and f^ is not undefined.
reset(f):	f exists.
get(f):	f is opened for reading and eof(f) is false.

The following errors are detected for these operations:

rewrite(f):

more args expected, trap 64, fatal:

f is a program-parameter and the corresponding file name is not supplied by the caller of the program.

rewrite error, trap 101, fatal:

the caller of the program lacks the necessary access rights to create the file in the file system or operating system problems like table overflow prevent creation of the file.

put(f):

file not yet open, trap 72, fatal:

reset or rewrite are never applied to the file. The checks performed by the run time system are not foolproof.

not writable, trap 96, fatal:

f is opened for reading.

write error, trap 104, fatal:

probably caused by file system problems. For instance, the file storage is exhausted. Because IO is buffered to improve performance, it might happen that this error occurs if the file is closed. Files are closed whenever they are rewritten or reset, or on program termination.

reset(f):

more args expected, trap 64, fatal:

same as for rewrite(f).

reset error, trap 100, fatal:

f does not exist, or the caller has insufficient access rights, or operating system tables are exhausted.

get(f):

file not yet open, trap 72, fatal:

as for put(f).

not readable, trap 97, fatal:

f is opened for writing.

end of file, trap 98, fatal:

eof(f) is true just before the call to get(f).

read error, trap 103, fatal:

unlikely to happen. Probably caused by hardware problems or by errors elsewhere in your program that destroyed the file information maintained by the run time system.

truncated, trap 99, fatal:

the file is not properly formed by an integer number of file elements. For instance, the size of a file of integer is odd.

non-ASCII char read, trap 106, non-fatal:

the character value of the next character-type file element is out of range (0..127). Only for text files.

BS 6.6.5.3: It shall be an error if a variant of a variant-part within the new variable becomes active and a different variant of the variant-part is one of the specified variants.

This error is not detected.

BS 6.6.5.3: It shall be an error to use dispose(q) if the identifying variable has been allocated using the form new(p,c1,...,cn).

This error is not detected. However, this error can cause more memory to be freed then was allocated. Dispose causes a fatal trap 73 when memory already on the free list is freed again.

BS 6.6.5.3: It shall be an error to use dispose(q,k1,...,km) if the identifying variable has been allocated using the form new(p,c1,...,cn) and m is not equal to n.

This error is not detected. However, this error can cause more memory to be freed then was allocated. Dispose causes a fatal trap 73 when memory already on the free list is freed again.

BS 6.6.5.3: It shall be an error if the variants of a variable to be disposed are different from those specified by the case-constants to dispose.

This error is not detected.

BS 6.6.5.3: It shall be an error if the value of the pointer parameter of dispose has nil-value or is undefined.

The same comments apply as for dereferencing NIL or undefined pointers.

BS 6.6.5.3: It shall be an error if a variable created using the second form of new is accessed by the identified variable of the variable-access of a factor, of an assignment-statement, or of an actual-parameter.

This error is not detected.

BS 6.6.6.2: It shall be an error if the value of sqr(x) does not exist.

This error is detected for real-type arguments (real overflow, trap 4, non-fatal).

BS 6.6.6.2: It shall be an error if x in ln(x) is smaller than or equal to 0.

This error is detected (error in ln, trap 66, non-fatal)

BS 6.6.6.2: It shall be an error if x in sqrt(x) is smaller than 0.

This error is detected (error in sqrt, trap 67, non-fatal)

In addition to these errors, overflow in the expression exp(x) is detected (error in exp, trap 65, non-fatal; real overflow, trap 4, non-fatal)

BS 6.6.6.3: It shall be an error if the integer value of trunc(x) does not exist.

It depends on the implementations whether this error is detected. The floating-point emulation detects this error (conversion error, trap 10, non-fatal).

BS 6.6.6.3: It shall be an error if the integer value of round(x) does not exist.

It depends on the implementations whether this error is detected. The floating-point emulation detects this error (conversion error, trap 10, non-fatal).

BS 6.6.6.4: It shall be an error if the integer value of ord(x) does not exist.

This error can not occur, because the compiler will not allow such ordinal types.

BS 6.6.6.4: It shall be an error if the character value of chr(x) does not exist.

Except when the *R* option is off, the compiler generates an EM range check instruction. The effect of this instruction depends on the EM implementation.

BS 6.6.6.4: It shall be an error if the value of succ(x) does not exist.

Same comments as for chr(x).

BS 6.6.6.4: It shall be an error if the value of pred(x) does not exist.

Same comments as for chr(x).

BS 6.6.6.5: It shall be an error if f in eof(f) is undefined.

This error is detected (file not yet open, trap 72, fatal).

BS 6.6.6.5: It shall be an error if f in eoln(f) is undefined, or if eof(f) is true at that time.

The following errors may occur:

file not yet open, trap 72, fatal;
not readable, trap 97, fatal;
end of file, trap 98, fatal.

BS 6.7.1: It shall be an error if a variable-access used as an operand in an expression is undefined at the time of its use.

The compiler performs some limited checks to see if identifiers are used before they are set. Since it can not always be sure (one could, for instance, jump out of a loop), only a warning is generated. When an expression contains a function-call, an error occur if the function is not assigned at run-time.

BS 6.7.2.2: A term of the form x/y shall be an error if y is zero.

It depends on the EM implementation whether this error is detected. On some machines, a trap may occur.

BS 6.7.2.2: It shall be an error if j is zero in ’i div j’.

It depends on the EM implementation whether this error is detected. On some machines, a trap may occur.

BS 6.7.2.2: It shall be an error if j is zero or negative in i MOD j.

This error is detected (only positive j in ’i mod j’, trap 71, non-fatal).

BS 6.7.2.2: It shall be an error if the result of any operation on integer operands is not performed according to the mathematical rules for integer arithmetic.

The reaction depends on the EM implementation. Most implementations, however, will not notice integer overflow.

BS 6.8.3.5: It shall be an error if none of the case-constants is equal to the value of the case-index upon entry to the case-statement.

This error is detected (case error, trap 20, fatal).

BS 6.9.1: It shall be an error if the sequence of characters read looking for an integer does not form a signed-integer as specified in 6.1.5.

This error is detected (digit expected, trap 105, non-fatal).

BS 6.9.1: It shall be an error if the sequence of characters read looking for a real does not form a signed-number as specified in 6.1.5.

This error is detected (digit expected, trap 105, non-fatal).

BS 6.9.1: When read is applied to f, it shall be an error if the buffer-variable f^ is undefined or the pre-assertions for get do not hold.

This error is detected (see get(f)).

BS 6.9.3: When write is applied to a textfile f, it shall be an error if f is undefined or f is opened for reading.

This error is detected (see put(f)). Furthermore, this error is also detected when f is not a textfile.

BS 6.9.3.1: The values of TotalWidth or FracDigits shall be greater than or equal to one; it shall be an error if either value is less then one.

When either value is less than zero, an error (illegal field width, trap 75, non-fatal) occurs. Zero values are allowed, in order to maintain some compatibility with the old Ack-Pascal compiler.

BS 6.9.5: It shall be an error if the pre-assertion required for writeln(f) do not hold prior to the invocation of page(f);

This error is detected (see put(f)).

Extensions to the Standard

External Routines

Except for the required directive *forward* the Ack-Pascal compiler recognizes the directive *extern*. This directive tells the compiler that the procedure block of this procedure will not be present in the current program. The code for the body of this procedure must be included at a later stage of the compilation process.

This feature allows one to build libraries containing often used routines. These routines do not have to be included in all the programs using them. Maintenance is much simpler if there is only one library module to be changed instead of many Pascal programs.

Another advantage is that these library modules may be written in a different language, for instance C or the EM assembly language. This is useful if you want to use some specific EM instructions not generated by the Pascal compiler. Examples are the system call routines and some floating point conversion routines. Another motive could be the optimization of some time-critical program parts.

The use of external routines, however, is dangerous. The compiler normally checks for the correct number and type of parameters when a procedure is called and for the result type of functions. If an external routine is called these checks are not sufficient, because the compiler can not check whether the procedure heading of the external routine as given in the Pascal program matches the actual routine implementation. It should be the loader's task to check this. However, the current loaders are not that smart. Another solution is to check at run time, at least the number of words for parameters. Some EM implementations check this.

For those who wish the use the interface between C and Pascal we give an incomplete list of corresponding formal parameters in C and Pascal.

Pascal	C
a:integer	int a
a:char	int a
a:boolean	int a
a:real	double a
a:^*type*	*type* *a
var a:*type*	*type* *a
procedure a(pars)	struct {
	void (*a)() ;
	char *static_link ;
	}
function a(pars):*type*	struct {

```
        type (*a)() ;
        char *static_link ;
    }
```

The Pascal run-time system uses the following algorithm when calling function/procedures passed as parameters.

```
if ( static_link )    (*a)(static_link,pars) ;
else                  (*a)(pars) ;
```

Separate Compilation

The compiler is able to (separately) compile a collection of declarations, procedures and functions to form a library. The library may be linked with the main program, compiled later. The syntax of these modules is

```
module = [constant-definition-part]
         [type-definition-part]
         [var-declaration-part]
         [procedure-and-function-declaration-part]
```

The compiler accepts a program or a module:

```
unit = program | module
```

All variables declared outside a module must be imported by parameters, even the files input and output. Access to a variable declared in a module is only possible using the procedures and functions declared in that same module. By giving the correct procedure/function heading followed by the directive ‘extern’ you may use procedures and functions declared in other units.

Assertions

When the **s** option is off, Ack-Pascal compiler recognizes an additional statement, the assertion. Assertions can be used as an aid in debugging and documentation. The syntax is:

```
assertion = ’assert’ Boolean-expression
```

An assertion is a simple-statement, so

```
simple-statement = [assignment-statement |
                    procedure-statement |
                    goto-statement |
                    assertion
                   ]
```

An assertion causes an error if the Boolean-expression is false. That is its only purpose. It does not change any of the variables, at least it should not. Therefore, do not use functions with side-effects in the Boolean-expression. If the **a** option is turned on, then assertions are skipped by the compiler. However, assignment to a variable and calling of a procedure with that name will be impossible. If the **s** option is turned on, the compiler will not know a thing about assertions, so using assertions will then give a parse error.

Additional Procedures.

Three additional standard procedures are available:

halt:

> a call of this procedure is equivalent to jumping to the end of your program. It is always the last statement executed. The exit status of the program may be supplied as optional argument. If not, it will be zero.

release:
mark:

> for most applications it is sufficient to use the heap as second stack. Mark and release are suited for this type of use, more suited than dispose. mark(p), with p of type pointer, stores the current value of the heap pointer in p. release(p), with p initialized by a call of mark(p), restores the heap pointer to its old value. All the heap objects, created by calls of new between the call of mark and the call of release, are removed and the space they used can be reallocated. Never use mark and release together with dispose!

Amoeba Interfacing.

If the **c** option is turned on, then some special features are available to simplify an interface with the Amoeba environment. First of all, the compiler allows you to use a different type of string constants. These string constants are delimited by double quotes (‘"’). To put a double quote into these strings, you must repeat the double quote, like the single quote in normal string constants. These special string constants are terminated by a zero byte (chr(0)). The type of these constants is a pointer to a packed array of characters, with lower bound 1 and unknown upper bound.

Secondly, the compiler predefines a new type identifier ’string’ denoting this just described string type.

The only thing you can do with these features is declaration of constants and variables of type ’string’. String objects may not be allocated on the heap and string pointers may not be dereferenced. Still these strings are very useful in combination with external routines. The procedure write is extended to print these zero-terminated strings correctly.

Double Length (32-bit) Integers.

If the **d** option is turned on, then the additional type ’long’ is known to the compiler. By default, long variables have integer values in the range −2147483647..+2147483647, but this can be changed with the −V option (if the back end can support this). Long constants can not be declared. Longs can not be used as control-variables. It is not allowed to form subranges of type long. All operations allowed on integers are also allowed on longs and are indicated by the same operators: ’+’, ’−’, ’*’, ’/’, ’div’, ’mod’. The procedures read and write have been extended to handle long arguments correctly. It is possible to read longs from a file of integers and vice-versa, but only if longs and integers have the same size. The default width for longs is 11. The standard procedures ’abs’ and ’sqr’ have been extended to work on long arguments. Conversion from integer to long, long to real, real to long and long to integer are automatic, like the conversion from integer to real. These conversions may cause a

conversion error, trap 10, non-fatal

Underscore As Letter

The underscore character ’_’ may be used in forming identifiers, if the u or U option is turned on. It is forbidden to start identifiers with underscores, since this may cause name-clashes with run-time routines.

Zero Field Width In Write.

Zero TotalWidth arguments are allowed. No characters are written for character, string or Boolean type arguments then. A zero FracDigits argument for fixed-point representation of reals causes the fraction and the character ’.’ to be suppressed.

Pre-processing

If the very first character of a file containing a Pascal program is the sharp (’#’, ASCII 23(hex)) the file is preprocessed in the same way as C programs. Lines beginning with a ’#’ are taken as preprocessor command lines and not fed to the Pascal compiler proper. C style comments, /*......*/, are removed by the C preprocessor, thus C comments inside Pascal programs are also removed when they are fed through the preprocessor.

Deviations from the standard

Ack-Pascal deviates from the standard proposal in the following ways:

1. Standard procedures and functions are not allowed as parameters in Ack-Pascal. You can obtain the same result with negligible loss of performance by declaring some user routines like:

```
function sine(x:real):real;
begin
        sine:=sin(x)
end;
```

2. The standard procedures read, readln, write and writeln are implemented as word-symbols, and can therefore not be redeclared.

Compiler Options

Some options of the compiler may be controlled by using ‘‘{$....}’’. Each option consists of a lower case letter followed by +, – or an unsigned number. Options are separated by commas. The following options exist:

a +/–   this option switches assertions on and off. If this option is on, then code is included to test these assertions at run time. Default +.

c +/– this option, if on, allows you to use C-type string constants surrounded by double quotes. Moreover, a new type identifier 'string' is predefined. Default –.

d +/– this option, if on, allows you to use variables of type 'long'. Default –.

i <num> with this flag the setsize for a set of integers can be manipulated. The number must be the number of bits per set. The default value is wordsize–1.

l +/– if + then code is inserted to keep track of the source line number. When this flag is switched on and off, an incorrect line number may appear if the error occurs in a part of your program for which this flag is off. These same line numbers are used for the profile, flow and count options of the EM interpreter *em*. Default +.

r +/– if + then code is inserted to check subrange variables against lower and upper subrange limits. Default +.

s +/– if + then the compiler will hunt for places in your program where non-standard features are used, and for each place found it will generate a warning. Default –.

t +/– if + then each time a procedure is entered, the routine *procentry* is called, and each time a procedure exits, the procedure *procexit* is called. Both *procentry* and *procexit* have a *string* as parameter. This means that when a user specifies his or her own procedures, the **c** option must be used. Default procedures are present in the run-time library. Default –.

u +/– if + then the underscore character ''_'' is treated like a letter, so that it may be used in identifiers. Procedure and function identifiers are not allowed to start with an underscore because they may collide with library routine names. Default –.

Some of these flags (**c, d, i, s, u, C** and **U**) The others may be switched on and off.

A very powerful debugging tool is the knowledge that inaccessible statements and useless tests are removed by the EM optimizer. For instance, a statement like:

```
if debug then
        writeln('initialization done');
```

is completely removed by the optimizer if debug is a constant with value false. The first line is removed if debug is a constant with value true. Of course, if debug is a variable nothing can be removed.

A disadvantage of Pascal, the lack of preinitialized data, can be diminished by making use of the possibilities of the EM optimizer. For instance, initializing an array of reserved words is sometimes optimized into 3 EM instructions. To maximize this effect you must initialize variables as much as possible in order of declaration and array entries in order of decreasing index.

References

[1] BSI standard BS 6192: 1982 (ISO 7185).

[2] A.S.Tanenbaum, J.W.Stevenson, Hans van Staveren, E.G.Keizer, *Description of a machine architecture for use with block structured languages*, Informatica rapport IR-81.

5.1.5. ACK Modula-2 Compiler Compliance Statements

Introduction

This document describes the implementation-specific features of the ACK Modula-2 compiler. It is not intended to teach Modula-2 programming. For a description of the Modula-2 language, the reader is referred to [1].

The ACK Modula-2 compiler is currently available for use with the VAX, Motorola MC680x0 and Intel 80x86 code-generators. For the 80x86, and MC680x0, floating point emulation is used. This is made available with the **–fp** option, which must be passed to *ack*.

The Language Implementation

This section discusses the deviations from the Modula-2 language as described in the *Report on The Programming Language Modula-2*, as it appeared in [1], from now on referred to as *the Report*. Also, the Report sometimes leaves room for interpretation. The section numbers mentioned are the section numbers of the Report.

Syntax (section 2)

The syntax recognized is that of the Report, with some extensions to also recognize the syntax of an earlier definition, given in [2]. Only one compilation unit per file is accepted.

Vocabulary and Representation (section 3)

The input ‘‘10..’’ is parsed as two tokens: ‘‘10’’ and ‘‘..’’.

The empty string ‘‘’’ has type

```
ARRAY [0 .. 0] OF CHAR
```

and contains one character: 0.

When the text of a comment starts with a ’$’, it may be a pragma. Currently, the following pragmas exist:

```
(*$F      (F stands for Foreign) *)
(*$R[+|-] (Run-time checks, on or off, default on) *)
(*$A[+|-] (Array bound checks, on or off, default off) *)
(*$U      (Allow for underscores within identifiers) *)
```

The Foreign pragma is only meaningful in a `DEFINITION MODULE`, and indicates that this `DEFINITION MODULE` describes an interface to a module written in another language (for instance C, Pascal, or `EM`). Run-time checks that can be disabled are: range checks, `CARDINAL` overflow checks, checks when assigning a `CARDINAL` to an `INTEGER` and vice versa, and checks that `FOR`-loop control-variables are not changed in the body of the loop. Array bound checks can be enabled, because many `EM` implementations do not implement the array bound checking of the `EM` array instructions. When enabled, the compiler generates a check before generating an `EM` array instruction. Even when underscores are enabled, they still may not start an identifier.

Constants of type `LONGINT` are integers with a suffix letter `D` (for instance `1987D`). Constants of type `LONGREAL` have suffix `D` if a scale factor is missing, or have `D` in place of `E` in the scale factor (for example, `1.0D`, `0.314D1`). This addition was made, because there was no way to indicate long constants, and also because the addition was made in Wirth's newest Modula-2 compiler.

Declarations and Scope Rules (section 4)

Standard identifiers are considered to be predeclared, and valid in all parts of a program. They are called *pervasive*. Unfortunately, the Report does not state how this pervasiveness is accomplished. However, page 87 of [1] states: ‘‘Standard identifiers are automatically imported into all modules’’. Our implementation therefore allows redeclarations of standard identifiers within procedures, but not within modules.

Constant Expressions (section 5)

Each operand of a constant expression must be a constant: a string, a number, a set, an enumeration literal, a qualifier denoting a constant expression, a type transfer with a constant argument, or one of the standard procedures `ABS`, `CAP`, `CHR`, `LONG`, `MAX`, `MIN`, `ODD`, `ORD`, `SIZE`, `SHORT`, `TSIZE`, or `VAL`, with constant argument(s); `TSIZE` and `SIZE` may also have a variable as argument.

Floating point expressions are never evaluated compile time, because the compiler basically functions as a cross-compiler, and thus cannot use the floating point instructions of the machine on which it runs. Also, `MAX(REAL)` and `MIN(REAL)` are not allowed.

Type Declarations (section 6)

Basic Types (section 6.1)

The type `CHAR` includes the ASCII character set as a subset. Values range from `0C` to `377C`, not from `0C` to `177C`.

Enumerations (section 6.2)

The maximum number of enumeration literals in any one enumeration type is `MAX(INTEGER)`.

Record Types (section 6.5)

The syntax of variant sections in [1] is different from the one in [2]. Our implementation recognizes both, giving a warning for the older one. However, see section 3.

Set Types (section 6.6)

The only limitation imposed by the compiler is that the base type of the set must be a subrange type, an enumeration type, `CHAR`, or `BOOLEAN`. So, the lower bound may be negative. However, if a negative lower bound is used, the compiler gives a warning of the *restricted* class (see the manual page of the compiler).

The standard type `BITSET` is defined as

```
TYPE BITSET = SET OF [0 .. 8*SIZE(INTEGER)-1];
```

Expressions (section 8)

Operators (section 8.2)

Arithmetic Operators (section 8.2.1)

The Report does not specify the priority of the unary operators `+` or `-`: It does not specify whether

```
- 1 + 1
```

means

```
- (1 + 1)
```

or

```
(-1) + 1
```

Some compilers implement the first alternative, and others that implement the second. Our compiler implements the second, which is suggested by the fact that their priority is not specified, which might indicate that it is the same as that of their binary counterparts. And then the rule about left to right decides for the second. On the other hand, one might argue that, since the grammar only allows for one unary operator in a simple expression, it must apply to the whole simple expression, not just the first term.

Statements (section 9)

Assignments (section 9.1)

The Report does not define the evaluation order in an assignment. Our compiler certainly chooses an evaluation order, but it is explicitly left undefined. Therefore, programs that depend on it, may cease to work later.

The types `INTEGER` and `CARDINAL` are assignment-compatible with `LONGINT`, and `REAL` is assignment-compatible with `LONGREAL`.

Case Statements (section 9.5)

The size of the type of the case-expression must be less than or equal to the word-size.

The Report does not specify what happens if the value of the case-expression does not occur as a label of any case, and there is no `ELSE`-part. In our implementation, this results in a run-time error.

For Statements (section 9.8)

The Report does not specify the legal types for a control variable. Our implementation allows the basic types (except `REAL`), enumeration types, and subranges. A run-time warning is generated when the value of the control variable is changed by the statement sequence that forms the body of the loop, unless run-time checking is disabled.

Return and Exit Statements (section 9.11)

The Report does not specify which result-types are legal. Our implementation allows any result type.

Procedure Declarations (section 10)

Function procedures must exit through a `RETURN` statement, or a run-time error occurs.

Standard Procedures (section 10.2)

Our implementation supports `NEW` and `DISPOSE` for backwards compatibility, but issues warnings for their use. However, see section 3.

Also, some new standard procedures were added, similar to the new standard procedures in Wirth’s newest compiler:

– `LONG` converts an argument of type `INTEGER` or `REAL` to the types `LONGINT` or `LONGREAL`.

– `SHORT` performs the inverse transformation, without range checks.

– `FLOATD` is analogous to `FLOAT`, but yields a result of type `LONGREAL`.

– `TRUNCD` is analogous to `TRUNC`, but yields a result of type `LONGINT`.

System-dependent Facilities (section 12)

The type `BYTE` is added to the `SYSTEM` module. It occupies a storage unit of 8 bits. `ARRAY OF BYTE` has a similar effect to `ARRAY OF WORD`, but is safer. In some obscure cases the `ARRAY OF WORD` mechanism does not quite work properly.

The procedure `IOTRANSFER` is not implemented.

Backwards Compatibility

Besides recognizing the language as described in [1], the compiler recognizes most of the language described in [2], for backwards compatibility. It warns the user for old-fashioned constructions (constructions that [1] does not allow). If the **–Rm2–3** option is passed to *ack*, this backwards compatibility feature is disabled. Also, it may not be present on some smaller machines, like the PDP-11.

Compile-Time Errors

The compile-time error messages are intended to be self-explanatory, and not listed here. The compiler also sometimes issues warnings, recognizable by a warning-classification between parentheses. Currently, there are 3 classifications:

(old-fashioned use)
: These warnings are given on constructions that are not allowed by [1], but are allowed by [2].

(strict)
: These warnings are given on constructions that are supported by the ACK Modula-2 compiler, but might not be supported by others. Examples: functions returning structured types, SET types of subranges with negative lower bound.

(warning)
: The other warnings, such as warnings about variables that are never assigned, never used, etc.

Run-time Errors

The ACK Modula-2 compiler produces code for an `EM` machine. Therefore, it depends on the implementation of the `EM` machine for detection some of the run-time errors that could occur.

The *Traps* module enables the user to install his own run-time error handler. The default one just displays what happened and exits. Basically, a trap handler is just a procedure that takes an INTEGER as parameter. The INTEGER is the trap number. This INTEGER can be one of the `EM` trap numbers, or one of the numbers listed in the *Traps* definition module.

The following run-time errors may occur:

array bound error
: The detection of this error depends on the `EM` implementation.

range bound error
: Range bound errors are always detected, unless run-time checks are disabled.

set bound error
: The detection of this error depends on the `EM` implementation. The current implementations detect this error.

integer overflow
: The detection of this error depends on the `EM` implementation.

cardinal overflow

This error is detected, unless run-time checks are disabled.

cardinal underflow
: This error is detected, unless run-time checks are disabled.

real overflow
: The detection of this error depends on the `EM` implementation.

real underflow
: The detection of this error depends on the `EM` implementation.

divide by 0
: The detection of this error depends on the `EM` implementation.

divide by 0.0
: The detection of this error depends on the `EM` implementation.

undefined integer
: The detection of this error depends on the `EM` implementation.

undefined real
: The detection of this error depends on the `EM` implementation.

conversion error
: This error occurs when assigning a negative value of type INTEGER to a variable of type CARDINAL, or when assigning a value of CARDINAL, that is > MAX(INTEGER), to a variable of type INTEGER. It is detected, unless run-time checking is disabled.

stack overflow
: The detection of this error depends on the `EM` implementation.

heap overflow
: The detection of this error depends on the `EM` implementation. Might happen when ALLOCATE fails.

case error
: This error occurs when non of the cases in a CASE statement are selected, and the CASE statement has no ELSE part. The detection of this error depends on the `EM` implementation. All current `EM` implementations detect this error.

stack size of process too large
: This is most likely to happen if the reserved space for a coroutine stack is too small. In this case, increase the size of the area given to `NEWPROCESS`. It can also happen if the stack needed for the main process is too large and there are coroutines. In this case, the only fix is to reduce the stack size needed by the main process, for example, by avoiding local arrays.

too many nested traps + handlers
: This error can only occur when the user has installed his own trap handler. It means that during execution of the trap handler another trap has occurred, and that several times. In some cases, this is an error because of overflow of some internal tables.

no RETURN from function procedure
: This error occurs when a function procedure does not return properly (‘‘falls through’’).

illegal instruction
: This error might occur when you use floating point operations on an implementation

that does not have floating point.

In addition, some of the library modules may give error messages. The *Traps*-module has a suitable mechanism for this.

Invoking the Compiler

Details of how to invoke the compiler can be found in *ack*(U) and *m2*(U).

This compiler itself has no version checking mechanism. A special linker would be needed to do that.

The Procedure Call Interface

Parameters are pushed on the stack in reverse order, so that the `EM` AB (argument base) register indicates the first parameter. For VAR parameters, its address is passed, for value parameters its value. The only exception to this rule is with conformant arrays. For conformant arrays, the address is passed, and an array descriptor is passed. The descriptor is an `EM` array descriptor. It consists of three fields: the lower bound (always 0), upper bound - lower bound, and the size of the elements. The descriptor is pushed first. If the parameter is a value parameter, the called routine must make sure that its value is never changed, for instance by making its own copy of the array. The Modula-2 compiler does exactly this.

When the size of the return value of a function procedure is larger than the maximum of `SIZE(LONGREAL)` and twice the pointer-size, the caller reserves this space on the stack, above the parameters. The callee then stores its result there, and returns no other value.

References

[1] Niklaus Wirth, *Programming in Modula-2, third, corrected edition*, Springer-Verlag, Berlin (1985)

[2] Niklaus Wirth, *Programming in Modula-2,* Stringer-Verlag, Berlin (1983)

6 Programming Tools

This chapter contains the reference manuals for the programming tools provided with Amoeba. Each section describes one tool. There are several important tools described. The first is the Amoeba Interface Language (AIL). This is a stub compiler for generating servers and their client stubs. The second is *amake* which is a parallel make program which given a list of sources, the name and the type of the target and a set of generation tools will create the desired target. The third is *bawk* which is an implementation of the AWK language.

6.1. The Amoeba Interface Language (AIL) Reference Manual

6.1.1. Overview

AIL is a tool to simplify and standardize the generation of RPC interfaces for Amoeba. Widespread use of AIL in Amoeba will more or less enforce certain conventions, such as the interpretation of header fields and the private parts of a capability, that are not enforced by the Amoeba kernel.

AIL is a stub compiler that generates the glue between an Amoeba server and a client program from an *interface* specification. The outputs it can currently generate are header files that client programs should include if they use the corresponding interface, client stubs that should be linked with the client program, a server mainloop that decodes the Amoeba message and calls the corresponding implementation routine for the requested remote procedure call and marshaling routines. All output is currently written in C. It should not be too difficult to modify AIL to generate output in any procedural language. AIL-generated stubs pass minimal information about the architecture of the client, such as the byte order. No information about user defined types is passed.

One could also regard AIL as a language in which one can express types and calling conventions (i.e., the syntax) of a procedure in a language independent way. The semantics of a procedure are not expressed in AIL. Procedures are grouped in classes, which in turn can inherit the procedures of other classes.

After defining procedures, types and their grouping one can generate translations into various languages. The mechanisms to generate these translations are not described here. The fact that AIL is a stub compiler should, according to this view, not be mentioned in this document, since it results from the fact that it contains translators for this purpose. This view was not adopted in this document, since it forces the user to read between the lines too much. Notwithstanding, the exact documentation of AIL's output is in a separate document: the *AIL Cookbook*.

Types

Currently, AIL supports all of the C-types, in that it can parse and comprehend them, but for some types AIL cannot generate marshaling routines automatically. They are: void, functions, pointers, enums and unions. If you insist on passing these, you can tell AIL to use your own, handwritten marshaling code. The type array has been extended, but a simple C-style array has the same meaning as in C. Most types have marshaling routines and size expressions associated with them. If they do not, AIL is incapable of generating code to pass it to a remote procedure. There is an important difference between the type systems of C and AIL. A typedef yields the same type as the type it is built from, except for the marshaling and size information. Thus, to marshal integers differently, make a typedef ‘‘myint’’, and specify the new marshaling and size information.

Procedures

In this document the word *procedure* is used to mean the call convention of an operation. A procedure syntactically resembles a STD C function prototype.

Usually it has a magic first argument represented as ‘‘*’’. In this case it defines both client and server stub, or more precisely, it defines the format of both the request and reply messages, from which the stubs can be generated. A number to be used as *h_command* value is picked from a range that is specified together with the class the procedure is defined in.

The client stub is called after the procedure. The server stub does not need a name, because it is not implemented as a function. The third routine associated with a procedure is the *implementation*, which should be written by every server implementor who wants to implement the class in which the procedure is defined. The name of an implementation is the name of the procedure, prefixed with *impl_*. An implementation has the same function prototype as the procedure where the type of the first argument, the ‘‘*’’ of the procedure, is implementation dependent. Currently it is always a pointer to a header.

If the ‘‘*’’ is absent, AIL still assumes that it describes a function that is logically part of the interface. An example is the function *dir_home*, which does not use *trans*, but should obviously be part of the directory interface. Other star-less procedures might be handwritten wrappers around a client stub. Thus, no command numbers or implementation routines are associated with these procedures. The assumption is that the basic RPC format will have to be published anyway, since that is Amoeba’s basic communication primitive.

Classes

A class groups together constant, type, marshaling, and procedure definitions. It can inherit the constants, types and procedures of other classes. Usually an integer range is specified, to allow the definition of procedures with the magic first star. AIL uses this range to pick a value for the *h_command* value. If the range is absent in the class definition, the class still represents a logical grouping of procedures, constants and types.
The interface is specified in an object-oriented style. This eases the development of servers that use standard interfaces. Let us suppose there is a class called *std_io* that holds the read and the write operators. A programmer that is building a server that knows about reading and writing its object, but also about destroying objects, simply defines a class that inherits the *std_io* class, and adds the destroy procedure. Of course, the programmer will have to provide the implementation of all the operators, including the inherited ones, but the old library routines for the read and write interface can be used.

AIL implements *multiple* inheritance. This allows the programmer to support an arbitrary number of classes, as long as their *h_command* ranges and the names defined in the classes are disjunct.

For each class AIL can generate a *server-mainloop*. The mainloop contains a big switch, surrounded by a *getreq* and a *putrep*. The cases in the switch are called *server stubs*. The task of a server stub is to get the arguments of a procedure, and call the corresponding implementation function, which should be supplied by the interface designer. The server mainloop contains such a server stub for every procedure that is defined or inherited by the class.

6.1.2. Language Specification

Scope rules

Identifiers declared within a class *C* are visible after their declaration, and in classes that inherit the class C. There is no concept of qualifiers as in Modula-2. This implies that if two separate classes declare identifier *foo*, it is impossible to inherit both in another class. Anything defined outside a class is defined in a global scope. These names are visible in classes that do not redefine the identifier.

The above rules do not apply exactly to union, enum, and structure tags. As in C, these always have a global scope. They are also the only exception to the rule that an identifier cannot be redeclared in the same scope. Declarations like:

```
typedef struct foo {
    struct foo *next;
    ...
} foo;
```

are allowed, and define *struct* foo in the global scope, and *foo* in the local one – which may be the global scope.

Syntax specification

Wherever the syntax of C was useful it was decided to stick with it. This mainly refers to type-declarations and expressions, but some other syntax constructs look C-ish too.

Notation used

The following notation is used in the presentation of the grammar:

Foo : ...	A rule defining the meaning of Foo
Foo	(capitalized words) nonterminals in the grammar
FOO	(all upper case) terminal symbols
'foo'	(quoted strings) literal terminal symbols
( )	used for grouping
Foo*	zero or more times Foo
Foo+	one or more times Foo
[Foo]	zero or one times Foo
{Foo DELIM}	one or more times Foo, separated by DELIM if more than one
Foo | Bar	separates alternatives

And here is the actual grammar:

```
AilSource    :   Definition*
Definition   :   TypeDefinition
             |   ClassDefinition
             |   GenerateClause
             |   ConstantDefinition
```

```
                |   MarshalDefinition

TypeDefinition    :   StructOrUnion [IDENT] ’{’ StructMember* ’}’ ’;’
                  |   ’typedef’ BaseType {TypeConstructor [MarshInfo]’,’} ’;’

StructOrUnion     :   ’struct’ | ’union’
StructMember      :   BaseType {TypeConstructor ’,’} ’;’
EnumMember        :   IDENT [ ’=’ ConstantExpression ]

BaseType    :   TypeModifier* IntrinsicType
            |   TypeModifier+
            |   StructOrUnion IDENT
            |   StructOrUnion IDENT ’{’ StructMember* ’}’
            |   ’enum’ IDENT
            |   ’enum’ IDENT ’{’ {EnumMember ’,’} ’}’
            |   IDENT

TypeModifier     :   ’long’ | ’short’ | ’signed’ | ’unsigned’
IntrinsicType    :   ’int’ | ’float’ | ’char’ | ’double’ | ’void’

TypeConstructor    :   IDENT
                   |   ’(’ TypeConstructor ’)’
                   |   ’*’ TypeConstructor
                   |   TypeConstructor ’[’ [ActualBound [ ’:’ MaximumBound] ’]’
                   |   TypeConstructor ’(’ ’)’

ActualBound        :   Expression
MaximumBound       :   Expression

ClassDefinition    :   ’class’ IDENT [ClassRange] ’{’
                           (InheritanceList | IncludeList)*
                           LocalDefinition*
                       ’}’ ’;’

ClassRange         :   ’[’ ConstantExpression ’..’ ConstantExpression ’]’
InheritanceList    :   ’inherit’ {IDENT ’,’} ’;’
IncludeList        :   ’include’ {STRING ’,’} ’;’

LocalDefinition        :   ConstantDefinition
                       |   TypeDefinition
                       |   OperationDefinition
                       |   MarshalDefinition
                       |   RightsDefinition
ConstantDefinition     :   ’const’ IDENT ’=’ ConstantExpression ’;’
                       |   ’const’ IDENT ’=’ STRING ’;’

MarshalDefinition    :   ’marshal’ IDENT MarshInfo ’;’
```

```
MarshInfo           :   'with' [ ActualSize [ ':' MaximumSize ] ]
                        [ 'in' [ClientMarshal] ',' [ServerUnmarshal] ]
                        [ 'out' [ClientUnmarshal] ',' [ServerMarshal] ]
ClientMarshal       :   IDENT
ClientUnmarshal     :   IDENT
ServerMarshal       :   IDENT
ServerUnmarshal     :   IDENT
```

The identifiers are supposed to be marshaling routines. They are called with a pointer in the transaction buffer, and the value they are supposed to marshal. They must return a pointer to where they left off. Server marshalers might get an extra parameter, indicating the architecture of the client. Currently, the most realistic way to learn the prototype of a marshaler is forcing AIL to write code that needs to call the marshaler, regrettably.

```
ActualSize      :   ['const'] Expression
MaximumSize     :   ['const'] Expression

RightsDefinition    :   'rights' {RightDef ','}
RightDef            :   IDENT '=' ConstantExpression
```

The constant expression must evaluate to a power of two, and must not be bigger than 2**8, which reflects the restrictions that Amoeba imposes.

```
OperationDefinition    :    IDENT [Identification] '(' ParameterList ')' [RightsList] ';'

Identification      :   '[' ConstantExpression AlternateNumber* ']'
AlternateNumber     :   ',' ConstantExpression
ParameterList       :   ['*' ',']  {Parameter ','}
RightsList          :   'rights' {IDENT ','}

Parameter        :   [AttributePart] [HeaderAddress] ParameterDef [MarshInfo]
AttributePart    :   ['var'] ['in'] ['out']
HeaderAddress    :   ':' IDENT
ParameterDef     :   BaseType TypeConstructor

GenerateClause   :   'generate' [ IDENT ] '{' {Generator ';'} '}' ';'
Generator        :   IDENT [ '(' {GeneratorArg ','} ')' ]
GeneratorArg     :   IDENT [':' SimpleType] ['=' Expression]

ConstantExpression    :   Expression
Expression            :   Expression DyadicOptr Expression
                      |   UnaryOptr Expression
                      |   '(' Expression ')'
                      |   FunctionCall
                      |   NUMBER
                      |   IDENT
```

Expressions

Expressions are modeled after C expressions. Some operators are missing. The remaining ones are listed below in precedence order:

||	denotes logical OR.
&&	denotes logical AND.
|	denotes bitwise inclusive OR.
ˆ	denotes bitwise exclusive OR.
&	denotes bitwise AND.
== and !=	denote equal and unequal.
>, <, >= and <=	test for greater than, smaller than, greater or equal, and smaller or equal respectively.
<< and >>	are the bitshift operators.
+ and –	denote addition and subtraction respectively.
*, / and %	denote multiplication, division and modulo.

Unary operators, which are all prefix, have the highest precedence. They are:

‘‘+’’, ‘‘–’’, ‘‘ ’’ and ‘‘!’’,

which mean nothing, unary minus, bitwise NOT and logical NOT. Parentheses can be used to override the precedence rules. Primary expressions are constants, identifiers and function calls. The identifiers in an expression must refer to an integer constant. Effectively, only integer expressions are allowed.

Lexical analysis.

Lexical analysis is the same as for C, except for the keywords. The input source is first preprocessed by the C preprocessor, then tokenized. The symbol *AIL* is predefined by the preprocessor to facilitate conditional compilation. Comments, spaces and tabs between tokens are ignored; within tokens they are illegal (except in strings, where they are meaningful). Comments are placed between /* and */. Comments do not nest. Comments are not recognized within strings, and vice versa.

The definitions below reflect the tokenization process. In the case of ambiguity the rule listed first is used. The longest production of a rule is always taken. Spaces are significant here!

IDENT	:	Letter [LetterOrOther]*
LetterOrOther	:	Letter | Digit | ’_’

NUMBER	:	OctalNumber | HexNumber | DecimalNumber
OctalNumber	:	’0’ Digit+ [Suffix]
HexNumber	:	’0’ (’x’ | ’X’) HexDigit+ [Suffix]
HexDigit	:	Digit | LetterAtoF
DecimalNumber	:	Digit+ [Suffix]

Currently, suffices do not mean a thing, but this will change:

Suffix	:	’l’ | ’L’ | ’u’ | ’U’

```
STRING          :   ’"’ CharOrEscape* ’"’
CharConstant    :   Quote CharOrEscape Quote
CharOrEscape    :   any character except \n, \, or the surrounding quote
                |   ’\n’ | ’\b’ | ’\f’ | ’\t’ | ’\r’
                |   ’\\’ Digit [Digit [Digit]]
                |   ’\\’
                |   ’\"’
                |   ’\’ Quote

Letter        :   ’a’ | ’b’ | ... | ’z’ | ’A’ | ’B’ | ... | ’Z’
LetterAtoF    :   ’a’ | ’b’ | ... | ’f’ | ’A’ | ’B’ | ... | ’F’
Digit         :   ’0’ | ’1’ | ... | ’9’
Quote         :   the single quote
DyadicOptr    :   ’||’ | ’&&’ | ’|’ | ’&’ | ’ˆ’ | ’!=’ | ’==’ |
                  ’<=’ | ’>=’ | ’<’ | ’>’ | ’+’ | ’–’ | ’*’ | ’/’ | ’%’
UnaryOptr     :   ’+’ | ’–’ | ’ ’ | ’!’
```

The keywords are: char, class, const, double, enum, float, generate, in, include, inherit, int, long, marshal, out, short, signed, struct, typedef, union, unsigned, var, void, with.

Types

The primary types of AIL are int, char, float, double and void. The meaning of int can be modified using long, short, signed and unsigned. Unlike most C implementations, AIL thinks a long long int is legal, and will feed this to your C compiler if you use one. An int and a long are four byte quantities on the network. A short is two bytes. For floats and doubles, no network format is defined yet. New types can be created by typedef, struct, union and enum, which are passed as integers. Arrays, pointers and functions are like in C, though neither pointers nor functions can be passed to procedures. There is an extended array type, which differentiates between an actual and a maximum size. Currently, the maximum must be constant. When transmitting such an array, the actual size is computed, and only that part of the array is transmitted.

Classes

A ClassRange defines an integer range from which AIL can pick command values. The classes mentioned in an InheritanceList are explicitly inherited. Any classes inherited by these classes are implicitly inherited. This difference is only relevant to some generators. The names defined in the inherited classes are visible in the current class. The IncludeList is used to tell AIL which header files must be included by the client_interface translation. A string starting with ‘‘<’’ and ending in ‘‘>’’ will be translated to an #include <stdio.h> style include, other strings are copied as is.

Procedures and parameters

Procedures define the prototype of a client stub. The implementation may have quite a different prototype, depending on other features of AIL you use in the generate clause for the server. The parameters to a procedure are identifiers with a type, and have certain attributes associated with them. The most obscure attribute specifies in which field of the Amoeba message header the parameter should travel. This attribute is there so that we can specify old client stubs in AIL. It is called the *HeaderAddress*.

The other attributes define in which direction the parameter values travel, and in which way they are passed to both client stub and implementation routine. They are *in*, *out*, and *var*.
The attribute *var* specifies that the parameter is passed by reference to both client stub and implementation routine.

The attribute *in* specifies that the parameter must be shipped in the request message.
Finally the attribute *out* means it must be shipped in the reply message. The attribute *out* implies *var*. If neither *in* nor *out* is specified, *in* is assumed. In effect, there are four possible combinations: in, var-out, var-in, and var-in-out.

The value of an out parameter after a communication failure is undefined. Its contents might have been overwritten during the processing of the reply by the Amoeba kernel.

AlternateNumbers are really a hack. Their use is not encouraged, and seldom necessary. They introduce numbers that should also be recognized by a server. Only the primary number is used in client stubs.

6.2. The AIL Cookbook

Introduction

This manual describes the parts of AIL that are target language specific: the code generators. This documentation is in a separate place since it is probably the fastest changing part of AIL. In particular, whenever support for a new language is added, this manual should be changed accordingly. It is assumed that the reader is familiar with the material in the *AIL Reference Manual*.

First of all, check that this documentation is consistent with the installed version of AIL. Type *ail –gG* to obtain a list of code generators that the installed version of AIL knows.

By convention, the generators that affect the behavior of other generators have capitalized names. An example of such a generator is *Language*, which sets the output language of AIL. Information like this is localized by resetting it to its default every time a new *generate* clause shows up.

Some code generators might have names that start with an underscore, indicating that they should not normally be used for some reason. Some may do things that are only useful when developing a new version of AIL, others may not be of astonishing quality, or are scheduled to disappear. They are listed here merely for completeness.

A concise description of the syntax to call a code generator is found in *The AIL Reference Manual*. As long as AIL does not support programming languages other than C, it is not necessary to document the obvious: where ‘‘header file’’ is written it means a C-file intended to be included whose name ends in ‘‘.h’’, and where it says ‘‘function’’ it refers to one written in C.

All code generators that create files put them in the output directory set by the command line **–o** option. The default output directory is the working directory. If the output directory is absent, it will be created.

The rest of this document describes the following code generators: Act_deact, class_list, client_interface, client_stubs, command, complete_interface, complete_stubs, contents, define, feature, flat_interface, Get_mem, Idempotent, impl_dummy, Impl_repl_buf, Language, marsh_funcs, Monitor_client, ordinal, output_files, Output_directory, Pass_acttype, Post_deact, server, type_list

Generator Act_deact

This one changes the output of generator server. The syntax is:

```
Act_deact(type-name, activate-function, deactivate-function)
```

It is used to call, for example, locking code, code to check the validity of the capability, or code to do additional cleanup between *putrep* and the next call to *getreq*. This code must be provided by the server implementor.

Type-name is supposed to be any valid type. AIL does not check this, because it might be declared in one of the ‘‘include’’ files. Note that this may lead to server code that does not compile.

The activate function must return an *errstat* as defined in *amoeba.h*. Its arguments are a pointer to the private part of the capability the RPC is meant for, and optionally a pointer to a variable of type type-name. The deactivate function is of type *void*, and optionally has one argument of type *type-name*:

```
errstat activate_function(private *_prv);
void deactivate_function(void);
```

As mentioned above, _obj parameters of type *type_name* * and *type_name* respectively are passed if generator *Pass_acttype* was used in the same generate clause.

After the request comes in, the activate function is called. If it returns something other than STD_OK, no implementation function is called, and the server returns the error status to the client. If it returns STD_OK, the activate function is supposed to have set *_obj. The implementation function is called, and then the deactivate function will be called with the variable set by the activate function, whether the implementation succeeded or failed.

See also : Pass_acttype, server, Post_deact.

Generator class_list

This generator is probably only useful for gurus. It makes a user-unfriendly dump of AIL's internal class data.

Generator client_interface

This generator creates a class specific header file. No parameters are allowed. It #includes the header files of all inherited classes and the files listed in the include line of the class definition, so in general a client_interface is needed for every class defined. Then it #defines the constants, declares the types, and declares the operators. These declarations will be prototypes if the language is *ansi*.

Then there is the prototype versus argument widening problem in C. If a function has been declared to have either a short, a char or a float argument, ANSI compilers can avoid argument widening when there is a prototype for the function in scope. If any operator has been declared that has such an argument, AIL will generate code that checks the `__STDC__` symbol. If this does not match the currently selected Language, the code will not compile, unless you have files with ridiculous names like traditional_c_required, or ansi_c_required.

To prevent multiple inclusions, the file tests and defines a preprocessor symbol whose name is a derivative of the name of the class. For example, the client interface for class *foo* – which is a C header file called *foo.h* – will define `__FOO_H__`.

See also : complete_interface, Language, Output_directory.

Generator client_stubs

This generates a client stub for each operator in a separate file. If invoked without arguments, AIL will generate the functions for all the operators defined within the class. Otherwise AIL assumes that the arguments are the names of the operators that should be generated.

Normally, this function contains code to marshal the parameters, call *trans*, and unmarshal the results. If the operator is declared to be idempotent, AIL generates code to retry in case of RPC failures. The function has exactly the same name as the operator, and is placed in a separate file with the name <operator-id>.c.

The function is written in the flavor of C selected by generator ‘‘Language’’, or the default language if none was selected.

AIL assumes that operators that do not have the special first argument ‘‘*’’ will be provided by the programmer. A request to write a complete client stub cannot be satisfied anyway, because AIL would not know which port to use. Instead, it will generate a dummy function that has the parameter list AIL expects it to have. It returns an error code right away. In this case the file is called <operator-id>.c.dummy.

See also : complete_stubs, Get_mem, Idempotent, Language, Monitor_client.

Generator command

This generator generates one or more stand alone programs that do one transaction. The arguments to the generator are the names of the operators for which a program should be created. If invoked without arguments, AIL will generate a program for each operator defined in the class. Each program appears in a separate file, called cmd_<operator-name>.c, e.g., *cmd_op.c* for operator *op*.

The program, when compiled and executed, expects the in-parameters of the operator as command line arguments. The first argument is converted to a capability using *name_lookup*(L). The out-parameters and their values are printed on *stdout*, except for any out capabilities. These are registered at the directory server using a path from the command line. Any previous entry with the same name is deleted, but not destroyed. If any error occurs, the program prints the most descriptive message possible given AIL’s limited knowledge of what it is doing, and exits with status 1. Hint: the first error to check is the number of parameters, which is fixed. Since these programs take at least one argument (the object capability), invoking it without arguments makes it print a usage string.

Currently, only integers, capabilities and character arrays are accepted as arguments. The way the latter are handled is a genuine bug, which will be around until AIL knows about strings. By then, the support for character arrays will probably disappear.

See also : Language, Output_directory.

Generator complete_interface

The generator *client_interface* only generates the header file for one class. In this file are #includes for inherited classes, which are not automatically generated by *client_interface*. This generator generates the header files of the effectively inherited classes. Like *client_interface*, it does not accept arguments.

See also : client_interface, Language.

Generator complete_stubs

This generator emits the same files as *client_stubs*, but for all the inherited classes, not the class self. It does not accept arguments. Note that a class might inherit more classes than just the ones listed in its inheritance list, since the classes mentioned might in turn inherit classes, which are then implicitly inherited by any class that inherits it.

See also : client_stubs, Idempotent, Language, Monitor_client.

Generator contents

Lists the contents of a class on *stdout*.

Generator define

This generator creates a header file that defines the *h_command* values it picked for each operator. To mimic C programmers, it folds the names of the operators to uppercase. The header file is called *<class>_def.h.*

Generator flat_interface

This generator creates a class specific header file, like *client_interface*. No parameters allowed. Unlike *client_interface*, it generates the definitions of the classes that are inherited explicitly, instead of including the associated header files. It #includes the header files of the implicitly inherited classes. For each explicitly inherited class, it includes the files listed in the include line of the class definition. Then it #defines the constants, declares the types, and declares the operators. These declarations will be prototypes if the language is *ansi*.

To prevent multiple inclusions, the file tests and defines a preprocessor symbol whose name is a derivative of the name of the class. For example, the client interface for class foo – which is a C header file called *foo.h* – will define `__FOO_H__`.

See also : Language, client_interface, Output_directory.

Generator feature

This generator generates several miscellaneous things. They are put together in a single file called *<class-id>_feature.c*.

The argument *def_inh* tells it to emit the declaration and initialization of a character pointer array called *def_inh*. These pointers point to the names of the classes in the *inherit* statement. The last pointer in the array is initialized to null. The argument *eff_inh* creates a similar array called *eff_inh*, containing the names of the effectively inherited classes. The first class listed is the superclass.

Generator Get_mem

For clients stubs, AIL by default does not generate code containing calls to either *alloca*, *malloc* or *free*. Specify *Get_mem(malloc)* to let the client generators use *malloc* and *free* to obtain and release dynamic storage. Specify *Get_mem(alloca)* to let them use *alloca*. There is an implicit *Get_mem(off)* before any generator is executed within a generate clause.

Generator Idempotent

Usage:

```
    Idempotent( [retry = expr, ] operators );
```

If an operator is declared to be idempotent, AIL will generate client stubs that retry <expr> times if *trans* returns RPC_FAILURE. The current default for *retry* is five. Note that *trans* is called at most *retry* + 1 times.

Please also note that the return value of a stub generated like this is the value for the *last* attempt. This means that `RPC_NOTFOUND` does not guarantee that the operation was never executed.

See also : client_stubs.

Generator impl_dummy

This generator makes dummy implementation routines for all the operators that should be implemented by a server for a class. The output is stored in a file called *<class-id>.dummy*.

If the language is set to *ansi*, AIL will generate a prototype style function, otherwise it will generate an old-fashioned K&R style function. The body contains the statement *return* `STD_ARGBAD`.

See also : Pass_acttype, Language.

Generator Impl_repl_buf

This generator takes several operator names as arguments, and tells AIL that the implementations for them supplies the buffer for the reply message. This is done to implement, for example, the Bullet Server's read request efficiently. An operator can only be implemented this way if there is exactly one parameter in the reply buffer. There might be more in the header of course. AIL assumes that no marshaling of this particular parameter is needed. Instead of passing a pointer in the buffer as usual, the server loop will pass two other parameters. The first one is a var pointer to `char` (a char ** in C), which must be set to point to the buffer. The other one is a `var long` (hence long * in C), and must indicate how big this buffer is. Obviously, this affects both server loop and implementation. The client stubs are not affected.

See also : server, impl_dummy.

Generator Language

This generator tells AIL which language should be generated. It takes one argument, which stands for the language. Currently, it knows about *traditional*, *ansi* and *lint*. All are C dialects. The default is *traditional*.

Not all generators actually obey this setting, because it does not make sense for some.

Generator marsh_funcs

This function generates the marshaling and unmarshaling routines for a typedef type. It expects one argument: the name of the typedef. The names of the functions must have been specified with a *marshal* statement.

See also : Language.

Generator Monitor_client

With this generator one can convince AIL to include monitor code in the named client stubs. If no operators are mentioned, all client_stubs will be monitored.

See also : client_stubs.

Generator ordinal

For tagged enumerated types, AIL can generate conversion routines from enumerated value to enumeration identifier and back. The identifiers are ordinary, statically allocated strings. The table of strings is called *tab_<tag>*. The conversion routine from enumerated type to string is called *str_<tag>*, and the conversion back is called *ord_<tag>*. They are to be found in the file *conv_<tag>.c*. The type of *tab_<tag>* is explicitly left undocumented, as is the behavior of the *ord_<tag>* function when invoked with an invalid parameter.

The functions are written in the flavor of C selected by generator ‘‘Language’’, or the default language if none was selected.

See also : Language.

Generator output_files

This generator prints a list of file names generated thus far. It takes one argument, which is considered to be the name of the file where it should print its list.

Generator Output_directory

This generator sets the current output directory for this generate clause. The output directory is reset to its default, or the command line value for each generate clause.

Generator Pass_acttype

This changes the output of a generator server. Before it may be called, an *Act_deact* must have been used. It makes the server pass to the implementation the value set by the activate function.

The arguments are the names of the operators to which the activate return value is to be passed. If no arguments are passed, the value is passed to all implementations.

See also : Act_deact, server.

Generator Post_deact

Before this generator is called, *Act_deact* must have been be used. There are no arguments. Normally, the deactivate function is called before the reply is sent to the client. If the deactivate function basically unlocks internal data structures, this is what you want. This generator makes *server* call the deactivate function *after* the call to *putrep*. This is done to facilitate single threaded servers and to be able to call *thread_exit* at a reasonable point.

Generator type_list

This prints on *stdout* a list of all known types, together with the address its descriptor happens to have within AIL.

Generator server

Used to create server main loops. The server loop is put in the file *ml_<class-id>.c*. Some aspects of the server loop can be modified by passing optional parameters.

> Specifying *rights_check* has the effect that the server will verify that a capability contains the appropriate rights for an operation before calling the implementation. Note that the generated code is insufficient to make a server secure. The capability should be

checked for the correct check field as well. Hint: this can be done in an activate function provided by the programmer. See *Act_deact*.

Specifying *monitor* has the effect that *monitor.h* is also included. See *monitor*(L) for the implications of this.

Another argument is *buf_size*, which should have a positive value, because it specifies the size of the message buffer.

Another way to provide a message buffer is passing it at run-time, which is indicated by the parameter *my_buf*. This results in a function that expects you to pass a *port* pointer, a buffer pointer and a buffer size, in that order. The buffer size is declared as a *bufsize*. Currently it is an unsigned 16-bit integer. This will probably change in Amoeba 6.0.

If neither buf_size or my_buf is specified, AIL is responsible for the buffer size. However, if any of the supported operators has a variable sized parameter, AIL does not succeed. It simply prints a warning and comes up with a compiled-in default.

The argument *no_loop* indicates that no loop should be generated. In this case the generated routine handles at most one request and returns zero if all went well.

The server loop looks like this:

First of all, *ailamoeba.h* and the client interface are included. The latter should be in the file *<class-id>.h*. Then the marshaling functions are declared (maybe these should be part of the client_interface?). Since C programmers might want to implement a marshaling function as a macro, every declaration is surrounded by #ifdef/#endif's. This is used by *ailamoeba.h* to speed some things up, by the way. The server main loop is implemented as a function called *ml_<class-id>*. It takes one argument unless *my_buf* was passed: a port-pointer. Unless *no_loop* is passed, the function does repeated calls to *getreq* on this port, and only returns if *getreq* fails. The return value is the return value from *getreq* is this case.

It decodes the request and calls the corresponding implementation function if all is well. This implementation function is supposed to have a name of the form *impl_<operator-id>*, so the implementation for *std_info* is *impl_std_info*.

The return value of the implementation is treated as an error code. It should be zero if all is well, and is shipped in the *h_command* field, which is #defined to *h_status*.

If the implementation function terminates successfully, the results are marshaled in a message buffer. After that, *putrep* is called.

See also : Act_deact, Impl_repl_buf, Language, Pass_acttype, Post_deact.

6.3. Amake User Reference Manual

6.3.1. Introduction

Amake is a software configuration manager that was designed to be a tool in the same spirit as *make*, i.e., it is a stand-alone tool that, when run by the user, should invoke precisely those commands that (re-)create a set of target files, according to a description file. The main idea is to make use of previous results as much as possible. Unlike standard *make*, *amake* is also able to exploit possible parallelism between the commands. Although, as the name suggests, *amake* was designed to be used on the Amoeba distributed operating system, it also runs under UNIX.

In contrast to a lot of ‘‘extended makes’’ the specification language is not a superset of *make*’s, but a totally new one. This paper focuses mainly on the syntax and semantics of the various constructs in the language. To make it as flexible as possible, *amake* has no built-in knowledge about specific languages or tools. The presence and usage of tools available can be declared in files that serve as a library of site-specific *amake* information. In order to make it possible to exchange a software configuration between various sites easily, a standard tool library is provided.

6.3.2. Why Amake

Amake is a software configuration manager developed to make use of the capacities — especially parallelism — of the Amoeba Distributed Operating System. While developing ideas about what features this new tool could and (most importantly) should have, it was found that keeping upward compatibility with *make* would constrain *amake* too much. The basic problem was that sticking to a make-like description file format would only allow us to introduce rules, written in *make* syntax, containing directives having a special meaning to *amake*. Although this is sufficient to provide the enhanced *make* with the information that, say, some additional action has to be taken when updating an explicit rule, it does not fundamentally change *make*’s concepts.

Two aspects of *make* are especially bothersome. First of all there is the (type, suffix) correspondence, upon which the implicit-rule construct is based. One would like to be able to group source objects according to other criteria than just sharing a common suffix. Another major problem with *make*, when used for large configurations, is that the description file is *task model* oriented, that is, it tells *how* instead of *which* things need to be done. As a result of this, *make* description files are system dependent and contain redundant information. For example, if all programs in a big software project need to be linked with some specific set of flags and libraries, this information has to be repeated in each Makefile.

Another problem with *make* is that its implicit rules, which it uses to derive which transformations it should perform automatically, only work when the source objects are in the *same* directory as the *makefile* and, consequently, the file objects produced. This is extremely bothersome when several versions of the resulting binaries have to be kept up-to-date. A well-known trick that tries to alleviate this problem is the creation of *shadow trees*, consisting of symbolic links to all the sources. However, apart from the fact that this takes up a lot of disk space, it in turn introduces the problem of keeping the symbolic links up to date (e.g., when a source file is added or moved).

As the aspects mentioned are fundamental to *make*, it was decided not try to implement yet another extended version of the existing program.

6.3.3. Overview

In *amake*, a software configuration can be specified by means of one or more *cluster* definitions. A cluster defines the targets to be constructed from a given set of sources. It is *amake*'s task to deduce which *tools* to use, in what order, and how to do it efficiently. The tools available can be defined by the user, but generally a reference to a common tool library (by means of a source inclusion mechanism) will suffice. In order to be able to decide if some file may be used or produced by a certain tool, each file is represented internally as an *object* with a set of valued *attributes*. (One possible value is ‘‘unknown,’’ indicating the absence of the attribute.)

A tool definition, which shows some resemblance to a function definition found in common programming languages, contains in its header references to the required values of certain attributes, such as ‘‘type.’’ Rather than letting the user specify the values of all the attributes needed, these values can most of the time be derived by *amake* itself, using *attribute derivations*. A well-known example is the *name-based* derivation: a file having suffix ‘‘`.c`’’ is assumed to be a C source file (hence having a `type` attribute with value `C-src`), unless explicitly specified otherwise. The *rules* defining derivations possible will generally be defined in a library, together with the tools that refer to the attribute values in question, but the user may of course provide additional ones. In derivations, as in each *amake* construct that allows expressions, built-in functions and previously declared tools may be used.

6.3.4. Example

For an illustration of the major differences between *make* and *amake* consider the simple compiler *comp* to be built from the source files *parse.y*, *lex.l*, *comp.c*, *defs.h* and the library *ident.a*. A *make* specification for this configuration is presented in figure 6.1.

```
OBJECTS = parse.o scan.o comp.o
LIBES = ident.a

comp: $(OBJECTS) $(LIBES)
	$(CC) -o comp $(OBJECTS) $(LIBES)

parse.o scan.o: defs.h
```

Fig. 6.1. Specification of a simple compiler in *make*

Note that *intermediate* files stemming from the application of an implicit rule are mentioned, rather than the sources themselves. This may introduce portability problems, because on one system the C-compiler might produce loadable objects (‘‘.o files’’), while on another it may generate assembly (‘‘.s’’) files. As there is no way to specify an implicit rule combining several files (possibly of different kinds as well), an explicit rule has to be provided in the manner shown on lines 4 and 5.

Also note the way *implicit inputs* have to be specified in *make*. Implicit inputs are source files that are read by the tool, in addition to the prime source file. In the example the files read by the C-compiler (or C-preprocessor) as a result of ‘‘#include’’ directives are implicit inputs. When one of the implicit files used in the construction of an intermediate file has changed, the associated command has to be re-invoked because it might deliver new results (i.e., when *defs.h* is modified, both *parse.c* and *scan.c* have to be recompiled). In *make* this implicit dependency has to be specified explicitly, as is done on line 7.

To contrast with *amake*, consider the *amake* specification of the same configuration in figure 6.2.

```
1 %include std-amake.amk;
2
3 %cluster
4 {
5     %targets comp[type = program];
6     %sources parse.y, scan.l, comp.c, defs.h, ident.a;
7 };
```

Fig. 6.2. Specification of a simple compiler in *amake*. Non-keywords are in italics.

The `%include` directive on line 1 causes *amake* to read the file ‘‘std-amake.amk’’, which contains a standard tool set constituting a C programming environment (containing a C-compiler, loader, library maintainer, *Yacc*, *Lex*, *Lint*, etc.). The cluster definition itself is clear, apart from the phrase ‘‘[type = program]’’ on line 3, which takes care of setting the attribute ‘‘type’’ of file *comp* (the *target* to be constructed) to ‘‘program’’. *Amake* needs this information, because, in general, various kinds of objects may be produced from a given set of sources (e.g., loadable objects may be combined to form either a runnable program or a library). In this example, *amake* is able to derive the *type* attribute of all source objects, using only suffixes.

A set of clauses describing implicit inputs is not needed in *amake*, because tools with an implicit input concept are supposed to deliver a list of them on a report file. If the tool is not able to do that, it is possible to specify in the tool’s definition that another program (such as *mkdep* for C-programs) should be called to deliver the information needed. *Amake* stores the set of *actual* inputs encountered when the tool was run in a (hidden) statefile, so the user is not bothered with keeping this information up-to-date.

Concluding, it is felt that *amake*’s way of describing software configurations is more convenient for the user, less system dependent, and — thanks to the automated implicit input and compilation flag administration — more secure.

6.3.5. Lexical elements

The lexical elements are described first, before turning our attention to the various language constructs, one by one. The main difference between the syntax of the language used in *amake* description files and the ones used in common programming languages is the fact that *keywords* rather than (file name or string) constants are quoted (prefixed with the character ‘‘%’’ in this case). Figure 6.3 shows the keywords of the *amake* specification language.

```
%and,     %boolean, %cluster, %computed, %condition,
%conform, %declare, %default, %derive,   %diag,
%do,      %export,  %false,   %generic,  %if,
%ignore,  %import,  %in,      %include,  %instance,
%list,    %not,     %or,      %out,      %return,
%sources, %string,  %targets, %tmp,      %tool,
%trigger, %true,    %unknown, %use,      %when
```

Fig. 6.3. Keywords

It should be noted that a large portion of the keywords deals with the construction of tool descriptions. Most people will just use an already existing tool set, so in practice only a small subset of the language will be encountered.

Literals, which may be used to serve as a string constant, or as (a component of) a file name, are sequences of upper- and lower case letters, digits, dots (‘‘.’’), minus-signs (‘‘−’’) and underlines (‘‘_’’). When a string containing special (but printable) characters needs to be specified, it has to be quoted by adding apostrophes (‘‘´’’) to the beginning and end of it. Apostrophes contained within a quoted string have to be doubled, so the string *I’m ready* has to be specified as ’I’’m ready’.

The names of global variables and parameters closely resemble literals, except that they may not contain dots or minus-signs. A reference to a previously defined parameter or variable can be made by prefixing the name with a dollar-sign (‘‘$’’). Note that in the declaration of a parameter or variable no dollar-sign is used, as is the case for the Bourne shell and *make*.

The operators and delimiters composed of special characters are shown in figure 6.4.

```
=       ==      <>      =>      +       ?
&       :       ,       ;       /       \
(       )       {       }       [       ]
```

Fig. 6.4. Operators and delimiters

Comments are introduced with the (unquoted) character ‘‘#’’ and continue till the end of the line. White space (spaces, tabs and newlines) is not significant, other than as a means to separate tokens.

6.3.6. Amake Constructions

The most important constituents of an *amake* ‘‘program’’ are the *cluster* and *tool definitions*, and the *attribute derivation rules*. They describe, respectively, the software configurations to be managed, the tools available, and how some attribute of an object could be computed, if it were still unknown. These basic constructs are used as an information base in the process of determining what actions *amake* has to perform, when updating a configuration. Although they may be considered as *declarative* language constructs (like *facts* in logic programs), the order of definition is sometimes significant, e.g., derivations rules are tried in the order in which they are declared.

Formally, at the highest level an *amake* specification file has the structure shown in figure 6.5. Note that each *amake-def* must be followed by a semicolon.

```
amake-program  ::=   amake-def-list
amake-def-list ::=   [ amake-def ";" ]*

amake-def  ::=       [ cluster-def          | default-def
                     | tool-def             | derivation-rule
                     | assignment           | declaration
                     | source-inclusion     | conditional
                     | generic              | instantiation
                     | import               | export
                     ]
```

Fig. 6.5. Global structure

In order to make it possible to share data between several constructs within a description file (e.g., clusters having partly the same sources) and among independent description files (a common tool library, for instance) there are also some imperative constructs available. They include the *assignment*, *source code inclusion*, *conditional definition*, a facility to create and instantiate *generic* pieces of *amake* code, and *environment* interaction.

Almost all top-level constructs may contain general expressions. In order to avoid getting too technical right away, expressions will be defined more formally in a later section. We will first go through the top-level language elements available.

6.3.7. Clusters

Clusters are used to describe a *configuration* of sources and targets that have to be created from the former. It is *amake*’s task to determine which tools it has to apply to achieve this. If there is more than one way to do this — for instance, two different compilers for the same language could be defined — *amake* points out the conflict, and the user will have to provide additional information, enabling *amake* to find a unique solution. The syntax of cluster definitions, as well as their semantics will be presented at the end of this section, but first a non-trivial example.

Example

To show a bit more of *amake*'s capabilities, consider the extended software configuration of figure 6.6.

```
 1 %include std-amake.amk;
 2
 3 SRC = { parse.y, scan.l, comp.c, defs.h, comp.a };
 4 FLAGS = '-DAMOEBA';
 5
 6 %cluster   # made by default.
 7 {
 8   %targets comp[type = program];
 9   %sources $SRC;
10   %use     cc-c(flags => $FLAGS, optimize => %true);
11 };
12
13 %cluster   # only made when asked for.
14 {
15   %targets comp_debug[type = program];
16   %sources $SRC + debug.c;# extra source
17   %use     cc-c(flags => $FLAGS + '-DDEBUG');
18 };
```

Fig. 6.6. Extended configuration example

This definition describes a common software development situation: we have a fast ‘‘production’’ version (defined on lines 6 through 11) and a debug version (defined on lines 13 through 18), which probably has some additional capabilities. Common *make* practice when we want to switch from a debug- to a fast version is the following procedure:

1) execute `make clean`, which ought(!) to throw away previously produced files

2) execute `make CFLAGS=-O`, which creates the version wanted.

In contrast, *amake* maintains its own *object pool* of files generated, without the user having to notice it (other than that some builds might be unexpectedly quick). Besides the advantage of not having to look at a directory cluttered up with objects that are — or should be! — of no interest to the programmer, this gives *amake* the opportunity to retain intermediate objects that may be of use in later runs.

