Part XI
Chapter 11

Internal Functions

The following manual pages describe the C language interfaces for those functions internal to the ANSA support environment. None of the routines in this section should be called directly by application programs. Calls to the top level routines are generated by the preprocessor and stub generator.

All application programs should be compiled with the switch:

```
-I/usr/local/include
```

and then linked with:

```
/usr/local/lib/libansa.a
```

All the definitions needed by the stubs are collected together in `<ansa/ansa.h>` which is automatically included by all stubs. Header files for ANSA service interfaces should be kept in `/usr/local/include/ansa` so that they can also be picked up by the standard `-I` switch.

Internal modules should be compiled with the switch:

```
-I/usr/local/include/ansa/capsule
```

and should include the header file `"capsule.h"`. Modules using certain internal routines may need additional include files as specified in the individual manual pages.

Please note that your installation may have put the include files and libraries somewhere other than `/usr/local`.

All interface types defined in `<ansa/ansa.h>` have a prefix of `ansa_`, in order to avoid name clashes with application code. Within the interpreter this is not necessary and the equivalent internal types without the prefix are defined in the file `"standard.h"` which is included by `"capsule.h"`.

The interface synopsis for each routine in the following manual pages is in the proposed ANSI C standard format using function prototypes. However, in order to maximize portability, the source as issued does not yet use function prototypes.

NAME

Binder - binds capsules together via their external interfaces

PURPOSE

The binder acts as a local agent for the trader; enabling server capsules to advertise interfaces (via the trader) and create the communications socket necessary to accept invocations of the interface operations. The binder also enables client capsules to locate server capsules supporting a particular interface (via the trader) and to create the communications plug and channel necessary to invoke the interface operations.

SYNOPSIS

```
#include <ansa/ansa.h>
#include "binder.h"

ansa_Status binder_export    (InterfaceRef   *trref,
                               String          type,
                               String          context,
                               String          instance,
                               Cardinal        concurrency,
                               InterfaceId     id,
                               Dispatch       *dispatcher,
                               InterfaceRef   *ifref) ;

ansa_Status binder_withdraw  (InterfaceRef   *trref,
                               InterfaceRef   *ifref) ;

ansa_Status binder_import    (InterfaceRef   *trref,
                               String          type,
                               String          context,
                               String          constraints,
                               InterfaceRef   *ifref) ;

ansa_Status binder_discard   (InterfaceRef   *ifref) ;

void        binder_terminate (void) ;

ansa_Status binder_bindRef   (InterfaceRef   *ifref,
                               ansa_ChannelId *plugPtr) ;

ansa_Status binder_bindSvc   (Cardinal        concurrency,
                               InterfaceId     ifid,
                               Dispatch       *dispatcher
                               InterfaceRef   *ifref,
                               ansa_ChannelId *plugPtr) ;
```

DESCRIPTION

The *binder_export* function calls the nucleus to make a new socket with the specified concurrency and dispatch routine. If a particular (non-zero) socket is specified the binder will attempt to allocate it; otherwise a suitable one will be chosen and its identity returned. It then calls the trader to register the export and stores the details of the export instance for use by *binder_withdraw* and *binder_terminate.*

The *binder_withdraw* function cancels a preceding export by requesting the trader to delete the relevant export entry and requesting the nucleus to withdraw the socket.

The *binder_import* function calls the trader to lookup the interface and then calls the nucleus to make a plug for the server capsule's socket. If a particular (non-zero) plug is specified the binder will attempt to allocate it; otherwise a suitable one will be chosen and its identity returned.

The *binder_discard* function cancels a preceding import by requesting the nulceus to discard the plug.

The *binder_terminate* function withdraws all outstanding exports done by the capsule.

The *binder_bindRef* and *binder_bindSvc* functions do the actual work for *binder_import* and *binder_export*, respectively. They have been broken out to permit access from client stubs.

FILES

ERRORS

SEE ALSO

TRADER (X), Support Environment (X), NUCLEUS (XI).

USAGE

The *binder_export*, *binder_withdraw*, *binder_import*, *binder_discard* and *binder_terminate* functions calls are generated by the preprocessor. The *binder_terminate* function is also called by the system module after receiving a termination signal from the operating system.

FUTURE DIRECTIONS

CHANGE HISTORY

Date	Release	Pages	Changes
1/04/88	Version 1.0		initial release
16/12/88	Version 2.0		`stackSize` argument moved from *binder_export* to *nucleus_tasks* `bufferSize` argument from *binder_export* calculated dynamically `concurrency` argument added to *binder_export* argument types made consistent with `<ansa/ansa.h>` multiple protocol support added
10/4/89	Version 2.5		InterfaceId support added bindRef and bindSvc routines provided

NAME

Buffer - interpreter buffer management

PURPOSE

Provides buffer management functions for capsule message buffers. These buffers are used by client and server stubs to marshal and unmarshal the arguments or results of invocations.

SYNOPSIS

```
#include <ansa/ansa.h>      /* or "capsule.h" */
#include "buffer.h"
void             buffer_init  (Cardinal header,
                               Cardinal data,
                               Cardinal trailer) ;
ansa_BufferLink buffer_make  (Cardinal bytes) ;
void             buffer_free  (ansa_BufferLink descriptor)  ;
void             buffer_reset (ansa_BufferLink descriptor)  ;
void             buffer_swap  (ansa_BufferLink old,
                               ansa_BufferLink new) ;
```