It is even possible to maintain both (fast and debug) versions in one run: there is an internal locking mechanism that allows both clusters to be updated. (If there are sufficient pool processors, one might as well use it). If this kind of use is intended, the targets must have different names or destination directories, of course.

As a final remark, situations as above, where some constructs — clusters in this case — are repeated with minor differences, can be handled more conveniently by making use of *generics*, which are described later on.

Syntax

The syntax of a cluster definition is shown in figure 6.7.

```
cluster-def        ::= %cluster cluster-name ?
                       "{"
                       [ %targets expression-list ";" ]?
                       [ %sources expression-list ";" ]?
                       [ %use     tool-usage-list ";"
                       | %do      statement-list
                       ]?
                       "}"

cluster-name       ::= expression

tool-usage-list    ::= tool-usage [ "," tool-usage ]*
tool-usage         ::= identifier "(" argument-list ")"

statement-list     ::= [ statement ";" ]*
statement          ::= expression
```

Fig. 6.7. Cluster definition syntax

Note that the cluster name may be an expression — rather than just a literal — which has to evaluate to a string. This makes it possible to include a cluster definition in a *generic*, which is handy in case a lot of similarly looking clusters have to be defined. We will give an example of this construct when generics are defined.

Cluster names can be given as argument to *amake*, in order to specify another cluster than the default one (see the section on defaults). A cluster name may also be absent, in which case *amake* takes the cluster to be a candidate for use as *subcluster*. Subclusters create targets that are sources of other clusters. Given the clusters to be updated, the subclusters are recursively determined.

The need for this construct arises when either a tool hierarchy creating the targets from the sources cannot be constructed or — a much more common case — when the user wants to create a *set* of targets, each defined by its own cluster containing its private sources, without having to specify them all on the command line.

The `%targets` and `%sources` specification each require an expression delivering a list of objects. Note that these expressions also may contain attribute assignments. In this case they are taken to be *local* attributes, i.e., it is possible in *amake* to let an object have type `C-src` in one cluster, and type `text` in another.

The `%use` clause serves two purposes. Firstly, it may be used to specify different defaults than supplied by the tool definition itself. Secondly, it can be used to remove a conflict that arises when *amake* finds that there is no unique set of tools that create the targets from the sources. Normally, when the standard tool set is used, this should not be necessary.

The `%do` clause gives an opportunity to override *amake*'s default behavior of deciding which tools to run. It resembles *make's* explicit rule mechanism. The *statement-list* is just a boolean expression to be evaluated; the semicolons are interpreted as `%and` operators. This statement list will generally consist of an `exec` call, which is executed each time the cluster is run. Alternatively an explicit tool invocation can be made. In this case it is only executed when the inputs of the tool (usually consisting of all the sources of the cluster) have changed. If the statement list does not evaluate to `%true`, the build of the cluster is said to have *failed*, which will be reported.

Clusters that have neither a `%use` nor a `%do` clause, are handled as `%use` clause clusters that do not change defaults, and thus let *amake* find out which tools it has to run in what order.

Default Clusters

As *amake* allows multiple clusters to be declared, there has to be a way to tell it which clusters are to be brought up to date. Apart from specifying clusters to be built as argument when *amake* is invoked, it is also possible to mark some of the clusters as ''default''. They will be constructed in case no cluster is mentioned. Default clusters are specified with the following syntax:

default-def `::=` **%default** *expression*

The expression given as argument should evaluate to the names of the clusters (or corresponding targets) to be built by default. Multiple default definition are allowed.

6.3.8. Attribute Derivation Rules

When *amake* has to decide whether a certain object may be used as the input or output of some tool, it first looks at attributes already known about the object. If the value of an attribute that is needed is not present, *amake* tries to *derive* it, using the *attribute derivations rules* relevant for that attribute. Such a rule has two parts: a specification of the attribute/value pairs to be attached, and a *condition* which places a restriction on the rule's applicability. A derivation is introduced as follows:

derivation-rule `::=` **%derive** *identifier* `"["` *attr-spec-list* `"]"`
%when *expression*

The identifier (say ''id'') serves as a parameter to the derivation rule, so that it can be referred to (by ''$id'') in the rest of the rule. When a rule is tried, the parameter is instantiated to the object for which the attribute(s) have to be computed.

Derivations rules are tried one by one, in the order in which they are defined. If one of the derivations *succeeds*, i.e., its condition evaluates to `%true`, the values specified are permanently attached to the object. If, however, the attribute can not be derived, the special value `%unknown` is given by default.

An often used derivation is that of the suffix based ''type'' attribute. Consider the examples in the next figure.

The third derivation is especially interesting. Tool definitions (described in the next section) require the attributes of in- and output objects to be explicitly defined; there is no way to specify that an object has got to have value ''C-incl'' *or* value ''C-src'' for attribute ''type''. By introducing new attributes and corresponding derivation rules, one can work around this restriction. The final derivation shows how we can prevent *amake* setting an attribute to

```
%derive f[type = C-src]  %when matches($f, '%.c');
%derive f[type = C-incl] %when matches($f, '%.h');
%derive f[C-src-or-incl] %when $f ? type == C-src
                         %or   $f ? type == C-incl;
%derive f[any-src]       %when %true;
```

Fig. 6.8. Derivation rule examples

`%unknown`, by default.

6.3.9. Tool definitions

The existence and characteristics of tools that might be used by *amake* have to be made known by means of *tool definitions*. Tool definitions form an interface to a set of commands (defined in the tool body) that are to be executed. They also provide more semantic information than the (textual) commands themselves.

The main properties of a tool definition are:

- It describes what attribute values the input objects are supposed to have. The attributes are examined in the algorithm *amake* uses to decide whether the tool should be applied to source or intermediate objects.
- It describes what attribute values will be attached to the output objects. This information is needed because the files produced possibly require processing by subsequent tools. The attributes can also be used to determine what name the default output objects will have. It often happens to be the case that the default output file has the same *basename* as one of the inputs.
- It is possible to create a relatively system independent interface by introducing boolean, string, or string list parameters, which may influence tool behavior.
- Temporary files can be specified so that *amake* can take measures not to let two tools, using the same file, interfere.
- It is possible to supply default values for most of the parameters, so that only differences with the standard behavior are to be specified in the cluster's `%use` clause.

Example

We will first give an example before turning to the general case. Figure 6.9 contains a tool definition of the UNIX program ''lex'', which produces a lexical scanner (in C) from a description file.

As can be seen from this example, tool definitions have a lot in common with functions or procedures found in common imperative programming languages. Lines 1-7 form the *tool header*, giving the name of the tool and (in the parameter list) describing the kind of objects used as in- and outputs, side-effects and options of the tool. Lines 8-11 contain the *tool body*, describing the actions to be executed. In this case, the result of the tool invocation is the boolean result of the evaluation of the tool body.

Going through the header, we notice that the parameter `src` is the only parameter which

```
1  %tool lex (
2      src:   %in     [type = lex-src];
3      dest:  %out    [type = C-src]    => match($src);
4      temp:  %tmp                      => lex.yy.c;
5      flags: %string %list             => {};
6      prog:  %in     [type = command]  => /usr/bin/lex;
7  )
8  {
9      exec($prog, args => $flags + $src);
10     move($temp, $dest);
11 }
```

Fig. 6.9. A tool definition for Lex

does not have a default. This should not be too surprising, because this parameter represents a possible lex input file — note it is marked `%in` — and it is *amake*'s very job to find out whether this tool can be applied to one of the files given as source in a cluster. Many tool definitions (e.g., the ones currently present in the standard tool definition set) have the same basic structure as the one presented here.

The next parameter (`dest` on line 3) is of type `%out`, i.e., it denotes a file which is supposed to be generated by the command block of the tool. The default name for this file is computed by the built-in `match` function. Actually, this function needs some help provided by a derivation rule, but this will be described in a later section. The result, however, is that an input file `lexer.l` will cause an output file `lexer.c` to be created.

Line 4 makes the hard-wired temporary file ‘‘`lex.yy.c`’’ explicit. As *amake* has a locking mechanism, which takes care of tool synchronization, this will prevent it to run 2 invocations of Lex at the same time.

The various options of the tool are represented on line 5 by the parameter `flags` of type `%string %list`. As the *command* Lex only has few options an alternative — perhaps more convenient, or at least more portable — approach would be to introduce a parameter of type `%boolean`, one for each option. The reason why this is not done here, is probably lazyness on part of the *amake* programmer.

The syntax of *amake*'s tool definitions is presented in figure 6.10.

A tool returns the value resulting from the evaluation of its body, unless a `%return` directive is used as file argument. In that case the result consists of the contents of this file (which can be referred to in the tool body by the parameter name) after the body has been executed.

Input and output objects

The input and output objects are described by a set of attributes, possibly including the `type` attribute, as was the case in the example above. The main advantage of having some special ‘‘type’’ attribute is that it allows us to create *generic*, suffix-based, derivation rules (generics are described in a later section). Note that an *attribute class* may contain more than one ‘‘attribute = value’’ pair, so that we can specify that a tool only takes, say, non-generated C-

```
tool-def        ::= tool-header trigger? tool-body
tool-header     ::= %tool identifier "(" [ param-decl ";" ]* ")"

param-decl      ::= identifier ":" type-spec
                    [ "=>" %computed param-ref
                    | "=>" default-spec
                    ]?
                    [ "=>" %conform param-ref ]?

type-spec       ::= base-type %list ? [ "[" attr-spec-list "]" ]?
base-type       ::= %boolean | %string | %in | %out | %tmp

default-spec    ::= expr-designator
trigger         ::= %trigger expression
tool-body       ::= "{" statement-list "}"
param-ref       ::= "$"identifier
```

Fig. 6.10. Tool definition syntax

source files as input.

It also is of prime importance that a tool definition describes all the inputs a certain invocation has, otherwise a tool might be started before all inputs (which might be generated by other tools; think about parser generators) are available.

Not specifying all inputs also means that *amake* cannot check whether all inputs are the same as in previous invocations. E.g., if in the example the parameter ‘‘prog’’ is left out, the tool lex will not be rerun when a new version of the program ‘‘/usr/bin/lex’’ is installed. It is possible, however, that this behavior is convenient in some cases, e.g., when a command which often changes (because it is being currently developed) is used.

A `%conform` clause makes it possible to describe commands where the creation of a file is dependent on the value of a parameter. In this case, the tool should be given a boolean parameter (probably with a default value) telling whether the side-effect should occur. In the tool body, the value of the boolean parameter can be used to decide which flag the command should be provided with.

Computed Inputs

Amake contains a very useful feature, allowing tools to make known the *actual* inputs it has read during a certain invocation. This can be specified with the `%computed` clause. Figure 6.11 contains an example of this construct. It is an adaptation of the C-compiler tool definition currently used in the construction of Amoeba libraries and utilities. For a source file `f.c`, the parameter ‘‘deps’’ will expand to `f.dep`. The `-d` option requests the C-compiler to report (on file `f.dep`) the header files encountered. This list of implicit inputs will, as soon as the tool body has been completed, be assigned to parameter ‘‘incl’’.

When a tool, making use of this feature, is invoked for the first time, *amake* is *pessimistic* (or at least not overly optimistic) in that it lets the tool wait until all the objects that could

```
%tool cc-c (
    src:   %in       [type = C-src];
    obj:   %out      [type = loadable]  => match($src);
    flags: %string %list                => $CFLAGS;
    prog:  %in       [type = command]   => $CC;
    deps:  %out      [type = dependents]=> match($src);
    incl:  %in %list [C-incl,implicit]  => %computed $deps;
)
{
    exec($prog, args=> '-d'$deps + '-c' + $flags + $src);
};
```

Fig. 6.11. A tool definition with computed inputs

possibly be inputs are available. This is important because some of them might be generated by other tools. For example, the tool Yacc may produce a header file included by several C source files.

If the same invocation is to be checked in a later *amake* run, *amake* knows both the specified and the computed input objects, and the tool only has to be rerun when one of these has changed. As each tool invocation reports the objects that were read, *amake* will never erroneously fail to recompile a source file, in the event that some of the header files have changed.

6.3.10. Expressions

Expressions in *amake* are typed. The types available are *boolean*, *string*, *object*, *unknown* and *lists* of these. Arguments of operators, functions and tools are automatically converted to the type required, if possible.

Overview

Objects are *amake*'s internal representation of (not necessarily existing) files. Properties of the files that are of interest to *amake* are represented as attributes. We must stress that the correspondence is not very strict: the ‘‘physical’’ object may have certain attributes which are not represented internally (e.g., contents, although it *is* possible to represent it), while the internal object will have attributes (such as type, and other properties used in *amake*'s algorithms) which are not necessarily represented anywhere on the file system. The hierarchical structure, which most file systems have, is represented internally, however.

As in UNIX and Amoeba, objects are identified with their *path name*. It is important to note that *amake* (in contrast to, e.g., the Bourne Shell and *make*) interprets the character ‘‘`/`’’ as a real *selection* operator, rather than just a character which is part of a path name. This has the advantage that an *amake* specification file might also be used, without change, in an OS environment that uses a different way to represent the file hierarchy.

In *amake*, strings and booleans are used mainly as attribute value and options to tools or commands. Boolean expressions can also be used to force control flow within a tool invocation. A special ‘‘unknown’’ value is given to an attribute whose value is neither set explicitly, nor derivable.

Expression Syntax and Semantics

Figure 6.12 shows how expression are built up, recursively.

```
expression ::=           literal
                       | string
                       | var-reference
                       | %true
                       | %false
                       | %unknown
                       | identifier "(" argument-list ? ")"
                       | "{" expression-list ? "}"
                       | "(" expression ")"
                       | expression "[" attr-spec-list "]"
                       | expression binop ? expression
                       | %not expression
                       | "/" expression ?
                       | expression "/"

binop ::=              + | \ | == | <> | & | / | ? | %and | %or

argument-list ::=      argument-spec [ "," argument-spec ]*
argument-spec ::=      [ identifier "=>" ]? expr-designator
expr-designator ::=    expression | designator
designator ::=         %ignore | %return | %diag
expression-list ::=    expression [ "," expression ]*

attr-spec-list ::=     attr-spec [ "," attr-spec ]*
attr-spec ::=          literal [ "=" expression ]?
```

Fig. 6.12. Expression syntax

Literals, strings and variable or parameter references were introduced in the section on lexical elements. The possible confusion whether the string ‘‘´a´’’ or the object ‘‘a’’ is meant, disappears when the value is used, for example when it appears as source of a cluster. The meaning of `%true`, `%false` and `%unknown` should be clear. The next entry represents both built-in function call and tool invocation. Lists (of objects, strings etc.) can be built with curly brackets (‘‘{’’ and ‘‘}’’). Parentheses can be used to force a grouping, differing from the default one.

Attributes can be attached to objects by means of the special ‘‘[’’ operator. The literal in the *attr-spec* is the name of the attribute to be set. If the expression specifying the value of the attribute is absent, the value `%true` is assumed by default. When evaluated, the attribute assignment expression puts the attributes on its right hand side on the object(s) on its left hand side, which also becomes the result of the expression. This makes it possible to let list building coincide with attribute setting, for example as in

```
SRC = { fool.c[C-src = %false], real.c } [author = versto];
```

The operators, their priority and their interpretation are summarized in the table of figure 6.13.

operator	priority	interpretation	domain	range
%not	8	not	boolean	boolean
/	7	select	object x string	object
&	6	concat	string x string	string
+	5	append	list x list	list
?	5	get	object x attribute	type
[	5	put	object x attribute x value	object
\	4	remove	list x list	list
==	3	equal	type x type	boolean
<>	3	unequal	type x type	boolean
%and	2	and	boolean x boolean	boolean
%or	1	or	boolean x boolean	boolean

Fig. 6.13. Operators and their interpretation

If the binary operator is omitted, it is interpreted as a concatenation. The ‘‘?’’ operator delivers the value of the attribute on its right hand side argument for the object on its left hand side. If it is not present, *amake* tries to derive it first. If that does not succeed either, the result is `%unknown`.

The `%and` and `%or` operators have the same interpretation as the && and `||` operators in the C programming language, i.e,

```
eval(b1 %and b2) = if eval(b1) then eval(b2) else %false fi
eval(b1 %or b2)  = if eval(b1) then %true else eval(b2) fi
```

Two lists are equal if and only if their respective components are equal, i.e., the order and multiplicity of elements *does* matter. Lists within lists are flattened, so ‘‘`{ a, { b } }`’’ evaluates to ‘‘`{ a, b }`’’. Be careful not to forget the commas separating the components in a list expression: the expression ‘‘`{ bet ray }`’’ evaluates to ‘‘`{ betray }`’’, because of an *invisible* concatenation operator between the two strings.

Since, as noted before, the slash is an ordinary operator, white space *is* allowed (but ignored) in path expressions. The entries in the grammar specification for expression containing the slash operator are there so that UNIX path names can be accepted without change: a slash on its own denotes the root directory, so ‘‘`/comp`’’ selects ‘‘`comp`’’ within the root directory. A trailing slash is ignored.

Type Conversion

Type conversion is done automatically when needed; the conversions possible are summarized in figure 6.14.

from	unknown	string	object	boolean
unknown	*	’unknown’		
string	’unknown’	*	*./component*	’true’, ’false’
object		*path name*	*	
boolean		’true’, ’false’		*

Fig. 6.14. Conversion table

In words: types boolean and unknown are converted to and from strings by means of ’true’, i.e., the keywords without the ‘‘`%`’’. Conversion from object to strings delivers the path name of the object. If the object resides in the current directory or one of its subdirectories, a relative path name is generated, otherwise an absolute path name. Conversion from string to object is done by assuming that an object within the current directory is meant.

When an operator or function expects a list argument, atomic arguments are automatically converted to the corresponding ‘‘singleton,’’ so ‘‘`{ a, b } + c`’’ is converted to ‘‘`{ a, b } + { c }`’’, which in turn evaluates to ‘‘`{ a, b, c }`’’.

Built-in functions and Tools

There are two kinds of calls: built-in function call and tool invocation. Tool definitions have been introduced in a previous section. Most of a tool’s parameters will generally be provided with defaults, and the (Ada-like) invocation syntax reflects this: only a subset has to be specified, and both positional and named argument syntax are accepted. Figure 6.15 shows the built-in functions currently available.

The Exec Function

The function `exec` takes care of the actual execution of a command on the system. It resembles the way a single command is specified in the Bourne shell — with syntax adapted to *amake*’s, of course — but we considered it general enough to be used as a system-independent interface. Just like tools, the exec function contains defaults to make life easier. Parameter `args`, requiring a string list, is used to specify file arguments and command options, and defaults to the empty list. Standard input is by default redirected to an empty file, and both standard output and standard error are redirected to a temporary diagnostic file.

After the command has finished, the diagnostic files are examined, and their contents shown on *amake*’s standard output. The reason why they are redirected by default in the first place, is to prevent output of several parallel jobs to appear intermixed on the terminal, thereby probably confusing the user.

The designators `%ignore`, `%diag` and `%return` can be used as file arguments to the exec function and tool invocations, and have special semantics. The keyword `%ignore` acts as */dev/null* on UNIX, i.e., empty file when used as input, and ‘‘kitchen sink’’ when used as

function	arguments	meaning
`exec`	`(prog, args, stdin, stdout, stderr)`	execute a command with possible redirection
`if`	`(condition, then, else)`	conditional evaluation
`echo`	`(strings)`	print arguments on *amake stdout*
`defined`	`(variable)`	has a variable been assigned to?
`exit`	`(integer)`	exit amake prematurely
`move`	`(from, to)`	rename a file on the file system
`select`	`(list, attr, value)`	select objects specified
`get`	`(object, attribute)`	return value of attribute, or %unknown
`vpath`	`(name, dirs)`	first d in dirs for which d/name exists
`lcomp`	`(object)`	last component of path
`basename`	`(object, pattern)`	remove prefix and/or suffix
`matches`	`(string, pattern)`	%true iff strings matches pattern
`match`	`(object)`	match basenames of objects

Fig. 6.15. Built-in functions

The following example illustrates the use of defaults and call syntax:

```
exec(/bin/cc, args => '-E' + $base.c, stdout => $base.e)
```

output file. Diagnostic files, to be shown on standard output when the command or tool has finished, are created with `%diag`. Usually, evaluation of commands and tools delivers a boolean value, indicating success or failure. Supplying `%return` as argument causes redirection to a temporary file, whose contents will be returned as a string list — the file is supposed to contain readable output — so

```
exec(/bin/cat, args => hello, stdout => %return)
```

effectively works as a ‘‘contents’’ function. Note that this is equivalent to the command

```
`/bin/cat hello`
```

in the Bourne shell.

General Utility Functions

Conditional evaluation is possible with the `if` function. If its first argument evaluates to `%true`, it delivers the value its second argument, otherwise that of its third. Only the argument to be delivered is evaluated (for other functions all arguments are evaluated first). The third argument is optional: a default value (empty list, empty string, or `%true`) will be assumed, depending on the type of the second argument.

When creating a description file that is supposed to be suitable for different environments, it is sometimes convenient to be able to act upon the fact whether some variable has been set. This can be achieved with the use of the function `defined`, usually in combination with the `if` function.

The function `echo` can be used to write a informative message on *amake*'s standard output. It delivers `%true` as return value. The `exit` function also prints its arguments, but after that it forces a premature *amake* exit. This is useful when, for example, some vital variable has not been set. Examples:

```
if ($OS == AMOEBA, ainstall($prog))
if (defined(OPTIMIZE), $OPTIMIZE, '-O')
if (%not defined(OS), exit('$OS not defined'), echo('O.K.'))
```

The function `get` is an alias for the ‘‘?’’ operator, i.e., it returns the value of a certain attribute of an object, possibly after deriving it.

There is also a function `select`, which filters precisely those objects from an object list that have a particular value for some attribute. Note that the attribute referred to might have to be derived first. For example:

```
select({parse.y, scan.c}, type, C-src) == {scan.c}
```

As the example indicates, this function can be handy when some special cluster only needs to have a subset from all sources as input (e.g., only the ‘‘lintable’’ sources).

Object functions

Sometimes it is not known in advance where a file resides (e.g., its place might change from time to time). The function `vpath` (the ‘‘v’’ stands for ‘‘view’’) returns the *first* occurrence of a file in a list of directories, having a specified string as last component. Example:

```
PATH = { /usr/local/bin, /bin, /usr/bin, /usr/ucb/bin };
CC = vpath(cc, $PATH);
```

It is common practice that system commands create output objects based on the name of the input(s). To give some way to describe this within a tool definition, and also to allow tools themselves to implement this kind of behavior, the function `basename` is provided. It can be used to strip off a known prefix and/or suffix.

The predicate `matches` returns a boolean telling whether its first argument matches the pattern given as second parameter. This function is often used in a derivations of the `type` attribute.

The function `match` (not to be confused with the previous one) involves some special trickery, and can only be used in tool contexts to derive the name of an output object based on that of an input object. This will be described in a later section.

Examples:

```
basename(file.c, '%.c') == file
basename(pre.file.post, 'pre.%.post') == file
basename(file.c, '%.y') == file.c

matches('down.under', '%.under') == %true
matches('down.under', '%.above') == %false
```

Notice that an unmatching pattern causes basename to return its first argument.

6.3.11. Assignments and Declarations

In *amake*, variables are primarily used to avoid having to repeat lists of strings or objects in different places. The variables are *not* textual macros, so assignments expect an expression as right hand side.

> *assignment* `::= [` *identifier* `"=" ]?` *expression*

As expressions are typed, so are variables. Note that there are no restrictions on the kind of expressions: tools may also be called, for instance. Variables may be assigned more than once.

Assignments are not necessarily executed directly the moment they are read; they might be cached until their value is used, or all input has been read. If an expression should evaluated because of its side effect rather than its resulting value, it is allowed just to mention the expression itself. In that case, the expression is evaluated right away. Example:

```
%if (%not defined(TOOLSET), {
   exit('panic: TOOLSET not defined'); # exit right away
});
```

Values of variables can also be taken from the environment and put in the environment of commands executed. See the section on import and export for details. The following variables have a special interpretation:

PWD	the directory containing current file being read
AMAKELIB	the search path for files to be %included
ROOT	the root object, ‘‘/’’

The variable `PWD` is particularly useful: it makes it possible to maintain *amake* source descriptions, completely independent of the place where the targets are built. The source description file, residing in the same directory as the sources, can refer to the sources by means of ‘‘`$PWD/sourcename`’’. An `Amakefile` in the configuration directory can then include the source lists required, and specify what should be constructed out of it.

Declarations

Declarations can be used to attach attributes to a list of objects.

> *declaration* `::=` **%declare** *expression-list*

Each expression in the expression list is required to be an attribute assignment. The expressions are evaluated one by one, which has the effect of the attributes being set. Example:

```
%declare $SRC[author = versto], foo.c[author = nobody];
```

It should be noted that the same effect could be achieved by means of assignments, but the advantage of using declarations is that they do not require invention of a variable name that is not used afterwards.

A possible application is the situation where a *derivation rule* is used to set the value of some attribute by default, and a *declaration* takes care of setting the attributes to some different value for the exceptions. So a more generic version of the example above would be:

```
%derive f[author = versto] %when %true; # default case
%declare foo.c[author = nobody];        # exception
```

6.3.12. Source Inclusions

The primary tool in structuring description files is the re-use of tool definitions, derivation rules, etc., by means of source file inclusion:

source-inclusion `::=` **%include** *expression-list*

The expression-list should evaluate to a list of *amake* source file names. If the resulting name is just a component, *amake* will read the first object with that last component, found in the search list described by the variable `AMAKELIB`. If a file to be included cannot be found, a fatal error will be triggered.

Conceptually the `%include` directive is replaced by the concatenation of the contents of the files. However, a file will never be included more than once.

The search path can be altered in various ways:

- simply assign it in the specification file itself. This is convenient if a certain configuration needs some non-standard set of description files.
- define AMAKELIB on the command line. This is useful when looking for the differences between two description sets. See the manual page for the exact syntax.
- insert a path in front of the standard path list with another option. This is handy when temporary use of a few private tool descriptions (placed in the directory mentioned) is desired. Again see the manual page for specifics.

6.3.13. Generics

Generics can be used to avoid having to repeat definitions that are equal up to some values (strings, objects, booleans). Generics have to be defined in a `%generic` clause, before they can be instantiated with an `%instance` directive, which contains the generic's name and the values for its parameters.

generic	`::=`	**%generic** *identifier* `"("` *id-list* `")"` `"{"` *amake-def-list* `"}"`
id-list	`::=`	*identifier* `[ ","` *identifier* `]*`
instance	`::=`	**%instance** *identifier* `"("` *argument-list* `")"`

Fig. 6.16. Syntax of generics

In the current implementation, instantiation of a generic has the effect of making (global) assignments to the parameters, followed by a copy of the body of the generic. This means that care must be taken not to let a generic's parameters coincide with one of other variables.

We will give two examples where the use of generics makes the specification simpler, and less error prone. The first deals with derivations, and is used in many tool libraries. The definition, which is part of the standard tool set, is shown in figure 6.17.

```
1 %generic deftypesuffix(tp, pat) {
2    %derive f [type = $tp]     %when matches($f, $pat);
3    %derive f [def-pat = $pat] %when $f ? type == $tp;
4 };
5
6 %derive f [base = basename($f, $f ? def-pat)]
7 %when   $f ? def-pat <> %unknown;
```

Fig. 6.17. Example use of generics (1)

A possible instantiation (in this case for the cc-c tool) would be:

```
%instance deftypesuffix(C-src,    '%.c');
%instance deftypesuffix(C-incl,   '%.h');
%instance deftypesuffix(loadable, '%.o');
```

An instantiated version of *deftypesuffix* takes care of linking the *type* and the *def-pat* (short for ‘‘default-pattern’’) attribute of an object. The first derivation of the generic (line 2) is used to derive the type of an object when its name matches some pattern (e.g., an object with name matching ‘‘`%.c`’’ is considered to have ‘‘`type = C-src`’’ by default). The other derivation included in the generic (line 3) has to do with the rather special semantics of the `match` function. This function is used to specify names of generated objects, having the basename with the input in common. To be able to do this, it requires

- the *def-pat* attribute of the generated object, supplying prefix and/or suffix of the name of the generated objects.
- the *base* attribute of the source object, which replaces the ‘‘%’’ in the pattern for the generated object.

The irony — or perhaps confusing detail — is, that the *base* attribute of the *source* object is also derived, using the *def-pat* attribute (see lines 6 and 7).

The second example deals with clusters. Consider the case when we have a whole bunch of programs to maintain, all using tools with the same defaults. Of course we could introduce a cluster for each configuration and copy the `%use` clauses. This has the problem that we have to change each cluster when we would like to override one tool default. A better way to do this is therefore first to create a generic as is done in figure 6.18.

In the instantiations we then only need to specify target and sources, for each of the configurations to be maintained. Note that in the definition of *gencluster* a `%default` clause is included in order to let *all* instances created with this definition to be updated by default.

Of course this generic can also be extended to have a number of tool parameters as argument, to use conditionals (see below), or to define several other clusters — such as *print*, *install*, *lint*, etc. — as well. For all not too complicated configurations, an *amake* specification along the lines of the example in figure 6.19 would then be sufficient.

```
%generic gencluster(tar, src) {
     %default $tar;
     %cluster
     {
         %targets $tar;
         %sources $src;
         %use     tool1(flags => $MYFLAGS1),
                  tooln(flags => $MYFLAGSn);
     };
};
```

Fig. 6.18. Example use of generics (2)

```
SRCDIR=/home/joe/src/prog;        # source directory

%include  toolset.amk;            # shared tool set
%include  mygenerics.amk;         # gencluster etc.
%include  $SRCDIR/Amake.srclist;  # source list

%instance gencluster(prog1, $SRC_PROG1);
%instance gencluster(prog2, $SRC_PROG2);
```

Fig. 6.19. Example use of generics (3)

6.3.14. Conditional

There is also a top-level conditional available, comparable to the ‘‘#if’’-construct in the C-preprocessor. It has the almost same syntax as the conditional expression (although it is debatable if that is really an advantage):

conditional `::=` **%if** `"("` *expression* `","` *def-list* `[","` *def-list*`]? ")"`
def-list `::=` `"{"` *amake-def-list* `"}"`

The condition may be an arbitrary boolean expression. Note that, in contrast to conditional expressions, the ‘‘else’’ part is optional. This is equivalent to defining the else part as being empty. Also note that the ‘‘then’’ and ‘‘else’’ part are arbitrary sequences of *amake* definitions, so nested conditionals are allowed.

A possible application is to assign default values for variables used in tool definitions. As the variables might well be defined before the tool itself is included, the default assignment in the tool library has to be shielded off. Examples:

```
%if (%not defined(CFLAGS), {
    CFLAGS = { ’-O’ };
});
```

```
%if (%not defined(TOOLSET), {
    exit('fatal error: TOOLSET not defined');
});
```

6.3.15. Import and Export

Amake also has two constructs dealing with the string environment each process has. Importing an *amake* variable is done by retrieving the corresponding value from *amake*'s environment, thereby converting it to an *amake* value. Exporting an variable is the opposite action, which causes processes started up by *amake* to have a corresponding value in their environment. The syntax is:

> *import* `::=` **%import** *identifier* *string*?
> *export* `::=` **%export** *identifier* *string*?

Because there is no system-enforced list element separator in the environment — not under UNIX, at least — the constructs have an optional string argument specifying it (each variable has its own list separator). The default separator is `':'`, because that is the one used for the various PATH variables interpreted by the Bourne shell and utilities. Example:

```
%import PATH;
%import HOME;
CFLAGS = { '-DAMOEBA', '-DACK' };
PATH = $PATH + $HOME/amakebin;
%export PATH;
%export CFLAGS ' ';
```

Note that the variable CFLAGS has been export with the space as separator. Otherwise it would have resulted in the value ‘‘`-DAMOEBA:-DACK`’’.

6.3.16. Concluding Remarks

For information about the actual invocation of *amake* (options, notes about the implementation, etc.), refer to the manual page.

6.4. BAWK — Basic AWK

AWK is a programming language devised by Aho, Weinberger, and Kernighan at Bell Labs (hence the name). *Bawk* is a basic subset of it written by Bob Brodt. *Bawk* programs search files for specific patterns and performs ‘‘actions’’ for every occurrence of these patterns. The patterns can be ‘‘regular expressions’’ as used in the *ed* editor. The actions are expressed using a subset of the C language.

The patterns and actions are usually placed in a ‘‘rules’’ file whose name must be the first argument in the command line, preceded by the flag **–f**. Otherwise, the first argument on the command line is taken to be a string containing the rules themselves. All other arguments are taken to be the names of text files on which the rules are to be applied, with **–** being the standard input. To take rules from the standard input, use **–f –**.

The command:

```
bawk rules prog.*
```

would read the patterns and actions rules from the file *rules* and apply them to all the arguments.

The general format of a rules file is:

```
<pattern> { <action> }
<pattern> { <action> }
```

There may be any number of these `<pattern> { <action> }` sequences in the rules file. *Bawk* reads a line of input from the current input file and applies every `<pattern> { <action> }` in sequence to the line.

If the `<pattern>` corresponding to any `{ <action> }` is missing, the action is applied to every line of input. The default `{ <action> }` is to print the matched input line.

6.4.1. Patterns

A `<pattern>` may consist of any valid C expression. If the `<pattern>` consists of two expressions separated by a comma, it is taken to be a range and the `<action>` is performed on all lines of input that match the range. A `<pattern>` may contain ‘‘regular expressions’’ delimited by an @ symbol. Regular expressions can be thought of as a generalized ‘‘wildcard’’ string matching mechanism, similar to that used by many operating systems to specify file names. Regular expressions may contain any of the following characters:

x	An ordinary character.
\	The backslash quotes any character.
ˆ	A circumflex at the beginning of an expr matches the beginning of a line.
$	A dollar-sign at the end of an expression matches the end of a line.
:x	A colon matches a class of characters described by the next character.
:a	‘‘:a’’ matches any alphabetic.
:d	‘‘:d’’ matches digits.
:n	‘‘:n’’ matches alphanumerics.

:	‘‘: ’’ matches spaces, tabs, and other control characters, such as newline.
*	An expression followed by an asterisk matches zero or more occurrences of that expression: ‘‘fo*’’ matches ‘‘f’’, ‘‘fo’’, ‘‘foo’’, ‘‘fooo’’, etc.
+	An expression followed by a plus sign matches one or more occurrences of that expression: ‘‘fo+’’ matches ‘‘fo’’, ‘‘foo’’, ‘‘fooo’’, etc.
−	An expression followed by a minus sign optionally matches the expression.
[]	A string enclosed in square brackets matches any single character in that string, but no others. If the first character in the string is a circumflex, the expression matches any character except newline and the characters in the string. For example, ‘‘[xyz]’’ matches ‘‘xx’’ and ‘‘zyx’’, while ‘‘[ˆxyz]’’ matches ‘‘abc’’ but not ‘‘axb’’. A range of characters may be specified by two characters separated by ‘‘−’’.

6.4.2. Actions

Actions are expressed as a subset of the C language. All variables are global and default to int’s if not formally declared. Only char’s and int’s and pointers and arrays of char and int are allowed. *Bawk* allows only decimal integer constants to be used—no hex (0xnn) or octal (0nn). String and character constants may contain all of the special C escapes (\n, \r, etc.).

Bawk supports the ‘‘if’’, ‘‘else’’, ‘‘while’’ and ‘‘break’’ flow of control constructs, which behave exactly as in C.

Also supported are the following unary and binary operators, listed in order from highest to lowest precedence:

Operator	**Type**	**Associativity**
() []	unary	left to right
! ++ — – * &	unary	right to left
* / %	binary	left to right
+ –	binary	left to right
<< >>	binary	left to right
< <= > >=	binary	left to right
== !=	binary	left to right
&	binary	left to right
ˆ	binary	left to right
|	binary	left to right
&&	binary	left to right
||	binary	left to right
=	binary	right to left

Comments are introduced by a ’#’ symbol and are terminated by the first newline character. The standard ‘‘/*’’ and ‘‘*/’’ comment delimiters are not supported and will result in a syntax error.

6.4.3. Fields

When *bawk* reads a line from the current input file, the record is automatically separated into ‘‘fields.’’ A field is simply a string of consecutive characters delimited by either the beginning or end of line, or a ‘‘field separator’’ character. Initially, the field separators are the space and tab character. The special unary operator ’$’ is used to reference one of the fields in the current input record (line). The fields are numbered sequentially starting at 1. The expression ‘‘$0’’ references the entire input line.

Similarly, the ‘‘record separator’’ is used to determine the end of an input ‘‘line,’’ initially the newline character. The field and record separators may be changed programmatically by one of the actions and will remain in effect until changed again.

Multiple (up to 10) field separators are allowed at a time, but only one record separator. In either case, they must be changed by *strcpy*, not by a simple equate as in the real AWK.

Fields behave exactly like strings; and can be used in the same context as a character array. These ‘‘arrays’’ can be considered to have been declared as:

```
char ($n)[ 128 ];
```

In other words, they are 128 bytes long. Notice that the parentheses are necessary because the operators [] and $ associate from right to left; without them, the statement would have parsed as:

```
char $(1[ 128 ]);
```

which is obviously ridiculous.

If the contents of one of these field arrays is altered, the ‘‘$0’’ field will reflect this change. For example, this expression:

```
*$4 = ’A’;
```

will change the first character of the fourth field to an upper- case letter ’A’. Then, when the following input line:

```
120 PRINT "Name         address       Zip"
```

is processed, it would be printed as:

```
120 PRINT "Name         Address       Zip"
```

Fields may also be modified with the *strcpy* function (see below). For example, the expression:

```
strcpy( $4, "Addr." );
```

applied to the same line above would yield:

```
120 PRINT "Name         Addr.       Zip"
```

6.4.4. Predefined Variables

The following variables are pre-defined:

FS	Field separator (see below).
RS	Record separator (see below also).
NF	Number of fields in current input record (line).
NR	Number of records processed thus far.
FILENAME	Name of current input file.
BEGIN	A special <pattern> that matches the beginning of input text.
END	A special <pattern> that matches the end of input text.

Bawk also provides some useful built-in functions for string manipulation and printing:

print(arg)	Simple printing of strings only, terminated by '\n'.
printf(arg...)	Exactly the printf() function from C.
getline()	Reads the next record and returns 0 on end of file.
nextfile()	Closes the current input file and begins processing the next file
strlen(s)	Returns the length of its string argument.
strcpy(s,t)	Copies the string ‘‘t’’ to the string ‘‘s’’.
strcmp(s,t)	Compares the ‘‘s’’ to ‘‘t’’ and returns 0 if they match.
toupper(c)	Returns its character argument converted to upper-case.
tolower(c)	Returns its character argument converted to lower-case.
match(s,@re@)	Compares the string ‘‘s’’ to the regular expression ‘‘re’’ and returns the number of matches found (zero if none).

6.4.5. Limitations

The maximum input line is 128 characters. The maximum action is 4K.

6.5. Debugging Tools

Amoeba has several aids for debugging and performance tuning of programs.

The GNU debugger *gdb* has been ported to Amoeba. Special support for multi-threaded programs has been added. The debugger is available with the third-party software.

For tracing server operations there is also special monitoring software. This is available through including the file *monitor.h* in the program sources and making calls to various macros where various ‘‘events’’ take place. The software automatically collects statistics about how often each event has occurred and also provides the possibility of logging the order in which events occurred from a particular point in time. A user utility (see *monitor*(U)) allows you to recover the event information from the server.

The monitoring system is described below.

6.5.1. Monitoring

It is possible to monitor various ‘‘events’’ in a server program. This includes Amoeba kernels since they have internal servers. This is currently only supported in the C programming language. There is a special include file, called *monitor.h*, which implements the mechanism for storing and retrieving the monitor information. A special macro can be placed in the code which acts as an ‘‘event marker’’. The macro keeps a count of the number of times it has been executed. The statistics kept by the monitoring can be collected using the command *monitor*(U). This command allows the possibility of keeping a circular buffer of events in the order in which they occur in addition to the count of the number of times an event has occurred. This option tends to use a lot of memory and shouldn’t be left on too long in important servers.

The macro that implements event marking is called `MON_EVENT`. The syntax of the call is:

```
MON_EVENT("description of an event")
```

Execution of this call generates a primitive event. A string describing the event is passed as an argument. A variable denoting the number of times this specific event has occurred is incremented. Memory within the server is used to buffer the event information.

The three RPC calls are automatically monitored. There are events for each call to *trans*, *getreq* and *putrep*. In addition special events are kept for any RPC errors.

The server does not need to implement a special command to allow clients to request the monitoring statistics. The include file redefines *getreq* call to implement the monitor command without the server seeing it.

At start up time the monitoring system allocates memory to store the event information using *malloc*(L). Some times this is impossible or undesirable since it may modify the behavior of the system being monitored in unacceptable ways. It is possible to pass the monitoring an event buffer using the `MON_SETBUF` macro. The syntax of the call is as follows.

```
MON_SETBUF(buffer, size)
```

A pointer to the private buffer and its size should be passed as arguments.

Note that it is not a good idea to add `MON_EVENT` calls to timing critical code. Although inexpensive there is some overhead involved.

7 Manual Pages

This chapter contains the library manual pages for C programming under Amoeba. They are listed in alphabetical order by entry name and classified into one of two categories. Either (L) for library routine or (H) for include file. At the beginning of each manual entry is a list of the libraries in which the routine(s) described can be found.

(in libraries: libajax.a, libamoeba.a)

Name

ansi_C – details of STD C library conformance

Synopsis

```
Support for STD C library functions and header files is discussed.
```

Description

Several STD C conformant compilers have been ported to Amoeba. The principle compilers are the ACK and the GNU compilers. Details of the ACK compiler conformance are provided in the languages section of the *Programming Guide*.

Amoeba provides its own C library rather than using those of a specific compiler. This library is the subject of the rest of this manual entry.

The various STD C functions are found in either *libamoeba.a*, *libajax.a* and *libmath.a*. Specific deviations from the C Standard are listed below where applicable. Many header files define additional names that are disallowed by STD C. The library also contains many reserved names that should not be overridden by applications (e.g., *trans*). STD C requires that only the names defined by STD C are reserved, but this requires that every Amoeba function name has a leading underscore. This has not been fully implemented yet, so be careful.

All supported header files are found in the include subdirectory *posix*. Since the POSIX standard defines extensions to many STD C headers, separating STD C headers from POSIX headers would be impossible.

Only functions and header files with peculiarities are mentioned. The rest are adequately described by the ISO C standard.

Notes regarding Amoeba's STD C header files and libraries

<errno.h>

Currently declares the integer variable *errno*, the constants `EDOM` and `ERANGE` and many constants required by POSIX. Warning: use of *errno* by concurrent threads is not safe; to fix this, it will eventually change into a macro that evaluates to a modifiable lvalue. For future compatibility, applications should not write

```
extern int errno;
```

but rather include *errno.h*.

<limits.h>

Defines all the constants required by STD C, as well as some required by POSIX.

<math.h>

The header file is STD C compatible, but in addition to the standard requirements, the library *libmath.a* contains the following functions: *gamma(), hypot(), j0(), j1(), jn().* The functions *frexp()* and *ldexp()* reside in *libamoeba.a*; they are included there rather than in *libmath.a* because a few standard I/O conversion routines need them.

<setjmp.h>

Apart from the standard requirements, this header file also defines the type *sigjmp_buf* and the functions *sigsetjmp()* and *siglongjmp()* required by POSIX. Note that part of these definitions are machine-dependent and are placed in the file *posix/machdep/*architecture*/_setjmp.h*. The directory where this file resides should be passed as a separate **−I** flag to the C compiler.

<stdio.h>

Apart from the standard requirements, the POSIX function *fdopen()* and function/macro *fileno()* are supported.

<string.h>

Warning: the function *strtok()* uses a static global variable; its use by concurrent threads is not safe.

<time.h>

The type *time_t* is in fact an integral type and measures seconds since January 1, 1970 (defined more precisely by POSIX). Additional information about these functions is found in *ctime*(L). Warning: many of these functions use global variables; their use by concurrent threads is not safe. In particular, *asctime(), ctime(), gmtime(), localtime()* return pointers to static storage that may be overwritten by other threads; most also share a hidden cache of timezone information that may be corrupted by concurrent accesses.

See Also

posix(L).

Name

ampolicy.h − system parameterization defines

Synopsis

```
#include "ampolicy.h"
```

Description

This include file contains system parameterization information which may be modified by the system administrator on a system-wide basis. In particular it contains path names for capabilities and data for certain servers and implies a lot about the structure of the file system. It comes set up to work with the directory structure defined in the command *newsoapgraph*(A).

Administration

Before compiling the system it is necessary to make all the desired alterations to this file. It determines to a large extent the directory structure required by the various programs. Make sure that any changes made to this file that fundamentally alter the structure of the directory graph are reflected in the shell scripts for building the directory graph, namely *newsoapgraph*(A) and *newuser*(A).

See Also

newsoapgraph(A), newuser(A).

(in libraries: libamoeba.a, libamunix.a)

Name

ar – ASCII representations of common Amoeba types

Synopsis

```
#include "amoeba.h"
#include "module/ar.h"

char * ar_cap(cap_p)
char * ar_port(port_p)
char * ar_priv(priv_p)

char * ar_tocap(s, cap_p)
char * ar_toport(s, port_p)
char * ar_topriv(s, priv_p)
```

Description

To aid in debugging a set of routines has been provided to produce human-readable representations of capabilities and various subparts thereof.

The first three functions return a pointer to a string containing the ASCII representation of the bytes in a *capability*, *port* and *private* (see *rpc*(L)), respectively. Note that the string is in static data and subsequent calls by other threads or the same thread will overwrite the contents of the string. Each routine has its own static data so there are no interactions between the individual routines.

The latter three functions convert a string of the format returned by the first three routines to a capability, port and private, respectively. The *ar_to* functions assign to the capability, port or private, pointed to by the second parameter. They return a pointer to the character beyond the last character of the string. If the string is supposed to contain no extra characters, you should check that the returned pointer points to a NUL character.

If the string passed to an *ar_to* function is illegal, NULL is returned.

The format used is explained below.

Functions

ar_port

```
        char *
        ar_port(port_p)
        port *port_p;
```

Ar_port returns a pointer to a string with the six bytes in the port represented as *x:x:x:x:x:x*, where *x* is the value of a byte in hexadecimal.

ar_priv

```
char *
ar_priv(p)
private *p;
```

Ar_priv uses the format *D(X)/x:x:x:x:x:x* , where *D* is the object number in decimal, *X* is the rights in hexadecimal, and *x* is a byte from the check field, in hexadecimal.

ar_cap

```
char *
ar_cap(cap_p)
capability *cap_p;
```

The format used by *ar_cap* is the concatenation of the formats of *ar_port* and *ar_priv* respectively, separated by a slash. That is, *x:x:x:x:x:x/D(X)/x:x:x:x:x:x* .

ar_tocap

```
char *
ar_tocap(s, cap_p)
char *s;
capability *cap_p;
```

Ar_tocap takes a string *s* in the format produced by *ar_cap* and makes a capability that matches that representation in the capability pointed to by *cap_p*. It returns a pointer to the first character after the parsing of *s* stopped.

ar_toport

```
char *
ar_toport(s, port_p)
char *s;
port *port_p;
```

Ar_toport takes a string *s* in the format produced by *ar_port* and makes a port that matches that representation in the capability pointed to by *port_p*. It returns a pointer to the first character after the parsing of *s* stopped.

ar_topriv

```
char *
ar_topriv(s, priv_p)
char *s;
private *priv_p;
```

Ar_topriv takes a string *s* in the format produced by *ar_priv* and makes a private part of a capability (i.e., object number, rights and check field) that matches that representation in the capability pointed to by *priv_p*. It returns a pointer to the first character after the parsing of *s* stopped.

Examples

The following code prints out an array of capabilities:

```
#include "amoeba.h"
#include "module/ar.n"

printcaps(caps, n)
    capability caps[];
    int n;
{
    int i;

    for (i = 0; i < n; ++i)
         printf("Cap #%d: %s\n", i, ar_cap(&caps[i]));
}
```

The following function prints ‘‘hello world’’ and stores the port consisting of the bytes 1, 2, 3, 4, 5, 6 in *p*.

```
Hello()
{
    port p;

    printf("%s\n", ar_toport("1:2:3:4:5:6hello world", &p);
}
```

See Also

c2a(U), rpc(L).

(in libraries: libamoeba.a, libamunix.a)

Name

bprintf – like sprintf but with bounds checking

Synopsis

```
#include "module/strmisc.h"

char * bprintf(begin, end, fmt, ...)
char * begin;
char * end;
char * fmt;
```

Description

Bprintf is a function similar to *sprintf* in the *STD C stdio* library. It takes the format string *fmt* and copies it to the buffer specified by *begin* and *end*, doing any conversions of extra arguments according to the rules of *printf*. However it does bounds checking so that it does not overrun the end of the buffer. It returns a pointer to the next free position in the buffer. This is useful for doing multiple *bprintf*'s to the same buffer. If it overruns the buffer it returns the NULL-pointer. If *begin* is the NULL-pointer and *end* is non-NULL then it will immediately return the NULL-pointer without modifying the buffer. This means that one can put many *bprintf* requests after another for the same buffer, using the result of the previous *bprintf* as the *begin* pointer for the next and never need to check for an error until after the last one. For example,

```
#define BSZ 30
char buf[BSZ];
char * end = buf + BSZ;
char * p:
int i = 20;

p = bprintf(buf, end, "foo %d", i);
p = bprintf(p, end, " test");
p = bprintf(p, end, "\n");
if (p == (char *) 0)
    ERROR("buffer overflow");
/* else all is well ... */
```

If the *end* pointer is the NULL-pointer then *bprintf* functions as though it was *printf*. Note that *bprintf* will NULL-terminate the resultant string only if it has not yet reached the *end* of the buffer. Thus, if the buffer is too small to hold all the data then it will not be NULL-terminated.

See Also

buffer(L).

(in libraries: libamoeba.a, libamunix.a)

Name

buffer – getting and putting data in architecture-independent format

Synopsis

```
#include "amoeba.h"
#include "module/buffers.h"

char *buf_get_cap(p, endp, &val)
char *buf_get_capset(p, endp, &val)
char *buf_get_int16(p, endp, &val)
char *buf_get_int32(p, endp, &val)
char *buf_get_objnum(p, endp, &val)
char *buf_get_pd(p, endp, &val)
char *buf_get_port(p, endp, &val)
char *buf_get_priv(p, endp, &val)
char *buf_get_right_bits(p, endp, &val)
char *buf_get_string(p, endp, &val)
char *buf_put_cap(p, endp, &val)
char *buf_put_capset(p, endp, &val)
char *buf_put_int16(p, endp, val)
char *buf_put_int32(p, endp, val)
char *buf_put_objnum(p, endp, val)
char *buf_put_pd(p, endp, &val)
char *buf_put_port(p, endp, &val)
char *buf_put_priv(p, endp, &val)
char *buf_put_right_bits(p, endp, val)
char *buf_put_string(p, endp, val)
```

Description

These functions get data from, or put data into, a character buffer in a format that is independent of any machine architecture; in particular of byte-order dependencies. They are primarily useful in loading and unloading RPC buffers (and reply buffers). All of them have the same calling sequence and return value, except for the type of the last argument.

In each case, *p* should point to the place within the buffer where data is to be stored or retrieved, and *endp* should point to the end of the buffer. (That is, the first byte after the buffer.)

For the *buf_get* functions, the *val* argument should be the address where the data retrieved from the buffer is to be stored. It must be a pointer to the type of value expected by the function. (See the individual function descriptions, below.)

For the *buf_put* functions, there are two possibilities: if the value to be put in the buffer is an integral type, or a character pointer, the value itself should be passed as the *val* argument. For larger, structured types, a pointer to the value should be supplied. (See the individual function descriptions, below.)

On success, the functions return a pointer to the next byte in the buffer following the value that was retrieved or inserted. Thus, in normal usage, a simple sequence of calls to *buf_get* functions or to *buf_put* functions can be used to get or put a sequence of values, in order. All that is necessary is to provide the output of a previous call as the first argument of the next call (see the example, below.)

The end pointer is used to detect whether there is sufficient space in the buffer to get or put the specified value. If not, NULL is returned to indicate failure. To avoid the necessity of an error check after each of a sequence of calls to *buf_get* or *buf_put* functions, this overflow error propagates: if a NULL pointer is passed as the *p* argument to any of the functions, it returns NULL.

Note that these routines guarantee not to require more than *sizeof(datatype)* bytes to store the data in the buffer. If similar routines are needed to marshal a *union* then the union should be encapsulated within a *struct* with an extra field which specifies the field of the *union* which is really being sent.

Functions

buf_get_cap

```
char *
buf_get_cap(p, endp, valp)
char *p, *endp;
capability *valp;
```

A capability is retrieved from the buffer pointed to by *p* and copied to the capability pointed to by *valp*.

buf_get_capset

```
#include "capset.h"

char *
buf_get_capset(p, endp, valp)
char *p, *endp;
capset *valp;
```

A capability-set is retrieved from the buffer pointed to by *p* and copied to the capability-set pointed to by *valp*. The suite for the capability-set is allocated by this function. The suite must not contain more than `MAXCAPSET` entries.

buf_get_int16

```
char *
buf_get_int16(p, endp, valp)
char *p, *endp;
int16 *valp;
```

An architecture-independent 16-bit integer is retrieved from the buffer pointed to by *p* and copied to the 16-bit integer pointed to by *valp*, in the form required by the local host architecture.

buf_get_int32

```
char *
buf_get_int32(p, endp, valp)
char *p, *endp;
int32 *valp;
```

An architecture-independent 32-bit integer is retrieved from the buffer pointed to by *p* and copied to the 32-bit integer pointed to by *valp*, in the form required by the local host architecture.

buf_get_objnum

```
char *
buf_get_objnum(p, endp, valp)
char *p, *endp;
objnum *valp;
```

An object number (as found in capabilities) is retrieved from the buffer pointed to by *p* and copied to the object number pointed to by *valp*.

buf_get_pd

```
char *
buf_get_pd(p, endp, valp)
char *p, *endp;
process_d *valp;
```

A process descriptor stored in an architecture-independent format is retrieved from the buffer pointed to by *p* and stored in the process descriptor structure pointed to by *valp*. All necessary allocation of sub-structures is performed by this function. See *process_d*(L) for more details.

buf_get_port

```
char *
buf_get_port(p, endp, valp)
char *p, *endp;
port *valp;
```

A port is retrieved from the buffer pointed to by *p* and copied to the port pointed to by *valp*.

buf_get_priv

```
char *
buf_get_priv(p, endp, valp)
char *p, *endp;
private *valp;
```

The private part of a capability (object number, rights and check field) is retrieved from the buffer pointed to by *p* and copied to the place pointed to by *valp*.

buf_get_right_bits

```
char *
buf_get_right_bits(p, endp, valp)
char *p, *endp;
rights_bits *valp;
```

The rights bits of a capability are retrieved from the buffer pointed to by *p* and copied to the place pointed to by *valp*.

buf_get_string

```
char *
buf_get_string(p, endp, valp)
char *p, *endp;
char **valp;
```

The char pointer specified by *valp* is modified to point to the character string beginning at location *p* in the buffer, and *p* is advanced to just beyond the NULL-byte that terminates the string. If no such NULL-byte is found, however, NULL is returned and *valp* is left undefined.

WARNING: This behavior is often not what is wanted, especially when the transaction buffer is to be re-used. Most of the time, it will be necessary to copy the characters of the string out of the buffer to a safer place. This is not done by *buf_get_string* nor is any storage allocated for the string, beyond the transaction buffer itself.

buf_put_cap

```
char *
buf_put_cap(p, endp, valp)
char *p, *endp;
capability *valp;
```

The capability pointed to by *valp* is copied into the buffer pointed to by *p*.

buf_put_capset

```
#include "capset.h"

char *
buf_put_capset(p, endp, valp)
char *p, *endp;
capset *valp;
```

The capability-set pointed to by *valp* (including the suite sub-structure) is copied into the buffer pointed to by *p*. The suite must not contain more than `MAXCAPSET` entries.

buf_put_int16

```
char *
buf_put_int16(p, endp, val)
char *p, *endp;
int16 val;
```

Val is stored in the buffer pointed to by *p*.

buf_put_int32

```
char *
buf_put_int32(p, endp, val)
char *p, *endp;
int32 val;
```

Val is stored in the buffer pointed to by *p*.

buf_put_objnum

```
char *
buf_put_objnum(p, endp, val)
char *p, *endp;
objnum val;
```

Val, which should be an object number as found in a capability, is stored in the buffer pointed to by *p*.

buf_put_pd

```
char *
buf_put_pd(p, endp, valp)
char *p, *endp;
process_d *valp;
```

The process descriptor pointed to by *valp* is copied into the buffer pointed to by *p*. See *process_d*(L) for more details.

buf_put_port

```
char *
buf_put_port(p, endp, valp)
char *p, *endp;
port *valp;
```

The port pointed to by *valp* is copied into the buffer pointed to by *p*.

buf_put_priv

```
char *
buf_put_priv(p, endp, valp)
char *p, *endp;
private *valp;
```

The private part of a capability (object number, rights and check field), pointed to by *valp*, is copied into the buffer pointed to by *p*.

buf_right_bits

```
char *
buf_put_right_bits(p, endp, val)
char *p, *endp;
rights_bits val;
```

Val, which should be the rights bit of a capability, is copied into the buffer pointed to by *p*.

buf_put_string

```
char *
buf_put_string(p, endp, valp)
char *p, *endp;
char *valp;
```

The string pointed to by *valp* is copied into the buffer pointed to by *p*, including the NULL-byte terminator.

Example

The following code copies a capability, *c*, a 32-bit integer, *i*, a character string, *s*, into a transaction buffer, then retrieves them into the variables *c2*, *i2*, and *s2*.

```
capability c, c2;
int32 i, i2;
char *s, *s2;
char buffer[1000];
char *p, *ep;

name_lookup(path, &c);
i = 17;
s = "foobar";

p = buffer;
ep = buffer + sizeof buffer;
p = buf_put_cap(p, ep, &c);
p = buf_put_int32(p, ep, i);
p = buf_put_string(p, ep, s);
if (!p) {
        error("Buffer overflow during put");
}

p = buffer;
ep = buffer + sizeof buffer;
p = buf_get_cap(p, ep, &c2);
p = buf_get_int32(p, ep, &i2);
p = buf_get_string(p, ep, &s2);
if (!p) {
        error("Buffer underflow during get");
}
```

(in libraries: libamoeba, libamunix.a)

Name

buildstack − build a stack segment for a process to be executed

Synopsis

```
#include "amoeba.h"
#include "module/proc.h"

unsigned long buildstack(buf, buflen, start, argv, envp, caps)
char *buf;
long buflen;
unsigned long start;
char **argv;
char **envp;
struct caplist *caps;
```

Description

Buildstack is a low-level utility function used by *exec_file*(L). It fills the buffer with a stack segment for a process to be executed. (‘Stack’ segment is a slight misnomer − the information filled in is better described as environment, but it happens to be placed above the stack in memory, and the rest of the segment is used as the stack for the main thread.)

Buildstack places the arguments, the string and capability environments and pointers to them in the buffer, starting from the high end, all as expected by the C run-time start-off, *head*(L). The argument *start* indicates the address where *buf* will be mapped in the new process’s virtual memory. See *head*(L) for a description of the lay-out.

The return value is the stack pointer to be used when starting the process.

Diagnostics

A NULL-pointer is returned if the buffer was too short.

See Also

exec_file(L), head(L).

(in libraries: libamoeba.a, libkernel.a, libamunix.a)

Name

bullet – the Bullet Server client interface stubs

Synopsis

```
#include "amoeba.h"
#include "cmdreg.h"
#include "stderr.h"
#include "server/bullet/bullet.h"

errstat b_disk_compact(supercap)
errstat b_fsck(supercap)
errstat b_sync(supercap)

errstat b_create(cap, buf, size, commit, newcap)
errstat b_delete(filecap, offset, size, commit, newcap)
errstat b_insert(filecap, offset, buf, size, commit, newcap)
errstat b_modify(filecap, offset, buf, size, commit, newcap)
errstat b_read(filecap, offset, buf, to_read, were_read)
errstat b_size(filecap, size)
```

Description

Along with the standard server stubs (see *std*(L)) the Bullet Server stubs provide the programming interface to the Bullet Server. They are divided here according to whether they are administrative or user functions. Bullet files can be created incrementally but do not exist for reading until they have been committed. The commit operation is atomic.

Types

The file *bullet.h* defines the type *b_fsize* which is the size of a bullet file in bytes. It is implemented as a suitably sized integer.

Access

Access to files and administrative functions is determined on the basis of rights in the capability given with each command.

There is a special capability known as the *super capability* which is primarily for use by the file system administrator. The *super capability* is used for checking the file system, enquiring the status of the file system and for initiating garbage collection. It cannot be used to alter or destroy files except for those found to be defective by the file system consistency checker and those which are garbage collected due to lack of use.

The following rights are defined in Bullet Server capabilities:

`BS_RGT_CREATE`	permission to create a file
`BS_RGT_READ`	permission to read, copy or get status of a file.
`BS_RGT_MODIFY`	permission to alter an uncommitted file.
`BS_RGT_DESTROY`	permission to destroy a file.
`BS_RGT_ADMIN`	permission to execute administrative activities
`BS_RGT_ALL`	permission to do anything

The last two are only of significance in the super capability.

Note that the only operations permitted on uncommitted files are *std_destroy*, *b_modify*, *b_insert* and *b_delete*. All other operations on uncommitted files will return the status `STD_CAPBAD`. Furthermore, any uncommitted file that remains unmodified for more than `BS_CACHE_TIMEOUT` minutes will be destroyed. NB. *std_touch* on an uncommitted file is not legal and would not prevent it from timing out even if it was legal. The only way to avoid timing out an uncommitted file is to make a null-modification to it.

Errors

All functions return the error status of the operation. The user interface to the Bullet Server returns only standard error codes (as defined in *stderr.h*). All the stubs involve transactions and so in addition to the errors described below, all stubs may return any of the standard RPC error codes (see *rpc*(L)). A valid capability is required for all operations and so all operations will return `STD_CAPBAD` if an invalid capability is used. For operations requiring rights, if insufficient rights are present in the capability then `STD_DENIED` will be returned. If illegal parameters such as NULL-pointers or negative buffer sizes are given, exceptions may occur rather than an error status being returned. The behavior is not well defined under these conditions.

Administrative Functions

b_disk_compact

```
errstat
b_disk_compact(supercap)
capability *supercap;
```

B_disk_compact goes through the entire virtual disk of the Bullet Server and moves all the files down so that there is no disk fragmentation. This is a very slow process and should only be attempted when there is no load on the system. The Bullet Server does respond to other requests while it is doing this but very slowly. The capability given as the parameter must have all rights for the command to succeed.

Required Rights:
    `BS_RGT_ALL` in supercap

Error Conditions:
    `STD_NOTNOW`:   the server is already doing a compaction.

b_fsck

```
errstat
b_fsck(supercap)
capability *supercap;
```

B_fsck does a check of the Bullet Server file system. Error messages appear on the console of the Bullet Server. The function returns a failure status of some sort if any faults are detected. Otherwise it returns `STD_OK`.

Required Rights:
    `BS_RGT_ALL` in supercap

b_sync

```
errstat
b_sync(supercap)
capability *supercap;
```

B_sync ensures that all committed files are written to disk. The function returns a failure status only if the capability is not the super capability for the server, if something goes wrong with the RPC or if there is a disk error. Otherwise it returns `STD_OK`.

Required Rights:
    NONE

User Functions

b_size

```
errstat
b_size(filecap, size)
capability *filecap;
b_fsize *size;
```

If there are no errors, *b_size* returns in *size* the number of bytes in the file specified by *filecap*. Otherwise *size* is unaltered.

Required Rights:
    `BS_RGT_READ` in filecap

b_read

```
errstat
b_read(filecap, offset, buf, to_read, were_read)
capability *filecap;
b_fsize offset;
char *buf;
b_fsize to_read;
b_fsize *were_read;
```

B_read returns in *buf* up to *to_read* bytes of the contents of the file specified by *filecap* starting at *offset* bytes from the beginning of the file. It will return less than *to_read* bytes if end of file is encountered before *to_read* bytes can be read. The parameter *to_read* may be zero. If there were no errors it returns in **were_read* the number of bytes returned in *buf*.

If an error is detected the value of **were_read* will not be altered from its original value.

Required Rights:
`BS_RGT_READ` in filecap

Error Conditions:

`STD_ARGBAD`:	offset > size of file
	offset < 0
	to_read < 0
`STD_NOMEM`:	buffer cache was full

In the remaining functions the type of *commit* is implemented as an int, but it contains two flags:

`BS_COMMIT`:	If set the file should be committed.
`BS_SAFETY`:	If set then wait till the file is written to disk.
	Otherwise return immediately.

If the `BS_COMMIT` flag is set in *commit* then the current contents of the file is the entire file and the file is created. If the `BS_SAFETY` flag is also set in *commit* then the function will wait until the file is written to disk before returning. Otherwise it will return immediately. (Note that `BS_SAFETY` is ignored if `BS_COMMIT` is not set.) If the `BS_COMMIT` flag is not set further alterations to the file are expected using either *b_modify*, *b_delete* or *b_insert* (see below). To avoid system abuse, if the `BS_COMMIT` flag is not set and if no further modification command is received within `BS_TIMEOUT` sweeps after the previous modification then the capability will be invalidated. (See *bullet*(A) for an explanation of sweeping.)

Note that when calculating file offsets the first byte in a file is numbered 0.

b_create

```
errstat
b_create(cap, buf, size, commit, newcap)
capability *cap;
char *buf;
b_fsize size;
int commit;
capability *newcap;
```

B_create requests the Bullet Server whose capability is *cap* to create a new file whose initial contents is the *size* bytes in *buf*. NB: The final size of the file is not specified in advance.

If *cap* is a valid capability for a committed file on the server in use then when the new file is committed the Bullet Server will compare the new file with the extant file. If they are the same it will discard the new file and return the capability for the extant file. Otherwise it will create the new file and return a new capability for it. Note that the capability for an existing file must have the read right for comparison to take place.

B_create returns the capability for the new file in *newcap*.

Required Rights:
    `BS_RGT_CREATE` in cap
    `BS_RGT_READ` in cap if it is for comparison

Error Conditions:
    `STD_NOSPACE`: resource shortage: inode, memory, disk space

b_modify

```
errstat
b_modify(filecap, offset, buf, size, commit, newcap)
capability *filecap;
b_fsize offset;
char *buf;
b_fsize size;
int commit;
capability *newcap;
```

If the file specified by *filecap* has been committed, *b_modify* makes an uncommitted copy of it. If the file was uncommitted it is used directly. The file is overwritten with the *size* bytes pointed to by *buf*, beginning at *offset* bytes from the beginning of the file. If *offset* + *size* is greater than the file size then the file will become larger. Note that *offset* cannot be larger than the current file size.

To commit a file that is already in the Bullet Server without adding anything further to it, simply do a *b_modify* with size 0. If the BS_COMMIT flag is not set it will restart the timeout. The capability for the new file is returned in *newcap*.

Note Well: This operation is *not* atomic unless the amount of data sent to the file server was less than or equal to `BS_REQBUFSZ` (as defined in *bullet.h*). If more data than this was to be sent and a failure status is returned it is possible that some part of the operation succeeded. The resultant state of the file is not able to be determined.

Required Rights:
`BS_RGT_MODIFY` and `BS_RGT_READ` in filecap

Error Conditions:

`STD_ARGBAD`:	offset < 0
	offset > file size
	size < 0
`STD_NOSPACE`:	resource shortage: inode, memory, disk space

b_insert

```
errstat
b_insert(filecap, offset, buf, size, commit, newcap)
capability *filecap;
b_fsize offset;
char *buf;
b_fsize size;
int commit;
capability *newcap;
```

If the file specified by *filecap* has been committed, *b_insert* makes an uncommitted copy of it. If the file was uncommitted it is used directly. *B_insert* inserts the *size* bytes in *buf* immediately before the position *offset* bytes from the beginning of the file. The file size will increase by *size* bytes.

The capability for the file is returned in *newcap*.

Note Well: This operation is *not* atomic unless the amount of data sent to the file server was less than or equal to `BS_REQBUFSZ` (as defined in *bullet.h*). If more data than this was to be sent and a failure status is returned it is possible that some part of the operation succeeded. The resultant state of the file is not able to be determined.

Required Rights:
`BS_RGT_MODIFY` and `BS_RGT_READ` in filecap

Error Conditions:

`STD_ARGBAD`:	size <= 0
	offset < 0
	offset > file size
`STD_NOSPACE`:	resource shortage: inode, memory, disk space

b_delete

```
errstat
b_delete(filecap, offset, size, commit, newcap)
capability *filecap;
b_fsize offset;
b_fsize size;
int commit;
capability *newcap;
```

If the file specified by *filecap* has been committed, *b_delete* makes an uncommitted copy of it. If the file was uncommitted it is used directly. *b_delete* deletes *size* bytes beginning at *offset* bytes from the beginning of the file. If *offset* + *size* is greater than the file size then it deletes to the end of the file and stops. Thus the size of the file decreases by up to *size* bytes.

The capability for the file is returned in *newcap*.

Required Rights:

`BS_RGT_MODIFY` and `BS_RGT_READ` in filecap

Error Conditions:

`STD_ARGBAD`:	size <= 0
	offset < 0
	offset > file size
`STD_NOSPACE`:	resource shortage: inode, memory, disk space

See Also

bfsck(A), bstatus(A), bsync(A), bullet(A), fromb(U), rpc(L), std(L), std_age(A), std_copy(U), std_destroy(U), std_info(U), std_restrict(U), std_status(U), std_touch(U), tob(U).

(in libraries: libamoeba.a, libkernel.a, libamunix.a)

Name

capcache – capability cache management routines

Synopsis

```
#include <module/cap.h>

int cc_init(n_entries)
int cc_restrict(cap, mask, new, restrict)

int cap_cmp(cap1, cap2)
```

Description

The capability cache routines are used to manage a least recently used (LRU) cache of known capabilities. The cache is dynamically allocated by the routine *cc_init* which takes as its parameter the number of entries the cache should have. Entries are looked up in the cache using the routine *cc_restrict*. If an entry is not found in the cache it is added. The idea is that instead of repeatedly performing `STD_RESTRICT` commands (see *std*(L)), the restricted versions of a capability are kept in the cache. The cache is protected from concurrent updates using mutexes (see *mutex*(L)).

The routine *cap_cmp* simply determines if two capabilities are identical.

cc_init

```
    int
    cc_init(n_entries)
    int n_entries;
```

This routine allocates sufficient memory for a capability cache with *n_entries* slots and initializes the cache to empty. It is only possible to have one capability cache per process. It returns 0 if the memory allocation failed. It returns 1 if the cache was successfully allocated or if a cache has already been allocated (even if it is a different size from the one requested!).

cc_restrict

```
    int
    cc_restrict(cap, mask, new, restrict)
    capability *cap;
    rights_bits mask;
    capability *new;
    int (*restrict)();
```

This routine attempts to look up the capability *cap* in the capability cache and returns in *new* the same capability but with the rights restricted to those set in *mask*.

If the required restricted capability is not in the cache then it is added. The function *restrict* is used to restrict the rights to create the new capability. (This is typically *std_restrict* — see *std*(L).) If the cache becomes full then entries are discarded from the cache on an LRU basis.

cap_cmp

```
int
cap_cmp(cap1, cap2)
capability *cap1, *cap2;
```

This routine compares the capabilities pointed to by *cap1* and *cap2*. If they are identical in all respects it returns 1. Otherwise it returns 0.

See Also

capset(L).

(in libraries: libamoeba.a, libkernel.a, libamunix.a)

Name

capset – routines for manipulating capability-sets

Synopsis

```
#include "amoeba.h"
#include "capset.h"

int cs_copy(new, cs)
void cs_free(cs)
errstat cs_goodcap(cs, cap)
int cs_member(cs, cap)
errstat cs_purge(cs)
int cs_singleton(cs, cap)
errstat cs_to_cap(cs, cap)
```

Description

Capability-sets are used by the Soap Server to hold several capabilities for a single name. This permits the multiple capabilities for a replicated object to be stored under a single name.

Types

The type *capset* is defined in *capset.h*. The definition of a capset is as follows.

```
typedef struct {
        short cs_initial;
        short cs_final;
        suite *cs_suite;
} capset;
```

and a *cs_suite* is a pointer to an array of the following structure.

```
typedef struct {
        capability s_object;
        short s_current;
} suite;
```

The *suite* array may have been allocated with *malloc*(L) so it is important to use the routines provided to manipulate them so as to avoid memory leaks. The structure member *s_current* in the *suite* tells whether the capability in *s_object* is valid. Invalid capabilities are not thrown out of the *suite*, but simply marked invalid so that their slot may be reused later. This saves lots of fooling around with *malloc*ed data.

The structure member *cs_final* in the *capset* holds the size of the array of *suite* structs. The structure member *cs_initial* must be greater than or equal to 0 and less than or equal to *cs_final*. It is not otherwise used and is obsolete.

cs_copy

```
int
cs_copy(new, cs)
capset *new, *cs;
```

Cs_copy copies the capability-set *cs* to the capability-set *new*. Any suites needed are allocated with *calloc* (see *malloc*(L)) and added to the structure. Note that **new* is not allocated, but must already exist. It returns 1 if the copy was successful and zero if it failed (due to the failure of *calloc*).

cs_free

```
void
cs_free(cs)
capset *cs;
```

Cs_free frees any memory allocated for suites and marks *cs* as containing no capabilities. It is not an error to call this routine for a capset that has already been freed.

cs_member

```
int
cs_member(cs, cap)
capset *cs;
capability *cap;
```

Cs_member returns 1 if the capability *cap* is a member of the capability-set *cs*. Otherwise it returns 0.

cs_purge

```
errstat
cs_purge(cs)
capset *cs;
```

cs_purge attempts to destroy all the objects whose capabilities are in the capability-set *cs*. If the destroy command for a capability succeeds it is marked as no longer being a valid member of the capability-set. If any of the destroy operations fail then it returns the error status of the first failure but continues to try to destroy the other objects. If the value of the function is not `STD_OK` the only way to see what was not destroyed is to look at *cs* and see which capabilities are still valid.

cs_singleton

```
int
cs_singleton(cs, cap)
capset *cs;
capability *cap;
```

Cs_singleton allocates a *suite* structure and stores the capability *cap* in it. It then takes the capability-set *cs*, which it assumes to be empty, and makes it contain the single *suite*. It returns 0 if it failed and 1 if it succeeds.

cs_goodcap

```
errstat
cs_goodcap(cs, cap)
capset *cs;
capability *cap;
```

Cs_goodcap attempts to get a usable capability from a capset, copying it to **cap*. If the capset is empty, it returns `STD_SYSERR`. If there are no caps for which *std_info* returns `STD_OK`, it copies the last capability in the set and returns the error status from *std_info* with it. The *goodport*(L) module is used to avoid attempting a transaction with a port that has previously returned `RPC_NOTFOUND.`

cs_to_cap

```
errstat
cs_to_cap(cs, cap)
capset *cs;
capability *cap;
```

Cs_to_cap gets a capability from the capset pointed to by *cs* and copies it to the capability pointed to by *cap*. It is functionally the same as *cs_goodcap*, above, except that when the capset contains only one capability, it copies that capability without checking whether it is good. Also, it always returns `STD_OK`, unless the capset is empty.

See Also

buffer(L), capcache(L), goodport(L).

(in libraries: libamoeba.a)

Name

circbuf – circular buffer - serialized, buffered data transfer between concurrent threads

Synopsis

```
#include "amoeba.h"
#include "semaphore.h"
#include "circbuf.h"

struct circbuf *cb_alloc(size)
void cb_close(cb)
void cb_free(cb)

int cb_putc(cb, c)
int cb_puts(cb, s, n)

int cb_getc(cb)
int cb_trygetc(cb, maxtimeout)

int cb_gets(cb, s, minlen, maxlen)

int cb_putp(cb, ps, blocking)
void cb_putpdone(cb, len)
int cb_getp(cb, ps, blocking)
void cb_getpdone(cb, len)

int cb_full(cb)
int cb_empty(cb)
void cb_setsema(cb, sema)
```

Description

A *circular buffer* provides serialized, buffered data transfer between concurrent threads. Threads within a process can read from a circular buffer, while other threads in the same process can write to it. The circular buffer module provides a set of routines to manage the circular buffers.

Data transfer through circular buffers is similar to transferring data through a pipe. Threads write data into the pipe, other threads read data from the pipe. A close on a circular buffer is similar to closing the input side of a pipe: it is no longer allowed to put new data into the circular buffer but the circular buffer can still be emptied.

Data transfer through a circular buffer can be accomplished by the routines *cb_getc*, *cb_trygetc* and *cb_gets* to read data, and *cb_putc* and *cb_puts* to write data. Each function copies data respectively from the circular buffer to a local buffer or from a local buffer to the circular buffer.

The routines *cb_putp*, *cb_putpdone*, *cb_getp* and *cb_getpdone* are routines meant for directly accessing the circular buffer data. The routines *cb_putp* and *cb_getp* return a pointer to the correct place where data can be put in/fetched from the circular buffer. Once data is moved to/from the circular buffer, it must be marked as done by the routines *cb_putpdone* and *cb_getpdone*. The main difference with stubs like *cb_puts* and *cb_gets* is the absence of an extra copy to a local buffer (internally, *cb_gets* and *cb_puts* use *cb_getp/cb_getpdone* and *cb_putp/cb_putpdone*). This entitles the stubs to be called the optimized stubs.

The following example shows the use of normal stubs and the use of the optimized stubs:

```
char buf[100];   /* Allocate a transaction buffer */
int bs;

/* Get some data in buf */
bs = trans(&some_header, 0, 0, &some_header, buf, 100);
if (ERR_STATUS(bs)) {
   ... etc ...
}
/* Put it in the circular buffer */
if (cb_puts(some_cb, buf, bs) != 0) {
   ... etc ...
```

Using the optimized stubs:

```
char *buf;
int bs, bs2;

/* Get a circular buffer data pointer */
bs = cb_putp(some_cb, &buf, 1);
/* Get some data in the circular buffer */
bs2 = trans(&some_header, 0, 0, &some_header, buf, bs);
if (ERR_STATUS(bs2)) {
   ... etc ...
}
/* Mark the data copied */
cb_putpdone(some_cb, bs2);
```

In the latter case, *rpc*(L) stores the received data directly in the circular buffer.

Functions

cb_alloc

```
struct circbuf *
cb_alloc(size)
int size;
```

Cb_alloc allocates a circular buffer of *size* bytes and returns a circular buffer reference pointer. If the memory could not be allocated it returns the NULL-pointer.

cb_close

```
void
cb_close(cb)
struct circbuf *cb;
```

Cb_close closes a circular buffer. *Cb_close* does not flush the circular buffer. When closed, threads are no longer allowed to write data to the circular buffer. The circular buffer can only be emptied.

cb_free

```
void
cb_free(cb)
struct circbuf *cb;
```

Cb_free destroys the circular buffer. All data in the circular buffer is lost. *Cb_free* may not be called while threads are still using the circular buffer; it will cause threads to hang or crash.

cb_putc, cb_puts

```
int
cb_putc(cb, c)
struct circbuf *cb;
int c;

int cb_puts(cb, s, n)
struct circbuf *cb;
char *s;
int n;
```

Cb_putc and *cb_puts* write data to the circular buffer. *Cb_putc* writes the character *c* to the circular buffer. *Cb_puts* writes *n* characters from the character array *s* to the circular buffer. *Cb_puts* is not an atomic operation (except when writing one character). Two concurrent *cb_puts* calls to the same circular buffer might result in mixed data. Use *cb_putp* and *cb_putpdone* for atomic writes to circular buffers. It is possible, however, to make sure that *cb_puts* is atomic: if all accesses to the buffer are through *cb_puts* and *cb_gets*, all these accesses put or get objects of the same size and the size of the circular buffer is a multiple of this size.

Cb_putc returns −1 when the circular buffer is closed, zero upon successful writes. *Cb_puts* returns 0 on success, −1 when the circular buffer is closed. Note that *cb_puts* will also return −1 if the buffer is closed during the *cb_puts* call, so some bytes may have been written.

cb_getc, cb_trygetc

```
int
cb_getc(cb)
struct circbuf *cb;

int
cb_trygetc(cb, maxtimeout)
struct circbuf *cb;
int maxtimeout;
```

Cb_getc and *cb_trygetc* read a byte from the circular buffer. They both return the character read. *Cb_trygetc* completes when it has read a character or when no character has become available after waiting for *maxtimeout* milliseconds.

Cb_getc and *cb_trygetc* return −1 if the circular buffer is closed and there are no input bytes available. *Cb_trygetc* returns 1 when the timer has expired.

Note: The routine *cb_trygetc* behaves on signals exactly as *mu_trylock* (see *signal*(L), *mutex*(L)). If the thread doing circular buffer actions has defined a signal catcher, the signal will be caught. Otherwise the signal will be ignored. On a caught signal, *cb_trygetc* will return a −1 value. When *maxtimeout* equals −1, *cb_trygetc* will be blocked until a byte arrives. That is, there is no maximum timeout. In this case, *cb_trygetc* is still interruptible by a signal.

cb_gets

```
int
cb_gets(cb, s, minlen, maxlen)
struct circbuf *cb;
char *s;
int minlen, maxlen;
```

Cb_gets reads at least *minlen* and at most *maxlen* bytes from the circular buffer into the character array *s*. *Cb_gets* will block when the number of bytes available is less than the minimum requested (available < minlen). *Cb_gets* returns the number of bytes read.

Cb_gets returns zero when the circular buffer is closed.

cb_putp, cb_putpdone

```
int
cb_putp(cb, ps, blocking)
struct circbuf *cb;
char **ps;
int blocking;
```

```
void
cb_putpdone(cb, len)
struct circbuf *cb;
int len;
```

Cb_putp returns in **ps* a pointer to the available free bytes in the circular buffer. The number of available free bytes is passed back as the function value. Data can be copied directly into the circular buffer by copying it to **ps*.

When *blocking* is zero, *cb_putp* does not block. It returns the number of free bytes in the circular buffer (including zero). When *blocking* is negative, *cb_putp* waits for available free bytes, but is interruptible by a signal (see *cb_trygetc*). When signaled and there are no bytes available, *cb_putp* returns zero. When *blocking* is positive, *cb_putp* waits for available free bytes and is not interruptible by a signal.

It is possible *cb_putp* returns less bytes than the amount actually free when the free buffer space crosses the circular buffer boundary (ie. the size of the circular buffer equals 10 bytes, bytes 4 and 5 are in use, *cb_putp* will return 5 instead of 8).

A *cb_putp* call that returned a value larger than zero must be followed by a *cb_putpdone* call informing how many bytes are used up in the circular buffer (*len* bytes).

Diagnostics

cb_putp returns −1 on closed circular buffers, otherwise it returns the amount of free bytes available. *Cb_putpdone* returns no status.

Note: A sequence *cb_putp()* − ‘‘store data’’ − *cb_close()* − *cb_putpdone()*, is allowed but the stored data is flushed.

cb_getp, cb_getpdone

```
int
cb_getp(cb, ps, blocking)
struct circbuf *cb;
char **ps;
int blocking;

void
cb_getpdone(cb, len)
struct circbuf *cb;
int len;
```

Cb_getp and *cb_getpdone* are the complement of *cb_putp* and *cb_putpdone*. *Cb_getp* returns in **ps* a pointer to the available data bytes. The number of available data bytes is passed back as function value. Data can be copied directly copied from the circular buffer via **ps*.

When *blocking* is zero, *cb_getp* does not block. It returns the number of data bytes immediately available (including zero). When *blocking* is negative, *cb_getp* waits for available data bytes, but is interruptible by a signal (see *cb_trygetc*). When signaled and there are no data bytes, *cb_getp* returns a zero value. When *blocking* is positive, *cb_getp* waits for available data bytes and is not interruptible.

It is possible that *cb_getp* returns less data bytes than actually available, in this case the data space crosses the circular buffer boundary (see *cb_putp*).

A *cb_getp* call that returned a value larger than zero must be followed by a *cb_getpdone* call, informing the number of bytes used up (*len* bytes).

Cb_getp returns the number of available data bytes (zero when no bytes are available). *Cb_getpdone* returns no status.

cb_full, cb_empty

```
int
cb_full(cb)
struct circbuf *cb;

int
cb_empty(cb)
struct circbuf *cb;
```

Cb_full returns the number of available data bytes in a circular buffer. *Cb_empty* returns the number of available free bytes in a circular buffer.

Cb_full returns a −1 when there are no available data bytes and the circular buffer is closed. *Cb_empty* returns −1 when the circular buffer is closed.

cb_setsema

```
void
cb_setsema(cb, sema)
struct circbuf *cb;
semaphore *sema;
```

Cb_setsema attaches an external semaphore to a circular buffer. Each time a data byte arrives, the semaphore *sema* is *upped* by *sema_up* (see *semaphore*(L)). When the circular buffer is closed, the semaphore is *upped* as well. The semaphore is initialized to the number of available data bytes.

Example

The following demonstrates the fundamentals of using the circular buffer routines in a multi-threaded environment.

```
static circbuf *cb;

main(argc, argv)
char **argv;
{
   int num_bytes, n;
   char s[10];

   /* Create the circular buffer */
   cb = cb_alloc(20);
   if (cb == 0) {
      fprintf(stderr,
                "%s: Cannot allocate circular buffer.\n",
               argv[0]);
      exit(1);
   }

   /* Spawn NR_THREADS threads which write to the circular
      buffer
   */
   for (n = 0; n != NR_THREADS; n++)
      if (thread_newthread(thread, STACK_SIZE, 0, 0) == 0) {
          fprintf(stderr, "Cannot spawn threads.0);
          exit(1);
      }

   /* while the circular buffer is not closed */
   while (cb_empty(cb) != -1) {
      /* get a string from the circular buffer, min size 5,
          max size 10
      */
      num_bytes = cb_gets(cb, s, 5, 10);
      do_something(s, num_bytes);
   }
   /* The circular buffer is closed, remove it */
   cb_free(cb);
}
```

```
thread()
{
   char s[10];
   int size;

   /* Get some data */
   while (size = buf_get_string(s, 10))
      /* Write the data to the circular buffer */
      if (cb_puts(cb, s, size) != 0)
         break;

   /* Once all is written, close the circular buffer */
   cb_close(cb);

   /* stop thread */
   thread_exit();
}
```

See Also

rpc(L), semaphore(L), thread(L).

(in libraries: libamoeba)

Name

ctime − convert date and time to ASCII

Synopsis

```
extern char *tzname[2];

void tzset()

#include <sys/types.h>

char *ctime(clock)
time_t *clock;

double difftime(time1, time0)
time_t time1;
time_t time0;

#include <time.h>

char *asctime(tm)
struct tm *tm;

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

time_t mktime(tm)
struct tm *tm;
```

Description

Tzset uses the value of the environment variable `TZ` to set time conversion information used by *localtime*. If `TZ` does not appear in the environment, the best available approximation to local wall clock time, as specified by the *tzfile*-format (see *zic*(A)) file **localtime** in the system time conversion information directory, is used by *localtime*. If `TZ` appears in the environment but its value is a NULL-string, Coordinated Universal Time (UTC) is used (without leap second correction). If `TZ` appears in the environment and its value is not a NULL-string:

if the value begins with a colon, it is used as a path name of a file from which to read the time conversion information;

if the value does not begin with a colon, it is first used as the path name of a file from which to read the time conversion information, and, if that file cannot be read, is used directly as a specification of the time conversion information.

When TZ is used as a path name, if it begins with a slash, it is used as an absolute path name; otherwise, it is used as a path name relative to a system time conversion information directory. The file must be in the format specified in *zic*(A).

When TZ is used directly as a specification of the time conversion information, it must have the following syntax (spaces inserted for clarity):

std offset[*dst*[*offset*][**,***rule*]]

Where:

std and dst — Three or more bytes that are the designation for the standard (*std*) or summer (*dst*) timezone. Only *std* is required; if *dst* is missing, then summer time does not apply in this locale. Upper- and lowercase letters are explicitly allowed. Any characters except a leading colon digits, comma minus plus and ASCII NUL are allowed.

offset — Indicates the value one must add to the local time to arrive at Coordinated Universal Time. The *offset* has the form:

hh[**:***mm*[**:***ss*]]

The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required and may be a single digit. The *offset* following *std* is required. If no *offset* follows *dst*, summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour must be between zero and 24, and the minutes (and seconds) — if present — between zero and 59. If preceded by a the timezone shall be east of the Prime Meridian; otherwise it shall be west (which may be indicated by an optional preceding Indicates when to change to and back from summer time. The *rule* has the form:

date/*time***,***date*/*time*

where the first *date* describes when the change from standard to summer time occurs and the second *date* describes when the change back happens. Each *time* field describes when, in current local time, the change to the other time is made.

The format of *date* is one of the following:

J*n* — The Julian day *n* ($1 \le n \le 365$). Leap days are not counted; that is, in all years — including leap years — February 28 is day 59 and March 1 is day 60. It is impossible to explicitly refer to the occasional February 29.

n — The zero-based Julian day ($0 \le n \le 365$). Leap days are counted, and it is possible to refer to February 29.

M*m***.***n***.***d* — The *d*'th day ($0 \le d \le 6$) of week *n* of month *m* of the year ($1 \le n \le 5$, $1 \le m \le 12$, where week 5 means ‘‘the last *d* day in month *m*’’ which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*'th day occurs. Day zero is Sunday.

The *time* has the same format as *offset* except that no leading sign or is allowed. The default, if *time* is not given, is **02:00:00**.

If no *rule* is present in TZ, the rules specified by the *tzfile*-format file **posixrules** in the system time conversion information directory are used, with the standard and summer time offsets from UTC replaced by those specified by the *offset* values in TZ.

For compatibility with System V Release 3.1, a semicolon may be used to separate the *rule* from the rest of the specification.

If the TZ environment variable does not specify a *tzfile*-format and cannot be interpreted as a direct specification, UTC is used with the standard time abbreviation set to the value of the TZ environment variable (or to the leading characters of its value if it is lengthy).

Ctime converts a long integer, pointed to by *clock*, representing the time in seconds since 00:00:00 UTC, January 1, 1970, and returns a pointer to a 26-character string of the form

```
Thu Nov 24 18:22:48 1986\n\0
```

All the fields have constant width.

Localtime and *gmtime* return pointers to ‘‘tm’’ structures, described below. *Localtime* corrects for the timezone and any timezone adjustments (such as Daylight Saving Time in the U.S.A.). Before doing so, *localtime* calls *tzset* (if *tzset* has not been called in the current process). After filling in the ‘‘tm’’ structure, *localtime* sets the **tm_isdst**'th element of **tzname** to a pointer to an ASCII string that is the timezone abbreviation to be used with *localtime*'s return value.

Gmtime converts to Coordinated Universal Time.

Asctime converts a time value contained in a ‘‘tm’’ structure to a 26-character string, as shown in the above example, and returns a pointer to the string.

Mktime converts the broken-down time, expressed as local time, in the structure pointed to by *tm* into a calendar time value with the same encoding as that of the values returned by the *time* function. The original values of the **tm_wday** and **tm_yday** components of the structure are ignored, and the original values of the other components are not restricted to their normal ranges. (A positive or zero value for **tm_isdst** causes *mktime* to presume initially that summer time (for example, Daylight Saving Time in the U.S.A.) respectively, is or is not in effect for the specified time. A negative value for **tm_isdst** causes the *mktime* function to attempt to divine whether summer time is in effect for the specified time.) On successful completion, the values of the **tm_wday** and **tm_yday** components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to their normal ranges; the final value of **tm_mday** is not set until **tm_mon** and **tm_year** are determined. *Mktime* returns the specified calendar time; If the calendar time cannot be represented, it returns **-1**.

Difftime returns the difference between two calendar times, *time1* - *time0,* expressed in seconds.

Declarations of all the functions and externals, and the ‘‘tm’’ structure, are in the **<time.h>** header file. The structure (of type) **struct tm** includes the following fields:

```
int tm_sec;       /* seconds (0 - 60) */
int tm_min;       /* minutes (0 - 59) */
int tm_hour;      /* hours (0 - 23) */
int tm_mday;      /* day of month (1 - 31) */
int tm_mon;       /* month of year (0 - 11) */
int tm_year;      /* year − 1900 */
int tm_wday;      /* day of week (Sunday = 0) */
int tm_yday;      /* day of year (0 - 365) */
int tm_isdst;     /* is summer time in effect? */
```

Tm_isdst is non-zero if summer time is in effect.

Tm_gmtoff is the offset (in seconds) of the time represented from UTC, with positive values indicating east of the Prime Meridian.

FILES

/profile/module/time/zoneinfo	timezone information directory
/profile/module/time/zoneinfo/localtime	local timezone file
/profile/module/time/zoneinfo/posixrules	used with POSIX-style TZ’s
/profile/module/time/zoneinfo/GMT	for UTC leap seconds

If */profile/module/time/zoneinfo/GMT* is absent, UTC leap seconds are loaded from */profile/module/time/zoneinfo/posixrules*.

Warnings

The return values point to static data whose content is overwritten by each call. The **tm_zone** field of a returned **struct tm** points to a static array of characters, which will also be overwritten at the next call (and by calls to *tzset*).

Avoid using out-of-range values with *mktime* when setting up lunch with promptness sticklers in Riyadh.

Example

The following program prints the current local time:

```
#include <stdio.h>
#include <time.h>
main()
{
    time_t t;
    time(&t);
    printf("%s", ctime(&t));
    exit(0);
}
```

See Also

posix(L), ansi_C(L), date(U), tod(L), zic(A).

(in libraries: libamoeba.a, libkernel.a, libamunix.a)

Name

dgwalk – routines for examining a directory graph

Synopsis

```
#include "amoeba.h"
#include "module/dgwalk.h"

errstat dgwalk(params)
errstat dgwexpand(path, proc)
```

Description

Dgwalk can be used to find all directory and directory entries that are reachable from one or more starting points. For each directory found, *dgwalk* calls a user-supplied function with as parameters one or more capabilities for the directory and an indication of the path names under which these capabilities can be found. *Dgwexpand* can then be used to unravel each directory and call a second, user-supplied routine for each entry found. *Dgwexpand* does not recognize the concept of links between entries in directories. Each entry is treated as a separate entity.

Dgwalk can function in two modes: *ad_hoc* and *all*. The basic difference between the two modes stems from the fact that the Amoeba name server implements a directed graph. The consequence is that a certain directory might have multiple paths into it with differing rights. In *all* mode *dgwalk* first tries to obtain all available information on all reachable directories and then calls a user-supplied function for every directory found. In *ad-hoc* mode, the default, *dgwalk* calls the user-supplied function as soon as each directory is found and expects a return value from that function indicating whether that directory has been satisfactorily dealt with or not. If the directory has been satisfactorily dealt with, *dgwalk* will not call the user-supplied function again for that directory, even if a path with more rights is found. If the directory has not been satisfactorily dealt with, *dgwalk* will call the user-supplied function again whenever the directory capability is met again. For directories not met again *dgwalk* will call a third user-supplied function, thereby indicating that *dgwalk* expects that this is the last time a user-supplied function is called for that entry. *Dgwalk* will keep calling the user-supplied functions for each directory not yet satisfactorily dealt with.

It is possible to construct graphs in which certain parts of the graph might not be found in ad-hoc mode.

dgwalk

```
    errstat
    dgwalk(params)
    dgw_params params;
```

The parameters to *dgwalk* are passed in a structure. This structure has the following definition:

```
      typedef struct {
            short  mode;
            dgw_paths *entry;
            capset **servers;
            int    (*dodir)();
            int    (*testdir)();
            int    (*doagain)();
      } dgw_params;
```

The fields of the parameter structure have the following meaning:

mode	Zero for *ad-hoc* mode and `DGW_ALL` for *all* mode.
entry	The set of entry points. This is a pointer to a list of structures, each describing a separate entry point. The use of the *dgw_paths* structure is described below.
servers	The set of directory servers to which the search for directories is to be restricted. If it is `NULL` no restriction applies. Otherwise *dgwalk* assumes that this field points to a NULL-terminated array of pointers to capability-sets. If this list is empty *dgwalk* will only call the user-supplied function for the entry points.
testdir	This routine is called in *all* mode whenever a new directory is examined during the determination of the graph. The return value determines whether this directory should be part of the graph `(DGW_OK)`, should not be examined further `(DGW_STOP)` or should be further examined whenever seen again `(DGW_AGAIN)`. When `DGW_QUIT` is returned *dgwalk* will not call any further user-supplied functions and return `STD_INTR`. The only parameter to these function calls is a pointer to a sequence of one or more paths as described below. This entry may be `NULL`.
dodir	This routine is called whenever a new directory is met in *ad-hoc* mode and for each directory found in *all* mode after determination of the graph. In *ad-hoc* mode the return values have the same meaning as described under *testdir*. The only recognized return value in *all* mode is `DGW_QUIT.` The only parameter to these function calls is a pointer to a sequence of one or more paths as described below.
doagain	This function is only called in *ad-hoc* mode when *dgwalk* has exhausted its own list of directories to examine. It is then called for all directories for which `DGW_OK` or `DGW_STOP` has not yet been returned by calls to the function specified in the *dodir* entry. In all other aspects it is functionally equal to *dodir*.

The functions passed to *dgwalk* all have one parameter: a list of paths. Its type is: `(dgw_paths *)`. The type *dgw_paths* has the following definition:

```
typedef struct {
        char     *entry;
        dgw_paths *dotdot;
        capabilitycap;
        dgw_paths*next;
        /* Internal */
        ....
} dgw_paths ;
```

The first three fields of the structure specify a capability for the directory and the way in which it was found. The capability *cap* for this directory was found in the parent directory specified by *dotdot* under *entry*. The entry *next* specifies one further way in which this directory was found. *Dgwalk* guarantees that no capability is present in this list with rights that are a subset of the rights of another capability in the list.

dgwexpand

```
errstat
dgwexpand(path, proc)
dgw_paths path;
void (*proc)();
```

This function can be called to examine the contents of a directory. The directory to be examined is specified by the *path* parameter that could be passed to *dgwexpand* by one of the parameter functions of *dgwalk*. *Dgwexpand* simply uses the first path in the list to get the contents of the directory and calls *proc* for each entry found. The function specified by *proc* is called with four parameters:

```
visit_entry(path, cset, parent_cset, entry)
        char     *path;
        capset   *cset;
        capset   *parent_cset;
        char     *entry;
```

Cset is the capability-set found under *entry* in the directory specified by *parent_cset*. *Path* gives the string with the full path name that can be used to access the entry.

Diagnostics

Both *dgwalk* and *dgwexpand* can return any of the Amoeba error messages. If they return `STD_NOMEM` there is a fair chance that they ran out of memory, but it might also indicate that one of the servers involved ran out of memory.

See Also

om(A).

(in libraries: libamoeba.a, libamunix.a)

Name

direct − capability-based functions for searching in and modifying directories

Synopsis

```
#include "module/direct.h"

errstat dir_append(origin, name, object)
char *dir_breakpath(origin, name, dir)
errstat dir_create(origin, name)
errstat dir_delete(origin, name)
errstat dir_lookup(origin, name, object)
capability *dir_origin(name)
errstat dir_replace(origin, name, object)
int dir_root(origin)

struct dir_open *dir_open(dir)
char *dir_next(dp)
errstat dir_close(dp)
errstat dir_rewind(dp)
```

Description

This module provides a portable interface to directory servers. It has the advantage of being independent of the particular directory server in use, but it does not make available all the functionality of every directory server, e.g., the Soap Server. It is similar to the module containing the same set of functions but with the prefix ‘‘name_’’ instead of ‘‘dir_’’. The main difference is that these functions take an extra argument, *origin*, which is the capability for a directory relative to which the given *name* is to be interpreted (see below).

In each function, the *name* should be a ‘‘path name’’ that specifies a directory or directory entry. To these functions, a directory is simply a finite mapping from character-string names to capabilities. Each (name, capability) pair in the mapping is a ‘‘directory entry’’. The capability can be for any object, including another directory; this allows arbitrary directory graphs (the graphs are not required to be trees). A capability can be entered in multiple directories or several times (under different names) in the same directory, resulting in multiple links to the same object. Note that some directory servers may have more complex notions of a directory, but all that is necessary in order to access them from this module is that they satisfy the above rules.

A path name is a string of printable characters. It consists of a sequence of 0 or more ‘‘components’’, separated by ‘‘/’’ characters. Each component is a string of printable characters not containing ‘‘/’’. As a special case, the path name may begin with a ‘‘/’’, in which case the first component starts with the next character of the path name. Examples: ‘‘a/silly/path’’, ‘‘/profile/group/cosmo-33’’.

If the path name begins with a ‘‘/’’, it is an ‘‘absolute’’ path name, otherwise it is a ‘‘relative’’ path name. The interpretation of relative path names is relative to the directory specified by the *origin* capability.

In detail, the interpretation of a path name relative to a directory *d* is as follows:

(1) If the path has no components (is a zero-length string), it specifies the directory *d*;

(2) Otherwise, the first component in the path name specifies the name of an entry in directory *d* (either an existing name, or one that is to be added to the mapping);

(3) If there are any subsequent components in the path name, the first component must map to the capability of a directory, in which case the rest of the components are recursively interpreted as a path name relative to that directory.

The interpretation of absolute path names is the same, except that the portion of the path name after the ‘‘/’’ is interpreted relative to the user’s ‘‘root’’ directory, rather than to the current working directory of the process. (Each user is given a capability for a single directory from which all other directories and objects are reached. This is called the user’s root directory.)

The components ‘‘.’’ and ‘‘..’’ have special meaning. Paths containing occurrences of these components are syntactically evaluated to a normal form not containing any such occurrences before any directory operations take place. See *path_norm*(L) for the meaning of these special components.

Errors

Most functions return the error status of the operation. Most of them perform transactions, so in addition to the errors described below, they may return any of the standard RPC error codes.

`STD_OK`:	the operation succeeded.
`STD_CAPBAD`:	an invalid capability was used.
`STD_DENIED`:	one of the capabilities encountered while parsing the path name had insufficient access rights (for example, the directory in which the capability is to be installed is unwritable).
`STD_NOTFOUND`:	one of the components encountered while parsing the path name does not exist in the directory specified by the preceding components (the last component need not exist for some of the functions, but the other components must refer to existing directory entries).

Functions

dir_append

```
errstat
dir_append(origin, name, object)
capability *origin;
char *name;
capability *object;
```

Dir_append adds the *object* capability to a directory server under *name* (interpreted relative to *origin*). It can subsequently be retrieved by giving the same *origin* and *name* to *dir_lookup*. The path up to, but not including, the last component of *name* must refer to an existing directory. This directory is modified by adding an entry consisting of the last component of *name* and the *object* capability. There must be no existing entry with the same name.

Required Rights:
    `SP_MODRGT` in the directory that is to be modified

Error Conditions:
    `STD_EXISTS`:   *name* already exists

dir_breakpath

```
char *
dir_breakpath(origin, name, dir)
capability *origin;
char *name;
capability *dir;
```

Dir_breakpath stores in *dir* the capability for the directory allegedly containing the object specified by *name* (interpreted relative to *origin*). It returns the simple name of the entry under which the object is stored, or would be stored if it existed. It is intended for locating the directory that must be modified to install an object in a directory server under the given *name*. In detail, *dir_breakpath* does the following:

If the *name* is a path name containing only one component, *dir_breakpath* stores the capability for either the *origin* (if the path name is relative) or the user's root directory (if the path name is absolute) in *dir*, and returns the *name* (without any leading ''/'').

Otherwise, the *name* is parsed into two path names, the first consisting of all but the last component and the second a relative path name consisting of just the last component. *Dir_breakpath* stores the capability for the directory specified by the first in *dir* and returns the second. The first path name must refer to an existing directory.

dir_close

```
errstat
dir_close(dp)
struct dir_open *dp;
```

Dir_close is called after reading a directory with *dir_open* and *dir_next*. It frees up the storage associated with the open directory specified by *dp*, which must be a value returned by a previous call to *dir_open*. The directory is now ''closed'' and any subsequent attempt to use *dp* will produce undefined behavior.

dir_create

```
errstat
dir_create(origin, name)
capability *origin;
char *name;
```

Dir_create creates a directory and stores its capability in a directory server under *name* (interpreted relative to *origin*). If *name* is a relative path name, the new directory is created using the server containing the *origin* directory, otherwise the new directory is created using the server containing the user's root directory. (Note that either of these might be different from the server that contains the directory which is to be modified by adding the new directory.)

The path up to, but not including the last component of *name* must refer to an existing directory. This directory is modified by adding an entry consisting of a new directory, named by the last component of *name*. There must be no existing entry with the same name.

Required Rights:
    `SP_MODRGT` in the directory that is to be modified

Error Conditions:
    `STD_EXISTS`: *name* already exists in the directory server

dir_delete

```
errstat
dir_delete(origin, name)
capability *origin;
char *name;
```

Dir_delete deletes the entry specified by *name* (interpreted relative to *origin*) from a directory server. The object specified by the capability associated with *name* in the directory entry is not affected. If desired, it should be separately destroyed (see *std_destroy*(L)).

Required Rights:
    `SP_MODRGT` in the directory that is to be modified

dir_lookup

```
errstat
dir_lookup(origin, name, object)
capability *origin;
char *name;
capability *object;
```

Dir_lookup finds the capability named by *name* (interpreted relative to *origin*) and stores it in *object*.

dir_next

```
char *
dir_next(dp)
struct dir_open *dp;
```

Dir_next returns the next name from the open directory specified by *dp*, which must be a value returned by a call to *dir_open*. At the end of the directory, 0 is returned. The return value points to static data and is overwritten by each call. The user is responsible for ensuring that multiple threads do not simultaneously read from the same open directory.

dir_open

```
struct dir_open *
dir_open(dir)
capability *dir;
```

Dir_open is called to ‘‘open’’ the directory specified by *dir*, in preparation for listing it. It returns a pointer to a temporary data structure that keeps track of a current position in the open directory. The names entered in the directory can subsequently be retrieved by calling *dir_next* with the value returned by *dir_open.* After listing the directory it should be closed with *dir_close*. Multiple directories can be open at the same time.

dir_origin

```
capability *
dir_origin(name)
char *name
```

If *name* is an absolute path name, *dir_origin* returns the capability for the user’s root directory. Otherwise it returns the capability for the current working directory which it obtains from the capability environment. This function is used to obtain an initial value for the *origin* parameter for the other functions in this module.

dir_replace

```
errstat
dir_replace(origin, name, object)
capability *origin;
char *name;
capability *object;
```

Dir_replace replaces the current capability stored under *name* (interpreted relative to *origin*) with the specified *object* capability. The *name* must refer to an existing directory entry. This directory entry is changed to refer to the specified *object* capability as an atomic action. The entry is not first deleted then appended.

Required Rights:
    `SP_MODRGT` in the directory that is to be modified

dir_rewind

```
    errstat
    dir_rewind(dp)
    struct dir_open *dp;
```

This routine resets the open directory pointed to by *dp* to the first entry in the directory.

dir_root

```
    errstat
    dir_root(origin)
    capability *origin;
```

Dir_root stores the capability for the user's root directory in *origin*. It returns 0 on success, -1 on failure.

Error Conditions:

`−1`:	cannot find the `ROOT` capability in the environment

Examples

```
capability rootcap, modify_dir, object;
char *old_name = "/home/abc";
/* Change the name /home/abc to /home/xyz: */
if (dir_root(&rootcap) == 0) {
    char *entry_name =
        dir_breakpath(&rootcap, old_name, &modify_dir);
    if (entry_name &&
        dir_lookup(&modify_dir, entry_name, &object) == STD_OK &&
        dir_delete(&modify_dir, entry_name) == STD_OK) {
        if (dir_append(&modify_dir, "xyz", &object) == STD_OK) {
            fprintf(stderr, "Successful name change\n");
            return;
        }
    }
}
fprintf(stderr, "Error -- no name change\n");
```

Several more examples can be found in the source code for the name-based directory functions (see *name*(L)), since they are implemented as little more than calls to this module.

See Also

name(L).

(in libraries: libamoeba.a, libamunix.a)

Name

environment − lookup, delete and modify string environment variables

Synopsis

```
void env_delete(env, name)
char *env_lookup(env, name)
char **env_put(env, env_alloc, newmap, override)
```

Description

This module is used for manipulating string environments. A string environment is just a mapping from character string names to character string values. Each process has a standard string environment, pointed to by the global *environ* variable.

These routines take the environment pointer as a parameter so that they can operate on any environment, not just the standard environment. The environment pointer should point to a NULL-terminated array of pointers to NULL-terminated character strings. Each string should have the form ‘‘name=value’’ (where ‘‘name’’ does not contain ‘‘=’’), specifying that ‘‘name’’ maps to ‘‘value’’. No two strings in the same environment should use the same name. (These routines will not add a duplicate name but it is possible to build such an environment by hand.)

env_delete

```
    void
    env_delete(env, name)
    char **env;
    char *name;
```

This function deletes the all environment variables with name *name* from the environment pointed to by *env*. If *name* is not in the environment then the environment will be unchanged.

env_lookup

```
    char *
    env_lookup(env, name)
    char **env;
    char *name;
```

This routine looks up the environment variable with name *name* in the environment *env* and returns a pointer to the *value* associated with that name. (I.e., the part after the ‘‘=’’ in the environment string.) It returns the NULL-pointer if *name* was not found in the environment.

env_put

```
char **
env_put(env, env_alloc, newmap, override)
char **       env;
unsigned int *env_alloc;
char *        newmap;
int           override;
```

This routine adds the ‘‘name=value’’ environment entry pointed to by *newmap* to the environment *env* if it does not already exist. If the specified ‘‘name’’ already exists it will only be replaced if *override* is non-zero. On entry, *env_alloc* should point to an integer containing the number of elements allocated (though not necessarily in use) in the environment. If the *env* array is not malloced the value should be zero. On return it contains the new number of elements allocated.

Since the environment may need to grow, it may use *malloc* or *realloc* to obtain a larger piece of memory.

On success (which includes refusing to replace an existing name), *env_put* returns a pointer to the new environment. On failure, such as a defective environment or *malloc* failure, it returns NULL.

See Also

exec_file(L), getcap(L).

(in libraries: libamoeba.a, libamunix.a)

Name

error − return names of standard exceptions and standard errors

Synopsis

```
#include "exception.h"

char *
exc_name(sig)
signum sig;

#include "stderr.h"

char *
err_why(err)
errstat err;
```

Description

Exc_name returns a pointer to a static string describing the exception specified by *sig*.

Err_why returns a pointer to a static string describing the error specified by *err*.

They only know about the more common errors and exceptions, and return a generic description, such as

```
    amoeba error −842780
```

in other cases. More specific errors are covered by other functions, such as *tape_why*.

Warnings

Both *exc_name* and *err_why* use an internal buffer to store their generic descriptions. This buffer will be overwritten by a subsequent call, possibly by another thread.

Example

The function *find_cap* tries to find a capability. It prints a descriptive message and exits if it cannot.

```
void
find_cap(name, capp)
char name[];
capability *capp;
{
    errstat err;

    err = name_lookup(name, capp);
    if (err != STD_OK) {
         fprintf(stderr, "Cannot lookup %s: %s\n",
             name, err_why(err));
         exit(1);
    }
}
```

See Also

exception(H), rpc(L), signals(L), tape(L).

Name

exception.h – predefined exception names

Synopsis

```
#include "exception.h"

#define EXC_ILL  (-2)     /* Illegal instruction */
#define EXC_ODD  (-3)     /* Mis-aligned reference */
#define EXC_MEM  (-4)     /* Non-existent memory */
#define EXC_BPT  (-5)     /* Breakpoint instruction */
#define EXC_INS  (-6)     /* Undefined instruction */
#define EXC_DIV  (-7)     /* Division by zero */
#define EXC_FPE  (-8)     /* Floating exception */
#define EXC_ACC  (-9)     /* Memory access control violation */
#define EXC_SYS  (-10)    /* Bad system call */
#define EXC_ARG  (-11)    /* Illegal operand (to instruction) */
#define EXC_EMU  (-12)    /* System call emulation */
#define EXC_ABT  (-13)    /* abort() called */

#define EXC_NONE 0        /* List terminator */
```

Description

This manual page offers a quick reference to the exceptions that a thread can generate. Exception handling is treated in *signals*(L).

The names for exceptions are defined in the include file *exception.h*. They are:

EXC_ILL	Illegal instruction.
EXC_ODD	Mis-aligned memory reference.
EXC_MEM	Access to non-existent memory. Note: on some machines this exception is not generated; rather *EXC_ACC* is generated instead.
EXC_BPT	Breakpoint instruction or trace-mode exception.
EXC_INS	Undefined instruction.
EXC_DIV	Divide by zero (or other arithmetic trap).
EXC_FPE	Floating point exception.
EXC_ACC	Memory access violation (e.g. writing read-only memory).
EXC_SYS	Bad system call or illegal system call arguments.
EXC_ARG	Illegal instruction operand. On some machines this shows up as *EXC_INS*.
EXC_EMU	System call emulation trap.
EXC_ABT	The library function *abort* was called (see *ansi_C*(L), *posix*(L)).

Exception number zero *(EXC_NONE)* is reserved. It is used as a list terminator in the vector passed by *sys_setvec* (see *signals*(L)).

See Also

ansi_C(L), signals(L), process(L).

(in libraries: libamoeba.a, libamunix.a)

Name

exec_file – start execution of a process

Synopsis

```
#include "amoeba.h"
#include "module/proc.h"
#include "am_exerrors.h"

errstat exec_file (
    capability     *prog,
    capability     *host,
    capability     *owner,
    int             stacksize,
    char          **argv,
    char          **envp,
    struct caplist *caps,
    capability     *process_ret
);

errstat exec_pd (
    process_d      *pd,
    int             pdlen,
    capability     *host,
    capability     *owner,
    int             stacksize,
    char          **argv,
    char          **envp,
    struct caplist *caps,
    capability     *process_ret
);
```

Description

The description of the process interface is split into several parts. The process descriptor data structure and its basic access methods are described in *process_d*(L). The low-level kernel process server interface is described in *process*(L). The high-level process execution interface is described here. See also *pd_read*(L).

Exec_file starts the execution of a new process. It takes many arguments, allowing precise control over the execution, but chooses sensible defaults for input parameters that are zero or NULL-pointers. Its first argument can be the capability of either an executable file or of a *directory*. The latter is used to implement heterogeneous process startup (i.e., executing a program without prescribing the architecture on which it is to be run). The directory should contain a set of one or more binaries, each for a different architecture. Of course, in order for this to work properly, the programs must be semantically the same (for example, files used or created by the program should be in byteorder independent format). The *run*(A) server is used to determine — given the architectures of the process descriptors available — which host is the most suitable to be used.

Exec_pd has, with the exception of heterogeneity, the same functionality as *exec_file*, but it uses a process descriptor passed as an argument instead of a capability for the executable.

The arguments have the following meaning:

prog	*(exec_file only)* Capability of the binary file or directory containing process descriptors to be executed (see *pd_read*(L) for a description of the directory format required). If this is a NULL-pointer, the capability named by argv[0] is looked up.
pd, pdlen	*(exec_pd only)* The process descriptor (in native byte order) and its length (in bytes). The process descriptor will be converted to standard network byte order by *pro_exec* (see *process*(L)).
host	Capability for the process server where the process should execute. The process server capability can be found by looking up `PROCESS_SVR_NAME` (defined in *proc.h*, typically ‘‘proc’’) in the processor directory. If this is a NULL-pointer, the selection of a suitable pool processor (and the corresponding process descriptor) is left to *exec_multi_findhost*(L).
owner	Capability of the process owner; a ‘‘checkpoint’’ transaction is sent to this capability if the process exits, receives a signal or causes an exception; see *process*(L). If this is a NULL-pointer, the owner found in the program’s process descriptor, if any, is retained (this is usually a NULL-capability, meaning the process has no owner).
stacksize	Stack size, in bytes. If this is zero, the size of the stack segment in the process descriptor is used; if that is also zero, a default is used (currently 16K).
argv	Null-terminated array of program argument strings, starting with the program name (using the same convention as for calling main). If this is a NULL-pointer, a single ‘‘–’’ is used as 0-th parameter. For *exec_file*, if the default for *prog* is to be taken from *argv* then *argv* and *argv*[0] must not be NULL-pointers.
envp	Null-terminated array of shell environment variables. Each string must be of the form *NAME=value.* If this is a NULL-pointer, the calling program’s shell environment is used, taken from the global variable *environ.*
caps	Capability environment. This is an array of *struct caplist* entries, the last of which should have `cl_name == NULL`; see *getcap*(L). If this is a NULL-pointer, the calling program’s capability environment is used, taken from the global variable *capv.* If this variable is also NULL (typically the case on UNIX), an error is returned.
process_ret	Should point to a variable where the capability for the new process is stored, if it was actually started. If this is a NULL-pointer, the process capability is lost.

If the run server is used to select a host, and the process cannot be executed because of lack of memory on the selected host, a new host is tried a couple of times.

Error Conditions:

`AMEX_NOPROG`	no program given
`AMEX_PDLOOKUP`	cannot find program
`AMEX_PDREAD`	cannot read process descriptor
`AMEX_PDSHORT`	not all bytes read from process descriptor
`AMEX_PDSIZE`	inconsistent process size
`AMEX_NOCAPS`	no capability environment
`AMEX_NOHOST`	cannot find suitable host processor
`AMEX_GETDEF`	the call to pro_getdef() failed (see *process*(L))
`AMEX_MALLOC`	cannot make local stack segment
`AMEX_STACKOV`	env+args too big for stack
`AMEX_SEGCREATE`	cannot create stack segment

These constants are defined in *am_exerrors.h*. Standard and RPC errors may also be produced, with conventional meaning; these are usually passed on from *pro_exec* (see *process*(L)).

Example

To execute the file *a.out* in a default environment, the following call suffices:

```
static char *arglist[] = {"a.out", (char *)0};
capability   process;
errstat      err;

err = exec_file((capability *)0,     /* prog */
                (capability *)0,     /* host */
                (capability *)0,     /* owner */
                0,                   /* stacksize */
                arglist,             /* argv */
                (char **)0,          /* envp */
                (struct caplist *)0, /* caps */
                &process             /* process_ret */
               );
```

See Also

ainstall(U), ax(U), buildstack(L), exec_findhost(L), getcap(L), pd_read(L), process(L), process_d(L), rpc(L).

(in libraries: libamoeba, libamunix.a)

Name

exec_findhost – find a suitable pool processor (host) to execute a process

Synopsis

```
#include "amoeba.h"
#include "module/proc.h"
#include "exec_fndhost.h"

errstat exec_findhost (
    process_d      *pd,
    capability     *procsvr_ret
);

errstat exec_multi_findhost (
    capability     *runsvr_pool,
    capability     *pooldir,
    process_d      *pd_array[],
    int             npd,
    int            *pd_chosen,
    capability     *procsvr_ret,
    char            hostname[PD_HOSTNAME]
);
```

Description

Exec_findhost finds a pool processor suitable for executing the specified process descriptor (see *process*(L) and *process_d*(L)). It is passed a process descriptor so it can use the architecture and required memory space to decide which processors are suitable.

Exec_multi_findhost is available for supporting *heterogeneous* process startup (i.e., executing a program without prescribing the architecture on which it is to be run). Also, it gives the caller more control over its operation by making it possible to specify a non-default run server or pool directory.

Exec_multi_findhost (and *exec_findhost*, which is implied whenever *exec_multi_findhost* is mentioned here) uses two strategies to find a suitable host. If possible, the run server (see *run*(A)) is asked for a suitable host. The run server performs load balancing for all users and all pool processors, so it can assign the most suitable host for the process (given what little it knows about the process) and distribute processes evenly over the available hosts. This takes at most two transactions: one to look up the capability of the run server (which is cached for future use), and one to ask the run server for a host.

If the run server's capability cannot be found, or if it does not respond in a timely fashion, *exec_multi_findhost* scans the pool directory to find a running host that has the architecture of one of the process descriptors provided. When called multiple times in the same process, a host will be chosen in a round-robin fashion. The first host is chosen at random to avoid the problem that every process (for example, a forked copy of the shell doing the actual command execution!) starts executing commands on exactly the same machine.

The arguments of *exec_multi_findhost* are:

runsvr_pool	Alternative run server pool (see *run*(A)). When this is a NULL-pointer (which is the normal case) the `RUN_SERVER` environment variable, when available, is used to look it up. Otherwise `DEF_RUNSVR_POOL`, as defined in *ampolicy.h* (typically */profile/pool/.run*), is used.
pooldir	When the run server cannot be found, or does not respond quickly enough, a host is selected from a *pool directory*. This is a directory containing subdirectories for each supported architecture; each subdirectory contains capabilities for the processors that form the processor pool for that architecture. The parameter *pooldir* can be used to specify an alternative pool directory. If this is a NULL-pointer (which it normally should be), `POOL_DIR` is used, defined in *ampolicy.h* (typically */profile/pool*).
pd_array, *npd*	An array of pointers to process descriptors having *npd* (>= 1) entries. (In order to get sensible results, the process descriptors should be architectural variants of the same program. A common way to obtain this, is to read the binaries from a directory using *pd_read_multiple* (see *pd_read*(L)).)
pd_chosen	Returns the index into *pd_array* of the process descriptor to be run on the selected host.
procsvr_ret	Returns the capability for the process server on the selected host.
hostname_ret	The name of the selected host (truncated, if needed) is returned in this buffer.

Exec_findhost just calls *exec_multi_findhost* with its single *pd* argument, and uses the default run server and pool directory (see the example section below).

Error Conditions:

FPE_BADARCH	Cannot determine architecture
FPE_NOPOOLDIR	Cannot find pool directory
FPE_BADPOOLDIR	Cannot open pool directory
FPE_NONE	No suitable processor in pool directory

These constants are defined in *exec_fndhost.h*.

Warning

Even when *exec_multi_findhost* thinks that a host is suitable, attempts to execute the given process on that host may fail for a variety of reasons (e.g., memory fragmentation being the most frequent cause, or processes on the host might have allocated extra memory in the meantime).

Example

The following function implements the functionality of *exec_findhost* using *exec_multi_findhost*.

```
errstat
findhost(pd, procsvr_ret)
process_d  *pd;
capability *procsvr_ret;
{
    char hostbuf[PD_HOSTNAME];
    int   pd_chosen;

    return exec_multi_findhost (
            (capability *)NULL, /* default run server      */
            (capability *)NULL, /* default pool directory  */
            &pd,                /* proc. desc. ptr array   */
            1,                  /* of length 1             */
            &pd_chosen,         /* pd chosen will be 0     */
            procsvr_ret,        /* process server selected */
            hostbuf             /* hostname (not used)     */
       );
}
```

See Also

exec_file(L), process(L), process_d(L), run(A).

(in libraries: libamoeba.a)

Name

exitprocess – terminate a process

Synopsis

```
#include "module/proc.h"

void exitprocess(status)
int status;
```

Description

Exitprocess terminates the current process. The value *status* is the so-called *exit status*, and is passed to the parent. If the process has more threads the other threads are aborted, just as if a *pro_stun* had been done (see *process*(L)). As soon as all threads have terminated a dummy process descriptor (see *process_d*(L)) is created, consisting only of the fixed length part with no thread map and no segment map. This process descriptor is sent to the owner, with a reason of `TERM_NORMAL`, and the exit status as detail.

When the last thread of a process calls *exitthread* (see *sys_newthread*(L)) or *thread_exit* (see *thread*(L)), this call behaves as if `exitprocess(0)` had been called.

Warnings

Most user programs should call *exit* (see *ansi_C*(L)) or *_exit* (see *posix*(L)), not *exitprocess*. Those functions do some additional cleanup like properly closing files before they terminate the process.

See Also

process(L), process_d(L), posix(L), ansi_C(L), sys_newthread(L), thread(L).

(in libraries: libamoeba)

Name

findhole – search for unused space in virtual memory

Synopsis

```
#include "module/proc.h"

char *
findhole(size)
long size;
```

Description

Findhole retrieves the virtual memory layout (using *getinfo*(L)) and looks for a place where a segment of *size* bytes can be mapped in, without overlapping other segments, and with some unmapped space (typically 32K) on each side. It returns the address where the segment should be mapped.

Diagnostics

Findhole returns zero if the *getinfo* call failed or if no fitting hole was found.

Warnings

There is no locking mechanism, so it is conceivable that another task fills the hole with something before this task has had a chance.

This interface should be replaced by one that finds a hole *and* maps a segment in.

See Also

getinfo(L), malloc(L), seg_map(L).

(in libraries: libamoeba.a, libamunix.a)

Name

fs – stream interface (OBSOLETE)

Synopsis

```
#include "file.h"

long fsread(cap, position, buf, size)
capability *cap;
long position;
char buf[];
long size;

long fswrite(cap, position, buf, size)
capability *cap;
long position;
char buf[];
long size;
```

Description

This interface is implemented by the UNIX-emulation pipe-server and by native Amoeba tty-servers.

Fsread reads *size* bytes from the stream pointed to by *cap*, starting from position *position*, and puts them in *buf*.

Fswrite writes *size* bytes from *buf* to the stream pointed to by *cap*, starting from offset *position*.

Diagnostics

If an error occurred, both functions will return the (negative) code. Otherwise they return the number of bytes actually copied. Which errors can be returned obviously depends on the server that manages *cap*.

Warnings

The offset parameters are currently ignored by every server that implements the fs-interface. They descend from an age in which the file interface was identical with the stream interface.

Example

This function fsreads an object and copies it onto another. It returns 0 on failure, and 1 on success. Note that this will not work for files, only for tty's and pipes.

```
#include "amoeba.h"
#include "stdio.h"
#include "assert.h"
#include "stderr.h"
#include "file.h"

copy(in, out)
    capability *in, *out;
{
    char buf[200];
    long position = 0;
    int done = 0;

    do {
        long r, w;/* #bytes read, written */
        r = fsread(in, position, buf, (long)sizeof(buf));
        if (r < 0) {
            fprintf(stderr, "fsread failed: %s\n", err_why(r));
            return 0;
        }
        if (r == 0) done = 1;
        else {
            w = fswrite(out, position, buf, r);
            if (w < 0) {
                fprintf(stderr, "fswrite failed: %s\n", err_why(w));
                return 0;
            }
            assert(r == w);
            position += r;
        }
    } while (!done);
    return 1;
}
```

(in libraries: libamoeba.a, libamunix.a)

Name

getcap – get environment capability

Synopsis

```
#include "amoeba.h"

capability *
getcap(name)
char name[];
```

Description

Getcap returns a pointer to the environment capability *name*.

Diagnostics

Getcap returns NULL when no capability *name* exists.

Environment Capabilities

Here is a list of commonly used environment capabilities:

`ROOT`	The root directory
`WORK`	The working directory
`STDIN`	Standard input
`STDOUT`	Standard output
`STDERR`	Diagnostics output
`_SESSION`	The session server
`TTY`	The current terminal

Warnings

There is no capability environment under UNIX so *getcap* always returns NULL.

Example

To find the capability for '/':

```
    capability *slashcap;
    slashcap = getcap("ROOT");
```

See Also

envcap(U).

(in libraries: libamoeba.a)

Name

getinfo – get information about the current process

Synopsis

```
#include "amoeba.h"
#include "module/proc.h"

int getinfo(retcap, procbuf, len)
capability *retcap;
process_d *procbuf
int len;
```

Description

Getinfo returns information about the calling process. If *retcap* is not a NULL-pointer the capability for the current process (with all right bits set) will be stored there. If *len* is non-zero *procbuf* should point to an area of *len* bytes. Upon return, *procbuf* will be filled with a stripped-down process descriptor for the current process. The process descriptor contains all the static information and the segment map, but it does not contain a thread map. The process descriptor is returned in native byte order. See *process_d*(L) for a description of the process descriptor. *Getinfo* returns the size of the process descriptor.

Note that the *segment identifiers* used by *seg_map* and friends (see *seg_map*(L)) are indices into the array of segment descriptors returned as part of the process descriptor.

Warnings

If *procbuf* is not big enough to contain the complete segment mapping table no segments will be returned at all. So, if *getinfo* is used to obtain a segment map the value of *pd_nseg* should be checked to see whether the call was successful.

Example

The following program uses *getinfo* to get a segment map, searches for the segment that contains the stack of the current thread and unmaps that segment. It is left to the imagination what will happen after the *seg_unmap* call returns.

```
#include "amoeba.h"
#include "module/proc.h"

#define MAXPD 1000

main() {
    char pdbuf[MAXPD];
    process_d *pd = (process_d *)pdbuf;
    segment_d *sd;
    segid i, seg_to_unmap;
    char c;

    getinfo((capability *)0, pd, MAXPD);
    if (pd->pd_nseg == 0) error("buffer too small\n");
    for (i=0; i<pd->pd_nseg; i++) {
         sd = &PD_SD(pd)[i];
         if ((char *)sd->sd_addr <= &c &&
             (char *)sd->sd_addr + sd->sd_len-1 >= &c)
                 seg_to_unmap = i;
    }
    printf("Unmapping segment %d\n", seg_to_unmap);
    seg_unmap(seg_to_unmap, (capability *)0);
    printf("We will not get here\n");
}
```

See Also

process_d(L), seg_map(L).

(in libraries: libamoeba.a, libkernel.a, libamunix.a)

Name

goodport – routines for avoiding repeated transactions to bad ports

Synopsis

```
#include "amoeba.h"
#include "module/goodport.h"

gp_badport(port, command)
gp_notebad(cap, status)
gp_trans(cap, func_call)
gp_std_copy(server, source, target)
gp_std_destroy(cap)
gp_std_info(cap, buf, n, len)
gp_std_restrict(cap, mask, new)
gp_std_status(cap, buf, n, len)
gp_std_touch(cap)
```

Description

One of the problems with the client-server model is that it is possible that a server for a particular type of object is unavailable. Every attempt to communicate with that server will result in a time-out while an attempt is made to locate the server. A succession of time-outs can result in very long delays and seriously impact perceived and actual performance. This module attempts to keep a history of servers which have recently been unavailable. It does this by remembering ports for which a locate time-out has occurred. By first checking to see if a port is in the list of unavailable ports it is possible to avoid the long delays. If an object is replicated then it is possible to immediately see if a particular port is known to be not responding and first attempt to obtain the object from one of the alternative servers for the object.

A port is defined to be *bad* if a previous transaction with that port has returned the error `RPC_NOTFOUND`. That is, an attempt to locate the server with that port has failed.

A fixed size cache of bad ports is maintained. If more bad ports are found than fit in the cache then the least recently added entry in the cache is deleted to make space for a new entry.

Types

The set of legal values for the *command* argument of *gp_badport* are defined in *goodport.h*.

gp_badport

```
int
gp_badport(port, command)
port *p;
int command;
```

Gp_badport is used to modify the list of *bad ports*. There are four values for *command* which are defined in *goodport.h*.

`GP_INIT`	Initialize the list of bad ports to empty. The function returns 1 in this case. The port argument *p* is not used and may be the NULL-pointer.
`GP_APPEND`	Add the port pointed to by *p* to the cache of bad ports. Returns 1 if the port was already in the cache and 0 if it added it.
`GP_DELETE`	Remove the port pointed to by *p* from the list of bad ports. Returns 1 if it deleted it and 0 if the *port* was not in the cache.
`GP_LOOKUP`	This routine returns 1 if the port pointed to by *p* is in the cache and 0 otherwise.

gp_notebad

```
errstat
gp_notebad(cap, status)
capability *cap;
errstat status;
```

Gp_notebad is a utility routine to add a port to the *bad port* cache. If a transaction returned an error status then the capability *cap* used for the transaction and the error returned *status* can be given to this routine. If *status* is `RPC_NOTFOUND` this routine will add the port of the server to the *bad port* cache. The function returns *status*.

gp_trans

```
errstat
gp_trans(cap, func_call)
capability *cap;
errstat (*func_call)();
```

Gp_trans is a macro defined in *goodport.h* that calls the function *func_call* if the port of the capability *cap* is not in the *bad port* cache. If it is in the cache then it returns the error `RPC_BADPORT`.

If the result of the *func_call* is `RPC_NOTFOUND` then it registers the port of the server in the *bad port* cache. It returns the error status of *func_call*.

This macro has been used to define several other macros for all the *std* functions that automatically log any bad ports in the *bad port* cache. These routines have the same parameters as the routines described in *std*(L) but the function names are prefixed with *gp_*.

See Also

std(L).

(in libraries: libamoeba.a)

Name

grp − the group communication primitives

Synopsis

```
#include "amoeba.h"
#include "group.h"

g_id_t   grp_create(p, resilience, maxgroup, lognbuf, logmaxmess)
errstat  grp_forward(gid, p, memid)
errstat  grp_info(gid, p, state, memlist, size)
g_id_t   grp_join(hdr)
errstat  grp_leave(gid, hdr)
int32    grp_receive(gid, hdr, buf, size, more)
g_indx_t grp_reset(gid, hdr, nmem)
errstat  grp_send(gid, hdr, data, size)
errstat  grp_set(gid, p, sync, reply, alive, large)
```

Description

RPC provides point-to-point communication between a single client and a single server. What is often needed is 1-to-n communication (for example in a replicated server). This can be simulated with n−1 RPCs but this is not very efficient. In most systems this will cost at least 2 (n−1) network packets (one packet for the message and one for the acknowledgement). If the message is larger than a single packet the cost will be even higher. Therefore a more efficient system of message passing for groups of processes has been provided. It provides optional fault tolerance and a total ordering of messages. Where possible it takes advantage of hardware multicast support (e.g., on Ethernet). Where no support is available it falls back to using point-to-point messages to send the information.

A *group* consists of one or more processes, called *members*, typically running on different processors and cooperating to provide some service or to implement some application program. Processes may be a member of more than one group. Groups are closed, which means that only group members can send a broadcast message to the group. Processes which are not a member and which wish to communicate with a group can use RPC (or can join the group).

A group is identified by a port. All group calls must supply this port. A group is explicitly created by calling *grp_create*. The process that called this primitive is the first member of the group. It becomes the *sequencer* for the group, ensuring a total ordering of messages and maintaining message history for use in recovery in the event of failures. Other processes can become a member by calling *grp_join*, and supplying as parameter the port with which the group has been created. There are a number of ways in which the creator of the group can pass the port to other processes so that they can join the group. One way is to publish the port in the directory server so that other processes can look it up. Another way is to use the program *gax*(U) which passes the port to a process through its capability environment. The knowledge of the port is sufficient to be allowed to join the group.

Both *grp_create* and *grp_join* return a small identifier, called the group id (*gid*). The gid and port have to be supplied with all subsequent group calls (the port is passed as part of the header or is supplied explicitly). The *gid* allows the system to do a fast lookup of the group state. The port must also be supplied, so that the kernel can check if the member is allowed to perform group calls.

Once a process is a member of the group, it can call *grp_send*, *grp_receive*, *grp_set*, *grp_info*, *grp_forward*, and *grp_leave*. *Grp_send* allows a member to send a message to the group. *Grp_receive* allows a member to receive a message sent to the group. *Grp_set* allows a member to set timer values. *Grp_info* allows a member to learn about the group state. *Grp_forward* allows a member to forward a message that is received with a *getreq* to another member of the group. This call is the link between the group communication calls and the *rpc*(L) calls. *Grp_leave* allows a member to leave the group. Once a member has successfully called *grp_leave*, the process is not a member of the group anymore. When the last member leaves the group, the group ceases to exist.

All primitives, except *grp_set* and *grp_info* block the calling thread until the call completes. A typical way of programming with the group communication primitives is to dedicate one thread to doing the *grp_receive* calls and another thread to doing the real work.

Error codes

The possible error codes returned by the group primitives, and their interpretation are shown in the next table:

Error code	Description
`STD_OK`	operation succeeded
`STD_ARGBAD`	illegal argument
`STD_EXISTS`	process already member of this group
`STD_NOMEM`	cannot allocate group buffers
`BC_ABORT`	a member has died; need grp_reset
`BC_BADPORT`	invalid group port specified
`BC_FAIL`	generic group failure code
`BC_ILLRESET`	illegal grp_reset
`BC_NOEXIST`	group does not exist (anymore)
`BC_TOOBIG`	message too large
`BC_TOOMANY`	no space for new group member

Note that the primitives *grp_create*, *grp_join grp_receive* and *grp_reset* return a signed integer rather than the standard error type *errstat*. In these cases the call must be considered successful when a non-negative integer is returned. Otherwise the return value should be interpreted as an error code.

Functions

grp_create

```
g_id_t
grp_create(p, resilience, maxgroup, lognbuf, logmaxmess)
port   * p;
g_indx_t resilience;
g_indx_t maxgroup;
uint32   lognbuf;
uint32   logmaxmess;
```

The primitive *grp_create* creates a new group. The parameter *resilience* specifies how many member failures must be tolerated without loss of any message. Thus, if after *resilience* crashes, the group is rebuilt with *grp_reset* it is still guaranteed that the remaining members will still receive all the messages sent to the group and that they still will receive them in the same order. Consider a group with *resilience* equal to zero. In this case, if a member successfully sends a message to the group and then crashes, it is not guaranteed that the remaining members will receive this messages after they have rebuilt the group. The parameter *maxgroup* limits the total number of group members permitted. The value of *resilience* may be greater than the maximum number of group members *maxgroup* but see the description of *grp_send* below for a discussion of this. Parameters *lognbuf* and *logmaxmess* specify the amount and size of message buffers that are to be allocated by the kernel of each member. They are both 2-logs of the actual amount. (Note that the actual buffers allocated are slightly larger than the size specified to allow space for the addition of a header to the message. The caller can thus send messages up to and including the specified buffer sizes.)

Regarding the parameter values, the following restrictions apply:

- *resilience* must be less than 32.
- *lognbuf* depends on size of the group (*nbuf* should be bigger than the number of members) but must be at least 4.
- *logmaxmess* must be at least 10 (i.e., maxmess should be at least 1K).
- *maxgroup* must be at least 1. Since the group protocol sometimes sends global state information to all members (e.g., during the join and recovery stage), the parameter *logmaxmess* may have to be increased when the number of members becomes larger than about 30.

If a process already is a member of a group, it is not allowed to create another group with the same port. If multiple processes create a group with the same port, no error is returned. If a process does a join on that port, it will become member of one group at random. This is similar to multiple servers performing a *getreq* on the same port. A request sent to that port will be received by one of the servers at random. Messages sent to that group are only received by members of that group. They will not go to other groups listening to the same port.

grp_forward

```
errstat
grp_forward(gid, p, memid)
g_id_t   gid;
port   * p;
g_indx_t memid;
```

Grp_forward integrates RPC with group communication. When a client does an RPC with a service, which consists of multiple servers listening to the same port, the client has no idea which server will receive the request; one of the servers will get the request. If the server that got the request is not able to serve the request (e.g., it does not store the requested data), the server can forward the request to another server in the group, instead of doing a *putrep*. The header and the user data are forwarded to the other server. When the other server's *getreq* succeeds, it will receive the header and user data that the client has sent to the first server and which is forwarded by the first server. This is transparent to the client process. It does not notice that the request is forwarded to another server.

Grp_forward can only be called by a member of the group with id *gid* (as returned by *grp_create* or *grp_join),* and port *p*. The RPC is forwarded to the group member specified by *memid*. That member must have a pending *getreq* waiting to receive a message or the forward will fail. The member ids for the various members are obtained using *grp_info* (described below).

It is an error to do a *putrep* for a request that has been forwarded.

grp_info

```
typedef struct {
    g_seqcnt_t  st_expect;    /* next seq # to be delivered */
    g_seqcnt_t  st_nextseqno; /* next seq # to be received */
    g_seqcnt_t  st_unstable;  /* next seq # to be acked */
    g_index_t   st_total;     /* total number of members */
    g_index_t   st_myid;      /* my member identifier */
    g_index_t   st_seqid;     /* sequencer identifier */
} grpstate_t, *grpstate_p;

errstat
grp_info(gid, p, state, memlist, size)
g_id_t     gid;
port     * p;
grpstate_p state;
g_indx_t   memlist[];
g_indx_t   size;
```

The *grp_info* primitive allows a group member to acquire information stored in the kernel about a group. If it succeeds it will return in the output structure *state* the number of members in the group, the member identifier of the caller and the sequence number of the next message expected. The identifiers of the members in the group are returned in the array *memlist* having *size* entries. The *memlist* can be used, for example, to find out which member has crashed by comparing the *memlist* returned before *grp_reset* with the *memlist* returned after *grp_reset*. A member will have the same member identifier during its entire life-time.

grp_join

```
g_id_t
grp_join(hdr)
header_p hdr;
```

Once a group has been created, other processes can become members of it by executing *grp_join*. Only a member can receive messages that are sent to its group. The group to be joined is specified by the port contained in the header *hdr*. Like *grp_create*, *grp_join* returns a group descriptor for use in subsequent group calls. In addition to adding a process to a group, *grp_join* also delivers the header *hdr* to all other members (excluding itself). This way, other members can find out that a new member has joined the group. The port of the group should be stored in *h_port*, when calling *grp_join.* A process is not allowed to join a group where it is already a member.

grp_leave

```
errstat
grp_leave(gid, hdr)
g_id_t   gid;
header_p hdr;
```

Once a process is a member of a group, it can leave the group by calling *grp_leave*. After a member has left the group, it will not receive subsequent broadcasts. In addition to leaving the group, *grp_leave* delivers *hdr* to all members (including the leaving member itself). This way, all members can find out that a member has left. The member receives its own message, so that it can check if it has processed all the messages until its leave message. After a member has left the group, it can join the group again. Like in all other group primitives that have a header parameter, the port in the header should be equal to the group's port. When the last member of a group calls *grp_leave*, the group ceases to exist.

grp_receive

```
int32
grp_receive(gid, hdr, buf, size, more)
g_id_t   gid;
header_p hdr;
bufptr   buf;
uint32   size;
int    * more;
```

To receive a broadcast, a member must call *grp_receive*. If a broadcast arrives and no such primitive is outstanding, the message is buffered. When the member finally does a *grp_receive*, it will get the next one in sequence. The parameters *hdr*, *buf* and *size* specify the header and the buffer in which the message should be delivered. If *size* is smaller than the data that arrived, the arriving data is truncated. The output parameter *more* is used to indicate to the caller that one or more broadcasts have been buffered and can be fetched using *grp_receive*. The port of the group should be stored in *h_port*, when calling *grp_receive.* If *grp_receive* succeeds, it returns the size of the buffer received. If a member never calls *grp_receive*, the group may block, because it may run out of memory to buffer messages. Messages are never intentionally thrown away until all the members have processed them.

grp_reset

```
g_indx_t
grp_reset(gid, hdr, nmem)
g_id_t   gid;
header_p hdr;
g_indx_t nmem;
```

The primitive *grp_reset* allows recovery from member crashes. If one of the members (or its kernel) does not respond to messages, the protocol enters a recovery mode. All outstanding calls return an error value (BC_ABORT) indicating that a member has crashed. The user application can now call *grp_reset* to transform the group into a new group that contains as many live members from the group as possible. This does not change the *gid*. Calling *grp_reset* is illegal when the group is not in recovery mode. The error `BC_ILLRESET` will be returned in this case.

The parameter *nmem* specifies the number of members that the new group must contain as a minimum. If fewer than *nmem* members of the group are still available then `BC_FAIL` will be returned. In this case it is possible to attempt another *grp_reset* with perhaps another value for *nmem* or perhaps at a later time when the network is restored. The group is not deleted when a reset attempt fails.

When *grp_reset* succeeds, it returns the number of members in the recovered group.

In addition to recovering from crashes, *grp_reset* delivers *hdr* to all newly recovered members. It may happen that multiple members initiate a recovery at the same moment. The new group consisting of the members that can communicate with each other, however, is built only once.

The way recovery is done is based on the design principle that policy and mechanism should be separated. In many systems that deal with fault tolerance, recovery from processor crashes is completely invisible to the user application. Here it is done differently. A parallel application that multiplies two matrices, for example, may want to continue even if only one processor is left. A banking system may require, however, that at least half of the group is alive. The user is able to decide on the policy. The group primitives only provide the mechanism.

grp_send

```
errstat
grp_send(gid, hdr, data, size)
g_id_t   gid;
header_p hdr;
bufptr   data;
uint32   size;
```

When a member wants to broadcast a message, it calls *grp_send*. It guarantees that the *hdr* and buffer *buf* of size *size* will be delivered to all members (including itself), even in the face of unreliable communication and finite buffers. Furthermore, when the resilience degree of the group is *k*, the protocol guarantees that in the event of a simultaneous crash of up to *k* members, it delivers the message to all remaining members or to none. If a group has less than *k* members, the message is delivered to all members. In general, the total number of failures from which the protocol can recover is equal to:

$$MIN(\, k,\ total\ number\ of\ members - 1).$$

Choosing a large value for *k* provides a high degree of fault-tolerance, but extracts a penalty in performance. The trade-off chosen is up to the user.

In addition to reliability, the protocol guarantees that messages are delivered in the same order to all members. Thus, if two members (on two different machines), simultaneously broadcast two messages, *A* and *B*, the protocol guarantees that either

(1) all members receive *A* first and then *B*, or
(2) all members receive *B* first and then *A*.

Random mixtures, where some members get *A* first and others get *B* first are guaranteed not to occur.

grp_set

```
errstat
grp_set(gid, p, sync, reply, alive, large)
g_id_t   gid;
port   * p;
interval sync;
interval reply;
interval alive;
uint32   large;
```

The *grp_set* primitive can be used to set a few parameters used in the implementation of groups. It is mainly present for testing and performance analysis.

The three interval parameters influence the rate at which members are synchronized. *Sync* determines how often the sequencer checks if the other members are up to date. *Reply* determines how soon a message is retransmitted. *Alive* determines how often members check each other to see if they are alive. The parameter *large* specifies above which message size a sending member itself should broadcast the message. Normally, a message to be broadcast is sent to the member responsible for ordering the messages. Letting the message be broadcast by the sending member has the advantage that the message only appears once on the network (though at the expense of more interrupts at the members, because there are two broadcasts instead of one).

Example

An example of a working program can be found in the source code in the file *src/test/performance/group/grp_perf.c*. A brief explanation of it is given below as a quick introduction.

The normal way to start a group program is with *gax*(U). It passes numerous command line arguments to a group program, including the number of group members (*ncpu*), the desired resilience (*resilience*) and which member of the group the current process is (*cpu*). These arguments need to be processed. It also provides several capabilities in the capability environment. The `GROUPCAP` tells which port the group will listen to.

```
capability *groupcap;
port group;

if((groupcap = getcap(GROUPCAP)) == NULL)
    panic("no GROUPCAP in cap env (use gax)");
group = groupcap->cap_port;
```

The routine *test()* in *grp_perf.c* creates the group and carries out the performance measurements. The member numbered 0 actually creates the group. The other members join the group after it has been created. Once all the members are present it can proceed with its work.

```
/* Start group. */
if (cpu == 0) {
    gd = grp_create(&group, (g_indx_t) resilience, ncpu,
                         (uint32) LOGBUF, logdata);
    if (gd < 0)
         panic("create failed (%d)0, (int) gd);
} else {
    if ((gd=grp_join(&hdr)) < 0)
         panic("%d: join failed %d0, cpu, gd);
}
```

The next step is to set some parameters and wait until all the expected group members are ready to do work. The variable *state.st_total* below reflects the total number of members in the group. The *handle()* function looks at each message received, in particular at group join requests, and keeps the *state* up to date.

Following this a thread is created to accept all incoming messages. It executes the routine *daemon()* which consists of the same basic loop above: *grp_receive()* followed by a call to *handle()*. It also has additional code to deal with recovery after the loss of group members.

Thereafter, one or more members send a set number of messages using the *grp_send()* primitive. It times how long it takes using *sys_milli()*. To ensure that the *daemon()* thread is scheduled the routine *threadswitch()* is called regularly. If the messages are not read fairly quickly after arrival, the group can choke up waiting for the ‘‘slow’’ member. Another solution to this is to enable preemptive scheduling and give the reader thread equal or higher priority than the sender thread.

```
        grpstate_t state;

        res = grp_set(gd, &group, (interval) SYNCTIMEOUT,
                          (interval) REPLYTIMEOUT,
                          (interval) debug_level, (uint32) large);
        if (ERR_STATUS(res)) {
            panic("%d: test: grp_set failed %d0,
                          cpu, ERR_CONVERT(res));
        }
        /* Find out who is in the group. */
        res = grp_info(gd, &group, &state, memlist,
                          (g_indx_t) MAXGROUP);
        if (ERR_STATUS(res)) {
            panic("%d: test: grp_info failed %d0,
                          cpu, ERR_CONVERT(res));
        }
        my_id = state.st_myid;

        /* Wait until the group is complete. */
        while (state.st_total < ncpu) {
            r = grp_receive(gd, &hdr, (bufptr) 0, (uint32) 0, &more);
            if (ERR_STATUS(r))
                panic("%d: test: grp_receive failed: %d0,
                          cpu, ERR_CONVERT(r));
            handle(&hdr, (char *) 0, 0);
        }
```

Once all the message timing has been completed it is necessary to terminate the group cleanly. This is done using *grp_leave()*.

```
        hdr.h_command = LEAVE;
        hdr.h_size = cpu;
        hdr.h_extra = my_id;
        s = grp_leave(gd, &hdr);
```

Once *test()* terminates the *daemon()* thread will receive the group leave command and terminate as well. Since it is the last active thread the process will then terminate.

See Also

gax(U), rpc(L).

(in libraries: head.o/libamoeba.a)

Name

head − C run-time start-off

Synopsis

```
#include "amoeba.h"
#include "caplist.h"

extern char *environ[];
extern struct caplist capv[];

main(argc, argv, envp, capv)
int argc;
char *argv[];
char *envp[];
struct caplist capv[];

_stackfix(sp)
```

Description

This document describes the way the C main program is called. The *main* routine is called with four arguments:

argc is the number of arguments;

argv is an array of strings containing the arguments, with *argv[0]* the name of the program;

envp is a NULL-terminated array of strings containing the string environment (all exported shell variables, usually);

capv is unique to Amoeba and contains the capability environment. See *getcap*(L) for a description of standard capability environment entries.

Since the string and capability environment are often needed by other library modules their addresses are also stored in the global variables *environ* and *capv*, respectively. In the light of compatibility with STD C, this is the recommended way of accessing them.

Stack format

The format of the stack segment ‘on the wire’, i.e., as it is created by *buildstack*(L), is in the byte order of the host that built it. From high to low addresses, the stack contains the capabilities, the names of the capabilities, the NULL-terminated *capv* array (with pointers to the previous items), the environment strings, the NULL-terminated *envp* array (with pointers to the environment strings), the argument strings, the *argv* array (with pointers to the argument strings), a pointer to the *capv* array, a pointer to the *envp* array, a pointer to the *argv* array and the argument count. Note that all addresses are real pointers, so the address where the stack segment will be mapped has to be known beforehand.

Since the stack is created in the byte-order of the originating machine it may have to be fixed before it can be handed to *main*. The routine *_stackfix* does this. It gets a pointer to the stack described above as an argument, checks *argc* to determine the byte order of the stack segment (argc is always supposed to be less than 2ˆ16), and swaps all pointers and integers if needed. Note that checkstack can only handle big-endian and little-endian architectures.

Implementation

The run-time start-off routine is very simple. First, it executes some architecture-dependent code to setup registers and terminate the frame pointer chain. Next, it calls *_stackfix* to fix the stack. Then it calls *main* with the aforementioned arguments. Finally it calls *exit* (see *ansi_C*(L)) with the return value from main as argument.

See Also

buildstack(L), exitprocess(L), getcap(L), ansi_C(L).

(in libraries: libamoeba.a, libkernel.a, libamunix.a)

Name

host_lookup − look up the capability for a host

Synopsis

```
#include "module/host.h"

errstat host_lookup(hostname, cap)
errstat ip_host_lookup(hostname, extension, cap)
errstat super_host_lookup(hostname, cap)
```

Description

These routines search for the capability of the host specified by the string *hostname*. They look in various places as described below. If the lookup is successful they return `STD_OK` and the capability for the host is in **cap*. Otherwise they return the error status and **cap* is not set.

super_host_lookup

```
    errstat
    super_host_lookup(hostname, cap)
    char *hostname;
    capability *cap;
```

If *hostname* begins with a / or ./ then it is assumed to be an absolute path name and *super_host_lookup* looks it up directly. Otherwise it assumes that it is a relative path name. It begins by looking for it in the directory `HOST_SUPER_DIR` as defined in the file *ampolicy.h* (typically */super/hosts*).

It returns the error status of the attempt to look up the name.

Under UNIX it performs one extra search before it looks in `HOST_SUPER_DIR`. It will attempt to look up the capability in a UNIX file with the name specified, relative to the directory `UNIX_HOST_DIR` as defined in the file *amupolicy.h* (typically `UNIX_HOST_DIR` is '.'). This is useful when porting Amoeba since it allows bootstrapping an Amoeba system from UNIX before an Amoeba directory server is available.

host_lookup

```
    errstat
    host_lookup(hostname, cap)
    char *hostname;
    capability *cap;
```

Host_lookup begins by calling *super_host_lookup*. If that succeeds it returns the value of *super_host_lookup*. Otherwise it looks in the directory `HOST_DIR` as defined in the file *ampolicy.h* (typically */profile/hosts*). For most users this last lookup is the one that will succeed. If this fails then it attempts to look up the host in the current directory.

The error status is either that of the last *name_lookup()* attempted (see *name*(L)) or `STD_NOMEM` if the *hostname* string is too long for internal buffering.

ip_host_lookup

```
     errstat
     ip_host_lookup(hostname, extension, cap)
     char *hostname;
     char *extension;
     capability *cap;
```

This function is used to look up one of the capabilities of an IP server (see *ipsvr*(A)). It uses *host_lookup* to find the capability for the specified host, which should be a host running an IP server. It then uses the extension to select the desired element of the server. The *extension* is a string which is either *ip*, *eth*, *tcp* or *udp*. Any other values will fail.

The function returns `STD_OK` on success. Otherwise it returns one of the standard error codes indicating the cause of failure.

Examples

The following code looks up the capability of the machine called `ihnp4`.

```
char * hostname = "ihnp4"
capability hcap;
errstat err;

if ((err = host_lookup(hostname, &hcap)) == STD_OK)
    /* use the capability for something */;
else
    printf("lookup of %s failed: %s\n", hostname, err_why(err));
```

To look up the *tcp* capability of the IP server running on the host *armada1E* use:

```
char * hostname = "armada1E";
capability tcpcap;
errstat err;

if ((err = ip_host_lookup(hostname, "tcp", &tcpcap)) == STD_OK)
    /* use the capability for something */;
else
    printf("ip lookup of %s.%s failed: %s\n",
                 hostname, "tcp", err_why(err));
```

See Also

name(L).

(in libraries: libamoeba.a)

Name

ip – Internet Protocol server's general-purpose client interface stubs

Synopsis

```
#include "stddef.h"
#include "amoeba.h"

#include "server/ip/hton.h"

u16_t htons(host_word)
u32_t htonl(host_dword)
u16_t ntohs(network_word)
u32_t ntohl(network_dword)
u16_t HTONS(host_word)
u32_t HTONL(host_dword)
u16_t NTOHS(network_word)
u32_t NTOHL(network_dword)

#include "server/ip/gen/oneCsum.h"

u16_t oneC_sum(prev, data, size)

#include "server/ip/tcpip.h"

char *tcpip_why(err)
errstat tcpip_keepalive(chan_cap, respite)
errstat tcpip_keepalive_cap(cap)
errstat tcpip_mkcap(tcpip_cap, obj, cap)
errstat tcpip_open(tcpip_cap, chan_cap)
errstat tcpip_read(chan_cap, buffer, bytes)
errstat tcpip_unkeepalive_cap(cap)
errstat tcpip_write(chan_cap, buffer, bytes)
```

Description

The Internet Protocol (IP) server implements four network protocols, as described in *ipsvr*(A). The four supported protocols are ETH, IP, TCP and UDP. The general-purpose routines described here and the protocol specific routines described in *ip_eth*(L), *ip_ip*(L), *ip_tcp*(L) and *ip_udp*(L) give access to the protocols of the IP server. Access to these services is provided using two types of capabilities: server capabilities and channel capabilities. The server capabilities are called *eth*, *ip*, *tcp*, *udp*, which correspond to the ETH, IP, TCP and UDP interfaces, respectively. The server capabilities can be used to obtain a channel to the corresponding server. This is done with *tcpip_open*. The channel capability can be used to transfer data using the protocol implemented by the server. This can typically be done with the *tcpip_* routines since they are generic to all interfaces. However the semantics of *tcpip_read* and *tcpip_write* vary slightly, depending on the protocol in use. The routines specific for each protocol server and the exact semantics of the *tcpip_* functions are described in the corresponding *ip_*xxx(L) manual page.

The descriptions of the various routines are divided into categories according to whether they are for byte-order conversion or general-purpose (i.e., they are used with all four interfaces). Before they are described, a brief introduction to the various types is given.

Types (general)

server/ip/types.h
: defines *u8_t*, *u16_t*, *u32_t* and *i32_t* (and *U8_t*, *U16_t*, *U32_t* and *I32_t* for use in prototypes).

Rights

The following rights are defined in *server/ip/tcpip.h*:

`IP_RIGHTS_OPEN`	The right to do a *tcpip_open* (this indicates a server capability).
`IP_RIGHTS_RWIO`	The right to do I/O (this indicates a channel capability).
`IP_RIGHTS_DESTROY`	The right to destroy a connection.
`IP_RIGHTS_LINGER`	The right keeps the server from destroying the connection when it is not used for some time.
`IP_RIGHTS_SUPER`	The right to get existing capabilities (using *tcpip_mkcap*).

Byte Order Conversion

htons, htonl, ntohs, ntohl

```
u16_t
htons(host_word)
u16_t host_word;

u32_t
htonl(host_dword)
u32_t host_dword;

u16_t
ntohs(network_word)
u16_t network_word;

u32_t
ntohl(network_word)
u32_t network_word;
```

These macros convert 16-bit and 32-bit quantities to and from the network byte order used by the TCP/IP protocols. The function of the macros is encoded in their name. *H* means host byte order, *n* means network byte order, *s* means a 16-bit quantity and *l* means a 32-bit quantity. Thus *htons* converts a 16-bit quantity from host byte order to network byte order. The difference between the lower case and upper case variants is that the lower case variants evaluate the argument at most once and the upper case variants can be used for constant folding. That is,

```
    htonl(f(x))
```

will call f(x) at most once and

```
    HTONS(0x10)
```

will be equivalent to 0x10 on a big-endian machine and 0x1000 on a little-endian machine.

General Functions

oneC_sum

```
    u16_t
    oneC_sum(prev, data, size)
    u16_t prev;
    u16_t *data;
    size_t size;
```

OneC_sum is used to calculate the one's complement checksum needed for IP network packets. The IP checksum is described in RFC-1071 (Computing the Internet checksum).

One_Csum expects three parameters:

prev	The checksum of previous blocks of data that are to be included in the checksum. The value of prev in first call to *oneC_sum* should be 0.
data	A pointer to the block of data. The data is interpreted as a series of 16-bit numbers in network byte order, but an odd number of bytes is also allowed.
size	The size of the data in bytes.

tcpip_why

```
    char *
    tcpip_why(err)
    errstat err;
```

This routine returns a pointer to a statically allocated string describing the error code *err*. If *err* is not one of the errors as described in the *Diagnostics* section below then *tcpip_why* will return the same string as *err_why* (see *error*(L)).

tcpip_keepalive

```
    errstat
    tcpip_keepalive(chan_cap, respite)
    capability *chan_cap;
    int *respite;
```

Tcpip_keepalive makes sure that a channel capability is not prematurely destroyed. In general channel capabilities are destroyed when not used for a while. *Tcpip_keepalive* returns the destruction timeout in *respite.* It is sufficient to call *tcpip_keepalive* again within *respite* milliseconds to prevent the channel from being destroyed.

See also *tcpip_keepalive_cap*.

tcpip_keepalive_cap

```
    errstat
    tcpip_keepalive_cap(cap)
    capability *cap;
```

Tcpip_keepalive_cap provides a more convenient interface to *tcpip_keepalive*. Each time a new TCP/IP capability is created, it can be handed to *tcpip_keepalive_cap*. *Tcpip_keepalive_cap* will keep the capability alive until it is either destroyed, *tcpip_unkeepalive_cap* is called for that capability, or the program exits. The first time *tcpip_keepalive_cap* is called, it creates a separate thread to manage all the capabilities that should be kept alive.

tcpip_unkeepalive_cap

```
    errstat
    tcpip_unkeepalive_cap(cap)
    capability *cap;
```

Tcpip_unkeepalive_cap removes the capability *cap* from the list of capabilities that should receive a *tcpip_keepalive*. Normally the IP server will destroy that capability after a while unless some other process keeps the capability alive.

tcpip_mkcap

```
    errstat
    tcpip_mkcap(tcpip_cap, obj, cap)
    capability *tcpip_cap;
    objnum obj;
    capability *cap;
```

A channel capability can be lost, or can be kept inside a process. *Tcpip_mkcap* recreates the capability which has object number *obj*. This provides a way to obtain an otherwise lost capability. To work out the object number of a particular channel use the *std_status*(U) command.

tcpip_open

```
errstat
tcpip_open(tcpip_cap, chan_cap)
capability *tcpip_cap;
capability *chan_cap;
```

Tcpip_open creates a new channel for a server capability. A pointer to the server capability must be passed via the first argument and the new channel capability is stored in the object pointed to by be the second argument.

tcpip_read

```
errstat
tcpip_read(chan_cap, buffer, bytes)
capability *chan_cap;
char *buffer;
size_t bytes;
```

Tcpip_read transfers data from the TCP/IP server to the client. The call blocks until enough data is available. The semantics of *tcpip_read* are different for each of the servers. See the *tcpip_read* section in the *ip*_xxx(L) manual pages for the exact semantics.

tcpip_write

```
errstat
tcpip_write(chan_cap, buffer, bytes)
capability *chan_cap;
char *buffer;
size_t bytes;
```

Tcpip_write transfers data from the client to the TCP/IP server. The call blocks until enough buffer space is available. The semantics of *tcpip_write* are different for each of the servers. See the *tcpip_write* section in the *ip*_xxx(L) manual pages for the exact semantics.

Diagnostics

The TCP/IP server introduces several new error codes. These are defined in *server/ip/tcpip.h*.

`TCPIP_PACKSIZE`	This indicates an attempt to read (*tcpip_read*) or write (*tcpip_write*) with a buffer that is too large or too small.
`TCPIP_OUTOFBUFS`	The TCP/IP server has insufficient memory to execute the request.
`TCPIP_BADIOCTL`	This indicates an attempt to execute a command the particular server does not understand. For example, a *tcp_ioc_getconf* on an ETH channel.
`TCPIP_BADMODE`	The request was refused because the channel is not fully configured, in the wrong state or the parameters are invalid.

`TCPIP_BADDEST`	This indicates an illegal destination address for a packet.
`TCPIP_DSTNORCH`	The destination was not reachable.
`TCPIP_ISCONN`	The channel is already connected so a second request is refused.
`TCPIP_ADDRINUSE`	This address is in use.
`TCPIP_CONNREFUSED`	The connection was refused by the other side.
`TCPIP_CONNRESET`	The connection was reset (non-gracefully terminated) by the other side.
`TCPIP_TIMEDOUT`	The connection was terminated due to an expired timer.
`TCPIP_URG`	Urgent data is present and the current receive mode does not allow urgent data to be transferred.
`TCPIP_NOURG`	No urgent data is present and a request came for urgent data.
`TCPIP_NOTCONN`	The request requires a connected channel and the channel is not connected.
`TCPIP_SHUTDOWN`	The connection was shutdown. That is, a *tcp_ioc_shutdown* has been executed so no more data can be transmitted.
`TCPIP_NOCONN`	The connection does not exist.
`TCPIP_ERROR`	A generic error code for extremely weird cases.

See Also

error(L), ip_eth(L), ip_ip(L), ip_tcp(L), ip_udp(L), ipsvr(A).

(in libraries: libamoeba.a)

Name

ip_eth – Internet Protocol server's Ethernet client interface stubs

Synopsis

```
#include "amoeba.h"
#include "server/ip/eth_io.h"
#include "server/ip/types.h"
#include "server/ip/gen/ether.h"
#include "server/ip/gen/eth_io.h"

errstat eth_ioc_getstat(chan_cap, ethstat)
errstat eth_ioc_getopt(chan_cap, ethopt)
errstat eth_ioc_setopt(chan_cap, ethopt)
```

Description

The Internet Protocol (IP) server implements four network protocols, as described in *ipsvr*(A). The routines described below give access to the Ethernet protocol of the IP server. This allows reading and writing of raw Ethernet packets. Access to this service is provided using two types of capabilities: a server capability and channel capabilities. The server capability is called *eth*. It is used to obtain a channel to the corresponding server. This is done with *tcpip_open* (see *ip*(L)). The channel capability can be used to transfer data using the protocol implemented by the server. This is also done with the generic *tcpip_read* and *tcpip_write* routines. Since their semantics vary slightly depending on channel type, these routines are described briefly below. The *eth_ioc_* routines are used to manage the options and status of the ETH channel.

Before they are described, a brief introduction to the various types is given. The general-purpose types and rights in the capabilities are described in *ip*(L).

Eth Types

server/ip/gen/ether.h
defines struct ether_addr (*ether_addr_t*), *ether_type_t* and *Ether_type_t* for use in prototypes.

server/ip/gen/eth_io.h
defines struct nwio_ethopt (*nwio_ethopt_t*) and struct nwio_ethstat (*nwio_ethstat_t*)

server/ip/gen/eth_hdr.h
defines struct eth_hdr (*eth_hdr_t*)

General Functions

tcpip_read

```
errstat
tcpip_read(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_read transfers Ethernet packets from the Ethernet channel specified by *chan_cap* to the client. The data is returned in *buffer* which has size *nbytes*. The call blocks until a packet is available. If successful it returns the number of bytes read. If unsuccessful the function returns an error status. It is possible to read just the data of the Ethernet packets or both the header and the data. See *eth_ioc_setopt* below for details.

Only reads with a buffer size greater than or equal to the maximum packet size are allowed.

Error Conditions:

See *ip*(L) for a description of the `TCPIP_` error codes.
`TCPIP_BADMODE`
`TCPIP_PACKSIZE`
`STD_INTR`

tcpip_write

```
errstat
tcpip_write(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_write transfers the *nbytes* of data in *buffer* from the client to the Ethernet channel specified by *chan_cap*. The call blocks until enough buffer space is available. The buffer must contain a complete packet and be at least of minimum size (60 bytes). On success this function returns the number of bytes written. On error it returns a negative error status.

Error Conditions:

See *ip*(L) for a description of the `TCPIP_` error codes.
`TCPIP_BADMODE`
`TCPIP_PACKSIZE`
`STD_INTR`

ETH Functions

eth_ioc_getopt

```
errstat
eth_ioc_getopt(chan_cap, ethopt)
capability *chan_cap;
struct nwio_ethopt *ethopt;
```

This call returns in *ethopt* the current options of an Ethernet channel specified by *chan_cap*. See *eth_ioc_setopt* for a description of the options.

eth_ioc_getstat

```
errstat
eth_ioc_getstat(chan_cap, ethstat)
capability *chan_cap;
struct nwio_ethstat *ethstat;
```

Eth_ioc_getstat returns the Ethernet address and some statistics about the channel specified by the capability *chan_cap*. The result is returned in the structure to which *ethstat* points. The *struct nwio_ethstat* defined in *server/ip/gen/eth_io.h* is used to pass the configuration description to the server.

```
typedef struct nwio_ethstat
{
        ether_addr_t nwes_addr;
        eth_stat_t nwes_stat;
} nwio_ethstat_t;
```

```
typedef struct eth_stat
{
    unsigned long
         ets_recvErr,        /* # receive errors */
         ets_sendErr,        /* # send error */
         ets_OVW,            /* # buffer overwrite warnings,
                               (packets arrive faster than
                                can be processed) */
         ets_CRCerr,         /* # crc errors of read */
         ets_frameAll,       /* # frames not aligned (# bits
                               not a multiple of 8) */
         ets_missedP,        /* # packets missed due to too
                               slow packet processing */
         ets_packetR,        /* # packets received */
         ets_packetT,        /* # packets transmitted */
         ets_transDef,       /* # transmission deferred (there
                               was a transmission of an
                               other station in progress */
         ets_collision,      /* # collisions */
         ets_transAb,        /* # transmissions aborted due
                               to excessive collisions */
         ets_carrSense,      /* # carrier sense lost */
         ets_fifoUnder,      /* # fifo underruns (processor
                               is too busy) */
         ets_fifoOver,       /* # fifo overruns (processor is
                               too busy) */
         ets_CDheartbeat,    /* # times unable to transmit
                               collision signal */
         ets_OWC;            /* # times out of window
                               collision */
} eth_stat_t;
```

eth_ioc_setopt

```
errstat
eth_ioc_setopt(chan_cap, ethopt)
capability *chan_cap;
struct nwio_ethopt *ethopt;
```

Before a capability of an Ethernet channel can be used to send or receive Ethernet packets, it has to be configured using *eth_ioc_setopt*. The structure *nwio_ethopt* is defined in *server/ip/gen/eth_io.h*:

```
typedef struct nwio_ethopt
{
        u32_t nweo_flags;
        ether_addr_t nweo_multi, nweo_rem;
        ether_type_t nweo_type;
} nwio_ethopt_t;

#define NWEO_NOFLAGS    0x0000L
#define NWEO_ACC_MASK   0x0003L
#       define NWEO_EXCL        0x00000001L
#       define NWEO_SHARED      0x00000002L
#       define NWEO_COPY        0x00000003L
#define NWEO_LOC_MASK   0x0010L
#       define NWEO_EN_LOC      0x00000010L
#       define NWEO_DI_LOC      0x00100000L
#define NWEO_BROAD_MASK 0x0020L
#       define NWEO_EN_BROAD    0x00000020L
#       define NWEO_DI_BROAD    0x00200000L
#define NWEO_MULTI_MASK 0x0040L
#       define NWEO_EN_MULTI    0x00000040L
#       define NWEO_DI_MULTI    0x00400000L
#define NWEO_PROMISC_MASK 0x0080L
#       define NWEO_EN_PROMISC  0x00000080L
#       define NWEO_DI_PROMISC  0x00800000L
#define NWEO_REM_MASK   0x0100L
#       define NWEO_REMSPEC     0x00000100L
#       define NWEO_REMANY      0x01000000L
#define NWEO_TYPE_MASK  0x0200L
#       define NWEO_TYPESPEC    0x00000200L
#       define NWEO_TYPEANY     0x02000000L
#define NWEO_RW_MASK    0x1000L
#       define NWEO_RWDATONLY   0x00001000L
#       define NWEO_RWDATALL    0x10000000L
```

The configuration is divided in a number of sections (covered by the xx_MASK macros). Options can be set in the *nweo_flags* field. The first section (`NWEO_ACC_MASK`) controls the access to a certain Ethernet packet type. If `NWEO_EXCL` is selected then this is the only channel that can send or receive Ethernet packets of the selected type. If `NWEO_SHARED` is selected then multiple channels (which all have to select `NWEO_SHARED`) can use the same Ethernet type. They all can send packets but incoming packets will be delivered to at most one of them. If `NWEO_COPY` is selected then multiple channels have access to the same Ethernet type and all receive a copy of an incoming packet.

The `NWEO_LOC_MASK` flags control the delivery of local packets, i.e., with a destination address equal to the Ethernet address of the machine where the TCP/IP server is running. If `NWEO_EN_LOC` is selected then these packets will be delivered and with `NWEO_DI_LOC` they will be discarded.

`NWEO_BROAD_MASK`, `NWEO_MULTI_MASK`, and `NWEO_PROMISC_MASK` do the same to broadcast packets, multicast packets and promiscuous mode packets as NWEO_LOC_MASK does for local packets, except that the precise multicast address is taken from the *nweo_multi* field.

The NWEO_REM_MASK flags control whether communication is restricted to a single destination or not. NWEO_REMSPEC restricts sending and receiving of packets to the single remote computer specified in the *nweo_rem* field. NWEO_REMANY allows sending to and receiving from any remote computer.

NWEO_TYPESPEC restricts sending and receiving of Ethernet packets to the protocol type specified in *nweo_type*. (For example, the RARP protocol has type number 0x8035.) The type has to be in network byte order (see *htons* in *ip*(L)). NWEO_TYPEANY allows any type.

If the Ethernet header is completely specified by the *nweo_flags* i.e., all of NWEO_EN_LOC, NWEO_DI_BROAD, NWEO_DI_MULTI, NWEO_DI_PROMISC, NWEO_REMSPEC and NWEO_TYPESPEC are specified, then NWEO_RWDATONLY can be used to send and receive just the data part of an Ethernet packet. The header information is filled in by the server from the pre-specified information. Otherwise NWEO_RWDATALL must be used. If NWEO_RWDATALL is specified then both Ethernet header and data must be provided for each packet.

Diagnostics

The complete set of error codes is described in *ip*(L).

See Also

error(L), ip(L), ip_ip(L), ip_tcp(L), ip_udp(L), ipsvr(A).

(in libraries: libamoeba.a)

Name

ip_ip – Internet Protocol server's IP client interface stubs

Synopsis

```
#include "amoeba.h"
#include "server/ip/ip_io.h"
#include "server/ip/types.h"
#include "server/ip/gen/in.h"
#include "server/ip/gen/ip_io.h"
#include "server/ip/gen/route.h"

errstat ip_ioc_getconf(chan_cap, ipconf)
errstat ip_ioc_getopt(chan_cap, ipopt)
errstat ip_ioc_getroute(chan_cap, route)
errstat ip_ioc_setconf(chan_cap, ipconf)
errstat ip_ioc_setopt(chan_cap, ipopt)
errstat ip_ioc_setroute(chan_cap, route)
```

Description

The Internet Protocol (IP) server implements four network protocols as described in *ipsvr*(A). The routines described below give access to the IP protocol in the IP server. Access to this service is provided using two types of capabilities: a server capability and channel capabilities. The server capability is called *ip*. The server capability is used to obtain a channel to the corresponding server. This is done with *tcpip_open* (see *ip*(L)). The channel capability can be used to transfer data using the protocol implemented by the server. This can also be done with the generic *tcpip_read* and *tcpip_write* routines. Since their semantics vary slightly depending on channel type, these routines are described briefly below. The *ip_ioc_* routines are used to manage the options and status of the IP channel.

Before they are described, a brief introduction to the various types is given. The general-purpose types and rights in the capabilities are described in *ip*(L).

IP Types

server/ip/gen/in.h
  defines *ipaddr_t*, *ipproto_t* and struct ip_hdropt (*ip_hdropt_t*).

server/ip/gen/ip_io.h
  defines struct nwio_ipconf (*nwio_ipconf_t*) and struct nwio_ipopt (*nwio_ipopt_t*)

server/ip/gen/ip_hdr.h
  defines struct ip_hdr (*ip_hdr_t*)

server/ip/gen/route.h
  defines struct nwio_route (*nwio_route_t*)

General Functions

tcpip_read

```
errstat
tcpip_read(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_read reads IP packets from the IP channel of a TCP/IP server, specified by *chan_cap*, into *buffer*, which has length *nbytes*. If the packet is larger than *nbytes* the packet is truncated to the buffer size and the error `TCPIP_PACKSIZE` will be returned. It blocks until it has a complete IP packet. If successful it returns the number of bytes read. Otherwise it returns a negative error status. If the read is interrupted the server returns `STD_INTR`.

It is possible to read the data only or the data plus IP header. See *ip_ioc_setconf* below for details.

Error Conditions:
See *ip*(L) for a description of the error codes.
`TCPIP_BADMODE`
`TCPIP_PACKSIZE`
`STD_INTR`

tcpip_write

```
errstat
tcpip_write(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_write transfers the IP packet in *buffer* from the client to the TCP/IP server's IP channel specified by *chan_cap*. The IP packet to be sent is *nbytes* long. It accepts only complete packets. It computes the checksum for the packet and inserts it at the appropriate place. The call blocks until enough buffer space for the whole packet is available in the TCP/IP server. On success it returns the number of bytes written. On failure it returns a negative error code.

Error Conditions:
See *ip*(L) for a description of the error codes.
`TCPIP_BADMODE`
`TCPIP_PACKSIZE`
`STD_INTR`

IP Functions

ip_ioc_getconf

```
errstat
ip_ioc_getconf(chan_cap, ipconf)
capability *chan_cap;
struct nwio_ipconf *ipconf;
```

Ip_ioc_getconf reports the Internet address and the netmask. For the *nwio_ipconf* structure see *ip_ioc_setconf* below.

ip_ioc_getopt

```
errstat
ip_ioc_getopt(chan_cap, ipopt)
capability *chan_cap;
struct nwio_ipopt *ipopt;
```

This call returns in *ipopt* the current options of the IP channel specified by *chan_cap*. See *ip_ioc_setopt* for a description of the options.

ip_ioc_getroute

```
errstat
ip_ioc_getroute(chan_cap, route)
capability *chan_cap;
struct nwio_route *route;
```

Ip_ioc_getroute can be used to query an IP server about it is routing table. The structure *nwio_route* is defined in *server/ip/gen/route.h*:

```
typedef struct nwio_route
{
        u32_t nwr_ent_no;
        u32_t nwr_ent_count;
        ipaddr_t nwr_dest;
        ipaddr_t nwr_netmask;
        ipaddr_t nwr_gateway;
        u32_t nwr_dist;
        u32_t nwr_flags;
        u32_t nwr_pref;
         ipaddr_t nwr_ifaddr;
} nwio_route_t;
```

```
#define NWRF_EMPTY        0
#define NWRF_INUSE        1
#define NWRF_STATIC       2
#define  NWRF_UNREACHABLE  4
```

The requested entry is taken from *nwr_ent_no*. Entries are counted from 0, so the value 0 can be used for an initial query. The size of the routing table is returned in *nwr_ent_count*. The *nwr_flags* field indicates if the entry is in use (`NWRF_INUSE`) and if the entry is static (`NWRF_STATIC` using *ip_ioc_setroute*) or generated by the IP server itself. A destination can be marked unreachable using `NWRF_UNREACHABLE`. The route is described by *nwr_dest*, *nwr_netmask*, *nwr_gateway*, *nwr_dist*, and *nwr_pref*. *Nwr_dest* and *nwr_netmask* select the destination addresses. A value of 0.0.0.0 (0x0) in both *Nwr_dest* and *nwr_netmask* means every host. A value of 255.255.255.255 (0xffffffff) in *nwr_netmask* means a single host. Other values of *nwr_netmask* are netmasks for the network specified by *nwr_dest*. *Nwr_gateway* is the gateway that should be used. *Nwr_dist* is the minimal distance (i.e. if the specified network is subnetted, destinations can be at varying distances.) Packets with a ‘‘time to live’’ smaller than *nwr_dist* will not reach the destination. If two routes have equal netmask and distance fields but different gateways then the gateway with highest value in *nwr_pref* is used.

ip_ioc_setconf

```
errstat
ip_ioc_setconf(chan_cap, ipconf)
capability *chan_cap;
struct nwio_ipconf *ipconf;
```

Ip_ioc_setconf can be used to inform the IP server about its Internet address and/or its netmask. Normally an IP server will discover its Internet address using the RARP protocol. *Ip_ioc_setconf* can be used in the case that the RARP failed, or the netmask has to be changed. Note that higher level protocols (TCP and UDP) assume that the Internet address of an IP device does not change, therefore TCP and UDP stop functioning if the Internet address is changed.

The structure *nwio_ipconf* is defined in *server/ip/gen/ip_io.h*:

```
typedef struct nwio_ipconf
{
        u32_t   nwic_flags;
        ipaddr_t nwic_ipaddr;
        ipaddr_t nwic_netmask;
} nwio_ipconf_t;

#define NWIC_NOFLAGS           0x0
#define NWIC_FLAGS             0x3
#       define NWIC_IPADDR_SET          0x1
#       define NWIC_NETMASK_SET         0x2
```

The function of *nwio_ipconf* depends on the value of *nwic_flags*. If `NWIC_IPADDR_SET` is set then the Internet address will be set to *nwic_ipaddr*. If `NWIC_NETMASK_SET` is set then the Internet address will be set to *nwic_netmask*.

ip_ioc_setopt

```
errstat
ip_ioc_setopt(chan_cap, ipopt)
capability *chan_cap;
struct nwio_ipopt *ipopt;
```

Before an IP channel can be used, it has to be configured using *ip_ioc_setopt*. The structure *nwio_ipopt* is defined in *server/ip/gen/ip_io.h*:

```
typedef struct nwio_ipopt
{
        u32_t nwio_flags;
        ipaddr_t nwio_rem;
        ip_hdropt_t nwio_hdropt;
        u8_t nwio_tos;
        u8_t nwio_ttl;
        u8_t nwio_df;
        ipproto_t nwio_proto;
} nwio_ipopt_t;

#define NWIO_NOFLAGS    0x0000L
#define NWIO_ACC_MASK   0x0003L
#       define NWIO_EXCL        0x00000001L
#       define NWIO_SHARED      0x00000002L
#       define NWIO_COPY        0x00000003L
#define NWIO_LOC_MASK   0x0010L
#       define NWIO_EN_LOC      0x00000010L
#       define NWIO_DI_LOC      0x00100000L
#define NWIO_BROAD_MASK 0x0020L
#       define NWIO_EN_BROAD    0x00000020L
#       define NWIO_DI_BROAD    0x00200000L
#define NWIO_REM_MASK   0x0100L
#       define NWIO_REMSPEC     0x00000100L
#       define NWIO_REMANY      0x01000000L
#define NWIO_PROTO_MASK 0x0200L
#       define NWIO_PROTOSPEC   0x00000200L
#       define NWIO_PROTOANY    0x02000000L
#define NWIO_HDR_O_MASK 0x0400L
#       define NWIO_HDR_O_SPEC  0x00000400L
#       define NWIO_HDR_O_ANY   0x04000000L
#define NWIO_RW_MASK    0x1000L
#       define NWIO_RWDATONLY   0x00001000L
#       define NWIO_RWDATALL    0x10000000L
```

The options are divided into several categories: `NWIO_ACC_MASK`, `NWIO_LOC_MASK`, `NWIO_BROAD_MASK`, `NWIO_REM_MASK`, `NWIO_PROTO_MASK`, `NWIO_HDR_O_MASK` and

`NWIO_RW_MASK`. A channel is configured when one option of each category is set.

The options covered by `NWIO_ACC_MASK` control the number of channels that can use one IP next-level protocol. If `NWIO_EXCL` is specified then only that channel can use a certain IP next-level protocol. If `NWIO_SHARED` is set then multiple channels that all have to specify `NWIO_SHARED` can use the same IP next-level protocol, but incoming packets will be delivered to at most one channel. `NWIO_COPY` does not impose any restrictions. Every channel gets a copy of an incoming packet.

`NWIO_LOC_MASK` and `NWIO_BROAD_MASK` control the delivery of packets. If `NWIO_EN_LOC` is specified then packets that are explicitly sent to the IP server are delivered. If `NWIO_EN_BROAD` is specified then broadcast packets are delivered. Either one or both of them can be disabled with `NWIO_DI_LOC` and `NWIO_DI_BROAD`.

`NWIO_REMSPEC` can be used to restrict communication to one remote host. This host is taken from the *nwio_rem* field. If any remote host is to be allowed then `NWIO_REMANY` can be used.

`NWIO_PROTOSPEC` restricts communication to one IP next-level protocol, specified in *nwio_proto*. `NWIO_PROTOANY` allows any protocol to be sent or received.

`NWIO_HDR_O_SPEC` specifies all IP header options in advance. The values are taken from *nwio_hdropt*, *nwio_tos*, *nwio_ttl*, and *nwio_df*. *Nwio_hdropt* specifies the IP options that should be present in an outgoing packet. *Ip_hdropt_t* is defined in *server/ip/gen/in.h*:

```
typedef struct ip_hdropt
{
        u8_t iho_opt_siz;
        u8_t iho_data[IP_MAX_HDR_SIZE-IP_MIN_HDR_SIZE];
} ip_hdropt_t;
```

The *iho_opt_siz* bytes in *iho_data* are appended to the IP header. *Nwio_tos* specifies the value of the ‘‘type of service’’ bits, *nwio_ttl* gives the value of the ‘‘time to live’’ field and *nwio_df* specifies whether fragmentation is disallowed or not. `NWIO_HDR_O_ANY` specifies that the header options must be specified with each write request.

`NWIO_RWDATONLY` specifies that the header should be omitted from a write request. This option can only be used when all header fields are specified in previous options: `NWIO_EN_LOC`, `NWIO_DI_BROAD`, `NWIO_REMSPEC`, `NWIO_PROTOSPEC` and `NWIO_HDR_O_SPEC`. A read operation will also only return the data part, so the IP options will be lost.

ip_ioc_setroute

```
errstat
ip_ioc_setroute(chan_cap, route)
capability *chan_cap;
struct nwio_route *route;
```

Ip_ioc_setroute adds a route to the routing table. See *ip_ioc_getroute* above for a description of the *nwio_route* structure. The fields *nwr_ent_no* and *nwr_ent_count* are ignored.

Diagnostics

The complete set of error codes is described in *ip*(L).

See Also

error(L), ip(L), ip_eth(L), ip_tcp(L), ip_udp(L), ipsvr(A).

(in libraries: libamoeba.a)

Name

ip_tcp – Internet Protocol server's TCP client interface stubs

Synopsis

```
#include "amoeba.h"
#include "server/ip/tcp_io.h"
#include "server/ip/types.h"
#include "server/ip/gen/in.h"
#include "server/ip/gen/tcp.h"
#include "server/ip/gen/tcp_io.h"

errstat tcp_ioc_connect(chan_cap, tcpcl)
errstat tcp_ioc_getconf(chan_cap, tcpconf)
errstat tcp_ioc_getopt(chan_cap, tcpopt)
errstat tcp_ioc_listen(chan_cap, tcpcl)
errstat tcp_ioc_setconf(chan_cap, tcpconf)
errstat tcp_ioc_setopt(chan_cap, tcpopt)
errstat tcp_ioc_shutdown(chan_cap)
```

Description

The Internet Protocol (IP) server implements four network protocols, as described in *ipsvr*(A). The routines described below give access to the ‘‘Transmission Control Protocol’’ (TCP) of the IP server. TCP implements a connection-oriented, byte-stream protocol on top of IP (see RFC-793).

Access to this service is provided using two types of capabilities: a server capability and channel capabilities. The server capability is called *tcp*. The server capability is used to obtain a channel to the corresponding server. This is done with *tcpip_open* (see *ip*(L)). The channel capability can be used to transfer data using the protocol implemented by the server. This is also done with the generic *tcpip_read* and *tcpip_write* routines. Since the semantics of the *tcpip_* functions vary slightly depending on channel type, these routines are described briefly below. The *tcp_ioc_* routines are used to manage the options and status of the TCP channel.

Before these routines are described, a brief introduction to the various types is given. The general-purpose types and rights in the capabilities are described in *ip*(L).

TCP Types

server/ip/gen/tcp.h
: defines *tcpport_t* and *Tcpport_t* for use in prototypes.

server/ip/gen/tcp_io.h
: defines struct nwio_tcpconf (*nwio_tcpconf_t*), struct nwio_tcpcl (*nwio_tcpcl_t*), struct nwio_tcpatt (*nwio_tcpatt_t*) and struct nwio_tcpopt (*nwio_tcpopt_t*).

server/ip/gen/tcp_hdr.h
  defines struct tcp_hdr (*tcp_hdr_t*) and struct tcp_hdropt (*tcp_hdropt_t*).

General Functions

tcpip_read

```
errstat
tcpip_read(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_read reads data from the TCP channel specified by *chan_cap* into *buffer*. The call blocks until either *nbytes* of data are available, or an error condition is detected. On success it returns the number of bytes read. If the connection has been closed then it returns 0. If urgent data is pending and the channel is not configured for urgent data then the error code `TCPIP_URG` is returned. The channel should then be reconfigured for urgent data using *tcp_ioc_setopt*. Further *tcpip_read*s should then be done until the error code `TCPIP_NOURG` is returned, indicating that there is no more urgent data. At this point the channel can be reconfigured for normal operation.

Error Conditions:
  See *ip*(L) for a description of the error codes.
  `STD_INTR`
  `TCPIP_NOCONN`
  `TCPIP_URG`
  `TCPIP_NOURG`

tcpip_write

```
errstat
tcpip_write(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_write writes the *nbytes* of data in *buffer* to the TCP channel specified by *chan_cap*. The call blocks until enough buffer space is available to hold the entire write. Urgent data can be sent by setting the `NWTO_SND_URG` option using *tcp_ioc_setopt* and then doing the write.

Error Conditions:

See *ip*(L) for a description of the error codes.

```
STD_INTR
TCPIP_NOTCONN
TCPIP_SHUTDOWN
```

TCP Functions

tcp_ioc_connect

```
errstat
tcp_ioc_connect(chan_cap, clopt)
capability *chan_cap;
struct nwio_tcpcl *clopt;
```

Tcp_ioc_connect tries to setup a connection with a remote TCP/IP server. The channel must be fully configured (see *tcp_ioc_setconf*) and values for the local port, the remote port and the remote address must have been specified using `NWTC_LP_SET` or `NWTC_LP_SEL`, `NWTC_SET_RA` and `NWTC_SET_RP`.

tcp_ioc_getconf

```
errstat
tcp_ioc_getconf(chan_cap, tcpconf)
capability *chan_cap;
struct nwio_tcpconf *tcpconf;
```

This call reports the current configuration of a TCP channel. See *tcp_ioc_setconf* for *struct nwio_tcpconf*. The *nwtc_flags* field shows the status of the *access*, *locport*, *remaddr* and *remport* fields. *Nwtc_locaddr* contains the Internet address of the TCP/IP server. *Remaddr* contains the Internet address of the remote TCP/IP server, once it has been set with `NWTC_SET_RA` or after a successful connect or listen (see *tcp_ioc_connect* and *tcp_ioc_listen*). *Nwio_locport* contains the local TCP/IP port, set with `NWTC_LP_SET` or the selected port set with `NWTC_LP_SEL`. *Nwtc_remport* contains the TCP port of the remote TCP/IP server as set with `NWIO_SET_RP` or after a successful connect or listen (see *tcp_ioc_connect* or *tcp_ioc_listen*).

A value of 0 (zero) is reported for *nwtc_remaddr*, *nwtc_locport* or *nwtc_remport* when no value is set, either explicitly or implicitly.

tcp_ioc_getopt

```
errstat
tcp_ioc_getopt(chan_cap, tcpopt)
capability * chan_cap;
struct nwio_tcpopt *tcpopt;
```

This returns in *tcpopt* the current options for a TCP channel specified by *chan_cap*. See *tcp_ioc_setopt* for a description of the various options.

tcp_ioc_listen

```
errstat
tcp_ioc_listen(chan_cap, clopt)
capability *chan_cap;
struct nwio_tcpcl *clopt;
```

Tcp_ioc_listen waits until a remote TCP/IP server tries to connect to this channel. The channel has to be configured (see *tcp_ioc_setconf*). An additional restriction is that the local port must be set (with `NWTC_LP_SET)` or selected (with `NWTC_LP_SEL`). When a remote address is set, only connections for that host are accepted, and when a remote port is set, only connections from that port are accepted. After a successful listen, *tcp_ioc_getconf* can be used to find out the address and port of the other side.

tcp_ioc_setconf

```
errstat
tcp_ioc_setconf(chan_cap, conf)
capability *chan_cap;
struct nwio_tcpconf *conf;
```

Before a TCP channel (created by calling *tcpip_open* with a TCP server capability) can be used or options set, it must configured using *tcp_ioc_setconf*. The parameters to *tcp_ioc_setconf* are the channel capability and a *struct nwio_tcpconf* as defined in *server/ip/gen/tcp_io.h*:

```
typedef struct nwio_tcpconf
{
        unsigned long nwtc_flags;
        ipaddr_t nwtc_locaddr;
        ipaddr_t nwtc_remaddr;
        tcpport_t nwtc_locport;
        tcpport_t nwtc_remport;
} nwio_tcpconf_t;
```

```
#define NWTC_NOFLAGS    0x0000L
#define NWTC_ACC_MASK   0x0003L
#       define NWTC_EXCL        0x00000001L
#       define NWTC_SHARED      0x00000002L
#       define NWTC_COPY        0x00000003L
#define NWTC_LOCPORT_MASK       0x0030L
#       define NWTC_LP_UNSET    0x00000010L
#       define NWTC_LP_SET      0x00000020L
#       define NWTC_LP_SEL      0x00000030L
#define NWTC_REMADDR_MASK       0x0100L
#       define NWTC_SET_RA      0x00000100L
#       define NWTC_UNSET_RA    0x01000000L
#define NWTC_REMPORT_MASK       0x0200L
#       define NWTC_SET_RP      0x00000200L
#       define NWTC_UNSET_RP    0x02000000L
```

A TCP channel is considered configured when one flag in each category has been selected. Thus one of `NWTC_EXCL`, `NWTC_SHARED` or `NWTC_COPY`, one of `NWTC_LP_UNSET`, `NWTC_LP_SET` or `NWTC_LP_SEL`, one of `NWTC_SET_RA` or `NWTC_UNSET_RA`, and one of `NWTC_SET_RP` or `NWTC_UNSET_RP`.

The `NWTC_ACC_` flags control the access to a TCP port. `NWTC_EXCL` means exclusive access. An attempt to configure a channel will be denied if the same port is specified as that of a channel that requested exclusive access. `NWTC_SHARED` indicates that several channels use the same port but cooperate. If the shared mode is specified for one channel then all other channels that use the same port should also be configured with the `NWTC_SHARED` flag. `NWTC_COPY` is specified when the programmer does not care about other channels. This is the default.

The locport flags control which TCP port is used for communication. `NWTC_LP_UNSET` indicates the absence of a local port. This is the default. `NWTC_LP_SET` means that the *nwtc_locport* field contains the local port to be used by TCP. This value must be in network byte order (see *Byte Order Conversion* in *ip*(L)). `NWTC_LP_SEL` requests the TCP server to pick a port. This port will be in the range from 32768 to 65535 and will be unique.

The *remaddr* flags specify which hosts are acceptable for connections. `NWTC_SET_RA` indicates that only connections to the host specified in *nwtc_remaddr* are acceptable. *Nwtc_remaddr* should be in network byte order (see *Byte Order Conversion* in *ip*(L)). `NWTC_UNSET_RA` allows any host on the other side of a connection. This is the default.

The *remport* flags specify which remote ports are acceptable for connections. `NWTC_SET_RP` indicates that only the port specified in *nwtc_remport* is acceptable. `NWTC_UNSET_RP` allows any port on the other side of a connection. This is the default.

tcp_ioc_setopt

```
errstat
tcp_ioc_setopt(chan_cap, tcpopt)
capability * chan_cap;
struct nwio_tcpopt *tcpopt;
```

The behavior of a TCP channel can be changed using *tcp_ioc_setopt*. The parameters to *tcp_ioc_setopt* are a channel capability and a *struct nwio_tcopt* as defined in *server/ip/gen/tcp_io.h*:

```
typedef struct nwio_tcpopt
{
        u32_t nwto_flags;
} nwio_tcpopt_t;
```

The *nwto_flags* field takes the following flags:

```
#define NWTO_NOFLAG             0x0000L
#define NWTO_SND_URG_MASK       0x0001L
#       define NWTO_SND_URG     0x00000001L
#       define NWTO_SND_NOTURG  0x00010000L
#define NWTO_RCV_URG_MASK       0x0002L
#       define NWTO_RCV_URG     0x00000002L
#       define NWTO_RCV_NOTURG  0x00020000L
#define NWTO_BSD_URG_MASK       0x0004L
#       define NWTO_BSD_URG     0x00000004L
#       define NWTO_NOTBSD_URG  0x00040000L
#define NWTO_DEL_RST_MASK       0x0008L
#       define NWTO_DEL_RST     0x00000008L
#       define NWTO_NOTDEL_RST  0x00080000L
```

`NWTO_SND_URG` and `NWTO_SND_NOTURG` control the sending of urgent data. If *NWTO_SND_URG* is set, data written with *tcpip_write* will be sent as urgent data. If `NWTO_SND_NOTURG` is set, data will be sent as normal data. Similarly, `NWTO_RCV_URG` and `NWTO_RCV_NOTURG` control the reception of urgent data. If the wrong kind of data is present *tcpip_read* returns the error `TCPIP_URG` or `TCPIP_NOURG`, as appropriate.

`NWTO_BSD_URG` and `NWTO_NOTBSD_URG` control whether data is sent and received according to the BSD semantics or according to the semantics described in the RFC. These options can only be specified if a successful *tcp_ioc_connect* or *tcp_ioc_listen* has been done.

The `NWTO_DEL_RST` can be used to delay the sending of an `RST` packet in response to a `SYN` packet. Normally, when a `SYN` arrives for a port and there is no listen for that particular port, the TCP server will send an `RST` packet. This results in a ‘‘Connection Refused’’ error at the sending host. If a server repeatedly accepts connections for the same port, it can use this bit tell the TCP server not to send an `RST` immediately. When a `SYN` packet arrives and the TCP server is about to send an `RST` packet, it will look for existing channels to the same port with the bit set. If such a channel exists, the TCP server will delay the sending of the `RST` packet to give the server an opportunity to open a new channel and start a listen.

tcp_ioc_shutdown

```
errstat
tcp_ioc_shutdown(chan_cap)
capability *chan_cap;
```

Tcp_ioc_shutdown tells the TCP/IP server that no more data will be sent over the channel specified by *chan_cap*. This command can be issued when the channel is connected to a remote TCP/IP server. The TCP/IP server will tell the remote TCP/IP server and the client of the remote TCP/IP server will receive an end-of-file indication.

Diagnostics

The complete set of error codes is described in *ip*(L).

See Also

error(L), ip(L), ip_eth(L), ip_ip(L), ip_udp(L), ipsvr(A).

(in libraries: libamoeba.a)

Name

ip_udp – Internet Protocol server's UDP client interface stubs

Synopsis

```
#include "amoeba.h"
#include "server/ip/udp_io.h"
#include "server/ip/types.h"
#include "server/ip/gen/in.h"
#include "server/ip/gen/udp.h"
#include "server/ip/gen/udp_io.h"

errstat udp_close(chan_cap, flags)
errstat udp_connect(udp_cap, chan_cap, srcport, dstport, dstaddr, flags)
errstat udp_destroy(chan_cap, flags)
errstat udp_ioc_getopt(cap, udpopt)
errstat udp_ioc_setopt(cap, udpopt)
errstat udp_read_msg(chan_cap, msg, msglen, actlen, flags)
errstat udp_reconnect(chan_cap, srcport, dstport, dstaddr, flags)
errstat udp_write_msg(chan_cap, msg, msglen, flags)
```

Description

These routines give access to the UDP protocol of the IP server (see *ipsvr*(A)). Access to this service is provided using two types of capabilities: a server capability and channel capabilities. The server capability is called *udp*. The server capability is used to obtain a channel to the corresponding server. This is done with *tcpip_open* (see *ip*(L)). The channel capability can be used to transfer data using the protocol implemented by the server. This is also done with the generic *tcpip_read* and *tcpip_write* routines. Since their semantics vary slightly depending on channel type, these routines are described briefly below. There is also a more convenient interface for reading and writing described in the *UDP Library Functions* section below.

The *udp_* routines are used to manage the options and status of the UDP channel.

Before they are described, a brief introduction to the various types is given. The general-purpose types and rights in the capabilities are described in *ip*(L).

UDP Types

server/ip/gen/udp.h
    defines *udpport_t* and *Udpport_t* for use in prototypes.

server/ip/gen/udp_io.h
    defines struct nwio_udpopt (*nwio_udpopt_t*).

server/ip/gen/udp_hdr.h
    defines struct udp_hdr (*udp_hdr_t*) and struct udp_io_hdr (*udp_io_hdr_t*).

General Functions

A more convenient interface to read and write UDP packets is described in the *UDP Library Functions* section below. The following interface is described for completeness.

NB. The TCP/IP server sends and receives the *udp_io_hdr* in network byte-order. This includes the *uih_ip_opt_len* and *uih_data_len* fields. The following two *tcpip_* routines do not decode these or any other fields to the local byte-order. The *udp_read_msg* and *udp_write_msg* functions do decode the length fields of the header making them simpler and safer to use.

tcpip_read

```
errstat
tcpip_read(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_read reads UDP packets from the UDP channel specified by *chan_cap* into *buffer*, which is of size *nbytes*. The call blocks until a complete packet is available. If successful it returns the number of bytes read.

It is possible to obtain just the data from the UDP packets or the data plus UDP headers. See *udp_ioc_setopt* below for details.

Error Conditions:

See *ip*(L) for a description of the error codes.
`STD_INTR`
`TCPIP_BADMODE`

tcpip_write

```
errstat
tcpip_write(chan_cap, buffer, nbytes)
capability *chan_cap;
char *buffer;
size_t nbytes;
```

Tcpip_write writes the UDP packet in *buffer*, of size *nbytes* to the UDP channel specified by *chan_cap*.

Error Conditions:

See *ip*(L) for a description of the error codes.
`TCPIP_BADMODE`

UDP Functions

udp_ioc_getopt

```
errstat
udp_ioc_getopt(cap, conf)
capability *cap;
struct nwio_udpopt *conf;
```

Udp_ioc_getopt returns the current options that result from the default options and the options set with *udp_ioc_setopt* When `NWUO_LP_SEL` or `NWUO_LP_SET` is selected the local port is returned in *nwuo_locport*. When `NWUO_RP_SET` is selected the remote port is returned in *nwuo_remport*. The local address is always returned in *nwuo_locaddr*, and when `NWUO_RA_SET` is selected the remote address is returned in *nwuo_remaddr*.

udp_ioc_setopt

```
errstat
udp_ioc_setopt(cap, conf)
capability *cap;
struct nwio_udpopt *conf;
```

A UDP channel must be configured using *udp_ioc_setopt* before any data can be read or written. *Udp_ioc_setopt* takes two parameters, a pointer to the capability of the UDP channel and pointer to a *nwio_udpopt* structure that describes the requested configuration. The *nwio_udpopt* structure is defined in *server/ip/gen/udp_io.h* as:

```
typedef struct nwio_udpopt
{
        unsigned long nwuo_flags;
        udpport_t nwuo_locport;
        udpport_t nwuo_remport;
        ipaddr_t nwuo_locaddr;
        ipaddr_t nwuo_remaddr;
} nwio_udpopt_t;

#define NWUO_NOFLAGS            0x0000L
#define NWUO_ACC_MASK           0x0003L
#define         NWUO_EXCL               0x00000001L
#define         NWUO_SHARED             0x00000002L
#define         NWUO_COPY               0x00000003L
#define NWUO_LOCPORT_MASK       0x000CL
#define         NWUO_LP_SEL             0x00000004L
#define         NWUO_LP_SET             0x00000008L
#define         NWUO_LP_ANY             0x0000000CL
#define NWUO_LOCADDR_MASK       0x0010L
#define         NWUO_EN_LOC             0x00000010L
#define         NWUO_DI_LOC             0x00100000L
```

```
#define NWUO_BROAD_MASK         0x0020L
#define         NWUO_EN_BROAD           0x00000020L
#define         NWUO_DI_BROAD           0x00200000L
#define NWUO_REMPORT_MASK       0x0100L
#define         NWUO_RP_SET             0x00000100L
#define         NWUO_RP_ANY             0x01000000L
#define NWUO_REMADDR_MASK       0x0200L
#define         NWUO_RA_SET             0x00000200L
#define         NWUO_RA_ANY             0x02000000L
#define NWUO_RW_MASK            0x1000L
#define         NWUO_RWDATONLY          0x00001000L
#define         NWUO_RWDATALL           0x10000000L
#define NWUO_IPOPT_MASK         0x2000L
#define         NWUO_EN_IPOPT           0x00002000L
#define         NWUO_DI_IPOPT           0x20000000L
```

A UDP channel is considered configured when one flag in each category has been selected. Thus one of NWUO_EXCL, NWUO_SHARED or NWUO_COPY, one of NWUO_LP_SEL, NWUO_LP_SET or NWUO_LP_ANY, one of NWUO_EN_LOC or NWUO_DI_LOC, one of NWUO_EN_BROAD, or NWUO_DI_BROAD, one of NWUO_RP_SET, or NWUO_RP_ANY, one of NWUO_RA_SET, or NWUO_RA_ANY, one of NWUO_RWDATONLY, or NWUO_RWDATALL, and one of NWUO_EN_IPOPT, or NWUO_DI_IPOPT. The NWUO_ACC flags control the access to a certain UDP port. NWUO_EXCL means exclusive access: no other channel can use this port. NWUO_SHARED means shared access: only channels that specify shared access can use this port, and all packets that are received are handed to at most one channel. NWUO_COPY imposes no access restriction and all channels get a copy of every received packet for that port.

The NWUO_LOCPORT flags control the selection of the UDP port for this channel. NWUO_LP_SEL requests the server to pick a port. This port will be in the range from 32768 to 65535 and it will be unique. NWUO_LP_SET sets the local port to the value of the *nwuo_locport* field. NWUO_LP_ANY does not select a port. Reception of data is therefore not possible but it is possible to send data. (See *tcpip_read,* and *tcpip_write* above).

The NWUO_LOCADDR flags control the reception of packets. NWUO_EN_LOC enables the reception of packets with the local IP address as destination. NWUO_DI_LOC disables the reception of packets for the local IP address.

The NWUO_BROAD flags control the reception of broadcast packets. NWUO_EN_BROAD enables the reception of broadcast packets and NWUO_DI_BROAD disables the reception of broadcast packets.

The NWUO_REMPORT flags let the client specify one specific remote UDP port or allow any remote port. NWUO_RP_SET sets the remote UDP port to the value of *nwuo_remport*. Only packets with a matching remote port will be delivered and all packets will be sent to that port. NWUO_RP_ANY allows reception of packets from any port and when transmitting packets the remote port must be specified.

The NWUO_REMADDR flags control the remote IP address. NWUO_RA_SET sets the remote IP address to the value of *nwuo_remaddr*. Only packets from that address will be delivered and all packets will be sent to that address. NWUO_RA_ANY allows reception of packets from any host and when transmitting packets the remote host has to be specified.

The NWUO_RW flags control the format of the data to be sent or received. With NWUO_RWDATONLY only the data part of a UDP packet is sent to the server and only the data part is received from the server. This option can only be set if the header information is completely specified by the other options. Otherwise NWUO_RWDATALL must be used. The NWUO_RWDATALL mode presents the data part of a UDP packet with a header that contains the source and destination IP address, source and destination UDP ports, the IP options, etc. The server expects such a header to be present in front of the data to be transmitted.

The NWUO_IPOPT flags control the delivery and transmission of IP options. When NWUO_EN_IPOPT is set, IP options will be delivered and sent. When NWUO_DI_IPOPT is set, IP options will be stripped from received packets and no IP options will be sent.

UDP Library Functions

The following routines provide a somewhat easier to use interface to UDP than the routines described above (*tcpip_open*, *udp_ioc_setopt*, *tcpip_read* and *tcpip_write*).

udp_connect

```
errstat
udp_connect(udp_cap, chan_cap, srcport, dstport, dstaddr, flags)
capability *udp_cap;
capability *chan_cap;
udpport_t srcport;
udpport_t dstport;
ipaddr_t dstaddr;
int flags;
```

Udp_connect combines the functionality of *tcpip_open* and *udp_ioc_setopt*. A pointer to a UDP server capability should be passed in *udp_cap*, and the channel capability will be returned in the capability pointed to by *chan_cap*. If *srcport* is 0 then an unused port will be selected, otherwise the local port will be set to *srcport*. If *dstport* is non-zero then communication will be restricted to remote ports equal to *dstport*, otherwise any data can be sent to or received from any remote port. The same applies to *dstaddr*: if *dstaddr* is non-zero then only *dstaddr* can be reached. Currently no flags are defined so *flags* should be 0.

udp_reconnect

```
errstat
udp_reconnect(chan_cap, srcport, dstport, dstaddr, flags)
capability *chan_cap;
udpport_t srcport;
udpport_t dstport;
ipaddr_t dstaddr;
int flags;
```

Udp_reconnect is the same as *udp_connect* except that an existing channel capability is (re)used.

udp_read_msg

```
errstat
udp_read_msg(chan_cap, msg, msglen, actlen, flags)
capability *chan_cap;
char *msg;
int msglen;
int *actlen;
int flags;
```

Udp_read_msg delivers a UDP packet. The data part of the UDP packet is prepended with a *udp_io_hdr*. The actual length of the (possibly truncated) packet is returned in *actlen*. No flags are defined so *flags* should be 0.

udp_write_msg

```
errstat
udp_write_msg(chan_cap, msg, msglen, flags)
capability *chan_cap;
char *msg;
int msglen;
int flags;
```

A UDP packet can be sent with *udp_write_msg*. *Msg* should point to a *udp_io_hdr* followed by the data part of the UDP packet. The *uih_dst_addr* and *uih_dst_port* fields of the *udp_io_hdr* should be filled in if no values were specified in the *udp_connect*, or *udp_reconnect* call.

udp_close

```
errstat
udp_close(chan_cap, flags)
capability *chan_cap;
int flags;
```

Udp_close cleans up the administration kept by the UDP library but does not destroy the capability. This function should be used if the capability was passed to another process and should continue to exist. No flags are defined so *flags* should be 0.

udp_destroy

```
errstat
udp_destroy(chan_cap, flags)
capability *chan_cap;
int flags;
```

Udp_destroy not only cleans up the administration kept by the UDP library but also destroys the channel capability.

Diagnostics

The complete set of error codes is described in *ip*(L).

See Also

error(L), ip(L), ip_eth(L), ip_ip(L), ip_tcp(L), ipsvr(A).

Name

libmod2 – introduction to the Modula 2 libraries

Description

This document describes the modules supplied with the ACK Modula-2 compiler.

ASCII.def

Contains mnemonics for ASCII control characters

Ajax.def

Currently only contains the function ‘‘isatty’’.

```
PROCEDURE isatty(fdes: INTEGER): INTEGER;
```

Arguments.def

This module provides access to program arguments and environment.

```
VAR      Argc: CARDINAL;
```

Number of program arguments, including the program name, so it is at least 1.

```
PROCEDURE Argv( argnum : CARDINAL;
                VAR argument : ARRAY OF CHAR
              ) : CARDINAL;
```

Stores the ‘‘argnum’th’’ argument in ‘‘argument’’, and returns its length, including a terminating NULL-byte. If it returns 0, the argument was not present, and if it returns a number larger than the size of ‘‘argument’’, ‘‘argument’’ was not large enough. Argument 0 contains the program name.

```
PROCEDURE GetEnv( name : ARRAY OF CHAR;
                  VAR value : ARRAY OF CHAR
                ) : CARDINAL;
```

Searches the environment list for a string of the form ‘‘name=value’’ and stores the value in ‘‘value’’, if such a string is present. It returns the length of the ‘‘value’’ part, including a terminating NULL-byte. If it returns 0, such a string is not present, and if it returns a number larger than the size of the ‘‘value’’, ‘‘value’’ was not large enough. The string in ‘‘name’’ must be NULL-terminated.

ArraySort.def

The interface is like the qsort() interface in C, so that an array of values can be sorted. This does not mean that it has to be an ARRAY, but it does mean that the values must be consecutive in memory, and the order is the ‘‘memory’’ order. The user has to define a comparison procedure of type CompareProc. This routine gets two pointers as parameters. These are pointers to the objects that must be compared. The sorting takes place in ascending order, so that f.i. if the result of the comparison is ‘‘less’’, the first argument comes in front of the second.

```
TYPE     CompareResult = (less, equal, greater);
         CompareProc = PROCEDURE(ADDRESS, ADDRESS)
                          : CompareResult;

PROCEDURE Sort(
         base: ADDRESS;(* address of array *)
         nel: CARDINAL;(* number of elements in array *)
         size: CARDINAL;(* size of each element *)
         compar: CompareProc);(* the comparison procedure *)
```

CSP.def - Communicating Sequential Processes

See the article

A Modula-2 Implementation of CSP,
M. Collado, R. Morales, J.J. Moreno,
SIGPlan Notices, Volume 22, Number 6, June 1987.

for an explanation of the use of this module.

```
FROM SYSTEM IMPORT BYTE;
TYPE Channel;

PROCEDURE COBEGIN;
```

Beginning of a COBEGIN .. COEND structure.

```
PROCEDURE COEND;
```

End of a COBEGIN .. COEND structure.

```
PROCEDURE StartProcess(P: PROC);
```

Start an anonymous process that executes the procedure P.

```
PROCEDURE StopProcess;
```

Terminate a Process (itself).

```
PROCEDURE InitChannel(VAR ch: Channel);
```

Initialize the channel ch.

```
PROCEDURE GetChannel(ch: Channel);
```

Assign the channel ch to the process that gets it.

```
      PROCEDURE Send(data: ARRAY OF BYTE; VAR ch: Channel);
```

Send a message with the data to the channel ch.

```
      PROCEDURE Receive(VAR ch: Channel; VAR dest: ARRAY OF BYTE);
```

Receive a message from the channel ch into the dest variable.

```
      PROCEDURE SELECT(n: CARDINAL);
```

Beginning of a SELECT structure with n guards.

```
      PROCEDURE NEXTGUARD(): CARDINAL;
```

Returns an index to the next guard to be evaluated in a SELECT.

```
      PROCEDURE GUARD(cond: BOOLEAN; ch: Channel;
                      VAR dest: ARRAY OF BYTE): BOOLEAN;
```

Evaluates a guard, including reception management.

```
      PROCEDURE ENDSELECT(): BOOLEAN;
```

End of a SELECT structure.

Conversion.def

This module provides numeric-to-string conversions.

```
      PROCEDURE ConvertOctal(num, len: CARDINAL;
                             VAR str: ARRAY OF CHAR);
```

Convert number ‘‘num’’ to right-justified octal representation of ‘‘len’’ positions, and put the result in ‘‘str’’. If the result does not fit in ‘‘str’’, it is truncated on the right.

```
      PROCEDURE ConvertHex(num, len: CARDINAL;
                           VAR str: ARRAY OF CHAR);
```

Convert a hexadecimal number to a string.

```
      PROCEDURE ConvertCardinal(num, len: CARDINAL;
                       VAR str: ARRAY OF CHAR);
```

Convert a cardinal number to a string.

```
      PROCEDURE ConvertInteger(num: INTEGER;
                      len: CARDINAL;
                             VAR str: ARRAY OF CHAR);
```

Convert an integer number to a string.

EM.def

This module provides an interface to some EM instructions and data.

```
TYPE TrapHandler = PROCEDURE(INTEGER);

PROCEDURE FIF(arg1, arg2: LONGREAL;
               VAR intres: LONGREAL) : LONGREAL;
```

Multiplies arg1 and arg2, and returns the integer part of the result in ‘‘intres’’ and the fraction part as the function result.

```
PROCEDURE FEF(arg: LONGREAL; VAR exp: INTEGER) : LONGREAL;
```

Splits ‘‘arg’’ in mantissa and a base-2 exponent. The mantissa is returned, and the exponent is left in ‘‘exp’’.

```
PROCEDURE TRP(trapno: INTEGER);
```

Generate EM trap number ‘‘trapno’’.

```
PROCEDURE SIG(t: TrapHandler): TrapHandler;
```

Install traphandler t; return previous handler.

```
PROCEDURE FILN(): ADDRESS;
```

Return current program file-name. This only works if file-name and line-number generation is not disabled during compilation.

```
PROCEDURE LINO(): INTEGER;
```

return current program line-number. This only works if file-name and line-number generation is not disabled during compilation.

Epilogue.def

The procedure in this module installs module termination procedures to be called at program termination. MODULA-2 offers a facility for the initialization of modules, but there is no mechanism to have some code executed when the program finishes. This module is a feeble attempt at solving this problem.

```
PROCEDURE CallAtEnd(p: PROC): BOOLEAN;
```

Add procedure ‘‘p’’ to the list of procedures that must be executed when the program finishes. When the program finishes, these procedures are executed in the REVERSE order in which they were added to the list. This procedure returns FALSE when there are too many procedures to be called (the list has a fixed size).

InOut.def

This is Wirth’s Input/Output module.

```
CONST    EOL = 12C;

VAR      Done : BOOLEAN;
         termCH : CHAR;
```

```
PROCEDURE OpenInput(defext: ARRAY OF CHAR);
```

Request a file name from the standard input stream and open this file for reading. If the filename ends with a '.', append the ‘‘defext’’ extension. Done := ‘‘file was successfully opened’’. If open, subsequent input is read from this file.

```
PROCEDURE OpenOutput(defext : ARRAY OF CHAR);
```

Request a file name from the standard input stream and open this file for writing. If the filename ends with a '.', append the ‘‘defext’’ extension. Done := ‘‘file was successfully opened’’. If open, subsequent output is written to this file. Files left open at program termination are automatically closed.

```
PROCEDURE OpenInputFile(filename: ARRAY OF CHAR);
```

Like OpenInput, but filename given as parameter. This procedure is not in Wirth's InOut.

```
PROCEDURE OpenOutputFile(filename: ARRAY OF CHAR);
```

Like OpenOutput, but filename given as parameter. This procedure is not in Wirth's InOut.

```
PROCEDURE CloseInput;
```

Close input file. Subsequent input is read from the standard input stream.

```
PROCEDURE CloseOutput;
```

Close output file. Subsequent output is written to the standard output stream.

```
PROCEDURE Read(VAR ch : CHAR);
```

Read a character from the current input stream and leave it in ‘‘ch’’. Done := NOT ‘‘end of file’’.

```
PROCEDURE ReadString(VAR s : ARRAY OF CHAR);
```

Read a string from the current input stream and leave it in ‘‘s’’. A string is any sequence of characters not containing blanks or control characters; leading blanks are ignored. Input is terminated by any character <= ' '. This character is assigned to termCH. DEL or BACKSPACE is used for backspacing when input from terminal.

```
PROCEDURE ReadInt(VAR x : INTEGER);
```

Read a string and convert it to INTEGER. Syntax: integer = ['+'|'-'] digit {digit}. Leading blanks are ignored. Done := ‘‘integer was read’’.

```
PROCEDURE ReadCard(VAR x : CARDINAL);
```

Read a string and convert it to CARDINAL. Syntax: cardinal = digit {digit}. Leading blanks are ignored. Done := ‘‘cardinal was read’’.

```
PROCEDURE Write(ch : CHAR);
```

Write character ‘‘ch’’ to the current output stream.

```
      PROCEDURE WriteLn;
```

Terminate line.

```
      PROCEDURE WriteString(s : ARRAY OF CHAR);
```

Write string ‘‘s’’ to the current output stream

```
      PROCEDURE WriteInt(x : INTEGER; n : CARDINAL);
```

Write integer x with (at least) n characters on the current output stream. If n is greater that the number of digits needed, blanks are added preceding the number.

```
      PROCEDURE WriteCard(x, n : CARDINAL);
```

Write cardinal x with (at least) n characters on the current output stream. If n is greater that the number of digits needed, blanks are added preceding the number.

```
      PROCEDURE WriteOct(x, n : CARDINAL);
```

Write cardinal x as an octal number with (at least) n characters on the current output stream. If n is greater that the number of digits needed, blanks are added preceding the number.

```
      PROCEDURE WriteHex(x, n : CARDINAL);
```

Write cardinal x as a hexadecimal number with (at least) n characters on the current output stream. If n is greater that the number of digits needed, blanks are added preceding the number.

MathLib0.def

This module provides some mathematical functions. See MathLib.def for a more extensive one.

```
      PROCEDURE sqrt(x : REAL) : REAL;
      PROCEDURE exp(x : REAL) : REAL;
      PROCEDURE ln(x : REAL) : REAL;
      PROCEDURE sin(x : REAL) : REAL;
      PROCEDURE cos(x : REAL) : REAL;
      PROCEDURE arctan(x : REAL) : REAL;
      PROCEDURE real(x : INTEGER) : REAL;
      PROCEDURE entier(x : REAL) : INTEGER;
```

Mathlib.def - Mathematical functions

Constants:

```
pi       = 3.141592*;
twicepi  = 6.283185*;
halfpi   = 1.570796*;
quartpi  = 0.785398*;
e        = 2.718281*;
ln2      = 0.693147*;
ln10     = 2.302585*;

longpi           = 3.141592*;
longtwicepi      = 6.283185*;
longhalfpi       = 1.570796*;
longquartpi      = 0.785398*;
longe            = 2.718281*;
longln2          = 0.693147*;
longln10 = 2.302585*;
```

Basic functions:

```
PROCEDURE pow(x: REAL; i: INTEGER): REAL;
PROCEDURE longpow(x: LONGREAL; i: INTEGER): LONGREAL;
PROCEDURE sqrt(x: REAL): REAL;
PROCEDURE longsqrt(x: LONGREAL): LONGREAL;
PROCEDURE exp(x: REAL): REAL;
PROCEDURE longexp(x: LONGREAL): LONGREAL;
PROCEDURE ln(x: REAL): REAL; (* natural log *)
PROCEDURE longln(x: LONGREAL): LONGREAL; (* natural log *)
PROCEDURE log(x: REAL): REAL;         (* log with base 10 *)
PROCEDURE longlog(x: LONGREAL): LONGREAL;(* log with base 10 *)
```

Trigonometric functions (arguments in radians):

```
PROCEDURE sin(x: REAL): REAL;
PROCEDURE longsin(x: LONGREAL): LONGREAL;
PROCEDURE cos(x: REAL): REAL;
PROCEDURE longcos(x: LONGREAL): LONGREAL;
PROCEDURE tan(x: REAL): REAL;
PROCEDURE longtan(x: LONGREAL): LONGREAL;
PROCEDURE arcsin(x: REAL): REAL;
PROCEDURE longarcsin(x: LONGREAL): LONGREAL;
PROCEDURE arccos(x: REAL): REAL;
PROCEDURE longarccos(x: LONGREAL): LONGREAL;
PROCEDURE arctan(x: REAL): REAL;
PROCEDURE longarctan(x: LONGREAL): LONGREAL;
```

Hyperbolic functions:

```
PROCEDURE sinh(x: REAL): REAL;
PROCEDURE longsinh(x: LONGREAL): LONGREAL;
PROCEDURE cosh(x: REAL): REAL;
PROCEDURE longcosh(x: LONGREAL): LONGREAL;
PROCEDURE tanh(x: REAL): REAL;
PROCEDURE longtanh(x: LONGREAL): LONGREAL;
PROCEDURE arcsinh(x: REAL): REAL;
PROCEDURE longarcsinh(x: LONGREAL): LONGREAL;
PROCEDURE arccosh(x: REAL): REAL;
PROCEDURE longarccosh(x: LONGREAL): LONGREAL;
PROCEDURE arctanh(x: REAL): REAL;
PROCEDURE longarctanh(x: LONGREAL): LONGREAL;
```

Conversions:

```
PROCEDURE RadianToDegree(x: REAL): REAL;
PROCEDURE longRadianToDegree(x: LONGREAL): LONGREAL;
PROCEDURE DegreeToRadian(x: REAL): REAL;
PROCEDURE longDegreeToRadian(x: LONGREAL): LONGREAL;
```

PascalIO.def

This module provides for I/O that is essentially equivalent to the I/O provided by Pascal with ‘‘text’’, or ‘‘file of char’’. Output buffers are automatically flushed at program termination. The CloseOutput routine is just there for compatibility with earlier versions of this module.

```
CONST    Eos = 0C;
```

End of string character

```
TYPE     Text;
VAR      Input, Output: Text;
```

Standard input and standard output are available immediately. Standard output is not buffered when connected to a terminal.

```
VAR      Notext: Text;
```

Initialize your Text variables with this.

```
PROCEDURE Reset(VAR InputText: Text;
                 Filename: ARRAY OF CHAR);
```

When InputText indicates an open textfile, it is first flushed and closed. Then the file indicated by ‘‘Filename’’ is opened for reading. If this fails, a run-time error results. Otherwise, InputText is associated with the new input file.

```
PROCEDURE Rewrite(VAR OutputText: Text;
                   Filename: ARRAY OF CHAR);
```

When OutputText indicates an open textfile, it is first flushed and closed. Then the file indicated by ‘‘Filename’’ is opened for writing. If this fails, a run-time error results.

Otherwise, OutputText is associated with the new output file.

```
        PROCEDURE CloseOutput();
```

To be called at the end of the program, to flush all output buffers.

All following input routines result in a run-time error when not called with either ‘‘Input’’, or a ‘‘Text’’ value obtained by Reset. Also, the routines that actually advance the ‘‘read pointer’’, result in a run-time error when end of file is reached prematurely.

```
        PROCEDURE NextChar(InputText: Text): CHAR;
```

Returns the next character from the InputText, 0C on end of file. Does not advance the ‘‘read pointer’’, so behaves much like ‘‘input^’’ in Pascal. However, unlike Pascal, if Eoln(InputText) is TRUE, it returns the newline character, rather than a space.

```
        PROCEDURE Get(InputText: Text);
```

Advances the ‘‘read pointer’’ by one character.

```
        PROCEDURE Eoln(InputText: Text): BOOLEAN;
```

Returns TRUE if the next character from the InputText is a linefeed. Unlike Pascal however, it does not produce a run-time error when called when Eof(InputText) is TRUE.

```
        PROCEDURE Eof(InputText: Text): BOOLEAN;
```

Returns TRUE if the end of the InputText is reached.

```
        PROCEDURE ReadChar(InputText: Text; VAR Char: CHAR);
```

Read a character from the InputText, and leave the result in ‘‘Char’’. Unlike Pascal, if Eoln(InputText) is TRUE, the newline character is put in ‘‘Char’’.

```
        PROCEDURE ReadLn(InputText: Text);
```

Skip the rest of the current line from the InputText, including the linefeed.

```
        PROCEDURE ReadInteger(InputText: Text;
                              VAR Integer: INTEGER);
```

Skip leading blanks, read an optionally signed integer from the InputText, and leave the result in ‘‘Integer’’. If no integer is read, or when overflow occurs, a run-time error results. Input stops at the character following the integer.

```
        PROCEDURE ReadCardinal(InputText: Text;
                               VAR Cardinal: CARDINAL);
```

Skip leading blanks, read a cardinal from the InputText, and leave the result in ‘‘Cardinal’’. If no cardinal is read, or when overflow occurs, a run-time error results. Input stops at the character following the cardinal.

```
        PROCEDURE ReadReal(InputText: Text; VAR Real: REAL);
```

Skip leading blanks, read a real from the InputText, and leave the result in ‘‘Real’’. Syntax of a real is as follows

```
        real ::= [(+|-)] digit {digit} [. digit {digit}]
                 [ E [(+|-)] digit {digit} ]
```

If no real is read, or when overflow/underflow occurs, a run-time error results. Input stops at the character following the real.

```
PROCEDURE ReadLongReal(InputText: Text;
                         VAR Real: LONGREAL);
```

Like ReadReal, but for LONGREAL

All following output routines result in a run-time error when not called with either ‘‘Output’’, or a ‘‘Text’’ value obtained by Rewrite.

```
PROCEDURE WriteChar(OutputText: Text; Char: CHAR);
```

Writes the character ‘‘Char’’ to the OutputText.

```
PROCEDURE WriteLn(OutputText: Text);
```

Writes a linefeed to the OutputText.

```
PROCEDURE Page(OutputText: Text);
```

Writes a form-feed to the OutputText

```
PROCEDURE WriteInteger(OutputText: Text;
                         Integer: INTEGER;
                         Width: CARDINAL);
```

Write integer ‘‘Integer’’ to the OutputText, using at least ‘‘Width’’ places, blank-padding to the left if needed.

```
PROCEDURE WriteCardinal(OutputText: Text;
                 Cardinal, Width: CARDINAL);
```

Write cardinal ‘‘Cardinal’’ to the OutputText, using at least ‘‘Width’’ places, blank-padding to the left if needed.

```
PROCEDURE WriteBoolean(OutputText: Text;
                         Boolean: BOOLEAN;
                       Width: CARDINAL);
```

Write boolean ‘‘Boolean’’ to the OutputText, using at least ‘‘Width’’ places, blank-padding to the left if needed. Equivalent to

```
WriteString(OutputText, " TRUE", Width)
```

or

```
WriteString(OutputText, "FALSE", Width)
```

```
PROCEDURE WriteString(OutputText: Text;
                        String: ARRAY OF CHAR;
                        Width: CARDINAL);
```

Write string ‘‘String’’ to the OutputText, using at least ‘‘Width’’ places, blank-padding to the left if needed. The string is terminated either by the character Eos, or by the upperbound of the array ‘‘String’’.

```
PROCEDURE WriteReal(OutputText: Text;
                     Real: REAL;
                     Width, Nfrac: CARDINAL);
```

Write real ‘‘Real’’ to the OutputText. If ‘‘Nfrac’’ = 0, use scientific notation, otherwise use fixed-point notation with ‘‘Nfrac’’ digits behind the dot. Always use at least ‘‘Width’’ places, blank-padding to the left if needed.

```
PROCEDURE WriteLongReal(OutputText: Text;
                 Real: LONGREAL;
                 Width, Nfrac: CARDINAL);
```

Like WriteReal, but for LONGREAL.

Processes.def

This is the standard process module, as described by Wirth. As discussed in ‘‘Unfair Process Scheduling in Modula-2’’, by D. Hemmendinger, SIGplan Notices Volume 23 nr 3, march 1988, the scheduler in this module is unfair, in that in some circumstances ready-to-run processes never get a turn.

```
TYPE SIGNAL;

PROCEDURE StartProcess(P: PROC; n: CARDINAL);
```

Start a concurrent process with program ‘‘P’’ and workspace of size ‘‘n’’.

```
PROCEDURE SEND(VAR s: SIGNAL);
```

One process waiting for ‘‘s’’ is resumed.

```
PROCEDURE WAIT(VAR s: SIGNAL);
```

Wait for some other process to send ‘‘s’’.

```
PROCEDURE Awaited(s: SIGNAL): BOOLEAN;
```

Return TRUE if at least one process is waiting for signal ‘‘s’’.

```
PROCEDURE Init(VAR s: SIGNAL);
```

Compulsory initialization.

RealConver.def

This module provides string-to-real and real-to-string conversions.

```
PROCEDURE StringToReal(str: ARRAY OF CHAR;
                        VAR r: REAL;
                        VAR ok: BOOLEAN);
```

Convert string ‘‘str’’ to a real number ‘‘r’’ according to the syntax:

```
real ::= ['+'|'-'] digit {digit} ['.' digit {digit}]
         ['E' ['+'|'-'] digit {digit}]
```

ok := ‘‘conversion succeeded’’ Leading blanks are skipped; Input terminates with a blank or any control character.

```
PROCEDURE StringToLongReal(str: ARRAY OF CHAR;
                    VAR r: LONGREAL;
                    VAR ok: BOOLEAN);

PROCEDURE RealToString(r: REAL;
                        width, digits: INTEGER;
                        VAR str: ARRAY OF CHAR;
                        VAR ok: BOOLEAN);
```

Convert real number ‘‘r’’ to string ‘‘str’’, either in fixed-point or exponent notation. ‘‘digits’’ is the number digits to the right of the decimal point, ‘‘width’’ is the maximum width of the notation. If digits < 0, exponent notation is used, otherwise fixed-point. If fewer than ‘‘width’’ characters are needed, leading blanks are inserted. If the representation does not fit in ‘‘width’’, then ok is set to FALSE.

```
PROCEDURE LongRealToString(r: LONGREAL;
                    width, digits: INTEGER;
                    VAR str: ARRAY OF CHAR;
                    VAR ok: BOOLEAN);
```

Like RealToString, only here ‘‘r’’ is a long real.

RealInOut.def

This is an InOut module for REAL numbers.

```
VAR Done: BOOLEAN;

PROCEDURE ReadReal(VAR x: REAL);
```

Read a real number ‘‘x’’ according to the syntax:

```
real ::= ['+'|'-'] digit {digit} ['.' digit {digit}]
         [('E'|'e') ['+'|'-'] digit {digit}]
```

Done := ‘‘a number was read’’. Input terminates with a blank or any control character. When reading from a terminal, backspacing may be done by either DEL or BACKSPACE, depending on the implementation of ReadString.

```
PROCEDURE ReadLongReal(VAR x: LONGREAL);
```

Like ReadReal, but for LONGREAL.

```
PROCEDURE WriteReal(x: REAL; n: CARDINAL);
```

Write x using n characters. If fewer than n characters are needed, leading blanks are inserted.

```
    PROCEDURE WriteLongReal(x: LONGREAL; n: CARDINAL);
```

Like WriteReal, but for LONGREAL.

```
    PROCEDURE WriteFixPt(x: REAL; n, k: CARDINAL);
```

Write x in fixed-point notation usign n characters with k digits after the decimal point. If fewer than n characters are needed, leading blanks are inserted.

```
    PROCEDURE WriteLongFixPt(x: LONGREAL; n, k: CARDINAL);
```

Like WriteFixPt, but for LONGREAL.

```
    PROCEDURE WriteRealOct(x: REAL);
```

Write x in octal words.

```
    PROCEDURE WriteLongRealOct(x: LONGREAL);
```

Like WriteRealOct, but for LONGREAL.

Semaphores.def

This module provides semaphores. On systems using quasi-concurrency, the only opportunities for process switches are calls to Down and Up.

```
    TYPE Sema;

    PROCEDURE Level(s: Sema) : CARDINAL;
```

Returns current value of semaphore s.

```
    PROCEDURE NewSema(n: CARDINAL) : Sema;
```

Creates a new semaphore with initial level ‘‘n’’.

```
    PROCEDURE Down(VAR s: Sema);
```

If the value of ‘‘s’’ is > 0, then just decrement ‘‘s’’. Else suspend the current process until the semaphore becomes positive again.

```
    PROCEDURE Up(VAR s: Sema);
```

Increment the semaphore ‘‘s’’.

```
    PROCEDURE StartProcess(P: PROC; n: CARDINAL);
```

Create a new process with procedure P and workspace of size ‘‘n’’. Also transfer control to it.

Storage.def

This module provides dynamic storage allocation. As Wirth’s 3rd edition mentions both Allocate and ALLOCATE, both are included to avoid problems.

```
FROM SYSTEM IMPORT ADDRESS;

PROCEDURE ALLOCATE(VAR a : ADDRESS; size : CARDINAL);
```

Allocate an area of the given size and return the address in ‘‘a’’. If no space is available, the calling program is killed.

```
PROCEDURE Allocate(VAR a : ADDRESS; size : CARDINAL);
```

Identical to ALLOCATE.

```
PROCEDURE DEALLOCATE(VAR a : ADDRESS; size : CARDINAL);
```

Free the area at address ‘‘a’’ with the given size. The area must have been allocated by ‘‘ALLOCATE’’ or ‘‘Allocate’’, with the same size.

```
PROCEDURE Deallocate(VAR a : ADDRESS; size : CARDINAL);
```

Identical to DEALLOCATE.

```
PROCEDURE Available(size : CARDINAL) : BOOLEAN;
```

Return TRUE if a contiguous area with the given size could be allocated. Notice that this only indicates if an ALLOCATE of this size would succeed, and that it gives no indication of the total available memory.

Streams.def

This module provides sequential IO through streams. A stream is either a text stream or a binary stream, and is either in reading, writing or appending mode. By default, there are three open text streams, connected to standard input, standard output, and standard error respectively. These are text streams. When connected to a terminal, the standard output and standard error streams are linebuffered. The user can create more streams with the OpenStream call, and delete streams with the CloseStream call. Streams are automatically closed at program termination.

```
FROM SYSTEM IMPORT BYTE;

TYPE     StreamKind = (text, binary, none);
         StreamMode = (reading, writing, appending);
         StreamResult = (succeeded, illegaloperation,
                 nomemory, openfailed,
                 nostream, endoffile);
         StreamBuffering = (unbuffered, linebuffered,
                    blockbuffered);
TYPE     Stream;

VAR      InputStream, OutputStream, ErrorStream: Stream;

PROCEDURE OpenStream(VAR stream: Stream;
                      filename: ARRAY OF CHAR;
                      kind: StreamKind;
                      mode: StreamMode;
                      VAR result: StreamResult);
```

This procedure associates a stream with the file named filename. If kind = none, result is set to ‘‘illegaloperation’’. ‘‘mode’’ has one of the following values:

reading:the file is opened for reading

writing:the file is truncated or created for writing

appending:the file is opened for writing at end of file, or created for writing.

On failure, result is set to ‘‘openfailed’’. On success, result is set to ‘‘succeeded’’.

```
PROCEDURE SetStreamBuffering(stream: Stream;
                      b: StreamBuffering;
                      VAR result: StreamResult);
```

This procedure is only allowed for output streams. The three types of buffering available are unbuffered, linebuffered, and blockbuffered. When an output stream is unbuffered, the output appears as soon as written; when it is blockbuffered, output is saved up and written as a block. When it is linebuffered (only possible for text output streams), output is saved up until a newline is encountered or input is read from standard input.

```
PROCEDURE CloseStream(VAR stream: Stream;
                       VAR result: StreamResult);
```

Closes the stream. ‘‘result’’ is set to ‘‘nostream’’ if ‘‘stream’’ was not associated with a stream.

```
PROCEDURE FlushStream(stream: Stream;
                       VAR result: StreamResult);
```

Flushes the stream. ‘‘result’’ is set to ‘‘nostream’’ if ‘‘stream’’ was not associated with a stream. It is set to ‘‘illegaloperation’’ if ‘‘stream’’ is not an output or appending stream.

```
      PROCEDURE EndOfStream(stream: Stream;
                             VAR result: StreamResult): BOOLEAN;
```

Returns true if the stream is an input stream, and the end of the file has been reached. ‘‘result’’ is set to ‘‘nostream’’ if ‘‘stream’’ was not associated with a stream. It is set to ‘‘illegaloperation’’ if the stream is not an input stream.

```
      PROCEDURE Read(stream: Stream; VAR ch: CHAR;
                     VAR result: StreamResult);
```

This procedure reads a character from the stream. Certain character translations may occur, such as the mapping of the end-of-line sequence to the character 12C. ‘‘result’’ is set to ‘‘nostream’’ if ‘‘stream’’ was not associated with a stream. It is set to ‘‘endoffile’’ if EndOfStream would have returned TRUE before the call to Read. In this case, ‘‘ch’’ is set to 0C. It is set to ‘‘illegaloperation’’ if the stream is not a text input stream.

```
      PROCEDURE ReadByte(stream: Stream; VAR byte: BYTE;
                         VAR result: StreamResult);
```

This procedure reads a byte from the stream. No character translations occur. ‘‘result’’ is set to ‘‘nostream’’ if ‘‘stream’’ was not associated with a stream. It is set to ‘‘endoffile’’ if EndOfStream would have returned TRUE before the call to ReadByte. In this case, ‘‘byte’’ is set to 0C. It is set to ‘‘illegaloperation’’ if the stream is not a binary input stream.

```
      PROCEDURE ReadBytes(stream: Stream;
                          VAR bytes: ARRAY OF BYTE;
                          VAR result: StreamResult);
```

This procedure reads bytes from the stream. No character translations occur. The number of bytes is determined by the size of the parameter. ‘‘result’’ is set to ‘‘nostream’’ if ‘‘stream’’ was not associated with a stream. It is set to ‘‘endoffile’’ if there are not enough bytes left on the stream. In this case, the rest of the bytes are set to 0C. It is set to ‘‘illegaloperation’’ if the stream is not a binary input stream.

```
      PROCEDURE Write(stream: Stream; ch: CHAR;
                      VAR result: StreamResult);
```

This procedure writes a character to the stream. Certain character translations may occur, such as the mapping of a line-feed or carriage return (12C or 15C) to the end-of-line sequence of the system. ‘‘result’’ is set to ‘‘nostream’’ if ‘‘stream’’ was not associated with a stream. It is set to ‘‘illegaloperation’’ if the stream is not a text output stream.

```
      PROCEDURE WriteByte(stream: Stream; byte: BYTE;
                          VAR result: StreamResult);
```

This procedure writes a byte to the stream. No character translations occur. ‘‘result’’ is set to ‘‘nostream’’ if ‘‘stream’’ was not associated with a stream. It is set to ‘‘illegaloperation’’ if the stream is not a binary output stream.

```
      PROCEDURE WriteBytes(stream: Stream;
                           bytes: ARRAY OF BYTE;
                           VAR result: StreamResult);
```

This procedure writes bytes to the stream. No character translations occur. The number of bytes written is equal to the size of the parameter. ‘‘result’’ is set to ‘‘nostream’’ if ‘‘stream’’ was not associated with a stream. It is set to ‘‘illegaloperation’’ if the stream is not a binary output stream.

```
PROCEDURE GetPosition(stream: Stream;
                       VAR position: LONGINT;
                       VAR result: StreamResult);
```

This procedure gives the actual read/write position in ‘‘position’’. ‘‘result’’ is set to illegaloperation if ‘‘stream’’ is not a stream.

```
PROCEDURE SetPosition(stream: Stream;
                       position: LONGINT;
                       VAR result: StreamResult);
```

This procedure sets the actual read/write position to ‘‘position’’. ‘‘result’’ is set to ‘‘nostream’’ if ‘‘stream’’ was not associated with a stream. It is set to ‘‘illegaloperation’’ if the stream was opened for appending and the position is in front of the current position, or it failed for some other reason (f.i. when the stream is connected to a terminal).

```
PROCEDURE isatty(stream: Stream;
                  VAR result: StreamResult): BOOLEAN;
```

This procedure returns TRUE if the stream is connected to a terminal. ‘‘result’’ is set to ‘‘nostream’’ if ‘‘stream’’ is not a stream, and set to ‘‘illegaloperation’ if the operation failed for some other reason.

Strings.def

This module provides string manipulations. Note: truncation of strings may occur if the user does not provide large enough variables to contain the result of the operation.

Strings are of type ARRAY OF CHAR, and their length is the size of the array, unless a 0-byte occurs in the string to indicate the end of it.

```
PROCEDURE Assign(source: ARRAY OF CHAR;
                  VAR dest: ARRAY OF CHAR);
```

Assign string source to dest.

```
PROCEDURE Insert(substr: ARRAY OF CHAR;
                  VAR str: ARRAY OF CHAR;
                  inx: CARDINAL);
```

Insert the string substr into str, starting at str[inx]. If inx is equal to or greater than Length(str) then substr is appended to the end of str.

```
PROCEDURE Delete(VAR str: ARRAY OF CHAR;
                  inx, len: CARDINAL);
```

Delete len characters from str, starting at str[inx]. If inx >= Length(str) then nothing happens. If there are not len characters to delete, characters to the end of the string are deleted.

```
      PROCEDURE Pos(substr, str: ARRAY OF CHAR): CARDINAL;
```

Return the index into str of the first occurrence of substr. Pos returns a value greater than HIGH(str) of no occurrence is found.

```
      PROCEDURE Copy(str: ARRAY OF CHAR;
                     inx, len: CARDINAL;
                     VAR result: ARRAY OF CHAR);
```

Copy at most len characters from str into result, starting at str[inx].

```
      PROCEDURE Concat(s1, s2: ARRAY OF CHAR;
                       VAR result: ARRAY OF CHAR);
```

Concatenate two strings.

```
      PROCEDURE Length(str: ARRAY OF CHAR): CARDINAL;
```

Return number of characters in str.

```
      PROCEDURE CompareStr(s1, s2: ARRAY OF CHAR): INTEGER;
```

Compare two strings, return -1 if s1 < s2, 0 if s1 = s2, and 1 if s1 > s2.

Termcap.def

This module provides an interface to termcap database. Use this like the C-version. In this Modula-2 version, some of the buffers that are explicit in the C-version, are hidden. These buffers are initialized by a call to Tgetent. The ‘‘ARRAY OF CHAR’’ parameters must be NULL-terminated. You can call them with a constant string argument, as these are always NULL-terminated in our Modula-2 implementation. Unlike the C version, this version takes care of UP, BC, PC, and ospeed automatically. If Tgetent is not called by the user, it is called by the module itself, using the TERM environment variable, or ‘‘dumb’’ if TERM does not exist.

```
      TYPE     STRCAP;
               PUTPROC = PROCEDURE(CHAR);

      PROCEDURE Tgetent(name: ARRAY OF CHAR) : INTEGER;
      PROCEDURE Tgetnum(id: ARRAY OF CHAR): INTEGER;
      PROCEDURE Tgetflag(id: ARRAY OF CHAR): BOOLEAN;
      PROCEDURE Tputs(cp: STRCAP; affcnt: INTEGER; p: PUTPROC);

      PROCEDURE Tgoto(cm: STRCAP; col, line: INTEGER): STRCAP;
```

Result exists until next call to Tgoto.

```
      PROCEDURE Tgetstr(id: ARRAY OF CHAR;
                        VAR res: STRCAP) : BOOLEAN;
```

Returns FALSE if the terminal capability does not exist; Result exists until next call to Tgetent.

Terminal.def

This module provides procedures to do terminal I/O.

```
PROCEDURE Read(VAR ch : CHAR);
```

Read a character from the terminal and leave it in ch.

```
PROCEDURE BusyRead(VAR ch : CHAR);
```

Read a character from the terminal and leave it in ch. This is a non-blocking call. It returns 0C in ch if no character was typed.

```
PROCEDURE ReadAgain;
```

Causes the last character read to be returned again upon the next call of Read.

```
PROCEDURE Write(ch : CHAR);
```

Write character ch to the terminal.

```
PROCEDURE WriteLn;
```

Terminate line.

```
PROCEDURE WriteString(s : ARRAY OF CHAR);
```

Write string s to the terminal.

Traps.def

This module provides a facility for handling traps.

Constants:

```
ERRTOOLARGE      (* stack size of process too large *)
ERRTOOMANY       (* too many nested traps + handlers *)
ERRNORESULT      (* no RETURN from function procedure *)
ERRCARDOVFL      (* CARDINAL overflow *)
ERRFORLOOP       (* value of FOR-loop control variable
                  * changed in loop
                  *)
ERRCARDUVFL      (* CARDINAL underflow *)
ERRINTERNAL      (* Internal error; should not happen *)
ERRUNIXSIG       (* received Amoeba signal *)
```

Procedures:

```
TYPE     TrapHandler = EM.TrapHandler;

PROCEDURE InstallTrapHandler(t: TrapHandler): TrapHandler;
```

Install a new trap handler, and return the previous one. Parameter of trap handler is the trap number. When a trap occurs, the default trap handler is re-installed before calling the new handler.

```
      PROCEDURE Message(str: ARRAY OF CHAR);
```

Write message ‘‘str’’ on standard error, preceded by filename and linenumber if possible.

```
      PROCEDURE Trap(n: INTEGER);
```

Cause trap number ‘‘n’’ to occur.

XXTermcap.def

Like Termcap.def this module provides interface to termcap database, only this module interfaces directly to C routines.

```
      TYPE     PUTPROC = PROCEDURE(CHAR);
      VAR      PC: CHAR;
               UP, BC: ADDRESS;
               ospeed: INTEGER[0..32767];

      PROCEDURE tgetent(bp, name: ADDRESS): INTEGER;
```

‘‘name’’ must be NULL-terminated.

```
      PROCEDURE tgetnum(id: ADDRESS): INTEGER;
```

‘‘id’’ must be NULL-terminated.

```
      PROCEDURE tgetflag(id: ADDRESS): INTEGER;
```

‘‘id’’ must be NULL-terminated.

```
      PROCEDURE tgetstr(id: ADDRESS; area: ADDRESS): ADDRESS;
```

‘‘id’’ must be NULL-terminated.

```
      PROCEDURE tgoto(cm: ADDRESS; col, line: INTEGER) : ADDRESS;
```

‘‘cm’’ must be NULL-terminated.

```
      PROCEDURE tputs(cp: ADDRESS; affcnt: INTEGER; p: PUTPROC);
```

‘‘cm’’ must be NULL-terminated.

random.def

This module provides random numbers.

```
      PROCEDURE Random(): CARDINAL;
```

Return a random CARDINAL.

```
      PROCEDURE Uniform (lwb, upb: CARDINAL): CARDINAL;
```

Return CARDINALs, uniformly distributed between ‘‘lwb’’ and ‘‘upb’’. ‘‘lwb’’ must be smaller than ‘‘upb’’, or ‘‘lwb’’ is returned.

```
      PROCEDURE StartSeed(seed: CARDINAL);
```

Initialize the generator. It is not essential to call this procedure, unless it is not desired that the system pick a starting value for itself.

(in libraries: libpc.a)

Name

libpc − library of external routines for Pascal programs

Synopsis

```
const     bufsize = ?;
type      br1 =  1..bufsize;
          br2 =  0..bufsize;
          br3 = -1..bufsize;
          ok = -1..0;
          buf = packed array[br1] of char;
          alfa = packed array[1..8] of char;
          string = ^packed array[1..?] of char;
          filetype = file of ?;
          long = ?;

{all routines must be declared extern}

function  argc:integer;
function  argv(i:integer):string;
function  environ(i:integer):string;
procedure argshift;

procedure buff(var f:filetype);
procedure nobuff(var f:filetype);
procedure notext(var f:text);
procedure diag(var f:text);
procedure pcreat(var f:text; s:string);
procedure popen(var f:text; s:string);
procedure pclose(var f:filetype);

procedure trap(err:integer);
procedure encaps(procedure p; procedure q(n:integer));

function  perrno:integer;
function  uread(fd:integer; var b:buf; len:br1):br3;
function  uwrite(fd:integer; var b:buf; len:br1):br3;

function  strbuf(var b:buf):string;
function  strtobuf(s:string; var b:buf; len:br1):br2;
function  strlen(s:string):integer;
function  strfetch(s:string; i:integer):char;
procedure strstore(s:string; i:integer; c:char);

function  clock:integer;
```

Description

This library contains some often used external routines for Pascal programs. The routines can be divided into several categories:

Argument control:

argc	Gives the number of arguments provided when the program is called.
argv	Selects the specified argument from the argument list and returns a pointer to it. This pointer is nil if the index is out of bounds (<0 or >=argc).
environ	Returns a pointer to the i-th environment string (i>=0). Returns nil if i is beyond the end of the environment list.
argshift	Effectively deletes the first argument from the argument list. Its function is equivalent to *shift* in the Bourne shell: argv[2] becomes argv[1], argv[3] becomes argv[2], etc. It is a useful procedure to skip optional flag arguments. Note that the matching of arguments and files is done at the time a file is opened by a call to reset or rewrite.

Additional file handling routines:

buff	Turn on buffering of a file. Not very useful, because all files are buffered except standard output to a terminal and diagnostic output. Input files are always buffered.
nobuff	Turn off buffering of an output file. It causes the current contents of the buffer to be flushed.
notext	Only useful for input files. End of line characters are not replaced by a space and character codes out of the ASCII range (0..127) do not cause an error message.
diag	Initialize a file for output on the diagnostic output stream (fd=2). Output is not buffered.
pcreat	The same as rewrite(f), except that you must provide the filename yourself. The name must be zero terminated. Only text files are allowed.
popen	The same as reset(f), except that you must provide the filename yourself. The name must be zero terminated. Only text files are allowed.
pclose	Gives you the opportunity to close files hidden in records or arrays. All other files are closed automatically.

String handling:

strbuf	Type conversion from character array to string. It is your own responsibility that the string is zero terminated.
strtobuf	Copy string into buffer until the string terminating zero byte is found or until the buffer if full, whatever comes first. The zero byte is also copied. The number of copied characters, excluding the zero byte, is returned. So if the result is equal to the buffer length, then the end of buffer is reached before the end of string.
strlen	Returns the string length excluding the terminating zero byte.
strfetch	Fetches the i-th character from a string. There is no check against the string length.

strstore	Stores a character in a string. There is no check against string length, so this is a dangerous procedure.

Trap handling:

These routines allow you to handle almost all the possible error situations yourself. You may define your own trap handler, written in Pascal, instead of the default handler that produces an error message and quits. You may also generate traps yourself.

trap	Trap generates the trap passed as argument (0..252). The trap numbers 128..252 may be used freely. The others are reserved.
encaps	Encapsulate the execution of *p* with the trap handler *q*. Encaps replaces the previous trap handler by *q*, calls *p* and restores the previous handler when *p* returns. If, during the execution of *p*, a trap occurs, then *q* is called with the trap number as parameter. For the duration of *q* the previous trap handler is restored, so that you may handle only some of the errors in *q*. All the other errors must then be raised again by a call to *trap*. Encapsulations may be nested: you may encapsulate a procedure while executing an encapsulated routine. Jumping out of an encapsulated procedure (non-local goto) is dangerous, because the previous trap handler must be restored. Therefore, you may only jump out of procedure *p* from inside *q* and you may only jump out of one level of encapsulation. Note that *p* may not have parameters.

Ajax calls (Amoeba UNIX-emulation):

uread	Equal to the read system call. Its normal name is blocked by the standard Pascal routine read.
uwrite	As above but for *write*.
perrno	Because external data references are not possible in Pascal, this routine returns the global variable errno, indicating the result of the last system call.

Miscellaneous:

clock	Return the number of ticks of user and system time consumed by the program.

Examples

The following program presents an example of how these routines can be used. This program is equivalent to the command cat(U).

```
{$c+}
program cat(input,inp,output);
var      inp:text;
         s:string;

function argc:integer; extern;
function argv(i:integer):string; extern;
procedure argshift; extern;
function strlen(s:string):integer; extern;
function strfetch(s:string; i:integer):char; extern;

procedure copy(var fi:text);
var c:char;
begin reset(fi);
  while not eof(fi) do
  begin
    while not eoln(fi) do
    begin
      read(fi,c);
      write(c)
    end;
    readln(fi);
    writeln
  end
end;

begin  {main}
  if argc = 1 then
         copy(input)
  else
    repeat
      s := argv(1);
      if (strlen(s) = 1) and (strfetch(s,1) = ’-’)
      then copy(input)
      else copy(inp);
      argshift;
    until argc <= 1;
end.
```

Another example gives some idea of the way to manage trap handling:

```
program bigreal(output);
const EFOVFL=4;
var trapped:boolean;

procedure encaps(procedure p; procedure q(n:integer)); extern;
procedure trap(n:integer); extern;

procedure traphandler(n:integer);
begin if n=EFOVFL then trapped:=true else trap(n) end;

procedure work;
var i,j:real;
begin trapped:=false; i:=1;
  while not trapped do
    begin j:=i; i:=i*2 end;
  writeln('bigreal = ',j);
end;

begin
  encaps(work,traphandler);
end.
```

Diagnostics

Two routines may cause fatal error messages to be generated. These are:

pcreat	Rewrite error (trap 77) if the file cannot be created.
popen	Reset error (trap 76) if the file cannot be opened for reading

Files

/profile/module/ack/lib/arch/tail_pc: the library itself.

See Also

ack(U), pc(U).

(in libraries: libamoeba.a, libkernel.a)

Name

malloc – memory allocation package

Synopsis

```
#include "stdlib.h"

void * malloc(size)
void * calloc(num, elmsize)
void * realloc(ptr, newsize)

void free(ptr)
```

Description

This module provides a memory allocation package for user processes. It provides linear performance. *Malloc*, *calloc* and *realloc* all return a pointer to memory guaranteed to be sufficiently aligned for any hardware alignment restrictions on any data type. In most implementations this is 8-byte alignment.

The package maintains a free list of any memory that it has available for use. If the package has insufficient memory to satisfy a request it attempts to create a new segment with size rounded up to the nearest click size larger than or equal to the amount of memory it needs to satisfy the current request. Any unused memory in the segment is kept on a free list for satisfying further requests for memory.

Note that writing to memory immediately before or after the piece allocated will corrupt the memory administration and/or result in an exception. If a program crashes in the *malloc* package then this is almost certainly happening (although see *free* below).

Functions

malloc

```
    void *
    malloc(size)
    size_t size;
```

Malloc allocates memory and returns a pointer to a piece of memory of *size* bytes. It returns the NULL-pointer if it was unable to allocate sufficient memory.

realloc

```
void *
realloc(ptr, newsize)
void *ptr;
size_t newsize;
```

Realloc attempts to grow/shrink the piece of memory pointed to by *ptr* to the size *newsize*. *Ptr* must be a pointer returned by either *malloc*, *calloc* or *realloc*. If it has to grow the piece of memory and there is no space to extend the current piece, it will have to allocate a totally new piece of memory and copy the data from the original piece. The original piece of memory will go onto the free list in this case. Therefore it is not wise to use *realloc* on memory which has pointers pointing into it.

Realloc returns the NULL-pointer if it cannot reallocate the memory and a pointer to the reallocated memory if it could.

calloc

```
void *
calloc(num, elmsize)
size_t num;
size_t elmsize;
```

Calloc allocates memory using *malloc* sufficient to hold *num* elements, each of *elmsize* bytes and initializes the memory to zero. It returns the return value of *malloc*.

free

```
void
free(ptr)
void *ptr;
```

Free is used to return memory allocated by one of the above three routines to the free list so that it can be reused in subsequent memory allocation requests. *Ptr* must be a pointer returned by one of the above three routines. It is an error to *free* the same piece of memory twice. This error will not be detected but it will probably result in an exception. Unlike with some other memory allocation packages, once a piece of memory has been *free*d it is not permitted to access it again, even before the next call to an allocation routine. When *free* is called, the package attempts to merge free blocks in its administration.

Warnings

The kernel malloc package is very similar but does not have the segment boundary restrictions. It can merge free memory blocks that span segment boundaries.

Name

monitor.h – monitoring macros and defines

Synopsis

```
#include "monitor.h"

#define MON_EVENT(what)
#define MON_SETBUF(buffer, size)

#define getreq
#define putrep
#define trans
```

Description

The monitoring package provides a way of collecting statistics from a server using a simple instrumentation mechanism. It is possible to put in calls to the macro `MON_EVENT`. Each time the macro is called during the execution of the program a counter corresponding to that ‘‘event’’ is incremented. The event is specified with an arbitrary string. The string used should be unique within the program for each distinguishable event to be monitored.

When *monitor.h* is included it redefines the RPC routines *trans*, *getreq* and *putrep* (see *rpc*(L)) so that they automatically keep a count of the number of times each is called. Furthermore, *getreq* has been modified to accept extra commands so that the statistics gathered can be requested and manipulated by the program *monitor(U).*

The monitoring information is stored in a buffer allocated using *malloc*(L). If for some reason it is not possible to use *malloc* in a program or a larger or smaller buffer is desired than is used per default then before any calls to `MON_EVENT`, *trans*, *getreq* or *putrep* a call to `MON_SETBUF` should be made with as arguments a buffer and the size of the buffer. The buffer must be statically allocated (i.e., a global variable) or allocated with *malloc*.

Example

The following shows a simple server that uses monitoring. Many details are left out to avoid obscuring the relevant part of the example. The include file *mysvr.h* contains the definitions of `CMD1` and `CMD2`.

```
#include "amoeba.h"
#include "monitor.h"
#include "mysvr.h"

extern port getport;

server()
{
    bufsize n;
    header hdr;

    for (;;) {
        hdr.h_port = getport;
        n = getreq(&hdr, NILBUF, 0);
        if (ERR_STATUS(n)) {
            MON_EVENT("getreq failed");
            continue;
        }
        switch (hdr.h_command) {
        case CMD1:
            MON_EVENT("Command one");
            do_command1(&hdr);
            break;
        case CMD2:
            MON_EVENT("Command two");
            do_command2(&hdr);
            break;
        default:
            MON_EVENT("Bad command");
            break;
        }
        putrep(&hdr, NILBUF, 0);
    }
}
```

See Also

malloc(L), monitor(U).

(in libraries: libamoeba.a)

Name

msg – message-based RPC routines

Synopsis

```
#include "rrpc/msg_rpc.h"

message  *msg_newmsg();
void     msg_delete(mp);
void     msg_rewind(mp);
int      msg_increfcount(mp);

void     msg_put(mp, type, val);
void     msg_put_int(mp, val);
void     msg_put_real(mp, val);
void     msg_put_string(mp, val);
void     msg_put_port(mp, val);

int      msg_next_type(mp)

int      msg_get(mp, type, val);
int      msg_get_int(mp, v);
int      msg_get_real(mp, v);
int      msg_get_port(mp, v);
int      msg_get_string(mp, v);

char     *MSG_SENDER_NAME(mp)
port     *msg_sender_port(mp);
int      MSG_ENTRY(mp)

int      msg_rpc(addr, entry, req_mp, reply);
message  *msg_rpc_rcv(rcv_port);
int      msg_rpc_reply(req_mp, reply_mp);

void     msg_printaccess(mp);
```

Description

The standard Amoeba communication primitives operate using user-defined buffers. The *msg_* routines provide an alternative based upon a *message* object. Messages are *typed*; a message consists of a sequence of values each tagged with its type. By using typed messages, the recipient of a message need not know beforehand the types of the values being sent to it. The use of types also provides a degree of protection against programming errors, ensuring that the contents of a message are interpreted correctly.

The supported types are denoted by integer constants as shown in the following table:

Name	Description
`TYPE_INT`	(32-bit) integer
`TYPE_REAL`	(float) floating-point
`TYPE_STRING`	NULL-terminated string
`TYPE_ADDRESS`	Amoeba port

Error codes

All routines with return type `int` return −1 upon failure.

msg_newmsg

```
message  *
msg_newmsg()
```

A message is created by invoking *msg_newmsg*. This routine returns a pointer to a message structure; this pointer is used for all subsequent references to the message. Values are stored in a message using the various *msg_put* routines. If a new message could not be allocated then this routine will not return but abort the calling program.

msg_put

```
void
msg_put(mp, type, val)
message *mp;
int type;
void *val;
```

Each invocation of a *msg_put_* routine appends the value to the end of the message. Routines are provided for each of the four types currently supported. In addition, *msg_put* may be called with the type and a pointer to the value; this routine provides a type-generic way of storing values in a message.

msg_put_int

```
void
msg_put_int(mp, val)
message *mp;
int val;
```

The integer value *val* is stored in the message pointed to by *mp*.

msg_put_real

```
void
msg_put_real(mp, val)
message *mp;
float val;
```

The floating-point value *val* is stored in the message pointed to by *mp*.

msg_put_string

```
void
msg_put_string(mp, val)
message *mp;
char *val;
```

The NULL-terminated string pointed to by *val* is stored in the message pointed to by *mp*.

msg_put_port

```
void
msg_put_port(mp, val)
message *mp;
port *val;
```

The Amoeba port value pointed to by *val* is stored in the message pointed to by *mp*.

msg_rpc

```
int
msg_rpc(addr, entry, req_mp, reply)
port *addr;
int entry;
message *req_mp;
message **reply;
```

A message-based RPC with a server is carried out by calling *msg_rpc*, passing as parameters the server's *put-port*, the procedure entry number, and the parameters for the remote procedure as encapsulated in a message. The final parameter to *msg_rpc* is an out-parameter, set upon successful return to point to a message containing the server's response. If the RPC fails for any reason then *msg_rpc* returns –1.

msg_get

```
int
msg_get(mp, type, val)
message *mp;
int type;
void *val;
```

The message received from the server is examined by using the various *msg_get* routines. The parameter *val* points to an object of sufficient size to hold a value of the indicated *type*. The value of the indicated type is extracted from the message *mp* and stored in the object pointed to by *val*.

msg_get_int

```
int
msg_get_int(mp, val)
message *mp;
int *val;
```

Reads the next value from the message, which must be an integer, and stores the value in object *val*.

msg_get_float

```
int
msg_get_float(mp, val)
message *mp;
float *val;
```

Reads the next value from the message, which must be an float, and stores the value in object *val*.

msg_get_port

```
int
msg_get_port(msg, val)
message *mp;
port *val;
```

Copies the next value from the message, which must be an Amoeba port, and stores the value in *val*.

msg_get_string

```
int
msg_get_string(msg, val)
message *mp;
char **val;
```

Sets *val* to point to the character string at the current read position within the message *msg*. The pointer is only valid for the lifetime of the message.

msg_next_type

```
int
msg_next_type(mp)
message *mp;
```

Msg_next_type may be called to determine the type of the next value; if no more values are present in the message, then *msg_next_type* returns −1.

All the *msg_get_* routines likewise return −1 if the message *mp* has been exhausted. A type mismatch occurs when the requested type and the type of the next value in the message differ. This results in a fatal library error: the program exits after printing a descriptive error message.

MSG_ENTRY

```
int
MSG_ENTRY(mp)
message *mp;
```

The intended entry number of a message is returned by `MSG_ENTRY`.

MSG_SENDER_NAME

```
char *
MSG_SENDER_NAME(mp)
message *mp;
```

If the sender or creator of a message has called *ctx_define_context* (see *msg_rpc*(L)), then `MSG_SENDER_NAME` returns the name of the process which sent the message

msg_sender_port

```
port *
msg_sender_port(mp)
message *mp;
```

If the sender or creator of a message has called *ctx_define_context* (see *msg_rpc(L)),* then *msg_sender_port* returns a pointer to the put-port of the process which sent the message.

msg_delete

```
void
msg_delete(mp)
message *mp;
```

Once the contents of a message are no longer needed, the space consumed by the message should be freed by calling *msg_delete*. Note that a client doing an RPC should invoke *msg_delete* on both its request message and the reply message returned by *msg_rpc*. The message library maintains with each message a reference counter, only freeing the space consumed by a message when the counter drops to zero.

msg_increfcount

```
int
msg_increfcount(mp)
message *mp;
```

Increments the reference counter for message *mp*. If a message pointer variable is set to point to some message, and all other references to the message are deleted, but it is desired that the contents of the message remain, then the message reference counter should be explicitly incremented by calling *msg_increfcount*. In general, *msg_increfcount* should be called after each time some pointer is set to point to a given message, and *msg_delete* should be called each time the value of a variable pointing to the given message is changed, or becomes undefined (for example, before leaving some program scope).

msg_rewind

```
void
msg_rewind(mp)
message *mp;
```

Since the *msg_get* routines sequentially access the contents of a message, it may be desirable to *rewind* a message to the beginning. This is accomplished by calling *msg_rewind*. Separate get and put pointers are maintained for each message; *msg_rewind* resets the get pointer to the beginning of the message.

msg_rpc_rcv

```
message *
msg_rpc_rcv(getport)
port *getport;
```

To receive a remote procedure call, a program calls *msg_rpc_rcv*, passing as a parameter the *get-port*. *Msg_rpc_rcv* returns the next message received on the specified port, blocking until such a request is received. Note that the *msg_rpc_rcv* routine is a wrapper on top of *getreq*.

msg_rpc_reply

```
int
msg_rpc_reply(req_mp, rep_mp)
message *req_mp;
message *rep_mp;
```

The reply message is transmitted by calling *msg_rpc_reply*, passing as parameters the corresponding request message and the reply message. Note that the *msg_rpc_reply* routine is a wrapper on top of *putrep*.

msg_printaccess

```
void
msg_printaccess(mp)
message *mp;
```

The routine *msg_printaccess* prints the contents of the message *mp* to *stdout* in a form convenient for program debugging.

Warning

The maximum size of a message is limited to a fixed constant.

Some of the above functions may be implemented as macros.

Floating point values are only correctly passed between machines of the same architecture, and the floating-point value must occupy no more than 32 bits.

The restrictions imposed by Amoeba on the use of *getreq* and *putput* routines apply to *msg_rpc_rcv* and to *msg_rpc_reply*.

Examples

There are examples showing how these routines are used in the sources. They are in the directory *src/test/lib/amoeba*.

See Also

msg_grp(L), msg_rpc(L).

(in libraries: libamoeba.a)

Name

msg_grp – message-based group management routines

Synopsis

```
#include "rrpc/msg_rpc.h"

GROUP    *msg_grp_join(gname, options, num_domains,
                        grp_state_xfer_func, grp_state_rcv_func,
                        monitor_func, arg);
void     msg_grp_leave(gp);
message  *msg_grp_receive(gp);
int      msg_grp_send(gp, entry, msg);
void     msg_define_grp_entry(gp, entry, func, entry_name);
int      msg_grp_num_members(gp);
```

Description

These routines provide a message-oriented group communication structure, built on top of the lower-level Amoeba group facilities (see *grp*(L)). Besides being based upon typed messages, these group facilities differ from their lower-level *grp*(L) counterparts in a number of key respects. This group facility provides an atomic group create/join; if multiple processes concurrently attempt to create a group, only one instance of the group will be created. The membership of a group is automatically maintained despite the failures of individual members; group members do not call a reset procedure to reform the group after partial failures. This facility also allows group messages to be received either via asynchronous call-backs or through blocking reads.

Error codes

These routines do not return an error condition. They block until they are able to return successfully.

msg_grp_join

```
GROUP *
msg_grp_join(gname, options, num_domains, grp_state_xfer_func,
              grp_state_rcv_func, monitor_func, arg)
char *gname;
int options;
int num_domains;
message *(*grp_state_xfer_func[])(int);
int (*grp_state_rcv_func[])(int, message *);
void (*monitor_func)(GROUP *gp, void *dummy)
void *arg;
```

To join a group, a process calls *msg_grp_join*. This routine atomically creates the group if it does not already exist. *Msg_grp_join* takes as parameters the name of the group as a string, an integer variable specifying any options, and parameters used for state transfer. The only options currently supported are `MSG_GRP_MANUAL_RCV`, and `MSG_GRP_RESILIENT`. The chosen options are ANDed together. The option `MSG_GRP_RESILIENT` indicates that the group is to be fully resilient to failures (see *grp*(L)). The option `MSG_GRP_MANUAL_RCV` will be described shortly. All processes seeking to join a given group must call *msg_grp_join* with the same parameters.

When a process joins an existing group, the library is capable of atomically transferring state from the existing members of the group to the new member. This automatic state transfer feature is useful when constructing fault-tolerant applications using active replication, also known as the *state-machine* approach. If all processes belonging to the group are identical and deterministic, then all replicas will have the same state at the time the join is processed. (This follows from the deterministic nature of the processes, the fact that the group join event is totally ordered with respect to all other group communication, and the fact that Amoeba threads are scheduled non-preemptively by default.) The library causes the state from the existing members to be transferred to the new member.

To carry out a state transfer, the state is divided into a number of domains. The parameter *num_domains* specifies the number of domains to be used. The user-specified routine *grp_state_xfer_func* is called once for each domain, with the domain number being passed as the parameter. This routine is expected to return a message pointer. The *grp_state_rcv_func* is called in the new member once for each domain, with two parameters: the domain number and a pointer to the message which was generated by calling *grp_state_xfer_func* in one of the existing group members with that domain number as the parameter. If no state transfer is required when joining a group, then *num_domains* should be specified as zero. In that case, the parameters *grp_state_xfer_func* and *grp_state_rcv_func* should be specified as null-parameters.

The *msg_grp_join* routine also allows a call-back routine to be specified, to be called whenever the membership of the group changes. The specified call-back routine will be called with two parameters: the group pointer and the value given as the *arg* parameter to the *msg_grp_join* call. If a call-back routine is specified, it will be called for the first time before the *msg_grp_join* routine returns.

The *msg_grp_join* routine returns a pointer to a group structure; this handle is used subsequently to identify the group.

msg_grp_leave

```
void
msg_grp_leave(gp)
GROUP *gp;
```

A process may explicitly depart from a group by calling *msg_grp_leave*. The failure of a process automatically causes the group to be reformed without the failed member. Unlike the lower level Amoeba group routines (see *grp*(L)), processes using these group routines do not reform the group upon member failures or departures. The group routines described here automatically maintain the membership of the group.

Note that a group join is totally ordered with respect to other group communication. Moreover, just as each message sent is processed sequentially, a group join is not processed until each message ordered before the join has been accepted by the program (either by calling *msg_grp_send* or by returning from the appropriate call-back routine). It is not possible that the state transfer routines will be called while the program is in the middle of executing the message receive call-back routine.

msg_grp_send

```
void
msg_send(gp, entry, msg)
GROUP *gp;
int entry;
message *msg;
```

A message is broadcast to the group by invoking *msg_grp_send*, passing as parameters the group pointer, an entry number, and a pointer to the message to be sent to the group. How messages are handled upon receipt depends on whether or not the option `MSG_GRP_MANUAL_RCV` was specified.

If `MSG_GRP_MANUAL_RCV` is not specified, then a separate thread is created which listens for messages. When a message is received which was sent to a specified entry point, then the user-defined call-back routine for that entry point is invoked, with a pointer to the newly received message as the parameter. The message is deleted by the library after the call-back routine returns; if the message is to persist, then the call-back routine should call *msg_increfcount* (see *msg*(L)).

msg_define_grp_entry

```
void
msg_define_grp_entry(gp, entry, func, entry_name)
GROUP *gp;
int entry;
void (*func)(message *);
char *entry_name;
```

Call-back routines for accepting group messages are defined by calling *msg_define_grp_entry*, passing as parameters the group pointer, the entry number to be defined, the address of the function to be called when a message is received, and a string denoting the name of the function. Entry numbers are small integers in the range of 0 to `MAX_MSG_ENTRIES`−1. If a message is received for an entry point for which no call-back function has been defined, then the message is simply discarded.

msg_grp_receive

```
message *
msg_grp_receive(gp)
GROUP *gp;
```

Messages may be manually received, by specifying the option `MSG_GRP_MANUAL_RCV` when joining the group. In this case, no thread is created to listen for messages. Instead, messages are accepted by calling the function *msg_grp_receive*, passing as a parameter the group pointer. This routine blocks until a a message is received. It then returns a pointer to the message.

msg_grp_num_members

```
int
msg_grp_num_members(gp)
GROUP *gp;
```

The current number of members in a group may be determined by calling *msg_grp_num_members* with the address of the group structure as a parameter.

Warning

If all members of a group fail, and new processes then immediately join an identically-named group, then the underlying Amoeba group structure gets confused, resulting in incorrect behavior. This problem can only be avoided by waiting a few seconds for Amoeba to clean up the kernel group information before recreating the group.

Examples

There are examples showing how these routines are used in the sources. They are in the directory *src/test/lib/amoeba*.

See Also

grp(L), msg(L), msg_rpc(L).

(in libraries: libamoeba.a)

Name

msg_rpc – fault-tolerant message-based remote procedure call routines

Synopsis

```
#include "rrpc/msg_rpc.h"

CTXT_NOD *
ctx_define_context(context_name, replicated, num_state_domains,
                     state_xfer_routine, state_rcv_routine);
void     ctx_define_entry(entry, func, entry_name);
CTXT_NODE *ctx_lookup_context(context_name);
int      ctx_rpc(ctxtp, entry, *req_mp, **reply);
```

Description

These routines, together with the message library routines (see *msg*(L)) provide a high-level mechanism for making remote procedure calls. An application using these routines may be made fault-tolerant by following some simple rules in constructing the program, and then simply running multiple replicas of each application component. In particular, the program components must be constructed as *state-machines*. Programs must be completely deterministic, with their outputs determined solely by the sequence of messages received.

Error codes

The *ctx_rpc* routine returns −1 if the remote procedure call fails, which happens if there is a total failure of the service.

ctx_define_context

```
CTXT_NODE
*ctx_define_context(context_name, replicated, num_state_domains,
                      state_xfer_routine, state_rcv_routine)
char *context_name;
int replicated;
int num_state_domains;
message *(*state_xfer_routine[])(int);
int (*state_rcv_routine[])(int, message *);
```

A program wishing to make or receive remote procedure calls using these library routines must first identify itself to the library. A program gives itself an identity by calling the routine *ctx_define_context*. The *context_name* parameter specifies the name by which the program (or replicated set of programs) is to be known by. If only a single replica of the program is to be run, then the *replicated* parameter should have the value of false (zero); this enables certain performance optimizations to be applied. Otherwise, the value of *replicated* should be true (one).

When a new server replica joins an existing service, it is required that the new replica start processing requests in the same state as the existing replicas. This generally will require state to be transferred from the existing replicas to the new replica. The state transfer takes place by transferring a sequence of state domains, with one message per state domain. The number of state domains to be transferred is specified as the parameter *num_state_domains*.

The state transfer takes place atomically whenever a new replica joins a service which already has replicas running with the same name, as defined by *ctx_define_context*. To accomplish the state transfer, the library repeatedly invokes the specified *state_xfer_routine* in an existing member of the group, calling the routine with the domain number as the parameter. The domains are numbered from zero to the highest allowed domain number (currently, two). In the new replica, the library calls the specified *state_rcv_routine* once for each domain message generated by *state_xfer_routine*, passing as parameters the domain number and a pointer to the corresponding message.

All replicas of a service must call *ctx_define_context* with the same parameter values.

ctx_define_entry

```
void
ctx_define_entry(entry, func, entry_name)
int entry;
message *(*func)(message *);
char *entry_name;
```

To receive server requests, a program declares to the library the server routines to be called when requests are received. Server procedures are identified by their entry number; up to sixteen (16) entry points may be defined, numbered from zero to fifteen. A server procedure is declared by calling *ctx_define_entry*. The parameters are the entry number to be associated with this routine, the address of the routine to call, and the name of the routine as a string. Upon receipt of a message for this entry point, the library automatically calls the specified routine, passing a pointer to the message as the parameter. The routine returns a pointer to the message to be returned to the client.

ctx_lookup_context

```
CTXT_NODE
*ctx_lookup_context(context_name)
char *context_name;
```

To make a remote procedure call, a program must first lookup the name of the appropriate service. A service is looked up by calling *ctx_lookup_context*, passing as a parameter the name of the service as a string. *Ctx_lookup_context* returns a pointer to an internal data structure; this pointer is used from then on to identify the service.

ctx_rpc

```
int
ctx_rpc(ctxtp, entry,, req_mp, reply_mp)
CTXT_NODE *ctxtp;
int entry;
message *req_mp;
message **reply;
```

A remote procedure call is carried out by calling *ctx_rpc*, which takes four parameters. The first parameter is the pointer identifying the service, as obtained by calling *ctx_lookup_context*. The second parameter is the entry number of the procedure and the third parameter is a pointer to the message which is to be passed to that procedure. The fourth parameter is a pointer to a message pointer. The *ctx_rpc* routine sets this pointer to point to the reply message.

The main thread of a server program will typically call *thread_exit* (see *thread*(L)) after executing the necessary initialization routines. The library will automatically invoke the server routines as necessary.

Warning

Note that these routines employ the *msg* routines (see *msg*(L)).

Examples

There are examples showing how these routines are used in the sources. They are in the directory *src/test/lib/amoeba*.

See Also

msg(L), msg_grp(L), rpc(L).

(in libraries: libamoeba.a, libamunix.a)

Name

mutex − thread synchronization primitives

Synopsis

```
#include "amoeba.h"
#include "module/mutex.h"

typedef ... mutex;

void mu_init(mu)
void mu_lock(mu)
void mu_unlock(mu)
int mu_trylock(mu, maxdelay)

void sig_mu_lock(mu)
int sig_mu_trylock(mu, maxdelay)
void sig_mu_unlock(mu)
```

Description

These operations implement mutexes (also known as binary semaphores). The use of mutexes is necessary to protect data structures that are accessed concurrently by multiple threads.

A mutex has two states: unlocked and locked. The basic operations on a mutex are *lock* and *unlock*. Locking an unlocked mutex changes its state to locked. Attempting to lock an already locked mutex blocks the thread that attempts the locking until the mutex is unlocked by another thread. If multiple threads are blocked, waiting for a locked mutex to become unlocked, exactly one of those will proceed and get the lock when it is unlocked; the others will remain blocked until the mutex is unlocked again. It is legal to unlock a mutex from a different thread than the thread that locked it; this can be used to emulate a sleep−wakeup mechanism.

Mutexes are *fair*: when two threads with the same priority are waiting for the same mutex, the system guarantees that the thread that has spent the longest time in the queue will get the lock first. In case the process has threads with different priorities, the thread that has the highest priority will get the lock first.

Note that a signal can arrive during the execution of a critical section of code. To avoid signals preempting a process during a critical section the *sig_mu_* functions have been provided. If they successfully lock the mutex with *sig_mu_lock* or *sig_mu_trylock* then all further signals are delayed until the process exits the critical section with a call to *sig_mu_unlock*. When *sig_mu_unlock* is called then the signal handlers will be called for any signals that arrived during the critical section. The semantics and parameters of the *sig_mu_* versions are identical to the *mu_* versions in all other respects and so only the *mu_* versions are described below.

N.B. A stun and a signal are not the same thing in the discussion below. Signals are used between threads whereas a stun is sent to a process and the *sig_mu_* routines will not prevent the preemption of the process while it is in the critical region.

Types

The `mutex` data type declared in *amoeba.h* is an opaque data type; its only use should be to declare mutexes. All operations on mutexes must be done through the functions below.

Functions

mu_init

```
    void
    mu_init(mu)
    mutex *mu;
```

A mutex must be initialized (in the unlocked state) before use. This can be done using this function or by declaring it as a global or static variable (in bss). No error conditions are detected. Calling *mu_init* on an already initialized mutex in the locked state has an undefined effect, but it will probably have unpleasant consequences.

mu_lock

```
    void
    mu_lock(mu)
    mutex *mu;
```

This function implements the *lock* operation described above. No error conditions are detected.

mu_unlock

```
    void
    mu_unlock(mu)
    mutex *mu;
```

This function implements the *unlock* operation described above. An exception (`EXC_SYS`) occurs if the mutex was not locked by any thread.

mu_trylock

```
    int
    mu_trylock(mu, maxdelay)
    mutex *mu;
    interval maxdelay;
```

This is a variant of the lock operation that gives up when it has been blocked unsuccessfully for *maxdelay* milliseconds, when the call is interrupted by a signal (see *signals*(L)), or when the process is continued after a stun (see *process*(L)). If *maxdelay* is zero, it succeeds only if the lock operation can proceed without blocking. If *maxdelay* is negative, an infinite delay is used, but the call may still be interrupted (unlike *mu_lock*). *Maxdelay* may be rounded up to a multiple of granularity of the kernel's internal clock. The return value is zero for success, negative for failure.

Warnings

When using *mu_trylock*, always beware of premature error returns caused by continuation after a stun; this can happen to any program when run under a debugger.

The mutex operations available in *libamunix.a* have limited use. Since only a single thread of control exists, when an operation would cause a thread to block forever, *abort* is called.

Example

The following code is idiomatic for a self-initializing module whose operations may be called from multiple threads without explicit initialization. This is the rationale for the requirement that statically initialized mutexes must be valid: imagine two threads calling *mod_routine* below at the same time; both threads will find that initialization is necessary and call *init*, but only one of the calls to *init* will execute the initialization code. Once *initialized* is set, future calls to *mod_routine* will skip the call to *init* altogether.

```
static mutex mu;
static int initialized;

static void init() {
        mu_lock(&mu);
        if (!initialized) {
                "initialize the mod module"
                initialized = 1;
        }
        mu_unlock(&mu);
}

/* Function callable by users of the mod module */
mod_routine() {
        if (!initialized)
                init();
        "do something mod-ish"
}
```

See Also

process(L), semaphore(L), signals(L).

Edsger W. Dijkstra, *Cooperating Sequential Processes*.

(in libraries: libamoeba.a, libamunix.a)

Name

name – name-based functions for manipulating directories

Synopsis

```
#include "module/name.h"

errstat cwd_set(path)
errstat name_append(path, object)
char *name_breakpath(path, dir)
errstat name_create(path)
errstat name_delete(path)
errstat name_lookup(path, object)
errstat name_replace(path, object)
```

Description

This module provides a portable, name-oriented interface to directory servers. It has the advantage of being independent of the particular directory server in use, but it does not make available all the functionality of every directory server, e.g., SOAP. It is similar to the module containing the same set of functions but with the prefix ‘‘dir_’’ instead of ‘‘name_’’. The main difference is that the ‘‘dir_’’ functions take an extra argument, *origin*, which is the capability for a directory relative to which the given *path* is to be interpreted.

In each function, the *path* should be a ‘‘path name’’ that specifies a directory or directory entry. To these functions, a directory is simply a finite mapping from character-string names to capabilities. Each (name, capability) pair in the mapping is a ‘‘directory entry’’. The capability can be for any object, including another directory; this allows arbitrary directory graphs (the graphs are not required to be trees). A capability can be entered in multiple directories or several times (under different names) in the same directory, resulting in multiple links to the same object. Note that some directory servers may have more complex notions of a directory, but all that is necessary in order to access them from this module is that they satisfy the above rules.

A path name is a string of printable characters. It consists of a sequence of 0 or more ‘‘components’’, separated by ‘‘/’’ characters. Each component is a string of printable characters not containing ‘‘/’’. As a special case, the path name may begin with a ‘‘/’’, in which case the first component starts with the next character of the path name. Examples: *a/silly/path*, */profile/group/cosmo-33*.

If the path name begins with a ‘‘/’’, it is an ‘‘absolute’’ path name, otherwise it is a ‘‘relative’’ path name.

The interpretation of relative path names is relative to the current working directory. (Each Amoeba process has an independent capability for a current working directory, usually inherited from its creator — see *cwd_set* below.)

In detail, the interpretation of a path name relative to a directory *d* is as follows:

(1) If the path has no components (is a zero-length string), it specifies the directory *d*;

(2) Otherwise, the first component in the path name specifies the name of an entry in directory *d* (either an existing name, or one that is to be added to the mapping);

(3) If there are any subsequent components in the path name, the first component must map to the capability of a directory, in which case the rest of the components are recursively interpreted as a path name relative to that directory.

The interpretation of absolute path names is the same, except that the portion of the path name after the ‘‘/’’ is interpreted relative to the user’s ‘‘root’’ directory, rather than to the current working directory of the process. (Each user is given a capability for a single directory from which all other directories and objects are reached. This is called the user’s root directory.)

The components ‘‘.’’ and ‘‘..’’ have special meaning. Paths containing occurrences of these components are syntactically evaluated to a normal form not containing any such occurrences before any directory operations take place. See *path_norm*(L) for the meaning of these special components.

Errors

All functions return the error status of the operation. They all involve transactions and so in addition to the errors described below, they may return any of the standard RPC error codes.

`STD_OK`:	the operation succeeded;
`STD_CAPBAD`:	an invalid capability was used;
`STD_DENIED`:	one of the capabilities encountered while parsing the path name did not have sufficient access rights (for example, the directory in which the capability is to be installed is unwritable);
`STD_NOTFOUND`:	one of the components encountered while parsing the path name does not exist in the directory specified by the preceding components (the last component need not exist for some of the functions, but the other components must refer to existing directories).

Functions

name_append

```
errstat
name_append(path, object)
char *path;
capability *object;
```

Name_append adds the *object* capability to the directory server under *path*, where it can subsequently be retrieved by giving the same name to *name_lookup*. The path up to, but not including the last component of *path* must refer to an existing directory. This directory is modified by adding an entry consisting of the last component of *path* and the *object* capability. There must be no existing entry with the same name.

Required Rights:
    `SP_MODRGT` in the directory that is to be modified

Error Conditions:
    `STD_EXISTS`:   *name* already exists

name_replace

```
errstat
name_replace(path, object)
char *path;
capability *object;
```

Name_replace replaces the current capability stored under *path* with the specified *object* capability. The *path* must refer to an existing directory entry. This directory entry is changed to refer to the specified *object* capability as an atomic action. The entry is not first deleted then appended.

Required Rights:
    `SP_MODRGT` in the directory that is to be modified

name_breakpath

```
char *
name_breakpath(path, dir)
char *path;
capability *dir;
```

Name_breakpath stores the capability for the directory allegedly containing the object specified by *path* in *dir*, and returns the name under which the object is stored, or would be stored if it existed. It is intended for locating the directory that must be modified to install an object in the directory service under the given *path*. In detail, *name_breakpath* does the following:

If the *path* is a path name containing only one component, *name_breakpath* stores the capability for either the current working directory (if the path name is relative) or the user's root directory (if the path name is absolute) in *dir*, and returns the *path* (without any leading ''/'').

Otherwise, the *path* is parsed into two path names, the first consisting of all but the last component and the second a relative path name consisting of just the last component. *Name_breakpath* stores the capability for the directory specified by the first in *dir* and returns the second. The first path name must refer to an existing directory.

cwd_set

```
errstat
cwd_set(path)
char *path;
```

Cwd_set changes the current working directory to that specified by *path.*

Required Rights:
    `NONE`

Error Conditions:
    `STD_NOTFOUND`: the directory does not exist

name_create

```
errstat
name_create(path)
char *path;
```

Name_create creates a directory and stores its capability in a directory server under *path*. If *path* is a relative path name, the new directory is created using the server containing the current working directory, otherwise the new directory is created using the server containing the user's root directory. (Note that either of these might be different from the server that contains the directory which is to be modified by adding the new directory.)

The path up to, but not including the last component of *path* must refer to an existing directory. This directory is modified by adding an entry consisting of a new directory, named by the last component of *path*. There must be no existing entry with the same name.

Required Rights:
    `SP_MODRGT` in the directory that is to be modified

Error Conditions:
    `STD_EXISTS`: *name* already exists in the directory server

name_delete

```
errstat
name_delete(path)
char *path;
```

Name_delete deletes the entry *path* from the directory server. The object specified by the capability associated with *path* in the directory entry is not destroyed. If desired, it should be separately destroyed (see *std_destroy*(L)).

Required Rights:
    `SP_MODRGT` in the directory that is to be modified

name_lookup

```
     errstat
     name_lookup(path, object)
     char *path;
     capability *object;
```

Name_lookup finds the capability named by *path* and stores it in *object*.

Examples

```
/* Change the name /home/abc to /home/xyz: */
capability mod_dircap, object;
char *old_name = "/home/abc";
char modify_dir[NAME_MAX+1];
char *entry_name = name_breakpath(old_name, &mod_dircap);

if (entry_name != NULL) {
        strncpy(modify_dir, old_name, entry_name - old_name);
        modify_dir[entry_name - old_name] = '\0';
        if (cwd_set(modify_dir) == STD_OK &&
                name_lookup(entry_name, &object) == STD_OK &&
                name_append("xyz", &object) == STD_OK &&
                name_delete(entry_name) == STD_OK) {
                fprintf(stderr, "Successful name change\n");
                return;
        }
}
fprintf(stderr, "Error -- no name change\n");
```

See Also

direct(L).

(in libraries: libamoeba.a)

Name

newproc – create a child process without forking

Synopsis

```
int newproc(file, argv, envp, nfd, fdlist, sigignore)
int newprocp(file, argv, envp, nfd, fdlist, sigignore)
char *file;
char *argv[];
char *envp[];
int   nfd;
int   fdlist[];
long  sigignore;
```

Description

These functions are extensions of the process creation primitives *fork* and *exec* provided by *posix*(L); they are much faster but less general, and intended to optimize the common cases.

Newproc is intended to replace the common case of a *fork* call followed (in the child) by some I/O redirection, signal fiddling, and an *exec* call. *Newprocp* does almost the same, but searches its file argument in the environment variable `PATH` using the same algorithm as the shell.

The arguments are:

file, argv, envp	The first three arguments correspond to the arguments to execve.
nfd, fdlist	These arguments specify I/O redirection. If nfd is negative, the child will inherit all open files from the parent except those with the close-on-exec flag on. Otherwise, for 0 <= i < nfd, file descriptor i in the child will be dupped from file descriptor fdlist[i] in the parent, if that refers to a valid file descriptor whose close-on-exec flag is off, and closed otherwise; file descriptors nfd and higher in the child are closed.
sigignore	The last argument is a signal mask: signals corresponding to one bits in the mask will initially be ignored by the child, while the default action will be used for the others. Remember that signal i corresponds to bit i-1, as defined by the *sigmask()* macro in *signal.h*.

Return Value

The return value is the process ID of the new process. This is a child of the calling process, indistinguishable from one created by *fork* and *exec*.

Diagnostics

If no process was started, errno is set to indicate the error and –1 is returned.

Environment Variables

PATH Used by *newprocp* to search the file. If this is not set, a system-dependent default path is used (typically *:/bin:/usr/bin*).

Example

The following program starts the program given by its first argument, passing the remaining arguments on, and waits for the process to finish. Environment, file descriptors and signals are left unchanged.

```
#include "stdio.h"

main(argc, argv)
int argc;
char **argv;
{
    int pid, sts;
    if (argc < 2)
        exit(2); /* Bad usage */
    pid = newprocp(argv[1], argv+1, (char **)0,
                   -1, (int *)0, -1L);
    if (pid < 0) {
        perror(argv[1]);
        exit(1);
    }
    printf("Child pid: %d\n", pid);
    if (waitpid(pid, &sts, 0) < 0) {
        perror("waitpid");
        exit(1);
    }
    printf("Child status: %d\n", sts);
    exit(0);
}
```

See Also

exec_file(L), posix(L).

(in libraries: libamoeba.a, libamunix.a)

Name

one_way − encrypt a port

Synopsis

```
#include "amoeba.h"

one_way(p, q)
port *p, *q;
```

Description

One_way computes an encrypted version of *p* and returns the result in *q*. It keeps an internal cache of encrypted ports for efficiency reasons.

See Also

priv2pub(L), posix(L), rpc(L).

(in libraries: libamoeba.a)

Name

opencap − open a file given by capability

Synopsis

```
#include "fcntl.h"

int opencap(cap, flags)
capability *cap;
int flags;
```

Description

This function returns an open file descriptor for the file referenced by the ‘capability’ argument. The ‘flags’ argument is interpreted as for *open()* (see *posix*(L)), but only the flags `O_RDONLY`, `O_WRONLY` and `O_RDWR` are supported. Furthermore, bullet files may only be opened for reading; other files (such as tty devices) may be opened for reading and/or writing. Errors are the same as for *open()*, insofar as they are relevant to the supported flags.

Example

The following code fragment (without error handling) looks up a capability and opens it, then copies some bytes from it to standard output:

```
capability cap;
int fd, n;
char buf[100];

if (name_lookup("/foo/bar", &cap) == STD_OK) {
    fd = opencap(&cap, O_RDONLY);
    if (fd >= 0) {
         n = read(fd, buf, 100);
         write(1, buf, n);
         close(fd);
    }
}
```

See Also

posix(L).

(in libraries: libamoeba.a, libamunix.a)

Name

path – functions for processing path names and lists of path names

Synopsis

```
#include "module/path.h"

int    path_capnorm(dircap_p, path, buf, len)
int    path_cs_norm(dircapset_p, path, buf, len)
char * path_first(dirlist, filename, path_buff)
char * path_lookup(dirlist, filename, path_buff, cap_ptr)
int    path_norm(cwd, path, buf, len)
int    path_rnorm(cwd, path, buf, len)
```

Description

This is a collection of functions for parsing and translating path names and colon-separated directory lists such as the PATH environment variable. This module implements the semantics of ‘‘.’’ and ‘‘..’’ as path components, by doing syntactic evaluation of such components in the user program. The directory servers do not recognize the special meaning of these names.

path_first

```
char *
path_first(dirlist, filename, path_buff)
char *dirlist;
char *filename;
char *path_buff;
```

Given a colon-separated directory list, a file name and a pointer to a character buffer, *path_first* sets *path_buf* to the result of appending *filename* (separated by ‘‘/’’) to the first directory in the pathlist. However, if the first directory is an empty string or the file name begins with ‘‘/’’, the file name is just copied to the buffer. In any case, a pointer to the portion of *dirlist* after the first directory (and terminating ‘‘:’’) is returned, if there are remaining directories, so that the result can be used in loops that are to iterate over the directories in the list. If there are no remaining directories, NULL is returned as a terminator for such loops.

Path_first also returns NULL on any call, if *filename* begins with ‘‘/’’ (since there is no point to further calls). The user is responsible for ensuring that the buffer pointed to by *path_buff* is large enough.

path_lookup

```
char *
path_lookup(dirlist, filename, path_buff, cap_ptr)
char *dirlist;
char *filename;
char *path_buff;
capability *cap_ptr;
```

Given a colon-separated directory list (*dirlist*), a *filename*, a pointer to a char buffer (*path_buff*), and a pointer to a capability (*cap_ptr*), *path_lookup* uses *path_first* and *name_lookup* repeatedly, to search for the given name in the list of directories (in left to right order). If found, the buffer is set to the full path name (the directory concatenated with the file name, or just the file name if it begins with ''/''), and the capability is filled in with the capability for the file.

It returns the value returned by the last call to *path_first*. This can be used as the directory list in a subsequent *path_lookup* call, if the found entry is not deemed suitable, or if it is intended that all matching entries be found. The first char of *path_buff* is set to `'\0'` to indicate failure. A NULL return value does not indicate failure, only that the search went all the way to the last directory in the list. The caller is responsible for ensuring that the buffer pointed to by *path_buff* is large enough.

path_norm

```
int
path_norm(cwd, path, buf, len)
char *cwd;
char *path;
char *buf;
int len;
```

Given an absolute or relative path name (*path*) and the name of the current working directory (*cwd*), *path_norm* builds, in the given character buffer (*buf*), a normal form for the path name. This normal form has any multiple slashes mapped to one, has no trailing ''/'' and has all occurrences of ''.'' and ''`..`'' expanded away. Also, if the path does not start with ''/'', and the current working directory is non-NULL, it is prefixed to the path before calculating the normal form. Thus the path is changed from relative to absolute, if and only if a non-NULL working directory is given.

It returns −1 if a normalized path name is not obtainable, either because path is NULL, or cwd is needed and is NULL. Note: ''`/..`'' is equivalent to ''/'', and paths ending in ''/'' are accepted.

path_rnorm

```
int
path_rnorm(cwd, path, buf, len)
char *cwd;
char *path;
char *buf;
int len;
```

Just like *path_norm*, above, except that it only uses *cwd* to turn a relative path into an absolute path if the path contains enough ‘‘`..`’’ components to require looking at *cwd*. Otherwise, it acts like *path_norm* with a NULL *cwd* argument.

path_capnorm

```
int
path_capnorm(dircap_p, path, buf, len)
capability **dircap_p;
char *path;
char *buf;
int len;
```

Similar to *path_rnorm*, except that it uses a capability pointer instead of the name of the current working directory. If the capability is the same as the working directory capability, the path is normalized as if *path_rnorm* had been called with the absolute path name of the working directory as first argument. Otherwise, the path is normalized using a NULL working directory string, in which case the path must not have a leading ‘‘`..`’’.

The capability is specified as a pointer to a pointer, because it needs to be modified if a path name relative to the working directory is normalized to an absolute path name. In this case, the capability pointer is changed to point to the capability for the user’s root directory.

path_cs_norm

```
int
path_cs_norm(dircapset_p, path, buf, len)
capset **dircapset_p;
char *path;
char *buf;
int len;
```

Similar to *path_rnorm*, except that it uses a pointer to a capability-set (*dircapset_p*) instead of the name of the current working directory. If any capability in the set is the same as the working directory capability, the path is normalized as if *path_rnorm* had been called with the absolute path name of the working directory as first argument. Otherwise, the path is normalized using a NULL working directory string, in which case the path must not have a leading ‘‘`..`’’.

The capset is specified as a pointer to a pointer, because it needs to be modified if a path name relative to the working directory is normalized to an absolute path name. In this case, the capset pointer is changed to point to the capset for the user’s root directory.

See Also

soap(L).

(in libraries: libamoeba.a, libamunix.a)

Name

pd_preserve − copy a process image to a file

Synopsis

```
#include "module/proc.h"

errstat
pd_preserve(filesvr, pd, pdlen, file)
capability *filesvr;
process_d *pd;
int pdlen;
capability *file; /*out*/
```

Description

Pd_preserve stores the process descriptor *pd*, which has length *pdlen*, and copies of the segments on file server *filesvr*. It returns a capability for the new file in *file*. The resulting file is in network byte order. In theory this file is executable again, but since the capability environment and command line arguments are copied as well, the resulting program is not very useful. It is used to create core dumps, which can be inspected by debuggers.

Diagnostics

In case of an error, *pd_preserve* returns one of the STD or RPC errors defined in *stderr.h*. In particular, STD_NOMEM is returned when *malloc* fails, but unfortunately, the file server is allowed to return this error when it ran out of memory as well.

Example

This function tries to store a process descriptor in a file called *core*.

```
#include "amoeba.h"
#include "module/proc.h"
#include "stderr.h"
#include "server/bullet/bullet.h"

void dumpcore(pd)
    process_d *pd;
{
    capability filesvr, file;
    if (name_lookup(DEF_BULLETSVR, &filesvr) == STD_OK &&
        pd_preserve(&filesvr, pd, pd_size(pd), &file) == STD_OK)
            (void) name_append("core", &file);
}
```

See Also

bullet(A), pd_read(L), process(L), process_d(L).

(in libraries: libamoeba, libamunix.a, libkernel.a)

Name

pd_read − read a process descriptor from a file

Synopsis

```
#include "module/proc.h"

process_d *
pd_read(file)
capability *file;

errstat
pd_read_multiple(cap, pd_array, npd)
capability  *cap;
process_d ***pd_array;
int         *npd;
```

Description

Pd_read reads a process descriptor from a file using *b_read* (see *bullet*(L)). The argument specifies a capability for the file. It converts the data read from the file from standard byte order to native byte order using *pdmalloc* and *buf_get_pd* (see *process_d*(L)), and returns a pointer to the malloced buffer containing the converted process descriptor.

Pd_read_multiple is used in heterogeneous process startup. It reads a *set* of process descriptors, one per architecture available. The argument *cap* may be the capability for either a file or a directory. If *cap* designates a file, *pd_read_multiple* behaves like *pd_read* and it will return a single process descriptor. If *cap* designates a directory, it should contain one or more binary files named ‘‘`pd.`*arch*’’, each holding a process descriptor for architecture *arch*, respectively. The process descriptors are returned in the output parameter *pd_array* by means of a dynamically allocated array of pointers to the process descriptors. Its length is returned in the output parameter *npd*.

The segment descriptors in the process descriptors returned by *pd_read_multiple* are patched to refer to the corresponding binary files. For backward compatibility reasons, the caller of *pd_read* is supposed to do this himself.

Diagnostics

Pd_read returns a NULL-pointer if an error occurred; this includes errors reported by *b_read*, *pdmalloc* and *buf_get_pd*.

Pd_read_multiple returns `STD_OK` if it could read and allocate at least one process descriptor. Otherwise it returns a standard error code telling what went wrong.

See Also

bullet(L), exec(L), process_d(L).

(in libraries: libajax.a, libamoeba.a)

Name

posix – the POSIX operating system interface support

Synopsis

Support for POSIX library functions and header files is discussed.

Description

The Amoeba library is evolving towards as much POSIX support as possible. Because most of these functions are adequately documented by POSIX, only those functions where Amoeba's POSIX implementation differs from the standard are described. A companion document, *ansi_C*(L), provides information about functions not defined by POSIX but by the C Standard. In cases where both standards have conflicting definitions, Amoeba follows POSIX.

Many of the POSIX library functions use the services of the session server (see *session*(U)) to assist in their implementation. The session server provides shared file descriptors, process management support and pipes (although named pipes or FIFOs are implemented by *fifosvr*(A)).

Specific deviations from POSIX are listed below. It should also be noted that the library contains many reserved names that are used by the POSIX emulation itself (e.g., the Amoeba transaction primitives). They should not be overridden by applications.

For easy reference, the sections below are numbered the same way as the chapters in the POSIX standard. All supported header files are found in the header subdirectory *posix*.

Chapter 2. Definitions and General Requirements

2.2 Conformance. While Amoeba is not yet a conforming implementation of POSIX, a large number of the facilities and interfaces defined by the POSIX standard are supported, so the POSIX standard is still an important yardstick to measure application portability to Amoeba.

2.2.3.3 Common Usage C Language-Dependent System Support. Although the C compiler available on Amoeba is fully derived from the Standard Conforming ACK STD C Compiler, the Amoeba libraries still require some work, in order to reach full conformance.

2.4 General Concepts

File access permissions. Amoeba's access control mechanisms are different from and partly incompatible with those specified by POSIX. Access rights for an object depend on the capability presented to the server; the *soap*(A) directory service strips rights from capabilities depending on the path name used. File access permissions reported by the POSIX emulation (e.g., by *stat*) are derived values at best. Execute permission is always granted.

File update time. Amoeba supports only a single time stamp for a file, which is reported as *atime*, *mtime* and *ctime*. It is the time that the capability was stored in the directory, and has a precision of minutes only.

User & group ids. No security is derived from uids and gids under Amoeba since access is controlled by capability. If some application needs to run with a particular uid and/or gid they can be set to other values in the string environment variable `_IDS`. The format is *uid:gid:euid:egid*, where *euid* and *egid* are the effective user and group ids, respectively. If `_IDS` is not defined then the uid is derived from the object number in the capability for the user's root directory (since this is unique per user) and the gid defaults to 1.

Path name resolution. Amoeba supports multiple indistinguishable links to directories. The special path name components dot and dot-dot are interpreted lexically, e.g., */foo/bar/../bletch/.* is reduced to */foo/bletch* before it is presented to the directory server. Path names relative to the current working directory are expanded using the special environment variable `_WORK`, which keeps track of its name. This environment variable is passed to child processes and updated by *chdir*.

2.8 Numerical Limits

Most of the constants mentioned in this section beyond those defined by the C standard are not yet provided in *limits.h*.

Chapter 3. Process Primitives

The following process attributes are not supported:

- Process time accounting attributes
- Session membership

P.3.1.1

- Directory stream positioning is not shared by parent and child after a *fork*.

P.3.1.2

- If the environment variable `PATH` is not present, the functions *execlp* and *execvp* use `:/bin:/usr/bin` as the search path.
- When computing the size of the arguments to *exec*, the sum is taken over the arguments including NULL-terminators, pointers and alignment bytes.
- A regular file can be executed as a shell script when it begins with a line of the form `#!` *shell optional-arg (e.g.,* `#! /bin/sh -x`*).*
- It is also possible to execute a directory containing a set of process descriptors (see *exec_file*(L)).
- An additional primitive to start a child process without forking is provided. See *newproc*(L).

P.3.3

- The implementation defines the following additional signals: `SIGAMOEBA`, `SIGSYS`, `SIGTRAP` and `SIGBUS`.
- `SIGCHLD` is not supported.
- If a subsequent occurrence of a pending signal is generated, the signal is delivered at most once.

- Job control is not implemented because the semantics required for job control signals are not all implemented.

P.3.3.2

- If the first parameter of *kill* is -1, all processes except the calling process are killed, which is what BSD does.

P.3.3.4

- An attempt to set the action for a signal that cannot be caught or ignored to SIG_DFL fails with *errno* set to `[EINVAL]`.

P.3.4.3

- *sleep* is not based on *alarm*. When a `SIGALRM` signal is scheduled and there is a handler present, *sleep* will return after the handler is finished.

Chapter 4. Process Environment

P.4.3.2

- *setsid* is a dummy since the whole concept of a session is not supported.

P.4.3.3

- No checks are made on the arguments of *setpgid* and thus no errors occur.

P.4.5.1

- If the function *time* cannot reach the time-of-day server, it returns -1, clears the variable pointed to by its argument if not a NULL-pointer, and sets *errno* to `[EIO]`.

P.4.5.2

- The function *times* returns a the number of milliseconds since system start-up. All fields of the supplied buffer are set to zero.

Chapter 5. Files and Directories

P.5.1.2

- The function *readdir* does not return the entries dot and dot-dot.
- After a *fork* both parent and child can use a previously opened directory stream without interfering each other.

P.5.2.2

- If the buffer supplied to to *getcwd* is `NULL`, *errno* is set to `[EINVAL]` and `NULL` is returned.

P.5.3.10

- When a disk file is concurrently opened for writing using separate *open* calls, only the version that is closed last will be retained.
- When a file is opened for reading, the version of the file that existed at the time the *open* call was made is accessed. Updates to the file while the file remains open for reading will not be seen by the reader.
- When a process that has a file open for writing crashes, the updated version of the file is not always committed to disk. In all cases, either the old or the new version will exist, never an intermediate version.
- There is no concept of file ownership.
- Directories cannot be opened for reading with *open*; *opendir* should be used instead.
- When a FIFO is opened with `O_RDWR`, the call to *open* fails with *errno* set to `[EINVAL].`
- When a file is opened with `O_CREAT` and the mode contains other bits than the file permission bits, these bits are ignored.
- If `O_EXCL` is set without `O_CREAT`, the call to *open* fails if the file exists.
- If *open* is called with `O_TRUNC` in combination with `O_RDONLY`, the `O_TRUNC` flag is ignored.

P.5.3.3

- *umask* ignores bits other than file permission bits.

P.5.3.4

- Links have totally different semantics. A link almost completely behaves as a copy of the file, only cheaper. The exception is that the Amoeba *std_destroy* request (see *std*(L)) destroys a file, breaking all unmodified links to it.
- Using *link* on arguments that name directories fails with *errno* set to `[EPERM]`.

P.5.4.1

- *mkdir* ignores bits other than file permission bits in its mode parameter.

P.5.4.2

- *mkfifo* ignores bits other than file permission bits in its mode parameter.

P.5.5.1

- Amoeba does not maintain file link counts; unlinked files stay around until garbage-collected or explicitly destroyed with *std_destroy* (see *std*(L)). When a file is updated via the Amoeba POSIX emulation, any old version is automatically destroyed. Note this also makes any link to the old version invalid. The reason for not relying fully on garbage collection in this case, is that each update of a big log file would create an unreachable copy, taking up lots of disk space. Garbage collection could be just too slow when this happens regularly.
- When the argument of *unlink* names a directory, the call fails with *errno* set to

`[EPERM].`

P.5.5.2

- *rmdir* is successful when *path* is a directory that is in use by another process or if it is the current working directory of the calling process.

P.5.6.2

- The functions *stat* and *fstat* return dummy data for certain members:

st_mode	Always reports read, write, modify permission for owner, group and others (the actual permissions on a file are only revealed when it is opened).
st_ino	Set to a hash function of the capability's private part (object number and random check word).
st_dev	Set to a hash function of the capability's port.
st_nlink	Set to one.
st_uid, st_gid	Set to one.
st_atime, st_mtime, st_ctime	All three are set to the time when the entry was last placed in its directory.

- The file times have an accurracy of about one minute.
- Currently *stat* requires read permission for the file that is inspected.
- Besides the required members, the stat structure contains the following members for BSD (layout) compatibility. The members are not fully supported.

Member type	Member name
dev_t	*st_rdev*
long	*_st_spare1*
long	*_st_spare2*
long	*_st_spare3*
long	*st_blksize*
long	*st_blocks*
long	*_st_spare4[2]*

- Capabilities of objects that do not represent a file or directory are interpreted as character special devices by *stat*, but they usually cannot be opened for reading or writing. The exception is terminal devices, which can be opened normally.

Chapter 6. Input and Output Primitives

- The semantics of calls interrupted by signals are probably incorrect.
- Not all *fcntl* options are supported: `F_GETLK`, `F_SETLK` and `F_SETLKW` are not supported; flags set with `F_SETFL` (`O_APPEND` and `O_NONBLOCK`) are not used by the system.

Chapter 7. Device- and Class-Specific Function

The 'controlling terminal' concept is implemented by the entry TTY in the capability environment of the process. The commands *ax*(U), and *login*(A) take care of setting it to the appropriate value.

Support for POSIX-style tty control is described in *tios*(L).

Chapter 8. Language-Specific Services for the C Programming Language

Remarks on the extensions listed in this chapter:

- TZ is supported (see *ctime*(L)).
- the only locale supported is 'C'.
- *fileno* and *fdopen* are supported.
- The type *sigjmp_buf* and the functions *sigsetjmp* and *siglongjmp* are not supported.
- *tzset* is supported (see *ctime*(L)).

Chapter 9. System Databases

Rudimentary support for the group and user databases exists, so programs expecting them will find them; however, the databases contain at best dummy information.

Chapter 10. Data Interchange Format

Amoeba currently only supports a restricted version of the *tar*(U) program. This program can be used to copy file hierarchies between Amoeba and POSIX systems. Other POSIX-compatible data interchange programs should be relatively easy to port as well. Given the different approaches of POSIX and Amoeba to access protection, care must be taken when moving files from one system to the other.

To copy an entire directory graph from one Amoeba system to another, the program *starch*(U) should be used instead. Besides being able to read and restore general graphs (rather than trees), it will also take care of restoring the capability protection masks to their original value.

Environment Variables

If the POSIX libraries are compiled without NDEBUG defined and the variable AJAX_DEBUG is present in the string environment, a terse message will be printed when a POSIX library function fails. Similarly, AJAX_TRACE can trigger various other debugging messages. These debugging messages are only intended when debugging the POSIX libraries and should not be considered a permanent part of the interface.

See Also

IEEE Std 1003.1-1990 (POSIX)

ansi_C(L), ctime(L), newproc(L), session(U), soap(A), std(L), tios(L).

(in libraries: libamoeba, libamunix.a)

Name

priv2pub – convert private port (get-port) to public port (put-port)

Synopsis

```
#include "amoeba.h"

void
priv2pub(priv, pub)
port *priv; /* in */
port *pub;  /* out */
```

Description

The Amoeba publications describe how a one-way function is applied by the F-box to convert a get-port to a put-port. This routine is used to implement the F-box in software. The *priv2pub* function is a one-way function (in that it is extremely difficult to invert) which is used to encrypt the port of a server.

Priv2pub converts a private port (also known as a *get-port*) to a public port (also known as *put-port*). A private port is the port used by a server in its *getreq* call; a public port is the port used by a client of that server in its *trans* call (see *rpc*(L)). When *getreq* is called the kernel calls *priv2pub* to encrypt the private port and only accepts requests to the encrypted port. Thus it is difficult for someone to pretend to be a server for which they do not know the *get-port*. Note that when *getreq* returns the *header* argument will contain the *put-port* since that is what the client sent.

It is legal for the *priv* and *pub* arguments to point to the same location. Neither should be a NULL-pointer.

See Also

one_way(L), rpc(L).

(in libraries: libamunix.a, libamoeba.a)

Name

process − kernel process server interface

Synopsis

```
#include "amoeba.h"
#include "cmdreg.h"
#include "fault.h" /* From src/h/machdep/arch/<architecture> */
#include "module/proc.h"

#define PROCESS_SVR_NAME        "..."
#define PROCESS_LIST_NAME       "..."

errstat pro_exec(svr, pd, proc_ret)
errstat pro_getdef(svr, log2ps_ret, first_ret, last_ret)
errstat pro_getload(svr, speed_ret, load_ret, mem_ret)
errstat pro_setload(svr, speed_ret, load_ret, mem_ret, exec_cap)

errstat pro_swapproc(svr, oldproc, newproc)

errstat pro_setowner(proc, owner)
errstat pro_getowner(proc, owner_ret)
errstat pro_setcomment(proc, comment)
errstat pro_getcomment(proc, buf, len)
errstat pro_stun(proc, stuncode)

#define PS_CHECKPOINT           ...
#define TERM_NORMAL             ...
#define TERM_STUNNED            ...
#define TERM_EXCEPTION          ...
```

Description

The description of the process interface is split in several parts. The process descriptor data structure and its basic access methods are described in *process_d*(L). The low-level kernel process server interface is described here. The high-level process execution interface is described in *exec_file*(L). See also *pd_read*(L).

Process Server Interface

The capability for the process server for a particular processor is found by looking up `PROCESS_SVR_NAME` (typically ‘‘proc’’) in its processor directory. An alternative way to get a process server capability is to use *exec_findhost*(L) (or, for heterogeneous process startup, *exec_multi_findhost*). The following three requests must be directed directly at this process server.

pro_exec

```
errstat
pro_exec(svr, pd, proc_ret)
capability *svr;
process_d *pd;
capability *proc_ret;
```

This starts a new process whose initial status is specified by the process descriptor pointed at by *pd*. See *process_d*(L) for a description of the contents of a process descriptor. Like almost all functions in the library with process descriptor arguments (except where explicitly specified otherwise), the argument must be in native (i.e., the current machine's) byte order; it will be converted to standard (network) byte order as needed by *pro_exec*. The capability for the new process is returned in **proc_ret*.

pro_getdef

```
errstat
pro_getdef(svr, log2ps_ret, first_ret, last_ret)
capability *svr;
int *log2ps_ret;
int *first_ret;
int *last_ret;
```

This returns three parameters of the memory management system of a process server. In **log2ps_ret* it returns the logarithm base 2 of the page size (the number *log2ps* such that *1<<log2ps* is the page size in bytes). In **first_ret* it returns the lowest page number usable in the address space of a user program. In **last_ret* it returns one more than the highest page number usable in the address space of a user program. These parameters are constants for a particular process server instance. They are expressed in pages rather than bytes to avoid overflow, since *first<<log2ps* or *last<<log2ps* may exceed the size of numbers that fit in a long int.

pro_getload

```
errstat
pro_getload(svr, speed_ret, load_ret, mem_ret)
capability *svr;
long *speed_ret;
long *load_ret;
long *mem_ret;
```

This returns three numbers that give an indication of the current load on a process server. In **speed_ret* it returns a rough indication of the CPU speed of the machine in instructions per second. The number returned is an estimate made by the kernel at boot time for a particular configuration; it should only be used to compare processors of the same architecture. In **load_ret* it returns a load average over the past few seconds, computed from the average number of runnable threads times 1024. This is a running average; more recent events have more effect. In **mem_ret* it returns the available memory, in bytes. This does not necessarily mean that a contiguous memory segment of this size is available; it is the sum of all free memory segments.

pro_sgetload

```
errstat
pro_sgetload(svr, speed_ret, load_ret, mem_ret, exec_cap)
capability *svr;
long *speed_ret;
long *load_ret;
long *mem_ret;
capability * exec_cap;
```

Pro_sgetload does all the things *pro_getload* does. In addition it also causes the kernel to modify its process server's capability. The new capability for the process server is returned via *exec_cap*. The process server accepts two capabilities at any one time: the old capability and the new capability. The *run*(A) server uses this routine to keep changing the process server's capability so that malicious users do not gain permanent access to a host just because they had access to it once.

The Process Owner Interface

The process owner (typically the process creator) is expected to run a small server loop to collect messages about any change in the process that it owns. In certain circumstances a process A may create a process B to replace itself with. If this happens the owner of process A needs to be informed that process B is now the process in question. This sort of thing happens in programs like *login*(A) which starts up a shell but needs to tell the system that it has simply mutated into a shell and has not gone away. (Otherwise the boot server will start another *login* while the user is still running a shell.) If the process owner is not interested in this information then they should still implement the server loop to avoid causing timeouts.

pro_swapproc

```
errstat
pro_swapproc(svr, oldproc, newproc)
capability *svr;
capability *oldproc;
capability *newproc;
```

This routine does a transaction with the owner capability, *svr*, of a process. It sends the capability of the original process which was owned and the capability of the new process which has replaced it.

Process Interface

The following requests must be directed at a process capability as returned by *pro_exec*. A process can obtain its own process capability using *getinfo*(L).

The process capabilities of all processes running on a particular processor can be found by looking up `PROCESS_LIST_NAME` (typically ‘‘ps’’) relative to its processor capability, and listing the contents of the directory thus found. Its directory entries have numeric names which are unique identifiers with no meaning outside this directory.

pro_setowner

```
errstat
pro_setowner(proc, owner)
capability *proc;
capability *owner;
```

This changes the owner capability of the process to the capability pointed to by *owner*. If *owner* is NULL or points to a capability with a NULL-port, the process has no owner from now on.

pro_getowner

```
errstat
pro_getowner(proc, owner_ret)
capability *proc;
capability *owner_ret;
```

This returns the owner capability of the process in the variable *owner*. If the process has no owner, *owner* is set to a capability with a NULL-port.

pro_setcomment

```
errstat
pro_setcomment(proc, comment)
capability *proc;
char *comment;
```

This sets the comment string of the process to the NULL-terminated string *comment*. The string stored with the process is truncated to a system-dependent limit (typically 31). By convention, the comment string contains the username of the user who started the process, a colon, the command name of the executing program with leading path components stripped, and the command line arguments, or as much of this as fits within the limit. A comment of this form is set automatically when a process is started by *exec_file*(L).

pro_getcomment

```
errstat
pro_getcomment(proc, buf, len)
capability *proc;
char *buf;
int len;
```

This returns the comment string of a process in the buffer *buf*, truncated to *len–1* bytes and terminated with a NULL-byte.

pro_stun

```
errstat
pro_stun(proc, stuncode)
capability *proc;
long stuncode;
```

This *stuns* the process specified by *proc*. The argument *stuncode* is made available to the process' owner when it receives a checkpoint (see below). A negative stun code implies an *emergency stop* to the kernel; this aborts transactions instead of waiting for them to complete, and is allowed even when the process is already stopped. If the owner has already received a checkpoint when an emergency stop is made, it will receive a client-to-server signal that aborts the checkpoint transaction, and another checkpoint is sent. It is not allowed to stun an already stopped process unless an emergency stop is being made. It may take a long time to stun a process; the call does not wait for the stun to complete but returns as soon as the stun is initiated.

A process may be stopped for other reasons than being stunned: when a thread causes an exception for which it has no catcher, and also when the process terminates normally.

The following happens when a process is stopped. All activities of all its threads are stopped. If a thread is serving (between *getreq* and *putrep*), this state is preserved. A client-to-server signal is sent to all servers serving outstanding transactions of the process. If an emergency stop is being made, transactions are then aborted; otherwise, they are allowed to complete and the process is not completely stopped until each server has sent a reply (or crashes). Calls to *putrep* may be aborted or not depending on the internal state of the RPC layer. Other blocking system calls (*getreq*, *mu_trylock*) are aborted. If a process has an owner, the owner is notified when all threads have stopped. This is done by constructing a process descriptor that contains the current state of the process and sending it to the owner capability in a `PS_CHECKPOINT` request, described below. If the owner returns a valid process descriptor, the process continues, unless it was stopped because of normal termination. If the process is continued, system calls that were aborted will return an error (`RPC_ABORTED` for the RPC calls), except *putrep* which returns no error code. Note that *mu_lock* will not fail; it is implemented as a library routine that retries *mu_trylock* until it succeeds (see *mutex*(L)). If the process has no owner, or the process cannot be continued after the owner replies, the stopped process is discarded and all the resources it occupies in the kernel are freed.

Process Owner Interface

The final part of the low-level process interface is the up-call made to the owner of a process when it is stopped. The owner should have some kind of server loop, waiting for requests on the port in the owner capability (which the owner should have created itself using *uniqport*(L)). The owner up-call is a request whose *h_command* value is `PS_CHECKPOINT`. The *h_extra* field in the RPC header specifies the reason why the process was stopped:

TERM_NORMAL	Normal termination: a thread called *exitprocess*(L) or the last thread called *exitthread* (see *sys_newthread*(L)). The *h_offset* field contains the exit status passed to *exitprocess()*; 0 if the process was terminated because the last thread called *exitthread()*.

TERM_STUNNED	The process was stunned by a call to *pro_stun*. The *h_offset* field contains the stun code passed as the second argument to *pro_stun*.
TERM_EXCEPTION	A thread has caused an exception for which it has no handler. The *h_offset* field contains the exception number (see *exception*(H)).

The request buffer contains a process descriptor in standard (network) byte order, containing the state of the process at the time it was stopped. This should be converted to native byte order using *buf_get_pd* (see *process_d*(L)).

If the owner wants to let the process continue (if it can), it should send the process descriptor back unchanged in the reply (in standard byte order), with *h_status* set to `STD_OK`. To let the process die, return a different error status or an empty reply buffer.

A debugger may change some parts of the process descriptor before letting the process continue; this is architecture-dependent and beyond the scope of this manual page.

When a stopped process is continued, the owner is not restored from the process descriptor; to change the owner of a stopped process, use *pro_setowner*.

Example

Code to run a process with these primitives would be too big to give a working example here. See the source code of *exec_file*(L) for a working example.

See Also

exception(H), exec_file(L), exec_findhost(L), exitprocess(L), getinfo(L), mutex(L), pd_read(L), process_d(L), run(A), sys_newthread(L).

(in libraries: libamunix.a, libamoeba.a, libkernel.a)

Name

process_d – process descriptor, segment descriptor, thread descriptor

Synopsis

```
#include "amoeba.h"
#include "module/proc.h"
#include "fault.h" /* From src/h/machdep/arch/<architecture> */

#define ARCHSIZE        ...

typedef struct {...}    process_d;
typedef struct {...}    segment_d;
typedef struct {...}    thread_d;
typedef struct {...}    thread_idle;

#define TDX_IDLE        ...
#define TDX_KSTATE      ...
#define TDX_USTATE      ...

#define PD_SD(p)        ...
#define PD_TD(p)        ...
#define TD_NEXT(t)      ...

int pd_size(pd)
char *pd_arch(pd)

char *buf_put_pd(buf, bufend, pd)
char *buf_get_pd(buf, bufend, pd)
process_d *pdmalloc(buf, bufend)
```

Description

The description of the process interface is split in several parts. The process descriptor data structure and its basic access methods are described here. The low-level kernel process server interface is described in *process*(L). The high-level process execution interface is described in *exec_file*(L). See also *pd_read*(L).

A *process descriptor* is the basic data structure used by the kernel and utilities that manipulate processes. A process descriptor contains some fixed parts and two tables of variable length: the segment descriptor list and the thread descriptor list.

Process Descriptors

```
#define ARCHSIZE        8

typedef struct {
        char            pd_magic[ARCHSIZE];
        capability      pd_self;
        capability      pd_owner;
        uint16          pd_nseg;
        uint16          pd_nthread;
} process_d;
```

The *process_d* data structure defines only the fixed parts of a process descriptor. Its members are:

pd_magic	Specifies the *architecture identifier* describing the type of processor on which the process runs or may run. The string is at most `ARCHSIZE−1` bytes long and NULL-padded to `ARCHSIZE` bytes, so it is always NULL-terminated.
pd_self	Ignored in executable files and by *pro_exec* (see *process*(L)); in a checkpoint, specifies the process capability of the process whose checkpoint it is. This is included to simplify applications that own many processes.
pd_owner	Specifies the owner capability. A NULL-port means the process has no owner.
pd_nseg	Specifies the number of segment descriptors included in the process descriptor. Must be at least one.
pd_nthread	Specifies the number of thread descriptors included in the process descriptor. Must be at least one.

Segment Descriptors

```
typedef struct {
        capability      sd_cap;
        long            sd_offset;
        long            sd_addr;
        long            sd_len;
        long            sd_type;
} segment_d;
```

The segment list of a process descriptor is an array of *segment descriptors*. Each array entry is a structure of type *segment_d*. To access the segment descriptors, the macro `PD_SD` must be used. If *p* is a pointer to a process descriptor, then `PD_SD(p)` is a pointer to its first segment descriptor, and for `0 <= i < p->pd_nseg`, the expression `PD_SD(p)[i]` is its *i*-th segment descriptor (not a pointer).

In executable files and for *pro_exec*, a segment descriptor describes a segment that must be created to form part of the address space of the new process. In a checkpoint, a segment descriptor describes an actually existing segment. Depending upon flags in *sd_type*, some segment descriptors do not describe a segment, and must be ignored. This is used to include the symbol table in the segment list, and for deleted segments in checkpoints.

The members of a segment descriptor are:

sd_cap — In executable files and for *pro_exec*, this specifies the capability of the file to be used to (partially) initialize the segment. In a checkpoint, it specifies the capability of the actual memory segment; memory segments may be read using *b_read* (see *bullet*(L)) and written using *ps_segwrite* (see *segment*(L)). If the capability is NULL, *pro_exec* will initialize the segment to zero. A NULL-capability in an executable file is interpreted different depending upon *sd_offset* and flags in *sd_type*: it may refer to a portion of the same executable file, require zero fill (e.g., bss), or be a placeholder for the stack segment.

sd_offset — In executable files and for *pro_exec*, this specifies the offset where the initializing data starts in the file specified by *sd_cap*. In executable files, the combination of a NULL *sd_cap* and a nonzero *sd_offset* implies that the initializing data for the segment resides in the executable file itself. This is normally the case for text and data segments; before handing the process descriptor to *pro_exec*, the file's capability must be stored in *sd_cap*.

sd_len — Specifies the length (in bytes) of the segment. The kernel may silently round this up to the next page or ‘‘click’’ boundary.* Zero-length segments are not supported; a zero length for the stack segment in an executable file indicates that a default stack size must be substituted by *exec_file*(L).

sd_type — Specifies various flags used when mapping the segment into an address space. The bits in this longword are divided in groups of flags; for each group there is a macro that defines which bits are part of the group, and macros specifying the individual bits or possible values for the group as a whole. Typical usage is as follows, taking the growth direction as an example:

```
if ((s->sd_type & MAP_GROWMASK) == MAP_GROWDOWN) {
    /* segment grows down */
}
```

The following bit fields are assigned:

MAP_GROWMASK
: Specifies the growth direction. Possible field values are `MAP_GROWNOT` for a fixed-size segment (e.g., the text and data segments), `MAP_GROWUP` for a segment which may grow up (e.g., the bss or heap segment), and `MAP_GROWDOWN` for a segment which may grow down (e.g., the stack segment).

MAP_TYPEMASK
: Specifies text or data type. Possible field values are `MAP_TYPETEXT` for a text segment, and `MAP_TYPEDATA` for a data (or bss, or stack) segment.

MAP_PROTMASK

* So, since segments may not overlap, a loader must know an upper bound for the click size of the target kernel, or it may assign overlapping address ranges to segments. In practice, since we have to work with existing loaders, this places an upper bound on the click size.

Specifies protection. Possible field values are `MAP_READONLY` for a read-only segment, and `MAP_READWRITE` for a readable and writable segment.

MAP_SPECIAL

Specifies special hacks. Possible field values are `MAP_INPLACE`, which is used to map segments that have a special meaning to the hardware, such as frame buffers for bit-mapped displays, and `MAP_AND_DESTROY`, which requests that the original segment be destroyed (through *std_destroy*) after it is copied into the new segment.

MAP_BINARY

Specifies flags used only in executable files. Possible field values are `MAP_SYSTEM`, which is used to indicate the stack segment (which must be initialized with arguments and environments), and `MAP_SYMTAB`, which is used to indicate the symbol table (not to be mapped at all).

Thread Descriptors

```
typedef struct {
        long            td_state;
        long            td_len;
        long            td_extra;
} thread_d;

typedef struct {
        long            tdi_pc;
        long            tdi_sp;
} thread_idle;
```

A thread descriptor is a variable-length object. Most of a thread descriptor is system-dependent or architecture-dependent. The simplest form, sufficient to create a new thread, only specifies an initial program counter and stack pointer. Thread descriptors describing threads that have been running contain more information, such as the full register set and some kernel state. Such thread descriptors may be passed to *pro_exec*; this can be used to migrate running processes.

The first thread descriptor is accessed through the `PD_TD` macro: if *p* is a pointer to a process descriptor, then the expression `PD_TD(p)` points to its first thread descriptor. If *t* is a pointer to a thread descriptor, then the expression `TD_NEXT(t)` is a pointer to the next thread descriptor; if *t* is the last thread descriptor, `TD_NEXT(t)` points to the first byte after the list of thread descriptors.

The *thread_d* structure only describes the first few bytes. Its members are:

td_state Contains kernel internal state flags for the thread. Set this to zero for a new process.

td_len Specifies the total length of this thread descriptor, in bytes, including the *thread_d* structure. For a new process, this should be set to

```
sizeof(struct thread_d) + sizeof(struct thread_idle)
```

td_extra A bit vector specifying which other, system-dependent structures follow. The order in which multiple structures occur is the same as the order in the following list. Some interesting bits are:

TDX_IDLE	A *struct thread_idle* follows. Its members are *tdi_pc*, specifying the initial program counter, and *tdi_sp*, specifying the initial stack pointer. This is the only bit that should be set for a new process.
TDX_KSTATE	A *struct thread_kstate* follows. Its contents are system-dependent.
TDX_USTATE	A *struct thread_ustate* follows. Its contents are architecture-dependent.

Access Functions

pd_size

```
int
pd_size(pd)
process_d *pd;
```

Computes the size of a process descriptor in bytes. This may be useful to allocate a buffer of sufficient size to copy a process descriptor into. The size depends on the number of segment descriptors and the number and size of thread descriptors.

pd_arch

```
char *
pd_arch(pd)
process_d *pd;
```

Returns the architecture string of a process descriptor. This is a pointer to the *pd_magic* member; the function is provided only for backward compatibility.

Packing and Unpacking Process Descriptors

When a process descriptor is written to a file or sent over the network, it must be converted to a standard byte order, and vice versa. The following three functions are used for this purpose. Note that other functions in the library with process descriptor arguments always expect or return the argument in the machine's native byte order, except where explicitly specified otherwise.

buf_put_pd

```
char *
buf_put_pd(buf, bufend, pd)
char *buf;
char *bufend;
process_d *pd;
```

Converts the process descriptor pointed at by *pd* to standard byte order, copying it into the buffer starting at *buf* and ending at *bufend*. The entire process descriptor is converted, not just its *process_d* part, with the exception that architecture-dependent parts of thread descriptors are copied as byte sequences without interpretation. It returns a pointer (pointing into the buffer) to the first byte after the converted process descriptor, or NULL if the conversion failed. Failure can be caused by insufficient space in the buffer or an ill-formatted process descriptor.

buf_get_pd

```
char *
buf_get_pd(buf, bufend, pd)
char *buf;
char *bufend;
process_d *pd;
```

Converts the buffer starting at *buf* and ending at *bufend* from standard byte order, copying it into the process descriptor pointed at by *pd* (which is normally allocated by *pdmalloc* below). It returns a pointer (pointing into the buffer) to the first byte after the consumed input, or NULL if the conversion failed. Failure can be caused by insufficient or ill-formatted data in the buffer.

pdmalloc

```
process_d *
pdmalloc(buf, bufend)
char *buf;
char *bufend;
```

Returns a pointer to storage allocated with *malloc*(L), cast to *(process_d *)*. It inspects the data in the buffer (which should be a process descriptor in standard byte order as created by *buf_put_pd*) to estimate an upper bound for the number of bytes to allocate. It returns NULL if the buffer contains ill-formatted data or if not enough memory was available. The caller should free the allocated storage after using it, using *free* (see *malloc*(L)).

Warnings

The structures defined here may be changed or extended later. Only the name and meaning of the members described here should be trusted; their exact type and relative position in the structure are subject to change. This implies that it is not portable to use initializers or to take the address of a member. Initializing a structure to zero and then filling in documented members will be portable.

Examples

To walk through the list of segment descriptors of a process descriptor *p*:

```
    segment_d *s;
    int i;
    for (i = 0; i < p->pd_nseg; ++i) {
        s = &PD_SD(p)[i];
        /* Here s points to segment descriptor number i */
    }
```

To walk through the list of thread descriptors of a process descriptor *p*:

```
    thread_d *t;
    int i;
    t = PD_TD(p);
    for (i = 0; i < p->pd_nthread; ++i) {
        /* Here t points to thread descriptor number i */
        t = TD_NEXT(t);
    }
```

See Also

bullet(L), exec_file(L), malloc(L), pd_read(L), process(L), segment(L).

(in libraries: libamoeba.a)

Name

profiling – find performance bottlenecks in Amoeba C programs

Synopsis

```
void    procentry(func)
void    procexit(func)
void    prof_dump(fp)
```

Description

The *profiling* module provides a method of finding performance bottlenecks in a C program that runs under Amoeba. It currently only works in conjunction with the ACK compiler (see *ack*(U)). The program to be profiled should be compiled using the **−p** option of *ack*(U) (which is linked to *cc* under Amoeba). The functions are not normally explicitly called by the programmer. They are compiled in at the appropriate places by the **−p** option of the compiler. The one exception is *prof_dump* which may be called by the programmer as described below.

There are two ways to obtain profiling information:

The first is to simply link the program with the standard version of *libamoeba.a*. To obtain statistics from such a program it is necessary to add a call to the function *prof_dump* just before the program exits, or where the summary of statistics is desired. Note that no statistics about calls to library routines will be reported in this case.

The second is to link the program with a specially compiled version of *libamoeba.a*, which has been compiled with profiling on. When the program runs it collects information about almost all the function calls made and when the program exits it prints a summary of all the statistics. Calls to assembly code routines and a very small number of special C routines (namely those used by *procentry* and *procexit)* are not reported in the statistics.

The profiling version of *libamoeba.a* can be made by adding the `-DPROFILING` flag to the `DEFINES` in the *Amakefile* for *libamoeba.a* and then remaking the library. This causes the entire library to be compiled with the **−p** option. System administrators may prefer to make a separate directory next to *lib/amoeba* in the configuration tree, copy the *Amakefile* from *lib/amoeba* to it and build the profiling version there.

If it is desired to profile libraries other than *libamoeba.a* then they should be compiled with the **−p** flag.

Functions

procentry, procexit

```
void
procentry(func)
char * func;

void
procexit(func)
char * func;
```

A call to these function is added by the compiler at the start and end respectively, of each C function call. They are given as argument the name of the function from which they are called. The programmer can, of course, redefine these routines for other debugging purposes. The version of these routines in *libamoeba.a* maintains, in core, statistics about the number of times the function is called in each thread of the process and the time taken for each call. A summary of this information can be printed using *prof_dump*.

prof_dump

```
void
prof_dump(fp)
FILE * fp;
```

This function prints the statistics gathered by the *procentry* and *procexit* routines in *libamoeba.a*. It prints them on the file referred to by *fp*. If *fp* is the NULL-pointer then it prints on *stderr*.

If the profiling version of the library is used then this routine is called when the program exits. If the standard version of *libamoeba.a* is used it is necessary to make explicit calls to *prof_dump* from within the program to obtain the statistics.

Profiling information is recorded and printed per thread. When printed it appears as follows:

```
------ thread 0 -------
clean_up            :   1 {   0,   0,   0 } [ list(1) ]
print_result        :  18 {   0,   8,  50 } [ list(18) ]
compare             :  54 {   0,   0,   0 } [ list(54) ]
list                :   1 { 650, 650, 650 } [ main(1) ]
main                :   0 {   0,   0,   0 } [ ]
```

To provide a short example, the above information was taken from a program with a single thread that was not linked with the profiling library. A fully profiled program would produce much more information. The first column is the name of the function that was called. The second column is the number of times that function was called. The next three numbers between the braces are the minimum, mean and maximum execution times in milliseconds taken by the function. The list between the brackets shows the names of the functions which called the function in question and the number of times each of them called it.

Warnings

The timings provided are not very accurate and the performance of the program will degrade when using the profiling tools. (They are themselves a bottleneck at present.)

See Also

ack(U).

(in libraries: libamoeba.a, libamunix, libkernel.a)

Name

prv – manipulate capability private parts

Synopsis

```
#include "amoeba.h"
#include "module/prv.h"

int prv_decode(prv, prights, random)
int prv_encode(prv, obj, rights, random)
objnum prv_number(prv)
```

Description

These functions are used by servers to make and validate capabilities and to extract the object number from a capability.

Functions

prv_decode

```
    int
    prv_decode(prv, prights, random)
    private *prv;
    rights_bits *prights;
    port *random;
```

Prv_decode is used to validate a capability when the original random number is known. It operates on the private part of the capability. It checks the check field in *prv* and if the check field is valid it returns in *prights* the rights in the capability. It returns 0 if it succeeded and −1 if the check field was invalid. If *prv_decode* fails then the capability was not valid and should be rejected.

prv_encode

```
    int
    prv_encode(prv, obj, rights, random)
    private *prv;
    objnum obj;
    rights_bits rights;
    port *random;
```

Prv_encode builds a private part of a capability in *prv* using the object number *obj*, the rights field *rights* and the check field pointed to by *random*. It returns 0 if it succeeded and −1 if the *rights* or the *obj* argument exceeded the implementation limits.

prv_number

```
     objnum
     prv_number(prv)
     private *prv;
```

Prv_number returns the object number from the private part *prv*.

Warnings

The current implementation restricts the object number to 24 bits and the rights field to 8 bits.

Example

The following function implements the std_restrict operation:

```
errstat
impl_std_restrict(cap, mask, newcap)
capability *cap;
rights_bits mask;
capability *newcap;
{
    objnum obj;
    rights_bits rights;
    struct foobar {
        port random;            /* Random checkword */
        ...
    } *foo, *FindFoo();

    /* Find object: */
    obj = prv_number(&cap->cap_priv);
    if ((foo = FindFoo(obj)) == NULL)
        return STD_CAPBAD;      /* Never heard of */
    /* Validate: */
    if (prv_decode(&cap->cap_priv, &rights, &foo->random) != 0)
        return STD_CAPBAD;      /* Do not trust cap */
    /* Build new cap: */
    rights &= mask;
    newcap->cap_port = cap->cap_port;
    if (prv_encode(&newcap->cap_priv, obj, rights, &foo->random) != 0)
        return STD_SYSERR;      /* Should not happen */
    return STD_OK;
}
```

See Also

rpc(L).

(in libraries: libamoeba.a)

Name

rawflip − routines to access FLIP protocol from user level programs

Synopsis

```
#include "amoeba.h"
#include "module/rawflip.h"

int flip_init(ident, receive, notdeliver)
int flip_end(ifno)
int flip_register(ifno, adr)
int flip_unregister(ifno, ep)
int flip_broadcast(ifno, pkt, ep, length, hopcnt)
int flip_unicast(ifno, pkt, flags, dst, ep, length, ltime)
int flip_multicast(ifno, pkt, flags, dst, ep, length, n, ltime)

void flip_oneway(prv_addr, pub_addr)
void flip_random(adr)
void flip_random_reinit()
```

Description

These routines provide for access to the FLIP network protocol from user programs. The calls are virtually identical to the ones as described in M.F. Kaashoek's thesis, *Group Communication in Distributed Computer Systems* and also to the calls used inside the Amoeba kernel, to ease porting code between user mode and kernel mode.

The interface consists of the actual routines for sending and receiving messages plus some miscellaneous support routines.

The FLIP Interface

flip_init

```
    int
    flip_init(ident, receive, notdeliver)
    long ident;
    int (*receive)();
    int (*notdeliver)();
```

Flip_init initializes a process' use of the FLIP protocol. Parameters are *ident*, a *long* passed back during upcalls and the *receive* and *notdeliver* functions used for callback from kernel space. To be able to do callback the *flip_init* function does a *thread_newthread* and all upcalls are done from the forked thread. This might be relevant when accessing shared data structures.

It is an error to call *flip_init* twice without an intervening call to *flip_end*.

The *flip_init* function returns a small integer *ifno* (interface number) on success or `FLIP_FAIL` on failure.

flip_end

```
int
flip_end(ifno)
int ifno;
```

Flip_end terminates a process' use of the FLIP protocol for the specified interface *ifno*. The extra thread created during *flip_init* will also terminate. Returns 0 on success, `FLIP_FAIL` on failure.

flip_register

```
int
flip_register(ifno, adr)
int ifno;
adr_p adr;
```

Flip_register registers the address pointed to by *adr* as an endpoint address for FLIP communication. From then on any packet sent to the one-way-encrypted address is given to this process using the *receive* function given to the *flip_init* function. Registering the all zero address indicates willingness to receive packets sent out by *flip_broadcast*. *Flip_register* returns a small integer *ep* (endpoint) on success or `FLIP_FAIL` on failure.

Note that repeated FLIP broadcasts with the same source address, message-id and offset within a short space of time will not be sent. Furthermore, messages sent to different incarnations of a process which reuses FLIP addresses may be discarded due to route caching effects. Therefore it is important to use a new FLIP source address for each new invocation of a process. The routine *flip_random* (described below) is one way to do this.

It is unwise to use the same address both for unicast and multicast. Cacheing effects in the kernel will make communication slower in that case. It is better to register an address for unicast and to use as a source for all messages.

flip_unregister

```
int
flip_unregister(ifno, ep)
int ifno;
int ep;
```

Flip_unregister stops FLIP listening to the FLIP address associated with the endpoint *ep*. Due to the asynchronous nature of the underlying network layer there is a possibility that some packets destined for that address are still underway and will be delivered through *receive*. Returns 0 on success, `FLIP_FAIL` on failure.

flip_broadcast

```
int
flip_broadcast(ifno, pkt, ep, length, hopcnt)
int ifno;
char * pkt;
int ep;
f_size_t length;
f_hopcnt_t hopcnt;
```

Flip_broadcast broadcasts the packet *pkt* with length *length* using the source address associated with *ep*. The distance the broadcast travels is limited by the *hopcnt* parameter. Note that a hop count of 1 does not imply that the broadcast will go over Ethernet. For example, the current implementation treats local communication and physical shared memory as networks and a hop count of at least 3 is required to send something over Ethernet. The actual value required should be determined experimentally. There is a maximum of 25 hops in the current implementation. Returns 0 on successful sending of the message, `FLIP_NOPACKET` if no free packets were currently available and `FLIP_FAIL` on failure. A 0 return value does not imply successful delivery of the message.

flip_unicast

```
int
flip_unicast(ifno, pkt, flags, dst, ep, length, ltime)
int ifno;
char * pkt;
int flags;
adr_p dst;
int ep;
f_size_t length;
interval ltime;
```

Flip_unicast sends a unicast packet *pkt* with length *length* to the flip address *dst* using the source address associated with *ep*. The locate timeout for finding the destination is set to *ltime* milliseconds. Possible *flags* which can be bitwise ORed are `FLIP_INVAL` to invalidate any existing route cache entry, `FLIP_SYNC` to wait for any necessary locate to finish before returning from *flip_unicast*, and `FLIP_SECURITY` to only route over trusted networks. If the destination is not found within the specified timeout, or if while sending the packet a routing error occurs, the packet is returned through the *notdeliver* function specified with *flip_init*. Returns 0 on successful sending of the message, `FLIP_NOPACKET` if no free packets were currently available and `FLIP_FAIL` on failure. A 0 return value does not imply successful delivery of the message.

flip_multicast

```
int
flip_multicast(ifno, pkt, flags, dst, ep, length, n, ltime)
int ifno;
char * pkt;
int flags;
adr_p dst;
int ep;
f_size_t length;
int n;
interval ltime;
```

Flip_multicast sends a multicast packet *pkt* with length *length* to the flip address *dst* using the source address associated with *ep*. The packet will only be sent if at least *n* recipients are found listening to this address. The locate timeout for finding the destination is set to *ltime* milliseconds. *Flags* are as in *flip_unicast*. If the destination is not found within the specified timeout, or if while sending the packet a routing error occurs, the packet is returned through the *notdeliver* function specified with *flip_init*. Returns 0 on successful sending of the message, `FLIP_NOPACKET` if no free packets were currently available and `FLIP_FAIL` on failure. A 0 return value does not imply successful delivery of the message.

receive

```
int
receive(ident, pkt, dstaddr, srcaddr, message-id, offset, length,
                total, flag)
long ident;
char *pkt;
adr_p dstaddr, srcaddr;
f_msgcnt_t messid;
f_size_t offset, length, total;
int flag;
```

The *receive* function specified by the user in *flip_init* is called when a packet arrives for one of the addresses registered by calling *flip_register*. The *ident* parameter is the same as specified in *flip_init* at the sender's side. The *message-id* identifies the message. It is unique per source address and per message. If a message arrives fragmented, all fragments will contain the same *message-id*. The *offset* and *length* identify which part of the message identified by *message-id* is contained in this packet. The total length of the message is *total*. The user is responsible for reassembly of messages fragmented by the network layer. The packet *pkt* points to data that is not available after *receive* returns, so must be copied to be retained. *Flag* can contain `FLIP_UNTRUSTED` if the packet crossed untrusted networks on the way.

notdeliver

```
int
notdeliver(ident, pkt, dstaddr, message-id, offset, length,
                 total, flag)
long ident;
char *pkt;
adr_p dstaddr;
f_msgcnt_t messid;
f_size_t offset, length, total;
int flag;
```

The *notdeliver* function specified by the user in *flip_init* is called when a message sent by this process is undeliverable. Parameters as in *receive*. *Flag* can also contain `FLIP_NOTHERE` if the packet was bounced back on the way to the destination because the address was not known. This is the usual case for *notdeliver*.

Miscellaneous Routines

To be able to use the raw FLIP interface it is necessary to have a few support routines for creating and encrypting FLIP addresses.

flip_oneway

```
void
flip_oneway(prv_addr, pub_addr)
adr_p prv_addr;
adr_p pub_addr;
```

Flip_oneway encrypts the FLIP address pointed to by *prv_addr* and returns the result in the FLIP address pointed to by *pub_addr*.

flip_random

```
void
flip_random(addr)
adr_p addr;
```

Flip_random generates a random FLIP address in the struct pointed to by *addr*. The *space* for the address is set to 0.

flip_random_reinit

```
void
flip_random_reinit()
```

Flip_random_reinit causes the next call to *flip_random* to get a new seed for the random number generator from the random number server. The random server is either the one specified by the string environment variable `RANDOM`, if set, or otherwise from the default random number server (see *rnd*(L)).

Example

The following example skips huge amounts of detail but gives an indication of how things work. Any messages that come in will be handled by the *received* function.

```
int ifno;
int ep;
addr_t my_address, addr_receiver;

static void received(long ident, char *pkt, adr_p dstaddr,
                 adr_p srcaddr, f_msgcnt_t messid, f_size_t offset,
                 f_size_t length, f_size_t total, int flag)
{
    /* fill in your receive protocol here */
}

static void notdelivered(long ident, char *pkt,
                 adr_p dstaddr, f_msgcnt_t messid, f_size_t offset,
                 f_size_t length, f_size_t total, int flag);
{
    /* fill in your 'not delivered' protocol here */
}

ifno = flip_init(1722, received, notdelivered);
if (ifno < 0) {
    /* error ... *./
}

/* Create an address */
flip_random(&my_address);

/* Register an endpoint */
ep = flip_register(ifno, &my_address);

/* Send a point-to-point message.  To get the address of the receiver
 * you can broadcast it or perhaps get it as a command line argument.
 */
while (((res = flip_unicast(ifno, message, 0, &addr_receiver,
            ep, buffer_size, 10000)) == FLIP_NOPACKET) && count < 5)
    count++;

/* Clean up */
flip_unregister(ifno, ep);
flip_end(ifno);
```

See Also

rnd(A), thread(L).

(in libraries: libamoeba.a)

Name

regex – BSD regex(3)/ed(U) compatible regular-expression routines

Synopsis

```
char *re_comp(s)
int re_exec(s)
int ignoreCase
int regex_bopat[10];
int regex_eopat[10];
```

Description

Public domain routines compatible with the BSD regex(3)/ed(U) compatible regular-expression routines.

Functions

re_comp

```
    char *
    re_comp(regexp)
    char *regexp;
```

Re_comp compiles the regular expression in the string argument into an internal format that is used by *re_exec* to perform pattern matching. *Re_comp* returns a NULL-pointer if the conversion was successful. In all other cases *re_comp* returns a pointer to a string containing the relevant error message.

If *re_comp* receives a NULL-pointer or an empty string as argument it will return a NULL-pointer and leave the previously compiled regular expression intact. If there was no previous regular expression *re_comp* will return an error.

Re_comp can compile the given regular expression into a case-insensitive regular expression. To achieve this effect the programmer must set the value of the global variable *ignoreCase*, defined inside the module containing *re_comp*, to a non-zero value. This is a BSD incompatible feature.

Re_comp recognizes the following regular expressions:

[1] `char`	matches itself, unless it is a special character (metachar): `. \ [ ] * + ^ $`
[2] `.`	matches any character.
[3] `\`	matches the character following it, except when followed by a left or right parenthesis, a digit `1` to `9` or a left or right angle bracket. It is used as an escape character for all other meta-characters, and itself. When used in a set, it is treated as an ordinary character.

[4] `[set]` matches one of the characters in the set. If the first character in the set is `^`, it matches a character NOT in the set. That is, `^` complements the set. A shorthand `S-E` is used to specify a set of characters `S` up to `E`, inclusive. The special characters `]` and `-` have no special meaning if they appear as the first characters in the set.

examples:	match:
`[a-z]`	any lowercase alpha
`[^]-]`	any character except `]` and `-`
`[^A-Z]`	any character except uppercase alpha
`[a-zA-Z]`	any alpha

[5] `*` any regular expression form [1] to [4], followed by closure character `*` matches zero or more matches of that form.

[6] `+` same as [5], except it matches one or more.

[7] a regular expression in the form [1] to [10], enclosed as `\(`form`\)` matches what form matches. The enclosure creates a set of tags, used for [8] and for pattern substitution. The tagged forms are numbered starting from 1.

[8] a `\` followed by a digit 1 to 9 matches whatever a previously tagged regular expression ([7]) matched.

[9] `\<` `\>` a regular expression starting with a `\<` construct and/or ending with a `\>` construct, restricts the pattern matching to the beginning of a word, and/or the end of a word. A word is defined to be a character string beginning and/or ending with the characters `A-Z a-z 0-9` and `_`. It must also be preceded and/or followed by any character outside those mentioned.

[10] a composite regular expression xy where x and y are in the form [1] to [10] matches the longest match of x followed by a match for y.

[11] `^` `$` a regular expression starting with a `^` character and/or ending with a `$` character, restricts the pattern matching to the beginning of the line, or the end of line. Elsewhere in the pattern, `^` and `$` are treated as ordinary characters.

re_exec

```
    char *
    re_exec(s)
    char *s;
```

Re_exec matches the regular expression compiled by *re_comp* to the string *s*. *Re_exec* returns 1 for success, 0 for failure and -1 when the compiled regular expression is invalid. When the match has been successful *re_exec* allows access to the tags created by `\(` and `\)`. The values of the tags can be accessed through the arrays *regex_bopat* and *regex_eopat* that contain offsets from the beginning of the string *s*. *regex_bopat[n]* points to the first character of the *n*th tag, *regex_eopat[n]* points to the last. Tag 0 contains the string passed to re_exec. The *regex_bopat* and *regex_eopat* arrays are not a part of the original BSD interface.

See Also

ed(U), grep(U).

Name

rnd − random server interface

Synopsis

```
#include "module/rnd.h"

void rnd_setcap(cap)
errstat rnd_defcap();
errstat rnd_getrandom(buf, size)
```

Description

This module is used for obtaining random numbers from a random number server (see *random*(A)).

rnd_setcap

```
void
rnd_setcap(cap)
capability *cap;
```

Rnd_setcap sets the random number server capability to be used by subsequent calls to *rnd_getrandom* to *cap*. It can be used to override the default random number server capability. Note that it sets the random number server per process and not per thread.

rnd_defcap

```
errstat
rnd_defcap()
```

Rnd_defcap selects the default random number server to be the random number server used by subsequent calls to *rnd_getrandom*. If the capability environment variable `RANDOM` is set it uses that capability. Otherwise it uses the capability specified by `DEF_RNDSVR` in the include file *ampolicy.h*. This is typically */profile/cap/randomsvr/default*. The value returned by *rnd_defcap* is the error status from the lookup of the default server.

Note that it is not necessary to call this routine before using *rnd_getrandom*. If no call has been made to *rnd_setcap* then *rnd_getrandom* will use the default server as defined by *rnd_defcap*.

rnd_getrandom

```
errstat
rnd_getrandom(buf, size)
char *buf;
int size;
```

Rnd_getrandom fills in the buffer pointed to by *buf*, which is *size* bytes long, with random data using a random number server. *Size* must not exceed `MAX_RANDOM`, which is typically 1 kilobyte.

Required Rights:
    NONE

Error Conditions:
    RPC errors
    `STD_OVERFLOW`: size > `MAX_RANDOM`

Environment Variables

`RANDOM` – capability for the random server.

Warnings

When *rnd_getrandom* fails, it undoes the effect of any previous call to *rnd_setcap*. That is, the next *rnd_getrandom* will call *rnd_defcap*.

See Also

ansi_C(L), uniqport(L).

(in libraries: libamoeba.a, libkernel.a, libamunix.a)

Name

rpc – the interface stubs for remote procedure calls

Synopsis

```
#include "amoeba.h"

bufsize  trans(request_hdr, request_buf, request_size,
                         reply_hdr, reply_buf, reply_size);
bufsize  getreq(hdr, buf, size);
void     putrep(hdr, buf, size);
interval timeout(length);
```

Description

The three calls *getreq*, *putrep*, and *trans* form the remote procedure call (RPC) interface for interprocess communication. All point to point communication is, at the lowest accessible level, structured along the client/server model; that is, a client thread sends a request message to a service, one of the threads in one of the server processes gets the request, carries it out and returns a reply message to the client. There is also the possibility for group communication. This is described in *grp*(L). See also *rawflip*(L).

For historical reasons, a remote operation (a request followed by a reply) is called a *message transaction* or just a *transaction*. A client thread invokes a transaction by a call to *trans*. A server thread receives a request via a call to *getreq* and returns a reply by calling *putrep*. *Trans*, *getreq*, and *putrep* are blocking; that is, a *trans* suspends a thread until the request is sent, carried out and a reply is received; *getreq* suspends a thread until a request has been received and *putrep* suspends a thread until the reply has been received by the appropriate client thread's kernel.

A request or reply message is described to the transaction system calls by means of three parameters: a pointer to a *header*, a pointer to a *buffer* and the *size* of that buffer. The *header* is a fixed-length data structure, containing addressing information, status information, an operation code and some parameters. The *buffer* is an 8-bit-character array whose *size* may vary from 0 to 30000_{decimal} bytes. The parameters of *getreq* specify where in memory the header and buffer of the request are to be received. The parameters of *putrep* specify the reply to be sent and the parameters of *trans* specify the request to be sent and where in memory the reply is to be received.

The following sections explain the port- and capability-based addressing mechanism used to specify services and objects, the exact structure of request and reply messages, the three system calls and the failure semantics of RPCs.

Ports and Capabilities

Each object is both identified and protected by a *capability*. Capabilities have the set of operations that the holder may carry out on the object coded into them and they contain enough redundancy and cryptographic protection to make it infeasible to guess an object's capability. Objects are implemented by server processes that manage them. Capabilities have the identity of the object's server encoded into them (the Service Port) so that, given a capability, the system can find a server process that manages the corresponding object. The structure of capabilities and ports is defined in the standard include file *amoeba.h*:

48	24	8	48 bits
Port	Object	Rights	Random

```
#define PORTSIZE 6

typedef struct {
        int8    _portbytes[PORTSIZE];
} port;

typedef struct {
        int8    prv_object[3];
        uint8   prv_rights;
        port    prv_random;
} private;

typedef struct {
        port    cap_port;
        private cap_priv;
} capability;
```

A server thread identifies itself to the system by giving its *port* in the header of the *getreq* system call. The port that it gives must be the *get-port*, also called the *private port*, (see *priv2pub*(L) for more details). A client identifies the object of its transaction − and with it the service that manages that object − by giving the object's capability in the *port* and *private* fields of the request header. The port that the client has is the *put-port* and this is also the port returned in the header of the *getreq* call when a message arrives. This is illustrated in the example below.

Both client and server thus specify a port and this port is used by the system to bring client and server together. To prevent other processes from impersonating a particular server, a port has two appearances called get-port and put-port. A server needs to specify a get-port when it does a *getreq* call and a client has to specify a put-port (as part of the capability) when it makes a *trans* call. Get-port and put-port are related via a one-way function, F, which transforms a get-port into a put-port. The function was chosen so that it is 'impossible' to compute a get-port from a put-port. The library function *priv2pub* does the one-way function transformation. The system guarantees that a client's request message containing put-port P in its capability will only be delivered to a server thread which specified get-port G in its *getreq* call, where $P = F(G)$.

Message Structure

Both request and reply messages consist of two parts, a header and a buffer. A message with an empty buffer is a *header-only* message. The header has a fixed length and a fixed layout:

```
typedef  uint16   command;
typedef  uint16   bufsize;

typedef  struct {
         port     h_port;/*  6 bytes */
         port     h_signature;/*  6 bytes */
         private  h_priv;/* 10 bytes */
         command  h_command;/*  2 bytes */
         int32    h_offset;/*  4 bytes */
         bufsize  h_size;/*  2 bytes */
         uint16   h_extra;/*  2 bytes, total 32 bytes */
} header;

#define  h_status h_command/* alias: reply status */
```

The meaning of the fields is as follows:

h_port	The port field contains the port-part of the capability of the object on which the request should be carried out. The port field is interpreted only in *trans* and *getreq*. With *trans* it contains the put-port of the service for which the request is intended. When calling *getreq* it contains the get-port of the service offered. When *getreq* returns it contains the put-port sent by the client. In replies, the contents of this field are passed from server to client uninterpreted.
h_signature	This field is reserved for future use, possibly connected with security.
h_command	This field is normally used in requests to hold an operation code which specifies which operation to carry out. In replies, it usually conveys the result of the operation back to the client (e.g., an error status). The field is not interpreted by the operating system.
h_offset	The name of this field was inspired by the use of transactions for reading and writing file objects. In the standard file read/write interface, it is used to specify offsets into a file. This 32-bit field may, however, be used as the application programmer sees fit. It is not interpreted by the operating system.
h_size	
h_extra	These fields can be used to communicate arbitrary 16-bit quantities between client and server. They are not interpreted by the operating system.

Request and reply messages consist of a header and optionally a buffer. The size of the buffer may be between 0 and 30000_{decimal} bytes. Requests and replies, or buffers in which requests and replies are to be received, are specified via three parameters, the address of the header, the address of the buffer, and the size of the buffer. The address of the header must always point to a valid area in memory (that is, an area that is mapped in readable or read/writable, depending on whether it is used for sending or receiving). If the length of the buffer is zero, the buffer address is ignored. If the length of the buffer is non-zero, the buffer address must point to an area of memory of that length, wholly contained in a single memory segment. In other words, buffers may not straddle segment boundaries. The segment in which a buffer lies must be readable or read/writable depending on whether it is used for sending or receiving a message.

When a request or reply is received that is larger than the available buffer, no error code is returned. The number of bytes actually received in the buffer is returned. Truncation can only be detected if the *h_size* field is used to transmit the length of the buffer sent. If this is done properly, an example test for truncation is:

```
if (ERR_STATUS(stat = getreq(&hdr, &buf, cnt))) {
        /* error */
        ...
}
if (stat != hdr.h_size) {
        /* truncation */
        ...
}
```

Programmers are strongly encouraged to use the *h_size* field to communicate the true length of a message buffer to the receiver.

getreq

```
bufsize
getreq(hdr, buf, size)
header *hdr;
bufptr buf;
bufsize size;
```

A server thread uses *getreq* to receive a request from a client. After processing the request, it must return a reply by a call on *putrep*, or forward the request to another server using *grp_forward* (see *grp*(L)). A server thread may carry out only one request at a time. Successful *getreq* and *putrep* calls must therefore alternate. A server thread may always, even while serving (that is, after *getreq* and before *putrep*), use *trans* calls to have operations carried out by other servers.

A server thread must provide the get-port on which it receives in the *h_port* field of the header parameter. The code for receiving a request thus becomes:

```
header.h_port = server_get_port;
status = getreq(&header, buf, size);
```

Getreq blocks a thread until a request is received (or until the thread is alerted by a signal). The returned value, *status*, contains the number of bytes in the buffer when the call is successful, or a negative error code. If the request attempted to pass more than *size* bytes, the message placed in the area pointed to by *buf* is truncated to *size*. Errors are discussed in the next section.

Upon successful completion, the header pointed to by *hdr* contains a copy of the header given in the corresponding *trans* call of a client. Therefore the *h_port* field will contain the put-port which was used by the client. A subsequent call to *getreq* must be sure to reinitialize *h_port* to the get-port.

putrep

```
void
putrep(hdr, buf, size)
header *hdr;
bufptr buf;
bufsize size;
```

After processing a request, a server must return a reply using *putrep*. All the fields (including the *h_signature* field) of the header pointed to by *hdr* are sent to the client unmodified. As a matter of convention, the *h_command* field is used as a *status* field and is, in fact, referred to as *h_status*. It is important that all paths through the server actually set an error status rather than returning the original command code.

It is the conventional wisdom to allocate command and error-status codes centrally in order to prevent strange things from happening when one mistakes one type of object or server for another. It also simplifies object inheritance using AIL. Currently the central allocation makes use of the files *cmdreg.h* and *stderr.h*.

Every request must be accompanied by a reply. It is therefore a programming error to call *getreq* when a call to *putrep* can legally be made (and vice versa). The blocking nature of *trans* automatically prevents a client thread from being involved in more than one transaction. As a consequence, a thread can be involved in at most two transactions at a time: once as a server and once as a client.

trans

```
bufsize
trans(request_hdr, request_buf, request_size,
        reply_hdr, reply_buf, reply_size)
header *request_hdr;
bufptr request_buf;
bufsize request_size;
header *reply_hdr;
bufptr reply_buf;
bufsize reply_size;
```

Trans attempts to deliver a request to a server and passes back the server's reply. *Trans* blocks until the transaction has either failed or a reply is received. *Trans* takes six parameters, three to specify the request to be sent to the server and three to specify the memory locations to place the reply. The request header pointed to by *request_hdr* must contain the put-port of the service to be called in the *h_port* field. The system guarantees not to deliver a request to a server that has not specified the associated get-port in the *h_port* field of its *getreq* call. Note that there is only a negative guarantee. The system cannot guarantee to deliver a request; it can merely make an effort. The system does guarantee, however, that a request will be received at most once and by no more than one server. *Trans* has **at-most-once** semantics.

When a *trans* fails, the contents of the reply header and reply buffer are undefined and may have been changed. The contents of the request header and request buffer are guaranteed to be unchanged, unless they overlap the reply header or reply buffer. It is common practice to use the same header and buffer for request and reply in one transaction.

The other fields in the header, except possibly *h_signature*, are not interpreted by the system. But again, convention dictates the use of the *h_private* field for the private part of the capability of the object to which the request refers (the port part is in the *h_port* field), the *h_command* field for a unique request code, the *h_status* field for a unique error reply code, and the *h_size* field for the size of request and reply. The value returned by *trans* contains the number of bytes in the buffer when the call is successful, or a negative error code. If the reply attempted to pass more than *reply_size* bytes, the message placed in the area pointed to by *reply_buf* is truncated to *reply_size*.

timeout

```
    interval
    timeout(length)
    interval length;
```

Before a request can be delivered by the system, the client's kernel must know where to send the request. It must *locate* a server. Details of the locate mechanism are not relevant here, but what is relevant is that there is a minimum time that the kernel will spend trying to locate a suitable server. This per-thread locate timeout can be modified by the *timeout* system call. Its argument *length* is the new minimum timeout in milliseconds. It returns the previous value of the timeout. The default timeout is system dependent; it is typically 5 seconds. It is not a good idea to set the timeout below 2 seconds (*length* = 2000) since it may result in locate failures even though the server is actually available.

Error Codes and Failure Semantics

All three calls, *getreq*, *putrep* and *trans* can fail in a number of ways. A distinction must be made between programming errors and failures. An error is caused by faults in the program, for instance, calling *getreq* when a *putrep* is still due. A failure is caused by events beyond the control of the programmer: server crashes, network failures, etc.

Errors are caused by program bugs. A program with an error in it should be debugged. Errors, therefore, cause an exception, which, when not fielded by the program, causes the program to abort. When they are fielded, an interrupt routine is called to handle the error. In addition, the system calls also return a negative error code. Under Amoeba the exception is `EXC_SYS`. Under UNIX the exception generated is `SIGSYS`.

All failures cause *trans* and *getreq* to return a negative error code. The include file *stderr.h* provides the mapping between the error codes and the names used in programs. The table below gives an overview of the errors and failures, in which calls they can occur (T for *trans*, G for *getreq* and P for *putrep*), the code returned on *getreq* or *trans* (*putrep* does not return a value), and whether an exception is raised when an error occurs.

Value returned	*Exception*	*Which call*	*Error description*
RPC_NOTFOUND		T	Server not found
RPC_ABORTED		T G	Signal received while blocked
RPC_FAILURE		T	Server or network failure
RPC_FAILURE	*	T G P	Buffer length > 30000
RPC_FAILURE	*	G	Getreq while serving
RPC_BADADDRESS	*	T G P	Use of invalid pointers
RPC_BADPORT		T G	Use of a NULL-port
	*	P	Putrep while not serving
RPC_TRYAGAIN		T	Out of resources

Server not found

When locating a server fails, the `RPC_NOTFOUND` error code is returned on *trans* calls. The reason can be that the server is not yet ready, or that there is no server at all. In any case, the request will not have been delivered anywhere. When this error code is returned, it is sometimes sensible to wait a few seconds and try once more. The server may have called *getreq* in the meantime.

Signal received while blocked

If a thread has indicated interest in a signal and that signal is raised while the thread is in a *getreq* or the locate stage of a *trans* call, the call is broken off and `RPC_ABORTED` is returned. No useful information should be expected in the receive header and buffer. Threads can be aborted in system calls as a consequence of debugging. For servers, a sensible action following an `RPC_ABORTED` error code is to restart the *getreq* or *trans* operation.

Server or network failure

When a request has been sent and no reply is received and the server's kernel does not respond to ‘‘are-you-alive?’’ pings, `RPC_FAILURE` is returned. The server may or may not have carried out the request; the server may even have sent a reply, but then it has not been received. The course to take when this failure happens depends very much on the semantics of the attempted operation.

Buffer length > 30000

The buffer length is an unsigned integer. It can not be less than zero. If it is larger than the maximum size of a request or reply, an exception is raised and `RPC_FAILURE` is returned if the exception is handled.

Getreq while serving, putrep while not serving

Programming errors, so an exception is raised. Putrep does not return a value, so when the exception is caught no error code is returned.

Use of invalid pointers

Using pointers to headers or buffers wholly or partially in non-existent memory, to receive headers or buffers in read-only memory, or to buffers straddling segment boundaries, an exception is raised. If the exception is caught, *trans* and *getreq* additionally return `RPC_BADADDRESS`.

Use of a NULL-port

When you forget to fill in the *h_port* field on *getreq* or *trans*, the system returns a `RPC_BADPORT` failure. This is not an exception, because ports are often filled in with values retrieved from remote sources such as the directory server. It seemed unreasonable to ask application programmers to write code to check for NULL-ports when the system has to check for them anyway.

Out of resources

This error can only occur under UNIX or UNIX-like systems. It is used when the Amoeba driver could not get access to enough resources to send the message. It is guaranteed that the request has not reached any server. If enough resources are freed the request can be safely repeated.

Two helpful macros are available for manipulating returned status codes: `ERR_STATUS` and `ERR_CONVERT`. They provide the proper casts from the 16-bit unsigned integers to signed integers for testing the sign bit. The macros are defined in *amoeba.h*.

Signals

The previous section specified that when a *trans* is blocked and a signal is received while the system is still trying to locate the server, *trans* will return the error code `RPC_ABORTED`. When the location of the server is known and the RPC has been sent to the server and no reply has been received as yet, the reaction of the system to signals to be caught is altogether different. In this case signals are propagated to the server. Thus the server will receive the signal and can choose to react accordingly. It can, as per default, ignore the signal or catch it. If the server ignores the signal the signal will have no effect in both server and client. If the server handles the signal it can choose its own way of reacting, for example by returning an error reply with *putrep*. If the server itself catches signals and is waiting for a reply to a *trans* the signal will again be propagated.

Signals can abort calls to *getreq*, but can not abort calls to *putrep*. See also *signals*(L).

Example

The example below shows the typical server main loop for a very simple file server and a sample of client code for calling that server for a read operation and a write operation. Its is assumed that the client's capability has the put-port associated with the server's get-port.

```
#include "amoeba.h"
#include "cmdreg.h"
#include "stderr.h"

/* Server threads typically sit in an endless loop
 * fielding and carrying out client requests
 */
for (;;) {
    /* Put the server's get-port in the header.  When the request
     * arrives, it will be replaced by the put-port of the client's
     * request.
     */
    hdr.h_port = getport;
    size = getreq(&hdr, buf, 100);
    /* Check for errors.  */
    if (ERR_STATUS(size)) {
         /* The macro ERR_CONVERT casts the 16-bit
          * unsigned into an ordinary signed int.
          */
         if (ERR_CONVERT(size) == RPC_ABORTED) {
             /* When debugging a program, signals are used to
              * checkpoint it. These abort getreq calls. When
              * this happens, just try again.
              */
             continue;
         }
         /* Other errors are bad */
         fprintf(stderr, "Getreq error (%s)\n",
                      err_why(ERR_CONVERT(size)));
         exit(-1);
    }
    /* Control gets here when a request is successfully received.
     * Dispatch to the implementation routines for the different
     * request types.
     */
    switch (hdr.h_command) {
    case F_READ:
         /* Read implementation code goes here For the example,
          * simulate the result of a successful read operation.
          */
         size = hdr.h_size = 100;
         hdr.h_status = STD_OK;
         break;
    case F_WRITE:
         /* Same for write */
         size = hdr.h_size = 0;
         hdr.h_status = STD_OK;
         break;
    default:
         /* Not a legal request type, return
          * an error code
          */
         hdr.h_status = STD_COMBAD;
    }
    /* Return a reply. No error codes are returned */
    putrep(&hdr, buf, size);
}
```

Below is the client code for calling the server to do a read and then a write operation.

```
#include "amoeba.h"
#include "stderr.h"

/* Set transaction timeout to 30 seconds */
timeout(30000);

/* Fill in transaction header */
hdr.h_port = cap.cap_port;/* Server port */
hdr.h_priv = cap.cap_priv;/* Rest of file cap */
hdr.h_command = F_READ;
hdr.h_size = 100; /* Read 100 bytes .. */
hdr.h_offset = 0; /* .. from beginning of file */

/* Call the server; request buffer is empty,
 * reply buffer is large enough to hold requested bytes
 */
size = trans(&hdr, NILBUF, 0, &hdr, buf, 100);

/* Size holds positive number of bytes read or
 * negative error code.
 */
if (ERR_STATUS(size)) hdr.h_status = size;

/* Even if the transaction succeeds, the server may
 * return an error code of its own. Convention is that
 * the only success code is STD_OK (zero).
 */
if (hdr.h_status != STD_OK) {
        /* Print an error message. The macro ERR_CONVERT
         * casts the error code into an int. The function
         * err_why converts standard error codes into a
         * string.
         */
        fprintf(stderr, "1st trans failed (%s)\n",
                err_why(ERR_CONVERT(hdr.h_status)));
        exit(-1);
}
/* The server returns number of bytes read in hdr.h_size.
 * If the reply is truncated, or if the server is faulty
 * the number of bytes received may not correspond to what
 * the server claims was sent.
 */
```

```
if (size != hdr.h_size) {
        /* In the case of file read, the client may be
         * assumed not to ask for more than the buffer
         * can hold, so this code is only reached if
         * the server is faulty.
         */
        fprintf(stderr, "message truncated\n");
        exit(-1);
}

/* Read trans succeeded, try a write trans.
 * Again, fill in header fields. Never trust what
 * server left intact.
 */
hdr.h_port = cap.cap_port;
hdr.h_priv = cap.cap_priv;
hdr.h_command = F_WRITE;
hdr.h_size = 100;
hdr.h_offset = 0;

/* This time, request buffer is not empty */
size = trans(&hdr, buf, 100, &hdr, NILBUF, 0);

/* Size returned should be zero (empty reply buffer) */
if (ERR_CONVERT(size) != 0) hdr.h_status = size;
if (hdr.h_status != STD_OK) {
        fprintf(stderr, "2nd trans failed (%s)\n",
                err_why(ERR_CONVERT(hdr.h_status)));
        exit(-1);
}
```

See Also

error(L), grp(L), priv2pub(L), signals(L).

(in libraries: libamoeba.a)

Name

sak – the Swiss Army Knife server interface stubs

Synopsis

```
#include "amoeba.h"
#include "caplist.h"
#include "server/sak/sak.h"

errstat sak_exec_trans(argv, envp, capl, fcap)
errstat sak_make_transfile(hdr, buf, req_size, rep_size, wait, fcap)

errstat sak_list_job(jobcap, sched, options)
errstat sak_submit_job(svrname, sched, fcap, options, jobcap)
```

Description

The *sak*(A) server provides a mechanism to execute a transaction at a later date under various conditions. These routines, along with the standard server stubs (see *std*(L)) provide the interface to the *sak* server. The first two routines do not actually do transactions with the server but package up the transaction that the server is to execute at a later date. The file containing this package is then submitted using *sak_submit_job*. *Sak_list_job* is used to examine the status of a job.

Access

Jobs are submitted using the server capability. No rights are required to submit a job. For each job submitted a capability is returned. The capability must be *touched* regularly (see *std*(L)) or it will be garbage collected after some time. The simplest way to ensure that jobs are *touched* is to install the capability in the directory graph. (*Cronsubmit*(U) does this, for example.)

Errors

In general, standard error codes are used but there is one error code specific to the *sak* server.

`SAK_TOOLATE` is returned when an attempt is made to submit a job after the time specified for the job to be executed.

Types

In the following, the variable *sched* specifies when the job is to be executed. *Sched* is an array of length MAX_SPEC. The entries are numbered,

MINU	minutes (0 - 59)
HOUR	hours (0 - 23)
MDAY	day-of-month (1 - 31)

MONTH	month (0 - 11)
WDAY	day-of-week (0 - 6, 0 is Sunday)
YEAR	year (any)

Any of the entries can be,

VALUE	followed by a value.
LIST	followed by a list of values.
RANGE	followed by two numbers specifying a range
ANY	meaning any possible value
NONE	meaning no value at all

All entries should have a `DELIMITER` appended (see the include file *sak.h).*

Specification of days can be done in one of two ways: day-of-month/month or day-of-week. If neither is specified as NONE, only one has to match for the transaction to be executed.

The *filecap* parameter is a capability for a transaction file created by *sak_make_transfile* which contains the transaction to be executed.

The *options* parameter is a structure which contains the following fields:

save_result	If set to 1, *sak* will write a log of all transactions executed to a file named *sak_statusfile*;
where	The capability for a directory which will hold the *sak* status file (default */home/sak*).
catchup	This specifies the action taken when the server was not able to execute the transaction at the specified time. If set to 1 it will execute it as soon as possible, if set to 0, it will report a failure. Jobs that are to be executed more then once should set this flag to 0.
name	A name of up to `MAX_NAME_LEN` characters which is used by the server in status reports and info requests.

Functions

sak_exec_trans

```
errstat
sak_exec_trans(argv, envp, capl, filecap)
char **argv;
char **envp;
struct caplist *capl;
capability *filecap;
```

Sak_exec_trans is used to generate a special file (on the default Bullet Server) which contains details of a transaction to be executed by the *sak* server. The function of this transaction will be to start the program specified by *argv[0]*, with arguments specified by the rest of *argv*, string environment specified by *envp* and capability environment specified by *capl*.

The resultant file should be submitted to the *sak* server using *sak_submit_job*.

When executed the transaction goes to a special thread within the *sak* server. The port used by this thread is found in DEF_EXECSVR (typically */profile/cap/saksvr/execdefault*). When a transaction is done to this port with the SAK_EXEC_FILE command specified, the command and argument list specified in the transaction buffer is executed using the string environment and capability environment also found in the transaction buffer. The arguments are as follows:

argv	a NULL-terminated argument list, *argv[0]* specifies the command to be executed.
envp	a NULL-terminated string environment list.
capl	a NULL-terminated capability environment list.

A capability for the created file is returned in *filecap*.

Required Rights:
    BS_RGT_CREATE on default Bullet Server.

Error Conditions:

STD_NOMEM:	cannot malloc a buffer for the data,

Any of the errors of *sak_make_transfile*.
Any of the errors of *name_lookup*(L).

sak_list_job

```
errstat
sak_list_job(jobcap, sched, options)
capability *jobcap;
int8 **sched;
struct sak_job_options *options;
```

Sak_list_job will return in *sched* a pointer to the schedule and in *options* the options associated with the job specified by *jobcap*.

Required Rights:
    OWNERRIGHTS

Error Conditions:

STD_COMBAD:	attempt to list super capability
STD_CAPBAD:	no such job
STD_DENIED:	insufficient rights
STD_NOMEM:	the job info did not fit in the server's internal buffer

sak_make_transfile

```
errstat
sak_make_transfile(hdr, buf, req_size, reply_size, wait, fcap)
header *hdr;
char *buf;
bufsize req_size;
bufsize rep_size;
interval wait;
capability *fcap;
```

Sak_make_transfile is used to create a transaction file (on the default Bullet Server) that can be used by *sak_submit_job*. The arguments are as follows:

hdr	is the header used in the transaction.
buf	is the buffer used in the transaction.
req_size	size of the transaction buffer.
reply_size	size of the expected reply buffer.
wait	timeout used for the transaction. *wait* has a maximum of 30 seconds.

A capability for the created file is returned in *filecap*.

Required Rights:
    `BS_RGT_CREATE` on default Bullet Server.

Error Conditions:
    Any of the errors returned by *b_create* or *b_modify* (see *bullet*(L)).

sak_submit_job

```
errstat
sak_submit_job(svrname, sched, filecap, options, jobcap)
char *svrname;
int8 **sched;
capability *filecap;
struct sak_job_options *options;
capability *jobcap;
```

Sak_submit_job submits a job to the *sak*(A) server specified by *svrname*. If *svrname* is a NULL-pointer, `DEF_SAKSVR` is used. The job (that is, transaction to be executed) is specified in the file *filecap* which should have been created using *sak_make_transfile*. The *sched* argument describes under what conditions the transaction should be executed. The *options* argument specifies what to do with any error status information. Details of the *options* and *sched* are given in the section *Types* above. A capability for the job is returned in *jobcap* if no errors occurred.

Required Rights:
    NONE

Error Conditions:

`SAK_TOOLATE`:	the specified time has already passed.
`STD_SYSERR`:	a system error occurred while initializing job.
`STD_NOMEM`:	insufficient resources within sak server.
`STD_ARGBAD`:	the time specification was not legal.
`STD_CAPBAD`:	invalid capability.
`STD_COMBAD`:	a job capability was used instead of a server capability.

See Also

bullet(L), sak(A).

(in libraries: libamoeba.a)

Name

seg_map – routines to control process virtual memory

Synopsis

```
#include "amoeba.h"
#include "module/proc.h"

segid seg_map(cap, addr, len, flags)
errstat seg_grow(id, newsize)
errstat seg_unmap(id, cap)
```

Description

These routines enable a user program to control its virtual address space. *Seg_map* creates a new segment, optionally filling it from a file or another segment, and maps the new segment in. *Seg_grow* grows or shrinks a segment. *Seg_unmap* unmaps a segment.

These calls are implemented with system calls, so no rights are involved, and a process can only manipulate its own address space in this way.

Functions

seg_map

```
    segid
    seg_map(cap, addr, len, flags)
    capability *cap;
    vir_bytes addr;
    vir_bytes len;
    long flags;
```

Seg_map creates a new segment of length *len* and maps it into the process' address space at address *addr*. *Addr* should be a multiple of ‘‘clicks’’. The size of a click is architecture dependent, but usually the same as the hardware page size. Also, architectures may place restrictions on the range of virtual addresses that may be used by user programs. The routine *findhole*(L) may be used to obtain an address where a segment may be mapped in and that satisfies all the necessary conditions.

If *cap* is non-NULL and not a NULL-capability the segment is initialized from the specified file or segment (using *b_read* calls, see *bullet*(L)). The *flags* specify how the segment should be mapped, whether it is growable, etc. A complete description of the bits in *flags* can be found in *process_d*(L).

On success, *seg_map* returns a segment identifier, a small integer that can be passed to *seg_grow* or *seg_unmap*.

Error Conditions:

`STD_ARGBAD`	Addr is not on a click boundary, the length is bigger than the maximum segment size, or the segment would overlap an existing segment.
`STD_COMBAD`	The flag word is invalid, i.e., no segment type or protection specified, or an attempt to map a non-local segment with `MAP_INPLACE` is made. Also returned on an attempt to map a zero-sized segment.
`STD_NOSPACE`	The machine has not enough physical memory available to create the segment, or some internal administrative table is full.
`STD_CAPBAD`	The capability specified does not refer to an existing file.

seg_unmap

```
errstat
seg_unmap(id, cap)
segid id;
capability *cap;
```

This call unmaps segment *id*. The segment id is either the value returned by a previous call to *seg_map*, or the index of the segment in the segment descriptor array returned by *getinfo*(L). If *cap* is not a NULL-pointer, a capability for the now unmapped segment is stored here. This segment remains attached to the program, and will be deleted when the program exits. If a NULL-pointer is specified for *cap*, the segment will be deleted immediately.

Error Conditions:

`STD_COMBAD`	The segment id does not refer to a currently mapped-in segment.

seg_grow

```
errstat
seg_grow(id, newsize)
segid id;
vir_bytes newsize;
```

This call changes the size of a mapped segment. Depending on the growth direction specified in the call to *seg_map*, the segment will be grown (or shrunk) at the high end or the low end.

Error Conditions:

`STD_COMBAD`	The segment does not exist, or it is a fixed size segment.
`STD_NOSPACE`	There is no physical memory available to grow the segment, or the new size specified would cause the segment to overlap with another segment.

Warnings

Due to problems in the current kernel it is often not possible to grow segments, even if enough physical memory is available. For this reason modules that do memory allocation (like *malloc*(L)) should always be prepared to allocate a completely new segment if it is impossible to grow a segment.

See Also

bullet(L), getinfo(L), process(L), process_d(L).

(in libraries: libamoeba.a, libamunix.a)

Name

segment – stubs to manage in-core segments

Synopsis

```
#include "amoeba.h"
#include "module/proc.h"

errstat ps_segcreate(server, size, orig, newcap)
errstat ps_segwrite(segcap, offset, buf, size)
```

Description

These stubs allow programmers to control in-core segments managed by Amoeba process servers. *Ps_segcreate* creates a segment, and *ps_segwrite* modifies it. Besides these calls the Bullet Server calls *b_read* and *b_size* (see *bullet*(L)) and all applicable standard calls (see *std*(L)) can also be applied to segments.

Rights bits

Of the rights used by the process server only three are applicable to segments:

`PSR_READ`	permission to read the segment (using *b_read*).
`PSR_WRITE`	permission to modify the segment.
`PSR_DELETE`	permission the delete the segment (using *std_destroy*).

Functions

ps_segcreate

```
     errstat
     ps_segcreate(server, size, orig, newcap)
     capability *server;
     long size;
     capability *orig;
     capability *newcap;
```

This routine asks the process server *server* to create a segment of length *size* bytes. The capability for the new segment is returned in *newcap*. If *orig* is not a NULL-pointer or a NULL-capability the new segment will be initialized from the file (or segment) *orig*, using *b_read* calls. Otherwise, the segment will be zero-filled.

Required Rights:

NONE

ps_segwrite

```
errstat
ps_segwrite(segcap, offset, buf, size)
capability *segcap;
long offset;
char *buf;
long size;
```

This routine modifies a segment. *Size* bytes from offset *offset* in segment *segcap* are written from *buf*. *Size* can be arbitrarily large; the stub routine converts big modification requests into many little ones.

Required Rights:
`PSR_WRITE`

Diagnostics

The routines return only standard error codes from *stderr.h*.

Warnings

Under some circumstances all rights (instead of only `PSR_READ` as expected) for the *orig* segment are required by the *ps_segcreate* call, and the same is true for the ‘‘initial contents’’ segment in the *seg_map* call (see *seg_map*(L)). For this reason it is currently not a good idea to strip rights from segment capabilities.

See Also

bullet(L), process(A), process(L), process_d(L), seg_map(L), std(L).

(in libraries: libamoeba.a, libamunix.a)

Name

semaphore – thread synchronization using counting semaphores

Synopsis

```
#include "amoeba.h"
#include "semaphore.h"

typedef ... semaphore;

void sema_init(psem, level)
int sema_level(psem)

void sema_up(psem)
void sema_down(psem)
int sema_trydown(psem, maxdelay)

void sema_mup(psem, count)
int sema_mdown(psem, count)
int sema_trymdown(psem, count, maxdelay)
```

Description

These operations implement counting semaphores. What follows is an intuitive explanation of semaphores, not a formal definition:

A semaphore contains a non-negative integer variable, usually called its level. The two important operations on semaphores are *up* and *down*, which increment and decrement the level, respectively. However, when a call to *down* would decrement the level below zero, it blocks until a call to *up* is made (by another thread) that increments the level above zero. This is done in such a way that the level will never actually go negative. You could also say that the total number of completed *down* calls for a particular semaphore will never exceed the total number of *up* calls (not necessarily completed), plus its initial level.

Types

The *semaphore* data type declared in *semaphore.h* is an opaque data type; its only use should be to declare semaphores. All operations on semaphores must be done through the functions below.

Errors

The semaphore package makes no consistency checks; illegal calls cause undefined behavior. The source may be compiled with a debugging flag which adds some consistency checks; if an illegal situation is detected a message is printed and the program is aborted.

Functions

sema_init

```
void
sema_init(psem, level)
semaphore *psem;
int level;
```

A semaphore must be initialized to a certain *level* by calling this function. This call is compulsory; semaphores initialized to all zeros have an illegal state. The initial level must not be negative.

sema_level

```
int
sema_level(psem)
semaphore *psem;
```

This function returns the semaphore's current level. Since this value may be invalid the next microsecond (if another thread changes the level), it is not useful for synchronization. It is provided in the interface to allow printing the value of a semaphore for debugging purposes.

sema_up

```
void
sema_up(psem)
semaphore *psem;
```

This function implements the *up* operation described above.

sema_down

```
void
sema_down(psem)
semaphore *psem;
```

This function implements the *down* operation described above.

sema_trydown

```
int
sema_trydown(psem, maxdelay)
semaphore *psem;
interval maxdelay;
```

This is a variant of the *down* operation that gives up when it has been blocked unsuccessfully for *maxdelay* milliseconds, or when the call is interrupted by a signal catcher (see *signals*(L)), or when the process is continued after a stun (see *process*(L)). If *maxdelay* is zero, it succeeds only if the *down* operation can proceed without blocking. If *maxdelay* is negative, an infinite delay is set, but the call may still be interrupted (unlike *sema_down*). *Maxdelay* may be rounded up to a multiple of the kernel's internal clock. The return value is zero for success, negative for failure.

sema_mup

```
void
sema_mup(psem, count)
semaphore *psem;
int count;
```

This function is equivalent to *count* calls to *sema_up*, but more efficient.

sema_mdown

```
int
sema_mdown(psem, count)
semaphore *psem;
int count;
```

This function is not quite equivalent to *count* calls to *sema_down*! When *count* <= 0, it is a no-op; otherwise, it is equivalent to between 1 and *count* calls to *sema_down*. It returns the actual number subtracted from the semaphore's level. This number is determined as follows. If, on entry into *sema_mdown*, the semaphore's level is > 0, the call proceeds immediately, otherwise it blocks until the level is raised above zero. In all cases, the return value is MIN(count, level). (Informally, you can think of all this as: *sema_mdown* tries to do as many calls as possible to *sema_down* with as little effort as possible.)

sema_trymdown

```
int
sema_trymdown(psem, count, maxdelay)
semaphore *psem;
int count;
interval maxdelay;
```

This call is a combination of the functionality of *sema_trydown* and *sema_mdown*. If it decides to block, it will block at most *maxdelay* milliseconds, or indefinitely but interruptible if *maxdelay* is < 0. If the level is still zero at the end of the delay period, the semaphore is not updated and the return value is negative; otherwise the effect and return value are the same as for *sema_mdown*.

Warning

Overflow of the *level* field is not detected.

Example

The code below sketches the solution to the standard problem of a producer and consumer with a limited buffer, using counting semaphores. It lacks a main program that executes the producer and consumer threads in parallel. Extension to multiple producers and/or consumers are left as exercises for the reader (hint: the *next_slot* variables must be shared per group and protected with mutexes).

```
#define N_SLOTS 10
typedef ... item;
item buffer[N_SLOTS];
semaphore slots_filled, slots_empty;

init()
{
    /* Buffer starts empty */
    sema_init(&slots_filled, 0);
    sema_init(&slots_empty, N_SLOTS);
}

producer()
{
    int next_slot = 0;

    for (;;) {
        sema_down(&slots_empty);
        "produce an item into buffer[next_slot]"
        next_slot = (next_slot + 1) % N_SLOTS;
        sema_up(&slots_filled);
    }
}

consumer()
{
    int next_slot = 0;

    for (;;) {
        sema_down(&slots_filled);
        "consume an item from buffer[next_slot]"
        next_slot = (next_slot + 1) % N_SLOTS;
        sema_up(&slots_empty);
    }
}
```

See Also

Edsger W. Dijkstra, *Cooperating Sequential Processes*.

A. Van Wijngaarden et.al., *Revised Report on the Algorithmic Language Algol-68*.

mutex(L), signals(L).

(in libraries: libamoeba.a)

Name

signals − signal and exception handling

Synopsis

```
#include "amoeba.h"
#include "exception.h"
#include "fault.h" /* From src/h/machdep/arch/<architecture> */
#include "module/signals.h"

typedef long signum;
#define SIG_TRANS 1

signum  sig_uniq()
void    sig_raise(sig)
errstat sig_catch(sig, catcher, extra)
void    sig_block()
void    sig_unblock()

void    my_catcher(sig, us, extra)

/* Low-level interface: */

typedef struct {...} sig_vector;

sys_setvec(v, n)
sys_sigret(us)

my_sys_catcher(sig, us, arg3, arg4)
```

Description

Signals are used for asynchronous communication between threads within a process. A thread may specify a *catcher* function for a particular *signal number*; another thread may asynchronously cause a call to this catcher in the catching thread by *generating* a signal. The signal catching mechanism is also used to catch exceptions and client-to-server signals; more about those later.

Do not confuse signals with *stuns*, which are used for asynchronous communication between different processes (see *process*(L)). Signals are also used to emulate the C/ UNIX signal mechanism (see *ansi_C*(L) and *posix(L)).*

When a signal is generated, it is broadcast to all threads in the same process that have a catcher for it. This may include the thread that generated the signal. Threads that have no catcher for the signal are completely unaffected. In threads that catch the signal, system calls are completed or aborted (see the next paragraph); the catcher is called just before the system call returns. If the catcher returns, the thread continues where it was interrupted. A thread may temporarily block signals; in this case the calls to catchers are postponed until signals are unblocked again (but system calls are still aborted).

RPC system calls (see *rpc*(L)) are aborted or continue depending on the state they are in when the signal is generated:

trans	When locating the server or sending the request, the call is aborted. When the server is already processing the request, a client-to-server signal is generated in the server (see below) and the call continues.
getreq	When waiting for an incoming request, the call is aborted. When a request is already arriving, it is allowed to complete.
putrep	Always allowed to complete.

Other system calls always continue, since they cannot block very long, with two exceptions.

mu_trylock	returns −1, for failure to acquire the mutex (see *mutex*(L)). Note that *mu_lock* is not a system call but loops until *mu_trylock* succeeds; thus, the catcher is called immediately but the *mu_lock* call does not return until it has acquired the mutex, which may be much later. Note that there are special versions of the mutex routines which block signals from having an effect until the critical region is complete. These are described in *mutex*(L).
seg_map	(see *segment*(L)) is aborted when it is busy in a transaction reading the initial data for a segment; otherwise it continues.

When a thread is busy in a *trans* and the server is already serving the request, the transaction is not aborted but a *client-to-server* signal is generated in the server (which may be executing on another machine). This client-to-server signal (`SIG_TRANS`) is not broadcast; it is only generated for the thread that is serving the transaction, and only if it has a catcher for it. When the server thread catches the client-to-server signal, it may decide to stop whatever it is doing and reply immediately. By convention, interrupted servers report a special error, `STD_INTR`. This is up to the server; many servers ignore this signal (although this is a bug if the server itself may wait or compute indefinitely). The client-to-server signal is generated under other circumstances as well, e.g., when a process is stunned or crashes.

When a thread causes an exception (e.g., it tries to divide by zero), and it has a catcher for the corresponding (negative) exception number, that catcher is called. Exceptions are not broadcast to other threads, nor can they be blocked by the blocking mechanism or ignored. When a thread causes an exception for which it has no catcher, the process is stunned and its owner receives a checkpoint. The owner usually prints a stack trace (e.g., *ax*(U)) or saves a ‘‘core image’’ (e.g., *session*(U)), and then kills the process. A debugger may allow close inspection of the process and let the user attempt repair and continuation.

Type

The type

```
signum
```

is used to specify signal numbers. This is an integral type; it is defined in *exception.h* (see *exception*(H)). Ordinary signals are identified by numbers larger than one. Signal number one (`SIG_TRANS`) identifies the client-to-server signal. Negative numbers identify exceptions; the names of the exceptions are defined in *exception.h*. Signal number zero is unused (it means ‘‘no signal’’ internally). Small signal numbers are reserved by the UNIX signal emulation; applications should use larger numbers.

High-level Functions

sig_uniq

```
signum
sig_uniq()
```

Each call returns a unique signal number in the range [2ˆ16 .. 2ˆ31). It should be called no more than 2147418112 times in one program.

sig_raise

```
void
sig_raise(sig)
signum sig;
```

This broadcasts the signal number *sig* to all threads in the same process that have catchers for it. The signal number should be larger than one.

sig_catch

```
errstat
sig_catch(sig, catcher, extra)
signum sig;
void (*catcher)();
void *extra;
```

This specifies that signal *sig* must be caught in the current thread by the function *catcher* with extra argument *extra*. If a NULL-pointer is passed as catcher, the signal will be ignored.

The catcher should be declared by the user as follows (the function name and parameter names may be freely chosen):

```
void
my_catcher(sig, us, extra)
signum sig;
thread_ustate *us;
void *extra;
```

where *sig* is the signal number, *us* is the pointer to machine dependent user state at the time of the call (only useful for code that attempts to repair exceptions, and not available to calls postponed by the blocking mechanism), and *extra* is the *extra* argument passed to the corresponding *sig_catch* call. The catcher is called in the catching thread. If and when it returns, the thread continues at the point where it was interrupted. System calls may be aborted as described earlier.

Note well: handlers are not cleared when an exception occurs. If exceptions are caught then the exception handler should not return without taking corrective action since an attempt will be made to restart the offending instruction. This will result in an infinite loop of calling the exception handler. One should certainly not ignore exceptions since the program will go into an infinite loop attempting to restart the offending instruction.

sig_block

```
void
sig_block()
```

This blocks calls to signal catchers in the current thread until a matching call to *sig_unblock* is made. Pairs of calls to *sig_block* and *sig_unblock* may be nested. They can be seen as brackets around critical sections; only the last (outermost) call to *sig_unblock* will unblock signals. Blocking signals only delays the call to the catcher; the other effects of catching a signal (aborting system calls, sending client-to-server signals) still happen immediately when the signal is generated. Exception catchers are unaffected by the blocking mechanism.

sig_unblock

```
void
sig_unblock()
```

This unblocks calls to signal catchers in the current thread, when not nested between another pair of calls to *sig_block* and *sig_unblock*. Any catcher calls pending for the current thread are performed at this point. Catchers called through *sig_unblock* get a NULL-pointer passed for their frame pointer argument, since the frame will probably be invalid by the time the call is made.

Low-level Functions

The following structure type is defined in *exception.h*:

```
typedef struct {
    signum  sv_type;
    long    sv_pc;
    long    sv_arg3;
    long    sv_arg4;
} sig_vector;
```

It is used for the vector argument to *sys_setvec*.

sys_setvec

```
sys_setvec(v, n)
sig_vector *v;
int n;
```

This is the kernel interface used by *sig_catch*. The argument *v* is a vector with *n* entries of type *sig_vector*. When a signal is generated, the kernel searches the vector for a catcher, from beginning to end, until either an entry is found whose *sv_type* matches the signal number, an entry is found whose *sv_type* is zero is found (an optional sentinel), or *n* entries have been inspected. If no matching entry is found or its *sv_pc* is a NULL-pointer, the signal is ignored; otherwise, the signal is caught and *sv_pc* points to the system catcher function. The system catcher should be declared as follows (the function name and parameter names may be freely chosen):

```
my_sys_catcher(sig, us, arg3, arg4)
signum sig;
thread_ustate *us;
long arg3, arg4;
```

where arguments *sig* and *us* are as for the user catcher, and arguments *arg3* and *arg4* are copied from *sv_arg3* and *sv_arg4* from the vector entry. The implementation of *sig_catch* uses *arg3* to hold the user catcher and *arg4* to store the extra argument.

The program may modify the vector entries while it is running; this makes specifying different catchers fast. Changing the size or location of the vector on the other hand requires a call to *sys_setvec* to tell the kernel about the new values.

The system catcher must not return directly; instead, it should call *sys_sigret(us)*, see below. It may also ‘‘return’’ via a call to *longjmp* (see *ansi_C*(L)). Calls that terminate the thread or process are also allowed. System catchers are unaffected by *sig_block* described above; the system catcher used by *sig_catch* implements signal blocking by saving the pending catcher call when necessary.

sys_sigret

```
sys_sigret(us)
thread_ustate *us;
```

This function must be called to return from a system catcher; the stack frame of a system signal catcher looks peculiar so an ordinary return statement would crash the program. The argument must be the *us* argument passed to the system catcher.

Diagnostics

Possible return values from *sig_catch* are:

`STD_OK`	all went well.
`STD_NOMEM`	no memory was available to update the signal administration.
`STD_ARGBAD`	an invalid signal number was given (e.g., zero).

Warnings

Resist the temptation to put calls to *printf* (see *ansi_C*(L)) in a catcher function, even for debugging. When such a catcher is called during another *printf* call (or any other stdio call), the stdio administration may be damaged and output garbled or further printing made impossible. It is safe to call *write* however (see *posix*(L)).

A thread which never makes a blocking system call cannot be interrupted by a signal when non-preemptive scheduling is used by the process. This happens because in that case no other thread is scheduled to generate a signal.

When more than one signal arrives for a thread while signals are blocked, it is not guaranteed that all catcher calls will be made; only the catcher call corresponding to the signal that arrived last is guaranteed.

Returning from an exception catcher is usually not a good idea. Unless you have repaired its cause, the same exception will be triggered again immediately. It usually requires a fair amount of machine-dependent hacking to find out what caused the exception, let alone to repair it. Breakpoint exceptions are not regenerated, but they are used by the debugger and should not be caught at all.

Do not use the *sys_setvec* or *sys_sigret* calls; they are considered part of the implementation and are documented here only for completeness' sake.

Code which does a lot of processing in a signal catcher is frowned upon. The most important reason for catching signals is to break out of unbounded waits; the catchers are only there to make emulation of UNIX signals possible.

The *read* function (see *posix(L))* interacts somewhat with signals; when it receives a `STD_INTR` error from the server, it assumes the request was interrupted by a signal catcher and sets *errno* to `EIO`.

Example

The following program sleeps two seconds and then prints the string *catcher called*:

```
#include "amoeba.h"
#include "exception.h"
#include "fault.h"
#include "module/signals.h"

int flag = 0;

/*ARGSUSED*/
void catcher(sig, us, extra)
signum sig;
thread_ustate *us;
void *extra;
{
    flag = 1;
}

void async()
{
    sleep(2);
    sig_raise(1234);
}

main()
{
    sig_catch(1234, catcher, (void *) 0);
    thread_newthread(async, 8000, (char *) 0, 0);
    sleep(5);
    if (flag) printf("catcher called\n");
    exit(0);
}
```

See Also

ax(U), exception(H), mutex(L), posix(L), process(L), rpc(L), session(U), ansi_C(L), thread(L).

(in libraries: libamoeba.a, libkernel.a, libamunix.a)

Name

soap – the Soap Server client interface stubs

Synopsis

```
#include "capset.h"
#include "server/soap/soap.h"

errstat sp_append(dir, name, cs, ncols, cols)
errstat sp_chmod(dir, name, ncols, cols)
errstat sp_create(server, columns, dir)
errstat sp_delete(dir, name)
errstat sp_discard(dir)
errstat sp_getmasks(dir, name, ncols, cols)
errstat sp_getseqno(dir, seqno)
errstat sp_install(n, entries, capsets, oldcaps)
errstat sp_lookup(dir, path, cs)
errstat sp_replace(dir, name, cs)
errstat sp_setlookup(n, in, out)
errstat sp_traverse(dir, path, last)

#include "sp_dir.h"
errstat sp_list(dir, dd)
```

Description

This manual page describes the Soap Server stub routines. Along with the standard server stubs (see *std*(L)) the Soap Server stubs provide the interface to the Soap Server. In addition to the stubs there are several derived and related functions to provide a simple interface for common operations. These are described in *sp_dir*(L), *sp_mkdir*(L) and *sp_mask*(L). For details about the server see *soap*(A).

Types

The type `SP_DIR` is defined in the include file *sp_dir.h*. It is analogous to the type `DIR` used for accessing directories.

Another important type is *capset*, which is used to store a set of capabilities, possibly the empty set. The type *capset* is defined in *capset.h*.

The type *sp_entry* is defined in *soap.h*. It is used for storing or retrieving one or more (name, capset) pairs in a single operation.

The type *sp_seqno_t* is used to represent the 64-bit directory sequence numbers, which are supported by some directory servers. It is defined in *soap.h*.

Access

Access to Soap objects is on the surface quite complicated, but extremely powerful and flexible. Soap directories are objects and each is therefore accessed using its capability. The following rights bits are defined for a directory entry:

`SP_COLMASK`	selects which column mask will be applied
`SP_DELRGT`	permission to delete a directory
`SP_MODRGT`	permission to modify a directory

When a capability-set is looked up in Soap, the rights bits in the capability for the directory determine the access permissions for the entries in that directory as follows. With each entry in a directory there is a set of masks which correspond to the columns of the directory. At present there is a maximum of four columns, but typically only three are used: *owner* (column 0), *group* (column 1) and *other* (column 2). The rights bits corresponding to columns (i.e., bits 0 to 3) select which column of the directory may be accessed. For all the columns to which the rights bits are on in the directory capability, the rights masks for those columns are bitwise OR-ed together. The resulting rights mask is used to generate a restricted version of the original capability-set stored for the name being looked up. This new capability-set is returned by Soap to the caller.

The generation of a restricted capability-set goes as follows. In case the original capabilities contain all rights, the Soap Server is able to generate the restricted versions by itself, using *prv_encode*. Otherwise, the capabilities and the restriction mask are passed to the corresponding servers to perform an `STD_RESTRICT` operation.

Note that to perform any operation on a directory entry one must have access to at least one column of the directory. Otherwise it is not possible to proceed further down that part of the directory hierarchy.

Time

Soap stores a *last updated* time-stamp with each directory entry. It uses a precision of seconds. This time-stamp cannot be relied upon to be accurate since it is derived from a Time of Day server (see *tod*(A)), but not necessarily always the same one. In general it will be moderately accurate but if one TOD server goes down and another is available it will be used. Therefore time-stamps may not be monotonically increasing. This causes programs such as *make*(U) to be even more unreliable than under UNIX and it is better to create programs that do not depend on a consistent view of the time of day.

Errors

All of the stub functions return an error status as the value of the function. Soap returns all the standard error codes (as defined in *stderr.h*) plus the following:

`SP_UNAVAIL`	This is returned if the Soap Server is not currently accessible. (It is usually equivalent to `RPC_NOTFOUND`).
`SP_UNREACH`	This is returned if during the lookup of a capability-set, the Soap Server encounters a capability for a directory within the same Soap Server, that does not exist. This can happen if someone deletes a directory to which multiple links exists. Note that when the object number of a deleted

	directory is reused by Soap for a new directory, the error code `STD_CAPBAD` will be returned instead, because the new directory will have a new check field.
`SP_NOTEMPTY`	This is returned if an attempt was made to destroy a directory that was not empty. This should not be confused with deleting a directory entry that points to a non-empty directory, which is perfectly legitimate.
`SP_CLASH`	This is returned if an attempt is made to replace a capability-set and the non-zero *original* capability provided is not a (restricted) member of it. (The old capability-set must be deleted before the new set can be installed if this is intentional.)

All of the stubs may return `SP_UNAVAIL`, any of the `RPC_` errors, `STD_CAPBAD`, `STD_DENIED`, `STD_ARGBAD` and `STD_NOSPACE`. The latter is inherent in the nature of capability-sets and even on delete operations it is possible to get this error.

Path names

Path names are always specified as a string relative to a directory whose capability-set is known. To perform operations relative to the current working directory (whatever that might be) the *dir* parameter in the library routines that take path name arguments can be specified as `SP_DEFAULT`.

Functions

sp_append

```
errstat
sp_append(dir, name, cs, ncols, cols)
capset     *dir;
char       *name;
capset     *cs;
int         ncols;
rights_bits cols[];
```

Sp_append adds the directory entry specified by *name*, relative to the directory specified by *dir*. The entry may not exist already. The new entry has the capability-set *cs*.

The column masks for the directory entry are specified by the *ncols* entries in the array *cols*. To avoid first having to look up the number of columns the directory has, any legal number of columns masks (1 up to `SP_MAXCOLUMNS`) is accepted. Masks referring to non-existent columns are ignored, and missing masks are set to zero.

Required Rights:

`SP_MODRGT`
Access to at least one column of the directory to which the new entry is being added.

Error Conditions:

`STD_EXISTS` the specified name is already in use.

sp_chmod

```
errstat
sp_chmod(dir, name, ncols, cols)
capset     *dir;
char       *name;
int         ncols;
rights_bits cols[];
```

Sp_chmod allows one to alter the column masks for the directory entry specified by the path name *name* relative to the directory with capability-set *dir*. The entry has to exist already. The new column masks are set to the *ncols* entries in the array *cols*. Like in *sp_append*, superfluous masks are ignored and missing masks are set to zero.

Required Rights:

`SP_MODRGT`
Access to at least one column of the directory containing the entry to be modified.

Error Conditions:

`STD_NOTFOUND`: the specified name does not exist.

sp_create

```
errstat
sp_create(server, columns, dir)
capset *server;
char   *columns[];
capset *dir;
```

Sp_create creates a new Soap directory on the Soap Server specified by *server*. The column names for the new directory are specified in *columns*, which is a NULL-terminated array of strings. (There are no default names so *columns* must be specified.) The number of strings determines the number of columns for the directory. The capability-set for the new directory is returned in *dir*.

NB. It does not enter the new directory into another directory. (That is the function of *sp_append*. See *sp_mkdir*(L) for a simple method of creating and installing a new directory.)

Required Rights:

Access to at least one column of the directory given as first argument.

sp_delete

```
errstat
sp_delete(dir, name)
capset *dir;
char   *name;
```

Sp_delete deletes the directory entry (which may itself be a directory capability) specified by *name*. *Name* is a path name relative to the directory whose capability-set is given in *dir*.

Required Rights:
`SP_MODRGT`
Access to at least one column of the directory containing the entry to be deleted.

sp_discard

```
errstat
sp_discard(dir)
capset *dir;
```

Sp_discard is equivalent to performing an *std_destroy* on the specified directory. This can only be performed on a directory that is empty (i.e., has no entries in it).

Required Rights:
`SP_DELRGT` for the directory to be destroyed.
Access to at least one column of the directory to be destroyed.

Error Conditions:
`SP_NOTEMPTY`: directory to be deleted is not empty.

sp_getmasks

```
errstat
sp_getmasks(dir, name, ncols, cols)
capset      *dir;
char        *name;
int          ncols;
rights_bits cols[];
```

Sp_getmasks returns in *cols* the column rights masks of the directory with path name *name* relative to the directory with capability-set *dir*. If *ncols* is less than the number of columns in the directory, only the masks for the first *ncols* columns will be returned in *cols*. If *ncols* is greater than the number of columns in the directory, the remaining column masks in *cols* will be set to zero.

Required Rights:
Access to at least one column of the directory containing the entry.

Error Conditions:

STD_NOTFOUND: the specified name does not exist.

sp_getseqno

```
errstat
sp_getseqno(dir, seqno)
capset     *dir;
sp_seqno_t *seqno;
```

Sp_getseqno returns in *seqno* the 64-bit sequence number of the directory with capability-set *dir*, if this is supported by the directory server. The sequence number is incremented with each directory modification. This information is useful for backup programs (e.g., *starch*(U)) when making an incremental file system dump.

Required Rights:

Access to at least one column of the directory.

Error Conditions:

STD_COMBAD sequence numbers are not supported by this server.

sp_install

```
errstat
sp_install(n, entries, capsets, oldcaps)
int         n;
sp_entry    entries[];
capset      capsets[];
capability *oldcaps[];
```

Sp_install implements atomical update of a set of existing directory entries. They may be entries in several different directories. The modify right is required for all the affected directories. The operation only succeeds if all the capability-sets can be installed. Otherwise no changes are made to any of the entries.

The parameter *n* specifies the number of elements in the three arrays. Each element of the array *entries* contains the *capset* for the directory containing the entry to be updated and the *name* of the directory entry within that directory which is to be updated. The type *sp_entry* is defined in *soap.h* as:

```
typedef struct {
        capset se_capset;
        char  *se_name;
} sp_entry;
```

For each directory entry described in *entries* the corresponding new capability-set in *capsets* is installed. However, this is only done if the corresponding capability in *oldcaps* is either NULL, or present (possibly as a restricted version) in the current capability-set. This allows for optimistic concurrency control by letting one read a set of values and then ensure that they have not changed when the new values are written back.

It is not possible to change the same entry twice in one command, since the optimistic concurrency control mechanism will reject the second change. Note that this can happen if there exist two distinct paths to the same directory.

NB. All the entries specified must be resident on the same Soap Server.

Required Rights:
  SP_MODRGT for the directories containing the entries.
  Access to at least one column of the directories containing the entries.

Error Conditions:
  SP_CLASH: a value in *oldcaps* did not match the corresponding current entry.

sp_list

```
#include "sp_dir.h"

errstat
sp_list(dir, dd)
capset  *dir;
SP_DIR **dd;
```

Sp_list returns in *dd* the entire list of directory entries in the directory specified by *dir*. The SP_DIR data structure is defined in *sp_dir.h* as follows:

```
typedef struct _dirdesc {
        capset dd_capset;
        int    dd_ncols;
        int    dd_nrows;
        char **dd_colnames;
        struct sp_direct *dd_rows;
        int    dd_curpos;
} SP_DIR;

struct sp_direct {
        char  *d_name;       /* name (up to MAX_NAME + 1)   */
        int    d_namlen;     /* redundant, but compatible   */
        long  *d_columns;    /* rights masks in the columns */
};
```

dd_capset is initialized with *dir*, *dd_ncols* with the number of columns in *dir*, and *dd_nrows* with the number of entries. The names of the columns are stored in *dd_colnames*, which is a string array of *dd_ncols* entries. The entry *dd_rows* is an array of *dd_nrows* entries, describing the (NULL-terminated) name in the row, the length of the name, and the rights masks. The entry *d_columns* is an array with *dd_ncols* entries containing the rights masks for each of the columns. The entry *dd_curpos* is a magic cookie for the implementation of the POSIX directory interface (see *sp_dir*(L) and *posix*(L)).

Required Rights:
    Access to at least one column of the directory to be listed.

sp_lookup

```
errstat
sp_lookup(dir, path, object)
capset *dir;
char   *path;
capset *object;
```

Sp_lookup returns in *object* the capability-set stored under the name *name* relative to the directory *dir*. Warning: if the NULL-string is given as the *path* then the capability in *dir* is for the directory required and so it is returned without checking to see if it is a valid capability.

Required Rights:
    Access to at least one column of the directory containing the object.

sp_replace

```
errstat
sp_replace(dir, name, cs)
capset *dir;
char   *name;
capset *cs;
```

Sp_replace performs a similar function to *sp_install* but for a single directory entry. It atomically updates the directory entry with path name *name* relative to directory *dir*. It overwrites the capability-set for that entry with the one in *cs*.

Required Rights:
    SP_MODRGT
    Access to at least one column of the directory containing the entry.

Error Conditions:
    STD_NOTFOUND: the specified path name does not exist.

sp_setlookup

```
errstat
sp_setlookup(n, in, out)
int       n;
sp_entry  in[];
sp_result out[];
```

For each directory entry described in *in*, the capability-set, the type capability, the update time, and an error status are returned in the corresponding entry in *out*. Both *in* and *out* contain *n* entries. The structure *sp_result* is defined in *soap.h* as follows:

```
typedef struct {
        short      sr_status;
        capability sr_typecap;
        long       sr_time;
        capset     sr_capset;
} sp_result;
```

The field *sr_status* is an error status. If it is not equal to `STD_OK` then the other data in the structure are invalid. The operation is atomic, that is, it returns the situation as it was at one moment in time.

NB. Some implementations dictate that all entries have to reside in the directory server as specified by the first entry in *in*. For all entries not available at this server the *sr_status* field will contain `STD_NOTFOUND`.

Required Rights:
    Access to at least one column of the directory containing each entry.

sp_traverse

```
errstat
sp_traverse(dir, path, last)
capset *dir;
char  **path;
capset *last;
```

Sp_traverse returns in **path* a pointer to the last component of the path name initially in **path*. It also returns in *last* the capability for the directory up until the last component of the path (relative to the directory *dir*).

There are several special cases. The path name / which will return a zero length path name in *path* and the capability-set for the root directory. The names *dot*(.) and *dot-dot*(..) at the end of a path name will not be normalized away. A path name which contains no / character will return in *last* the capability-set in *dir* and *path* will be unchanged.

Required Rights:
    Access to at least one column of the directory containing each element of the path.

See Also

chm(U), chbul(A), del(U), dir(U), mkd(U), get(U), put(U), rpc(L), soap(A), std(L), std_age(A), std_destroy(U), std_info(U), std_copy(U), std_restrict(U), std_touch(U).

(in libraries: libamoeba.a, libkernel.a, libamunix.a)

Name

sp_dir – routines which implement the POSIX-like directory commands for the Soap Server

Synopsis

```
#include "sp_dir.h"

void sp_closedir(dd)
SP_DIR *sp_opendir(name)
struct sp_direct *sp_readdir(dd)
void sp_rewinddir(dd)
void sp_seekdir(dd, pos)
long sp_telldir(dd)
```

Description

This set of routines implement an interface to the Soap Server which is similar to that required by POSIX. However the management of the *errno* variable is not done. The POSIX versions of these functions are available in the Amoeba libraries.

sp_closedir

```
    void
    sp_closedir(dd)
    SP_DIR *dd;
```

This routine closes the directory pointed to by *dd* and frees any associated resources. *Dd* must point to a directory opened by *sp_opendir*.

sp_opendir

```
    SP_DIR *
    sp_opendir(name)
    char *name;
```

This routine opens the directory whose path name is *name*, allocates storage, fills it with the details of the directory and returns a pointer to the allocated data. This pointer must be used in subsequent operations on the directory. If the *name* does not exist, is not a directory, or if memory allocation fails or reading the directory fails, the routine returns the NULL-pointer.

sp_readdir

```
struct sp_direct *
sp_readdir(dd)
SP_DIR *dd;
```

This routine returns a pointer to the next directory entry in the directory pointed to by *dd*. If there are no more entries in the directory it returns the NULL-pointer. The structure of the directory entry is defined in *sp_dir.h*.

sp_rewinddir

```
void
sp_rewinddir(dd)
SP_DIR *dd;
```

This routine moves the current position within the directory specified by *dd* to the beginning. A subsequent read will return the first entry in the directory.

sp_seekdir

```
void
sp_seekdir(dd, pos)
SP_DIR *dd;
long pos;
```

This routine moves the current position within the directory specified by *dd* to the entry number *pos*. A subsequent read will return the entry at position *pos*.

sp_telldir

```
long
sp_telldir(dd)
SP_DIR *dd;
```

This routine returns the current position in the directory specified by *dd*.

See Also

posix(L), soap(L).

(in libraries: libamoeba.a, libamunix.a)

Name

sp_mask – returns the information from the `SPMASK` environment variable

Synopsis

```
#include "amoeba.h"
#include "capset.h"
#include "soap/soap.h"

int
sp_mask(ncols, cols)
int ncols;
long cols[];
```

Description

Sp_mask reads the string environment variable `SPMASK` and returns the column masks described therein in *cols*. The parameter *ncols* gives the size of the array *cols*. The function returns the actual number of columns it filled in. If the environment variable is not defined it returns 0xFF for the number of columns specified by *ncols*.

The value of `SPMASK` is used by several utilities to provide a default set of column masks for new directory entries.

Environment Variables

`SPMASK` is expected to be a colon separated list of rights masks. The default base is 10 but base 16 can be used by preceding the digits with 0x. For example,

```
SPMASK=0xff:0xf:0x2:0x8
```

See Also

soap(A), soap(L).

(in libraries: libamoeba.a, libamunix.a)

Name

sp_mkdir – create and install a Soap directory

Synopsis

```
#include "amoeba.h"
#include "capset.h"
#include "soap/soap.h"

int
sp_mkdir(start, path, colnames)
capset *start;
char *path;
char **colnames;
```

Description

This function creates a soap directory. It is merely a convenience, replacing calls to *sp_traverse*, *sp_create*, *sp_mask*, *sp_append* and some error checking.

The arguments *start* and *path* together specify the name of the directory to be created. Normally, *start* is `SP_DEFAULT` and *path* is the absolute or relative path name for the directory to be created. If *start* is not `SP_DEFAULT`, *path* specifies the path name relative to the directory specified by the capability-set *start*.

The argument *colnames* specifies the number of columns of the new directory and their names. It should be a NULL-pointer or a pointer to a NULL-terminated array of strings giving the column names. If it is a NULL-pointer, the new directory has three columns with names *owner*, *group* and *other*. A directory can have at most `SP_MAXCOLUMNS` columns (this is typically 4).

The directory is appended to its parent directory with column masks computed by *sp_mask*(L).

Diagnostics

Error returns are those returned by the Soap routines called. In particular, *path* must not be empty, and its last component must be a non-existent entry in an existing directory with create and append rights. If an error occurs, all resources allocated by the call are released.

Environment Variables

`SPMASK` as used by *sp_mask*(L).

Example

The following is the source to the utility *mkd*. It creates directories named by its arguments.

```
#include "amoeba.h"
#include "cmdreg.h"
#include "stderr.h"
#include "capset.h"
#include "soap/soap.h"

#include "stdio.h"

main(argc, argv)
int argc;
char **argv;
{
    char *err_why();

    int i;
    int err;
    int sts;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s name ...\n", argv[0]);
        exit(1);
    }
    sts = 0;
    for (i = 1; i < argc; ++i) {
        err = sp_mkdir(SP_DEFAULT, argv[i], (char **)NULL);
        if (err != STD_OK) {
            fprintf(stderr, "%s: can’t create %s (%s)\n",
                argv[0], argv[i], err_why(err));
            sts = 1;
        }
    }
    exit(sts);
}
```

See Also

soap(L), sp_mask(L).

(in libraries: libamoeba.a, libkernel.a, libamunix.a)

Name

std – interface for standard server commands

Synopsis

```
#include "amoeba.h"
#include "module/stdcmd.h"

errstat std_age(cap)
errstat std_copy(servercap, origcap, newcap)
errstat std_destroy(cap)
errstat std_getparams(cap, parambuf, bufsz, paramlen, nparams)
errstat std_info(cap, infobuf, bufsz, len)
errstat std_restrict(cap, mask, newcap)
errstat std_setparams(cap, parambuf, bufsz, nparams)
errstat std_status(cap, buf, n, len)
errstat std_touch(cap)
errstat std_ntouch(svr_port, n, privbuf, num_done)
```

Description

The *std* module provides the interface to the standard commands which are supported by nearly all servers. The purpose of these routines is to provide a standard way for programs to deal with common aspects of servers. For precise details about rights bits and whether a particular server handles a particular *std* call, consult the manual entry for the server.

Errors

The value of all the *std* functions is the error status. All the *std* stubs return only standard error codes (as defined in *stderr.h*). They all use transactions and therefore, in addition to the errors described below, may all return the RPC error codes which relate to transaction errors.

Warning

Some older servers may return `STD_CAPBAD` instead of `STD_DENIED` when there are insufficient rights for an operation.

std_age

```
    errstat
    std_age(cap)
    capability *cap;
```

Std_age is used to tell a server to perform a garbage collection cycle. The parameter *cap* is the super capability for the server.

Required Rights:
All rights bits must be set in *cap* to perform the operation.

Error Conditions:

`STD_CAPBAD`:	the capability was invalid.
`STD_DENIED`:	insufficient rights.

std_copy

```
errstat
std_copy(server, orig, new)
capability *server;
capability *orig;
capability *new;
```

Std_copy requests the server specified by *server* to make a copy of the object specified by the capability *orig* and returns in *new* the capability for the new copy of the object. The server should check that the type of object that it is being asked to copy is the type of object that it manages. Servers that manage physical objects (such as disks) may not perform this command.

Required Rights:
The *server* capability must have the server's create right (if any) and the *orig* capability must give read permission.

Error Conditions:

`STD_CAPBAD`:	the *server* capability or *orig* capability was invalid.
`STD_DENIED`:	the create right was not defined in *server* capability, the read right was not defined in the *orig* capability.
`STD_COMBAD`:	the server specified by *server* does not perform the copy operation.

std_destroy

```
errstat
std_destroy(cap)
capability *cap;
```

Std_destroy requests the server to destroy the object specified by *cap*.

Required Rights:
Write and/or destroy rights.

Error Conditions:

`STD_CAPBAD`:	invalid capability.
`STD_DENIED`:	insufficient rights.

std_getparams

```
errstat
std_getparams(cap, parambuf, bufsz, paramlen, nparams)
capability *cap;
char       *parambuf;
int         bufsz;
int        *paramlen;
int        *nparams;
```

Std_getparams requests from the server a description of the runtime parameters associated with the object specified by the capability *cap*. The description is returned in the buffer *parambuf* which is *bufsz* bytes long. The size of the buffer returned by the server is returned in *paramlen*; the number of parameters is returned in *nparams*. The output parameters are only valid when `STD_OK` is returned. For each parameter there are four consecutive (NULL-terminated) strings in the buffer: its *name*, its *type*, a short *description* telling its meaning, and finally its current *value*. Note that non-string values, such as integers and booleans, will have to be converted to strings by the server itself.

Required Rights:

It is up to the server to decide which rights it requires in order to reveal parameter settings for an object. Note also that the server may have both per-object and global parameters. The latter should be retrieved from the server by presenting its super capability.

Error Conditions:

`STD_CAPBAD`:	invalid capability.
`STD_DENIED`:	insufficient rights.

std_info

```
errstat
std_info(cap, info, n, len)
capability *cap;
char       *info;
int        n;
int        *len;
```

Std_info requests from the server for (short) character array that describes the object specified by the capability *cap*. The string is returned in the buffer *info* which is *n* bytes long. The size of the buffer returned by the server is returned in **len*. If the function returns `STD_OK` then the value returned in **len* will be less than or equal to *n* and *buf* will contain the complete status message.

If the buffer provided was too small to contain the data, the error `STD_OVERFLOW` will be returned along with the first *n* bytes of the information. The value returned in **len* will be the size the buffer needs to be to contain all the data.

If any other error is returned then the value of **len* will not be modified.

Required Rights:
    The object’s read right.

Error Conditions:

`STD_CAPBAD`:	invalid capability.
`STD_DENIED`:	insufficient rights.
`STD_OVERFLOW`:	the info buffer was too small.

std_restrict

```
errstat
std_restrict(cap, mask, new)
capability  *cap;
rights_bits mask;
capability  *new;
```

Std_restrict requests the server managing the object specified by *cap* to produce a capability with the rights removed that are not set in *mask.* The new capability is returned in *new*.

It should be noted that *std_restrict* does not do a transaction with the server and verify the capability if all the rights are present in the capability. This means that it can return `STD_OK` and a restricted capability even when the original capability had an invalid check field. In this case the restricted capability will also be invalid.

Required Rights:
    None.

Error Conditions:

`STD_CAPBAD`:	invalid capability.

std_setparams

```
errstat
std_setparams(cap, parambuf, paramlen, nparams)
capability *cap;
char       *parambuf;
int         bufsz;
int         nparams;
```

Std_setparams is used to set a number of the server’s runtime parameters associated with the object specified by capability *cap*. The requested parameter assignments are specified in *parambuf* which is *bufsz* bytes long. The number of parameter assignments in the buffer is specified by *nparams*. For each assignment, there are two consecutive (NULL-terminated) strings in the buffer: the parameter’s *name*, and its new *value*. Note that the server itself is responsible for converting a supplied string value to integer, when a parameter’s type requires this.

Required Rights:

It is up to the server to decide which rights it requires for changing the parameter settings of an object. Note also that the server may have both per-object and global parameters. The latter should be changed by means of the server’s super capability.

Error Conditions:

`STD_CAPBAD`:	invalid capability.
`STD_DENIED`:	insufficient rights.
`STD_ARGBAD`:	parameter value out of range, or too many parameter settings.

std_status

```
errstat
std_status(cap, buf, n, len)
capability *cap;
char       *buf;
int        n;
int        *len;
```

Std_status requests the server for a character array containing current status information about the server specified by the capability *cap*. The data is returned in the buffer *buf* which is *n* bytes long. The size of the buffer returned by the server is returned in **len*. If the function returns `STD_OK` then **len* will be less than or equal to *n* and *buf* will contain the complete status message.

If the buffer provided was too small to contain the data, the error `STD_OVERFLOW` will be returned along with the first *n* bytes of the status information. The value in **len* will be the size the buffer needs to be to contain all the data.

If any other error is returned then the value in *len* will not be modified.

NB. The character array is not NULL-terminated.

Required Rights:

Some servers require more rights than others depending on the sensitivity of the data. Generally read permission is required, but some servers require that the super capability be presented.

Error Conditions:

`STD_CAPBAD`:	invalid capability.
`STD_DENIED`:	insufficient rights.
`STD_OVERFLOW`:	input buffer was too small.

std_touch

```
errstat
std_touch(cap)
capability *cap;
```

Std_touch is used to tell the server not to garbage collect the object specified by *cap*.

Required Rights:
None.

Error Conditions:
`STD_CAPBAD`: invalid capability.

std_ntouch

```
errstat
std_ntouch(svr_port, n, privbuf, num_done)
port    *svr_port;
int     n;
private *privbuf;
int     *num_done;
```

Std_ntouch is used to tell the server whose service port is *svr_port* not to garbage collect the *n* objects specified by the private parts of capabilities in the array *privbuf*. On completion it returns the number of objects successfully touched in *num_done*. There is no upper limit on the value of *n*, but *std_ntouch* may use more than one RPC to execute the touches if *n* is sufficiently large. The value of the function is `STD_OK` if all the specified objects were successfully touched. If any of the capabilities presented were rejected by the server then the error status of the last error generated is returned.

Required Rights:
None.

Error Conditions:
`STD_ARGBAD`: n <= 0.
`STD_CAPBAD`: invalid capability.

See Also

std_age(A), std_copy(U), std_destroy(U), std_info(U), std_params(A), std_restrict(U), std_status(U), std_touch(U).

(in libraries: libamoeba.a (sparc))

Name

sun4m_timer − microsecond resolution timer on the sun4m machines

Synopsis

```
#include "machdep/dev/sun4m_timer.h"

errstat sun4m_timer_init(hostname)
long sun4m_timer_diff(start, stop)
struct sun4m_timer_regs * sun4m_timer_getptr()
uint32 sun4m_timer_micro()
```

Description

The timer resolution provided by *sys_milli*(L) varies from milliseconds to deciseconds, depending on the architecture on which it is executed. The SPARCstations of the type `sun4m` provide a microsecond resolution timer which can be accessed by user programs. If the timer is present on a host, its capability is published under the name *usertimer* in the kernel directory of the host where it is available. Programs running on that host can map the timer into their address space, thus avoiding the time delays due to system call overhead.

sun4m_timer_init

```
    errstat
    sun4m_timer_init(hostname)
    char * hostname;
```

Sun4m_timer_init attempts to map the user timer capability from the kernel running on the host *hostname* into the address space of the current process. It is mapped read/write. If *hostname* is the NULL-pointer then it uses the name in the string environment variable `AX_HOST`. If this is not defined the error `STD_ARGBAD` is returned. `AX_HOST` is defined in the string environment by the programs *ax*(U) and *gax*(U).

The process must be running on the host with the specified name. Otherwise it is not possible to map in the segment. The *usertimer* can be mapped in by multiple processes at the same time. Therefore it advisable to let the timer run freely, though it is possible to write the timer registers, in principle.

Error Conditions:

`STD_ARGBAD`:	No valid hostname specified.
`STD_NOMEM`:	no free virtual address space to map in timer segment.

errors from host_lookup.
errors from dir_lookup.
errors from seg_map.

sun4m_timer_diff

```
long
sun4m_timer_diff(start, stop)
union sun4m_timer * start;
union sun4m_timer * stop;
```

Sun4m_timer_diff converts the time difference `stop − start` into microseconds.

Error Conditions:
None.

sun4m_timer_getptr

```
struct sun4m_timer_regs *
sun4m_timer_getptr()
```

Sun4m_timer_getptr returns a pointer to the mapped-in timer registers. If the registers are not mapped in it returns NULL.

Error Conditions:
None.

sun4m_timer_micro

```
uint32
sun4m_timer_micro()
```

Sun4m_timer_micro reads the current timer and returns the time in microseconds.

Error Conditions:
If the timer registers are not mapped in, calling this function will cause an exception.

Warnings

If more than one process maps in the *usertimer* and one or them modifies the behavior of the timer then the other will not obtain meaningful timing information.

Example

An example of the use of these routines can be found in the *test* suite for Amoeba, in the file *src/test/performance/cpu/timertest.c*.

See Also

sys_milli(L).

(in libraries: libamoeba.a, libamunix.a)

Name

syssvr – the kernel systhread server client interface stubs

Synopsis

```
#include "amoeba.h"
#include "cmdreg.h"
#include "stderr.h"
#include "server/systask/systask.h"

errstat sys_boot(cap, kernelcap, commandline, flags)
errstat sys_chgcap(cap, timeout, chkfields)
errstat sys_kstat(cap, flag, buf, bufsz, result)
errstat sys_printbuf(cap, buf, bufsz, offset, num_bytes)
```

Description

These stubs provide a part of the programmer's interface to the *syssvr* in the kernel. The *syssvr* also supports certain *soap*(L), *bullet*(L) and *std*(L) commands. See *syssvr*(A) for more details.

Errors

All functions return the error status of the operation. They return only standard error codes (see *stderr.h*). All the stubs involve transactions and so in addition to the errors described below, all stubs may return any of the standard RPC error codes. If illegal parameters (such as NULL-pointers or negative buffer sizes) are given, exceptions may occur rather than an error status being returned. The behavior is not well defined under these conditions.

Functions

sys_boot

```
errstat
sys_boot(cap, kernelcap, commandline, flags)
capability *cap;
capability *kernelcap;
char *commandline;
int flags;
```

Sys_boot requests that the kernel whose *syssvr* listens to capability *cap* should load the kernel binary specified by *kernelcap* and start running it. The kernel is probably stored on a file server. The *commandline* and *flags* will be passed to the new kernel in implementations where this is possible. This request has the effect of terminating all current activity on the host. In the future, one of the flags might be to migrate existing activity to other hosts before rebooting.

Required Rights:
```
SYS_RGT_BOOT
```

sys_chgcap

```
errstat
sys_chgcap(cap, timeout, chkfields)
capability *cap;
interval timeout;
port chkfields[3];
```

Sys_chgcap requests that the kernel whose *syssvr* listens to capability *cap* should change the check fields of the kernel directory server, the syssvr and the process server to those specified in the array *chkfields*. Unless changed by a subsequent call to *sys_chgcap*, the kernel directory server capability will revert to its default value after *timeout* seconds. This function is used by the reservation system to reserve a kernel for exclusive use.

Note that if *chkfield*[0] is NULL then the kernel directory server capability will revert immediately to its default value. This can be used to undo the effects of a previous *sys_chgcap* before the *timeout* has expired.

Required Rights:
```
SYS_RGT_ALL
```

sys_kstat

```
errstat
sys_kstat(cap, flag, buf, bufsz, num_bytes)
capability *cap;
char flag;
char *buf;
bufsize bufsz;
int *num_bytes;
```

Each Amoeba kernel provides a set of internal routines which print the current state of kernel data as ascii strings into a buffer. *Sys_kstat* requests that the kernel print some kernel data selected by *flag* into *buf* (whose size is *bufsz).* The number of bytes of data returned in *buf* is returned in *num_bytes*.

Since kernels may have different configurations there is a special flag, *?* which will return in *buf* a list of flags supported by the kernel and a short description of each. With one exception this set of flags corresponds exactly to those commands that can be given at the console using the *home* key (CTRL−_) followed by the appropriate letter. The *flag r* is always illegal for the *sys_kstat* function.

Required Rights:

These depend on the sensitivity of the data or operation requested. Typically `SYS_RGT_READ` is a minimum.

Error Conditions:

```
    STD_ARGBAD
```
an invalid flag was given.

sys_printbuf

```
errstat
sys_printbuf(cap, buf, bufsz, offset, num_bytes)
capability *cap;
char *buf;
bufsize bufsz;
int *offset;
int *num_bytes;
```

Each Amoeba maintains a copy of messages printed on its console which originated within the kernel. This is kept in a circular buffer which is guaranteed to fit in one RPC call. *Sys_printbuf* requests the kernel running on machine with the capability *cap* to return the contents of its console buffer in *buf*. *Bufsz* specifies the size of *buf* in bytes. The function returns in *offset* the starting position of the circular buffer and *num_bytes* tells how many bytes of data were returned in *buf*.

Required Rights:

```
    SYS_RGT_READ
```

Example

The main use of these routines is in the programs *kstat*(A) *printbuf*(A) and *reboot*(A). However, below is a simple program that demonstrates the use of some of the routines.

```
#include "amtools.h"
#include "sys/printbuf.h"

main(argc, argv)
int argc;
char *argv[];
{
    capability mach;
    errstat err;
    char buf[30000];
    int len, offset;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s mach-cap\n", argv[0]);
        exit(2);
    }
    if ((err = name_lookup(argv[1], &mach)) != STD_OK) {
         fprintf(stderr, "%s: cannot lookup %s (%s)\n",
                  argv[0], argv[1], err_why(err));
         exit(1);
    }
    err = sys_kstat(&mach, ’?’, buf, sizeof(buf), &len);
    if (err == STD_OK)
         printf("%.*s\n", len, buf);
    else
         fprintf(stderr, "%s: cannot kstat %s (%s)\n",
                  argv[0], argv[1], err_why(err));

    err = sys_printbuf(&mach, buf, sizeof(buf), &offset, &len);
    if (err == STD_OK) { /* print circular buffer */
         char * p = buf + offset;
         do {
             if (p >= buf + len)
                  p = buf;
             printchar(*p++);
         } while (p != buf + offset);
         putchar(’\n’);
    } else
         fprintf(stderr, "%s: cannot printbuf %s (%s)\n",
                  argv[0], argv[1], err_why(err));
}
```

The function *printchar* is not shown here. It simply takes care of unprintable characters in a nice way.

See Also

bullet(L), kstat(A), printbuf(A), soap(L), std(L), std_age(A), std_destroy(U), std_info(U), std_copy(U), std_restrict(U), std_touch(U), syssvr(A).

(in libraries: libamoeba.a)

Name

sys_milli – return time in milliseconds

Synopsis

```
#include "module/syscall.h"

unsigned long
sys_milli()
```

Description

Sys_milli returns the number of milliseconds since an unknown origin. It wraps after 2 ** 32 milliseconds, which is about 50 days. It is used for measuring time intervals by calling it once at the start of an event and again at the end and comparing the difference.

Warnings

The granularity of the hardware clock is sometimes bigger than a millisecond.

Example

This function returns the time it takes to compute the sine of its argument, in milliseconds. By computing it several times, it increases the accuracy of the measurement by reducing errors due to the granularity of the system clock.

```
#define MUCH 10000

unsigned long sin_time(f)
    double f;
{
    unsigned long start;
    register int i;
    double r, sin();

    start = sys_milli();
    for (i = MUCH; i--; r = sin(f))
         ;
    return (sys_milli() - start + MUCH / 2) / MUCH;
}
```

See Also

sun4m_timer(L).

(in libraries: libamoeba.a)

Name

sys_newthread − low-level thread management

Synopsis

```
#include "thread.h"

int sys_newthread(func, sp, local)
void exitthread(done)
struct thread_data *sys_getlocal()
void thread_en_preempt(thread_local_addr)
```

Description

Beware! This manual page is only presented for completeness. Application programs should use *thread*(L) to create threads and to manipulate glocal data.

The calls *sys_newthread*, *exitthread*, *sys_getlocal* and *thread_en_preempt* form the lowest level interface to Amoeba's multiple threads facility. In the current version of Amoeba, threads by default run till completion: another thread in a process is only run if a thread exits, executes a blocking system call (for example, see *rpc*(L)) or calls *threadswitch*(see thread_scheduling (L)). Preemptive scheduling must be requested explicitly using *thread_enable_preemption* (see *thread_scheduling*(L)). Programs should not rely on the current default scheduling policy. They should always properly protect their access to shared data with mutexes (see *mutex*(L)).

To allow the implementation of *glocal* data (see *thread*(L)), each thread has an associated ‘‘local value’’. This local value is stored in the kernel's per-thread data. It can be retrieved at any time by calling *sys_getlocal()*. A thread's local value is set once and for all when the thread is created; the main thread's local value is zero, for other threads the value is the third argument passed to *sys_newthread*.

A copy of the local value of the current thread is usually kept in the global variable *_thread_local*. When the program only uses non-preemptive scheduling, this variable can be kept up-to-date by the system call stubs (executing in user space): on entry, each stub saves the current value of *_thread_local* on the stack; before returning, it restores *_thread_local* from the stack. Certain architectures use a different strategy whereby *_thread_local* variable is not saved; instead, a −1 is stored in it when the call returns. The routines that implement *thread*(L) know this and test *_thread_local* for −1, and use *sys_getlocal* to restore the true local value. (There are also interactions with signal catchers that are taken care of by the implementation of *signals*(L).)

When preemptive scheduling is used, a different mechanism is required. The system call *thread_en_preempt*, which should only be called via *thread_enable_preemption*(see *thread_scheduling*(L)), has a parameter that will cause the kernel to update the *_thread_local* pointer whenever a thread is being rescheduled.

Because the thread's local value and the *_thread_local* variable are used by the implementation of *thread*(L), they should not be touched by application programs. In fact all calls described in this manual page are unsafe in programs that use *thread*(L); and since *thread*(L) is used by *stdio* and many other functions described in *ansi_C*(L) and *posix*(L), they are unsafe in almost all programs. Furthermore, the current thread package may be replaced by one that runs entirely in user space and these routines would then be obsolete.

Functions

sys_newthread

```
int
sys_newthread(func, sp, local)
void (*func)();
struct thread_data *sp;
struct thread_data *local;
```

Sys_newthread is the low level kernel stub to spawn a new thread in the current process. The thread's execution starts with a call to (*func)() with no arguments. The function should never return. The stack pointer of the new thread is initialized to *sp*; the local value of the new thread is initialized to *local*. Note that the stack pointer must point to at least 512 bytes of space. Otherwise the calling program will be killed.

The new thread does not run immediately; the current thread keeps running until it exits, blocks or calls *threadswitch.*

Sys_newthread returns −1 upon failure (caused by lack of resources to create another thread in the kernel) and zero upon success.

exitthread

```
void
exitthread(done)
long *done;
```

Exitthread is the low level kernel stub to exit a thread. When the thread is created by *thread_newthread* it is advisable to exit the thread using *thread_exit* (see *thread*(L)), since *exitthread* does not cleanup previously allocated stacks or memory blocks. The parameter *done* specifies the address of a variable that will be set to one by the kernel as soon as the thread has exited. This allows others threads to see when it is safe to free the resources that were allocated for the thread. When the calling thread is a server and it is serving a client, the client will receive an `RPC_FAILURE` (see *rpc*(L)). *Exitthread* never returns.

sys_getlocal

```
struct thread_data *
sys_getlocal()
```

Sys_getlocal returns the *_thread_local* value for a thread (see above).

thread_en_preempt

```
void
thread_en_preempt(thread_local_addr)
struct thread_data **thread_local_addr;
```

Thread_en_preempt enables preemptive scheduling for the calling process. It also requests the kernel to update the pointer whose address is given as argument whenever a new thread within the process is scheduled. The pointer is set to the ‘‘local value’’ of the thread to be run next. Typically, the user level stub *thread_enable_preemption* will just do

```
thread_en_preempt(&_thread_local);
```

See Also

ansi_C(L), posix(L), rpc(L), signals(L), thread(L), thread_scheduling(L).

(in libraries: libamoeba, libamunix.a)

Name

tape – tape server stubs

Synopsis

```
#include "amoeba.h"
#include "module/tape.h"

errstat tape_erase(cap)
errstat tape_fpos(cap, pos)
errstat tape_fskip(cap, count)
errstat tape_load(cap)
errstat tape_read(cap, buf, record_size, xread)
errstat tape_rewind(cap, immediate)
errstat tape_rpos(cap, pos)
errstat tape_rskip(cap, count)
errstat tape_status(cap, buf, size)
errstat tape_unload(cap)
errstat tape_write(cap, buf, record_size, xwritten)
errstat tape_write_eof(cap)
char *tape_why(err)
```

Description

Along with the standard server stubs (see *std(L))* the tape server stubs provide the programmer's interface to the tape server.

Errors

All functions return the error status of the operation. The user interface to the tape server returns standard error codes (as defined in *stderr.h*) and tape errors which are listed below. Since all the stubs involve transactions they may return any of the standard RPC error codes. A valid capability is required for all tape operations. Tape stubs will return `STD_CAPBAD` if an invalid capability is used. If illegal parameters (such as NULL-pointers or negative buffer sizes) are given, exceptions may occur rather than an error status being returned. The behavior is not well defined under these conditions.

The following tape errors are defined:

`TAPE_NOTAPE`	The tape unit is off-line or there is no tape available.
`TAPE_WRITE_PROT`	The tape is write protected.
`TAPE_CMD_ABORTED`	The last command has been aborted.
`TAPE_BOT_ERR`	While repositioning, hit begin of tape marker.
`TAPE_EOF`	Found an EOF marker.
`TAPE_REC_DAT_TRUNC`	Tape record longer than expected.

`TAPE_POS_LOST`	Tape record position lost.
`TAPE_EOT`	End of tape reached.
`TAPE_MEDIA_ERR`	Tape format error.

The user should be aware that the maximum transfer size (record reads/writes) is dependent on hardware (the tape controllers) and the maximum transaction buffer size of Amoeba (30,000 bytes, truncated to 29K).

tape_load

```
errstat
tape_load(cap)
capability *cap;
```

Tape_load loads a tape on the tape unit identified by *cap* and positions it at the beginning of medium.

tape_read

```
errstat
tape_read(cap, buf, record_size, xread)
capability *cap;
bufptr buf;
bufsize record_size;
bufsize *xread;
```

Tape_read reads *record_size* bytes from the tape specified by *cap* to buffer *buf*. The number of bytes actually read is returned in *xread*. If *record_size* bytes are requested from a tape while the tape record is larger, *tape_read* will fail (`TAPE_REC_DAT_TRUNC`).

tape_write

```
errstat
tape_write(cap, buf, record_size, xwritten)
capability *cap;
bufptr buf;
bufsize record_size;
bufsize *xwritten;
```

Tape_write writes *record_size* bytes from buffer *buf* to the tape specified by *cap*. The number of bytes actually written is returned in *xwritten.*

tape_write_eof

```
errstat
tape_write_eof(cap)
capability *cap;
```

Tape_write_eof writes an *End-Of-File* marker on the tape specified by *cap.*

tape_rewind

```
errstat
tape_rewind(cap, immediate)
capability *cap;
int immediate;
```

Tape_rewind rewinds the tape specified by *cap* to its beginning. When *immediate* is set, *tape_rewind* returns immediately without waiting for the rewind to complete.

tape_unload

```
errstat
tape_unload(cap)
capability *cap;
```

Tape_unload rewinds and unloads the tape specified by *cap*. This command does not wait until the tape is physically rewound and unloaded. Some tape controllers cannot unload a tape by themselves; they rewind the tape and wait for an operator to unload the tape manually.

tape_rskip

```
errstat
tape_rskip(cap, count)
capability *cap;
int32 count;
```

Tape_rskip skips *count* records on the tape specified by *cap*. Positive counts result in forward record skipping, negative counts result in backward record skipping. Since EOF markers are normal tape records for certain tape controllers skipping records might result in skipping files without noticing. After a record skip, the tape file position may be uncertain.

tape_rpos

```
errstat
tape_rpos(cap, pos)
capability *cap;
int32 *pos;
```

Tape_rpos returns the current tape record position in the variable pointed to by *pos*.

tape_fpos

```
errstat
tape_fpos(cap, pos)
capability *cap;
int32 *pos;
```

Tape_fpos returns the current tape file position in the variable pointed to by *pos*.

tape_fskip

```
errstat
tape_fskip(cap, count)
capability *cap;
int32 count;
```

Tape_fskip skips *count* EOF markers on the tape specified by *cap.* Positive counts result in forward file skipping, negative counts result in backward file skipping. A forward tape file skip causes the tape to be positioned at the beginning of the desired file, a backward file skip causes the tape to be positioned at the end of the desired file (actually at the EOF marker).

tape_erase

```
errstat
tape_erase(cap)
capability *cap;
```

Tape_erase formats the complete tape specified by *cap.* The current data is destroyed.

tape_status

```
errstat
tape_status(cap, buf, size)
capability *cap;
bufptr *buf;
bufsize size;
```

Tape_status returns a NULL-terminated ASCII string in buffer *buf* which reports the status of the tape specified by *cap.* The ASCII string size is *size.*

tape_why

```
char *
tape_why(err)
errstat err;
```

Tape_why returns a NULL-terminated ASCII string which describes the tape error *err.* It calls *err_why* (see *error*(L)) if it does not recognize the error code.

Examples

The following example loads a tape, skips 10 records, reads a record of *size* bytes and unloads the tape.

```
do_tape(cap, buf, size)
capability *cap;
bufptr  buf;
bufsize size;
{
    errstat rv;
    bufsize xread;

    /* load tape */
    if ((rv = tape_load(cap)) != STD_OK) {
         printf("Cannot load tape: %s\n", tape_why(rv));
         exit(1);
    }

    /* skip 10 records forwards */
    if ((rv = tape_rskip(cap, 10)) != STD_OK) {
         printf("Cannot skip on tape: %s\n", tape_why(rv));
         exit(1);
    }

    /* read a record of size bytes */
    if ((rv = tape_read(cap, buf, size, &xread)) != STD_OK) {
         printf("Cannot read tape: %s\n", tape_why(rv));
         exit(1);
    }

    /* unload the tape */
    if ((rv = tape_unload(cap)) != STD_OK) {
         printf("Cannot unload tape: %s\n", tape_why(rv));
         exit(1);
    }
    return(xread);
}
```

See Also

error(L), std(L).

(in libraries: libamoeba.a)

Name

thread – thread creation and thread memory management module

Synopsis

```
#include "amoeba.h"
#include "thread.h"

int thread_newthread(func, stsize, param, psize)
int thread_exit()
char *thread_alloc(index, size)
char *thread_param(size)
char *thread_realloc(index, size)
```

Description

The *thread* module provides the programming interface to create, destroy and manage concurrent threads. Each thread can start executing a separate routine; they do not all have to execute the same function. Each thread in a multi-threaded process shares the same address space. It has its own stack and program counter but otherwise shares the text, data and bss segments of the process. Because it is sometimes useful to have data global within a thread but not accessible outside the thread, *glocal* data is provided. See the description of *thread_alloc* below for details of how to allocate and use *glocal* data.

The threads are currently scheduled non-preemptively by default. Preemptive scheduling must be enabled explicitly using *thread_enable_preemption* (see *thread_scheduling*(L)). It is important to protect accesses to global data with mutexes (see *mutex*(L)). It is possible for a thread to request that it be rescheduled using *threadswitch* (see *thread_scheduling*(L)). This can be very useful in the presence of non-preemptive scheduling.

NB. If a program is multi-threaded it is not safe to use UNIX emulation routines in more than one thread of the program. UNIX is not multi-threaded and therefore the emulation is only likely to be correct if confined to a single thread of a program. For example, a program with several threads where one is in *read* waiting for input from a terminal and another does a *fork* may well hang until the read is satisfied. The *exit* routine has been modified to force a close on all descriptors, even if they are held by another thread and no guarantees are made on the correctness of any resulting input or output if *exit* is called in such circumstances.

Functions

thread_newthread

```
int
thread_newthread(func, stsize, param, psize)
void (*func)();
int stsize;
char *param;
int psize;
```

Thread_newthread spawns a new thread and starts it at function *func*. *Thread_newthread* allocates a thread stack of *stsize* bytes. *Stsize* must be at least 512 bytes or the calling program will be killed. Parameters can be passed to the new thread via the *thread_newthread* parameters *param* and *psize*. *Param* is a pointer to the data structure to pass, *psize* is the size of the data structure. *Param* must be allocated by a member of the *malloc* family (see *malloc*(L)) since the clean up when the thread exits will free this memory. Memory allocated using *thread_alloc* cannot be used! When no parameters are passed, *param* must be a NULL-pointer and *psize* must be zero. Once the thread exits, the allocated stack and parameter area are freed. The function *func* is called as follows:

```
(*func)(param, psize)
char *param;
int psize;
```

If the called function returns, the thread exits.

Thread_newthread returns zero upon failures (insufficient memory or out of threads), otherwise a positive value is returned.

Note that not all threads are created equal. When a process first starts it consists of one thread which starts the routine *main()*. If main returns then *exit()* is called which will terminate the entire process immediately. If it is desired that *main* terminate and the other threads continue then it must not return but call *thread_exit* (described below).

Typically, when a new thread is created the parent continues to execute until it blocks. However, if preemptive scheduling is enabled (see *thread_scheduling*(L)) then the newly created process will have the same priority as the current thread. This means at the next event (such as an interrupt) the new thread may be scheduled.

thread_exit

```
int
thread_exit()
```

Thread_exit stops the current thread. It then frees any glocal memory (allocated by *thread_alloc/thread_realloc*), the parameter area and the allocated stack before exiting. *Thread_exit* does not return. When the calling thread was a server thread, and it was still serving a client, the client will receive an `RPC_FAILURE` (see *rpc*(L)). If *thread_exit* is called in the last thread of a process, the process exits as well (see *exitprocess*(L)).

thread_alloc

```
char *
thread_alloc(index, size)
int *index;
int size;
```

The first time *thread_alloc* is called (with **index == 0)*, *thread_alloc* allocates glocal data of *size* bytes and returns the module reference number in **index*. The allocated data is initialized to zero. The value of the function is a pointer to the glocal data. Successive calls to *thread_alloc* with the previously assigned module reference number result in returning a pointer to the previously allocated memory.

Thread_alloc returns a NULL-pointer on insufficient memory or when a successive call to *thread_alloc* has a different size parameter than in the original call. In this case, the already allocated memory is not modified or freed.

Consider an example. Suppose a function in a single-threaded program that uses a (static) global variable. For example,

```
static long sum;
long add(x)
long x;
{
    sum += x;
    return sum;
}
```

Now suppose this function must be used in a multi-threaded program, where the threads perform independent computations. This means a separate *sum* variable is required for each thread. If the number of threads is known in advance, the threads could be numbered and an array indexed by thread numbers could maintained (assuming a thread can find out its own thread number, which is not trivial unless a parameter is added to *add* for this purpose). In general, however, this is too complex. A simpler solution is to use *thread_alloc*.

```
#include "thread.h"
static int ident;
long add(x)
long x;
{
    long *p_sum;
    p_sum = (long *) thread_alloc(&ident, sizeof(long));
    *p_sum += x;
    return *p_sum;
}
```

Because there may be several functions in a program that need a block of glocal memory for private use, *thread_alloc* has an *ident* parameter that indicates the identity of the memory block (not the calling thread!). This must be the address of a global integer variable, statically initialized to zero.† This variable is used for *thread_alloc*'s internal

† To be precise, the variable must have global lifetime ('static duration' in Standard C jargon); it may have local scope, i.e., it may be a static variable declared in a function.

administration; its contents must never be touched by the calling function. Functions that need to use the same block of glocal data must use the same *ident* variable. For consistency, all calls using the same *ident* variable must pass the same block size. To change the block size, *thread_realloc* can be used.

thread_realloc

```
char *
thread_realloc(index, size)
int *index;
int size;
```

Thread_realloc is used to change the size of the block of glocal memory associated with the module reference number **index* that was previously allocated by *thread_alloc*. It allocates a new glocal memory block of *size* bytes and initializes it to zero. It then copies to the new memory block however much of the old data that will fit in it. When *thread_realloc* is called with **index* equal to zero it operates as *thread_alloc*.

Thread_realloc returns a NULL-pointer on insufficient memory. It does not free the original glocal data in this case so that it can still be accessed or the reallocation retried later.

Note Well: As a temporary hack, *thread_realloc* never frees the previous copy of the data. This must be done in the calling program using *free* (see *malloc*(L)). It is not recommended that this routine be used until this feature is removed.

thread_param

```
char *
thread_param(size)
int *size;
```

Thread_param returns the parameter pointer which was initially passed to the initial routine of the current thread. It returns the size of the parameter in **size*.

Warnings

The *thread* module uses the global variable *_thread_local* (see *sys_newthread*(L)) for its administration.

Example

The following example shows thread creation and the use of glocal memory. *Ref_nr* is the module reference number. Each time the signal handler is activated, the glocal data is fetched which belongs to the running thread and this module.

```
#define NTHREADS 5
#define STKSIZE  8096
#define  SIZE     100

static int ref_nr;
```

```
void
worker_thread(param, size)
char * param;
int size;
{
    char *ptr;
    int i;

    if (size == sizeof (int)) {
         i = *(int *) param;
         printf("Thread %d started\n", i);
    }
    /* Initially allocate memory and fetch module ref. number */
    if ((ptr = thread_alloc(&ref_nr, SIZE)) == 0) {
        fprintf(stderr, "worker_thread: cannot thread_alloc.\n");
        thread_exit();
    }
    strcpy(ptr, "Peter was here");
    ...
}

main()
{
    int i;
    char * p;

    for (i = 0; i < NTHREADS; i++) {
         if ((p = malloc(sizeof (int))) == 0) {
             printf("malloc failed\n");
             exit(1);
         }
         *(int *)p = i;
         if (!thread_newthread(worker_thread, STKSZ, p, sizeof (int))) {
             printf("thread_newthread failed\n");
             exit(1);
         }
    }
    /* Do not wait for threads to exit */
    thread_exit();
}
```

See Also

exitprocess(L), malloc(L), rpc(L), sys_newthread(L), thread_scheduling(L).

Name

thread_scheduling – routines to control scheduling between threads

Synopsis

```
#include "thread.h"

void threadswitch()
void thread_enable_preemption()
void thread_get_max_priority(max)
void thread_set_priority(new, old)
```

Description

These routines can be used to alter the scheduling algorithm used for selecting threads within a process. They do not alter the scheduling algorithm between processes. Nor can they alter the scheduling algorithm between kernel threads. The threads in the kernel always have a higher priority than those of user processes. All user processes have equal priority and are scheduled round-robin with time-slicing.

The default scheduling between threads within a process is non-preemptive. Threads run until they block or terminate. Threads block when they make a blocking system call. For example, an RPC (see *rpc*(L)), a *grp*(L) function or a *mu_lock* (see *mutex*(L)). Since most library routines, e.g., *sema_up*, *sema_down*, *printf* and *open*, do an RPC or use mutexes to protect shared variables, it is highly likely that a library call will block. When non-preemptive scheduling between threads is used it is sometimes necessary for a process to voluntarily give up control so that another thread can run. That functionality is provided by *threadswitch*.

However, it is possible to enable preemptive scheduling between threads and to assign different priorities to threads. In this case the highest priority thread that is runnable will be scheduled when the process is selected to run. The lowest priority for a thread is 0. The highest is that returned by *thread_get_max_priority*. Time-slicing takes place between threads. The effect of priorities is confined to within the process. It is not the case that the highest priority thread from all processes is selected.

Functions

threadswitch

```
    void
    threadswitch()
```

Threadswitch causes the current thread to stop running and the scheduler to be called. If preemptive scheduling is disabled (the default case) then if there is another runnable thread in the same process it will be the next thread to run in that process. Note, however, that it may not be run immediately. Under certain circumstances a thread in another process may be run first. When no other thread is runnable, the calling thread is immediately restarted.

If preemptive scheduling is activated then it gives up control to the next thread of the same priority. The next thread will run if there is no runnable thread with higher priority. Once again, it is possible that a thread from another process may be run first.

thread_get_max_priority

```
void
thread_get_max_priority(max)
long * max;
```

Thread_get_max_priority returns in **max* the maximum legal priority which can be assigned to a thread. This call may be done at any time, regardless of whether preemptive scheduling is enabled or not.

thread_set_priority

```
void
thread_set_priority(new, old)
long new;
long * old;
```

Thread_set_priority sets the priority of the current thread to *new* and calls the scheduler. It returns in **old* the previous priority of the thread. The value of *new* must be in the range 0 to the maximum priority (as returned by *thread_get_max_priority*). It is a programming error to attempt to set a thread's priority if preemptive scheduling is not enabled or to set it to a value outside the range of legal values.

thread_enable_preemption

```
void
thread_enable_preemption()
```

Thread_enable_preemption enables preemptive priority scheduling between threads for the current process. Once preemptive scheduling has been enabled it cannot be disabled. Note that all threads present in a process will have priority 0 immediately after this function is called.

For programs with threads of different priorities it is a good idea to enable preemption before the other threads are created. In this way, *main()* can alter its priority from its default of 0 to a level higher than the intended priorities of the child threads. Initially, each new child thread will have the same priority as *main()* but must set its priority to the correct (lower) level before doing anything else. This should lead to *main()* being resumed immediately if a child thread happens to run. Thus *main()* can complete initialization of threads before any of the child threads begins work. Then it can set its priority to the correct level for the program to perform its function. This is much simpler than using mutexes to enforce the completion of initialization before a program starts its work.

Warning

If a process that has enabled preemptive scheduling forks (using the Ajax emulation) then the child process will be scheduled preemptively but the threads will not inherit the priorities of the parent. They will all have priority zero.

Example

The following demonstrates how to enable preemptive scheduling and to assign priorities to threads.

```
#include "amoeba.h"
#include "thread.h"

void
mate(p, sz)
char * p;
int sz;
{
    long old;

    thread_set_priority((long) 1, &old);
    ...
}

main()
{
    long old;
    long max;

    thread_enable_preemption();
    thread_get_max_priority(&max);
    thread_set_priority(max, &old); /* set priority to max */
    if (thread_newthread(mate, 4096, (char *) 0, 0) == 0) {
         printf("thread_newthread failed0);
         exit(1);
    }
    ...
}
```

See Also

thread(L).

(in libraries: libamoeba.a, libamunix.a, libkernel.a)

Name

tios – termios-style terminal control interface

Synopsis

```
#include "amoeba.h"
#include "termios.h" /* or "posix/termios.h" */

errstat tios_drain(tty)
errstat tios_flow(tty, action)
errstat tios_flush(tty, queue_selector)
errstat tios_getattr(tty, tp)
errstat tios_getwsize(tty, width, length)
errstat tios_sendbrk(tty, duration)
errstat tios_setattr(tty, optional_actions, tp)

char *tc_marshal(buf, t, same_byte_order)
char *tc_unmarshal(buf, tp, same_byte_order)

#include "sys/ioctl.h" /* or "posix/sys/ioctl.h" */

void tc_frombsd(tp, sgp, tcp, ltcp, local_mode_p)
void tc_tobsd(tp, sgp, tcp, ltcp, local_mode_p)

void tc_tosysv(tp, sysv_tp)
void tc_fromsysv(tp, sysv_tp)
```

Description

This module contains the capability-based counterparts of the *termios* function calls (see *posix*(L)). These operations are supported by several servers, including *ax*(U), *xterm* and the console driver in the kernel. The functions *tc_marshal* and *tc_unmarshal* help servers with packing and unpacking a *struct* termios The functions *tc_frombsd* and *tc_tobsd* provide conversion from and to structures used by BSD's tty ioctl's. Similarly the *tc_fromsysv* and *tc_tosysv* functions provide conversion from and to the structures used by UNIX System V ioctl's.

*tios_**

```
    errstat
    tios_drain(tty)
    capability *tty;
```

```
errstat
tios_flow(tty, action)
capability *tty;
int action;

errstat
tios_flush(tty, queue_selector)
capability *tty;
int queue_selector;

errstat
tios_getattr(tty, tp)
capability *tty;
struct termios *tp; /* out */

errstat
tios_sendbrk(tty, duration)
capability *tty;
int duration;

errstat
tios_setattr(tty, optional_actions, tp)
capability *tty;
int optional_actions;
struct termios *tp; /* in */
```

There is a one-to-one correspondence between these calls and the POSIX calls, replacing the file descriptor parameter to the POSIX call by a capability pointer parameter to the *tios_* call, and the *tc* name prefix with *tios_*. See the POSIX standard for an explanation of the semantics. See *posix*(L) for Amoeba-dependent information.

Error Conditions:

`STD_COMBAD`:	operation not supported by server.
`STD_CAPBAD`:	invalid capability.
`STD_DENIED`:	insufficient rights.

RPC errors are also possible, as well as other errors depending on the server implementation.

tios_getwsize

```
errstat
tios_getwsize(tty, width, length)
capability *tty;
int *width; /* out */
int *length; /* out */
```

This routine returns the dimensions of the screen area of the terminal device. This provides support for window systems where windows may have a variable size and screen-based programs need to enquire of the size of the screen area.

tc_marshal, tc_unmarshal

```
char *
tc_marshal(buf, t, same_byte_order)
char *buf; /* out */
termios t; /* in; by value! */
int same_byte_order;

char *
tc_unmarshal(buf, tp, same_byte_order)
char *buf; /* in */
termios *t; /* out */
int same_byte_order;
```

Function *tc_marshal* packs a struct termios value into the buffer argument, and returns a pointer to the next free byte in the buffer. Function *tc_unmarshal* unpacks the buffer into a struct termios variable, and returns a pointer to the next data byte in the buffer. Both use the flag ‘‘same_byte_order’’ to determine whether to swap bytes or not. There is no buffer overflow detection, nor are other error conditions detected.

tc_frombsd, tc_tobsd

```
void
tc_frombsd(tp, sgp, tcp, ltcp, local_mode_p)
struct termios *tp; /* out */
struct sgttyb *sgp; /* in */
struct tchars *tcp; /* in */
struct ltchars *ltcp; /* in */
int *local_mode_p; /* in */

void
tc_tobsd(tp, sgp, tcp, ltcp, local_mode_p)
struct termios *tp; /* in */
struct sgttyb *sgp; /* out */
struct tchars *tcp; /* out */
struct ltchars *ltcp; /* out */
int *local_mode_p; /* out */
```

These functions convert a struct termios from/to a couple of BSD-style tty control blocks. No error conditions are detected. NOTE: These functions are **not** present in the library if it was compiled with the flag SYSV.

tc_fromsysv, tc_tosysv

```
void tc_tosysv(tp, sysv_tp)
struct termios *tp;
char *sysv_tp;

void tc_fromsysv(tp, sysv_tp)
struct termios *tp;
```

These functions convert an Amoeba style termios struct from/to a UNIX System V style termios struct. NOTE: These functions are only present in the library if it was compiled with the flag `SYSV`.

Warnings

Most servers only implement *tios_(get,set)attr*, not the other *tios* stubs.

Code compiled for UNIX must include Amoeba's *termios.h* to get the proper definition of *struct termios*, but UNIX's *sys/ioctl.h* to get the proper definition of the ioctl structs.

The *tc_(un)marshal* interface may go away or change in the future. If you decide to write a server that implements the *tios_* functions, you should consider using *ail*(U) instead of *tc_(un)marshal*.

There should be no reason to use *tc_(to,from)bsd*.

Note

The kernel library contains the *tc_** functions but not the *tios_** ones.

Example

For an example of using *tc_(un)marshal*, see the source for *ax*(U). For examples of using the *tios_* stubs and the only uses of *tc_(from,to)bsd*, see the source for ioctl (see *posix*(L)) and *ax*(U).

See Also

IEEE Std 1003.1-1988 (POSIX), section 7.

BSD manual pages *ioctl*(2), *tty*(4).

ail(U), ax(U), posix(L).

(in libraries: libamoeba.a, libamunix.a)

Name

tod – the Time-Of-Day server interface stubs

Synopsis

```
#include "amoeba.h"
#include "module/tod.h"

errstat tod_defcap()
void tod_setcap(tod_cap)
errstat tod_gettime(sec, msec, tz, dst)
errstat tod_settime(sec, msec)
```

Description

This module provides the interface to the time of day service. There may be more than one time of day server on an Amoeba network and it is possible to select a particular server, or simply use the default server when getting or setting the time.

Functions

tod_defcap

```
errstat
tod_defcap()
```

This routine is used to set the time of day server to be used by calls to get or set the time. It finds the capability for a time-of-day server as follows. It uses the `TOD` capability environment variable if it exists. Otherwise it looks up `DEF_TODSVR` as defined in *ampolicy.h* (which is typically */profile/cap/todsvr/default*). The capability is stored in static data inside the module. Be careful when trying to use multiple time of day servers.

The function returns `STD_OK` if it successfully looked up the capability in the directory server. Otherwise it returns the error status from the directory server look-up.

tod_setcap

```
void
tod_setcap(tod_cap)
capability *tod_cap;
```

Tod_setcap sets to *tod_cap* the pointer to the capability for the time of day server to be used in subsequent calls to *tod_gettime* and *tod_settime*. The address of the capability is stored in static data inside the module. Be careful when trying to use multiple time of day servers.

tod_gettime

```
errstat
tod_gettime(sec, msec, tz, dst)
long *sec;
int *msec;
int *tz;
int *dst;
```

Tod_gettime gets the time from the time-of-day server. It fills the long integer pointed to by *sec* with the number of seconds since the epoch, January 1, 1970, and the integer pointed to by *msec* with the number of milliseconds since the value in *sec*. The integer pointed to by *tz* is filled in with the local time zone, measured in minutes of time westward from Greenwich. The integer pointed to by *dst* is filled in with a non-zero value if Daylight Saving time is in effect.

Tod_gettime calls *tod_defcap* if it has not already been called, so it is not necessary to explicitly call *tod_defcap* or *tod_setcap* before *tod_gettime*.

Required Rights:
None

Error Conditions:

`RPC` errors	if the server is not available.

tod_settime

```
errstat
tod_settime(sec, msec)
long sec;
int msec;
```

Tod_settime sets the time of the time-of-day server as selected by either *tod_defcap* or *tod_setcap*.

Tod_settime calls *tod_defcap* if it has not already been called, so it is not necessary to explicitly call *tod_defcap* or *tod_setcap* before *tod_settime*.

Required Rights:
All rights bits must be on.

Error Conditions:

`STD_DENIED`	if any rights are not on.
`STD_CAPBAD`	if the capability is invalid.
`RPC` errors	if the server is not available.

Environment Variables

`TOD` – environment capability for a tod server.

Warnings

The granularity of the clock is server dependent. Time values are truncated as opposed to rounded.

If the capability whose address is handed to *tod_setcap* is overwritten, the new value will be used in subsequent calls to *tod_gettime* and *tod_settime*.

If *tod_gettime* or *tod_settime* fails, it undoes the effect of any previous *tod_setcap*.

See Also

date(U), ctime(L), posix(L), rpc(L).

(in libraries: libamoeba.a, libamunix.a)

Name

uniqport – generate a random port

Synopsis

```
#include "module/rnd.h"

void uniqport(p)
void uniqport_reinit()
```

Description

Routines to generate unique RPC and group communication ports and check fields.

uniqport

```
    void
    uniqport(p)
    port * p;
```

Uniqport produces a non-NULL, random *port*. In general the result should be unique to the system. It is useful for servers that need to create new check fields or a new port to listen to. It uses a random number server to get the seed for the random number generator.

uniqport_reinit

```
    void
    uniqport_reinit()
```

Uniqport_reinit causes the next call to *uniqport* to choose a new seed for the generation of random numbers.

Environment Variables

If set, `RANDOM` determines which random number server to use when generating the seed. Otherwise the default random server is used.

Warnings

It is possible that the port generated is not unique within the system. If it is to be used as a *port* then it may be necessary to test to see if anyone is already listening to that port before using it. This is done by doing a *std_info* (see *std*(L)) on the port with a short locate timeout (say 2 seconds).

Example

```
#include "amoeba.h"
#include "module/rnd.h"

port listen;

uniqport(&listen);
```

See Also

rnd(L), rpc(L).

(in libraries: libamoeba.a, libamunix.a)

Name

vdisk – the virtual disk server client interface stubs

Synopsis

```
#include "amoeba.h"
#include "cmdreg.h"
#include "stderr.h"
#include "module/disk.h"

errstat disk_info(cap, buf, size, cnt)
errstat disk_read(cap, l2vblksz, start, numblocks, buf)
errstat disk_size(cap, l2vblksz, size)
errstat disk_write(cap, l2vblksz, start, numblocks, buf)
errstat disk_getgeometry(cap, geom)
```

Description

Along with the standard server stubs (see *std*(L)) the virtual disk stubs provide the programmer's interface to the Virtual Disk Server. The Virtual Disk Server allows the user to specify the size of the block they wish to use in the call. This is convenient for certain applications which work most effectively with a particular block size and do not wish to be concerned with the physical reality. The block size must be a power of 2. To enforce this it must be specified in the call as the logarithm base 2 of the desired block size. It must be greater than `D_PHYS_SHIFT` (see *disk/disk.h*).

Types

The file *server/disk/disk.h* (which is included by *module/disk.h*) defines the type *disk_addr* which is a disk address. It is implemented as a large integer.

Access

Access to the disk is determined on the basis of rights in the capability given with each command.

The following rights are defined in Virtual Disk Server capabilities:

`RGT_WRITE`	permission to write to the disk.
`RGT_READ`	permission to read the disk.

Errors

All functions return the error status of the operation. The user interface to the Virtual Disk Server returns only standard error codes (as defined in *stderr.h*). All the stubs involve transactions and so in addition to the errors described below, all stubs may return any of the standard RPC error codes. A valid capability is required for all operations and so all operations will return `STD_CAPBAD` if an invalid capability is used. If the required right(s) are not present then `STD_DENIED` will be returned. If an invalid blocksize is specified then `STD_ARGBAD` is returned. If illegal parameters (such as NULL-pointers or negative buffer sizes) are given, exceptions may occur rather than an error status being returned. The behavior is not well defined under these conditions.

Functions

disk_getgeometry

```
errstat
disk_getgeometry(cap, geom)
capability *cap;
geometry *geom;
```

Disk_getgeometry returns in *geom* the disk geometry information for the first *piece* of a virtual disk. It is only intended for use on *bootp* partitions (which are physical disks). These consist of exactly one piece. However if applied to a virtual disk consisting of more than one piece it returns the geometry of the first piece.

Required Rights:
`RGT_READ`

Error Conditions:

`STD_SYSERR`:	If the data returned was insufficient for a full geometry struct.

disk_info

```
errstat
disk_info(cap, buf, size, cnt)
capability *cap;
dk_info_data *buf;
int size;
int *cnt;
```

Disk_info returns information about the physical disk partitions that comprise the virtual disk specified by *cap*. The data returned consists of:

unit number of the virtual disk (type: *long)*
disk address of the first block in the partition available to the user (type: *disk_addr)*
number of disk blocks available in the partition (type: *disk_addr)*

It returns the data in *buf* (which contains room for *size* sets of data). The structure *dk_info_data* is defined in the include file *server/disk/disk.h*. *Disk_info* returns in *cnt* the number of units of information returned. NB. If more partitions are available than fit in the buffer provided it will return the number asked for without complaint or warning.

Required Rights:
    `RGT_READ`

Error Conditions:

`STD_SYSERR`:	If the data returned was insufficient for a full dk_info_data struct.

disk_read

```
    errstat
    disk_read(cap, l2vblksz, start, numblocks, buf)
    capability *cap;
    int l2vblksz;
    disk_addr start;
    disk_addr numblocks;
    bufptr buf;
```

Disk_read reads exactly *numblocks* of the size specified by *l2vblksz* (i.e., the disk block size is $2^{l2vblksz}$), starting at block number *start*, from the virtual disk specified by *cap*. The data is returned in the buffer pointed to by *buf*. If it fails to read exactly the right amount of data then it will return an error status.

Required Rights:
    `RGT_READ`

Disk_size

```
    errstat
    disk_size(cap, l2vblksz, size)
    capability *cap;
    int l2vblksz;
    disk_addr *size;
```

Disk_size returns in *size*, the number of blocks of the size specified by *l2vblksz* (i.e., the disk block size is $2^{l2vblksz}$), available on the virtual disk specified by *cap*.

Required Rights:
    `RGT_READ`

disk_write

```
errstat
disk_write(cap, l2vblksz, start, numblocks, buf)
capability *cap;
int l2vblksz;
disk_addr start;
disk_addr numblocks;
bufptr buf;
```

Disk_write writes the *numblocks* blocks in *buf*, whose block-size is specified by *l2vblksz* (i.e., the disk block size is $2^{l2vblksz}$), to the virtual disk specified by *cap*, beginning at block *start*.

Required Rights:

```
RGT_WRITE
```

See Also

dread(A), dwrite(A), std(L), std_age(A), std_copy(U), std_destroy(U), std_info(U), std_restrict(U), std_touch(U), vdisk(A).

(in libraries: libamoeba.a)

Name

virtcirc – virtual circuits, full-duplex interprocess communication channels

Synopsis

```
#include "amoeba.h"
#include "semaphore.h"
#include "vc.h"

int vc_avail(vc, which)
void vc_close(vc, which)
struct vc *vc_create(iport, oport, isize, osize)
int vc_getp(vc, buf, blocking)
void vc_getpdone(vc, size)
int vc_putp(vc, buf, blocking)
void vc_putpdone(vc, size)
int vc_read(vc, buf, size)
int vc_readall(vc, buf, size)
void vc_setsema(vc, sema)
void vc_warn(vc, which, rtn, arg)
int vc_write(vc, buf, len)
```

Description

The *virtual circuit* module provides a collection of routines to manage virtual circuits. Virtual circuits are full-duplex interprocess communication (IPC) channels. A virtual circuit consists of an input and an output channel, which can be concurrently read or written. Virtual circuit users write data to the output channel and read data from the input channel. A close on an output channel causes an end of file (EOF) to be transmitted. This does not flush the current contents of the local output and remote input channels. A close on an input channel causes a hangup (HUP) to be transmitted and the current contents of the local input and remote output channels are flushed.

Internally, the input and output channels consist of a thread and a circular buffer. The input channel is a client thread which receives data from the remote server and writes it to the input circular buffer. The output channel is a server thread which reads data from the output circular buffer and transmits it to the remote client.

The virtual circuit structure and various other constants (for example, `VC_IN`, `VC_OUT`) are defined in the header file *vc.h*.

vc_create

```
struct vc *
vc_create(iport, oport, isize, osize)
port *iport, *oport;
int isize, osize;
```

Vc_create creates a virtual circuit. It starts an input channel of size *isize* and an output channel of size *osize*. The local output channel connects to the remote input channel via the put port *oport*. The local input channel waits for a remote output channel to connect to the local get port *iport*. *Vc_create* returns a virtual circuit reference pointer. The local and remote *vc_create* must be called within the time limit of a locate timeout (typically 5 secs, see *rpc*(L)). Otherwise a lost connection will be detected.

A NULL-pointer is returned upon failure (insufficient memory, out of threads).

vc_close

```
void
vc_close(vc, which)
struct vc *vc;
int which;
```

Vc_close closes one or both channels of the virtual circuit. A closed channel may not be read or written any more. *Which* can either be `VC_IN`, `VC_OUT` or `VC_BOTH`. `VC_IN` closes the input channel, `VC_OUT` closes the output channel and `VC_BOTH` closes both channels. To break down a connection completely, both channels of the virtual circuit must be closed. Closing `VC_OUT` does not flush the contents of the local output and remote input channels. Closing `VC_IN` does flush the local input and remote output channels. By default *vc_close* operates synchronously: it waits until the connection is completely broken before returning. When `VC_ASYNC` is added to the parameter *which*, *vc_close* operates asynchronously: it does not wait until the connections are broken down. Once both channels are closed, the virtual circuit reference pointer may not be accessed any more.

vc_read, vc_readall

```
int
vc_read(vc, buf, size)
struct vc *vc;
bufptr     buf;
int        size;

int
vc_readall(vc, buf, size)
struct vc *vc;
bufptr     buf;
int        size;
```

The routines *vc_read* and *vc_readall* read *size* bytes from the input channel into the buffer *buf*. The routine *vc_read* reads at least one byte and at most *size* bytes. The routine *vc_readall* reads exactly *size* bytes, unless an `EOF` is encountered.

Vc_read and *vc_readall* return the number of bytes read (including zero). If an `EOF` is encountered, the number of bytes read will be smaller than requested.

vc_write

```
int
vc_write(vc, buf, size)
struct vc *vc;
bufptr     buf;
int        size;
```

Vc_write writes *size* bytes from the buffer *buf* to the output channel.

Vc_write returns −1 on failure (i.e., broken connection, closed remote input channel). Otherwise it returns *size*.

vc_getp, vc_getpdone

```
int
vc_getp(vc, buf, blocking)
struct vc *vc;
bufptr    *buf;
int        blocking;

void vc_getpdone(vc, size)
struct vc *vc;
int        size;
```

Vc_getp and *vc_getpdone* are the virtual circuit equivalents of the circular buffer routines *cb_getp* and *cb_getpdone* (see *circbuf*(L)). *Vc_getp* and *vc_getpdone* fetch data from the virtual circuit input channel into the buffer *buf*.

If an `EOF` is encountered, *vc_getp* returns zero.

vc_putp, vc_putpdone

```
int
vc_putp(vc, buf, blocking)
struct vc *vc;
bufptr    *buf;
int        blocking;

void vc_putpdone(vc, size)
struct vc *vc;
int size;
```

Vc_putp and *vc_putpdone* are the virtual circuit equivalents of *cb_putp* and *cb_putpdone*. Instead of writing data to a circular buffer, data is written from the buffer *buf* to the virtual circuit output channel.

Vc_putp returns −1 upon failure (i.e., broken connection, closed remote input channel).

vc_avail

```
int
vc_avail(vc, which)
struct vc *vc;
int which;
```

Vc_avail returns the number of bytes immediately available in the input or output channel. If `VC_IN` is selected, the number of free bytes in the input channel is returned. If `VC_OUT` is selected, the number of free bytes in the output channel is returned.

Vc_avail returns −1 when the specified channel is closed.

vc_setsema

```
void
vc_setsema(vc, sema)
struct vc *vc;
semaphore *sema;
```

Vc_setsema attaches an external semaphore on the input channel. Each time a byte arrives on the input channel or when the input channel is closed, the semaphore is *upped* via *sema_up* (see *semaphore*(L)). The semaphore is initialized to the number of data bytes available in the input channel. This routine is the virtual circuit equivalent of *cb_setsema* (see *circbuf*(L)).

vc_warn

```
vc_warn(vc, which, rtn, arg)
struct vc *vc;
int which;
void (*rtn)();
int arg;
```

Vc_warn sets a warning routine for interesting events. When the input channel is selected (*which* == `VC_IN`), the routine *rtn* will be called whenever the input channel is not empty or when the input channel breaks/closes down. If the output channel is selected (*which* == `VC_OUT`), the routine will be called whenever the output channel state changes to empty or when the output channel breaks/closes down.

The specified argument *arg* will be passed as argument to *rtn*.

Example

The following example creates a virtual circuit, the client writes a password through the virtual circuit (‘‘Foo’’), which is checked by the server, and some data is transferred. On illegal passwords or failures, the connection is directly broken by the server.

```
#include "amoeba.h"
#include "vc.h"

some_server(icap, ocap)
capability *icap, *ocap;
{
    struct vc *vc;
    char buf[4], *bp;
    int size;

    /* create the virtual circuit */
    vc = vc_create(icap, ocap, ISIZE, OSIZE);
    if (vc == 0)
         exit(1);

    /* read 4 bytes */
    if (vc_readall(vc, buf, 4) != 4) {
         /* on failures, close down asynchronously */
         vc_close(vc, VC_BOTH|VC_ASYNC);
         exit(1);
    }

    /* Check password */
    if (strcmp(buf, "Foo")) {
         /* not ok, close down */
         vc_close(vc, VC_BOTH|VC_ASYNC);
         exit(1);
    }

    /* get a buffer pointer to received data */
    if ((size = vc_getp(vc, &bp, 1)) == 0) {
         vc_close(vc, VC_BOTH|VC_ASYNC);
         exit(1);
    }

    /* well, do something */
    do_something(bp, size);

    /* mark the data received */
    vc_getpdone(vc, size);
    /* and close down */
    vc_close(vc, VC_BOTH|VC_ASYNC);
    exit(0);
}
```

```
some_client(icap, ocap)
capability *icap, *ocap;
{
    struct vc *vc;
    char buf[10];

    /* create the virtual circuit */
    vc = vc_create(icap, ocap, ISIZE, OSIZE);
    if (vc == 0)
         exit(1);

    /* Close input channel */
    vc_close(vc, VC_IN);

    /* Write the password to the server */
    if (vc_write(vc, "Foo", 4) != 0) {
         /* close on failures */
         vc_close(vc, VC_OUT);
         exit(1);
    }

    /* write some data to the server */
    if (vc_write(vc, buf, 10) != 0) {
         vc_close(vc, VC_OUT);
         exit(1);
    }

    /* And close down */
    vc_close(vc, VC_BOTH);
    exit(0);
}
```

See Also

circbuf(L), rpc(L), semaphore(L).

(in libraries: libamoeba.a)

Name

x11 – the X server interface stubs

Synopsis

```
#include "amoeba.h"
#include "server/x11/Xamoeba.h"

errstat x11_reinit(server)
errstat x11_shutdown(server)
```

Description

These routines provide control over X servers to user programs. They are not part of the standard X interface provided by the X11 libraries.

Functions

x11_reinit

```
    errstat
    x11_reinit(server)
    capability *server;
```

X11_reinit sends a message to the X server asking it to re-initialize itself. As a result all connections are dropped, all windows are closed and various resources are released. It is commonly used when logging out, or before logging in, to ensure a clean display without dangling connections.

x11_shutdown

```
    errstat
    x11_shutdown(server)
    capability *server;
```

X11_shutdown asks the addressed X server to terminate. This results in an orderly shutdown of all connections, after which the server exits. It is useful when shutting down a workstation.

Diagnostics

Only RPC errors can be returned.

Example

The following code fragment (without any error checking) does a lookup of the X server specified on the command line and asks it to terminate.

```
#include "amoeba.h"
#include "ampolicy.h"
#include "server/x11/Xamoeba.h"

main(argc, argv)
int argc;
char **argv;
{
    capability scap;
    char sname[100];

    sprintf(sname, "%s/%s", DEF_XSVRDIR, argv[1]);
    name_lookup(sname, &scap);
    x11_shutdown(&scap);
}
```

See Also

X documentation.

8 Index

The italic references in the index are to programs or routines described in this manual.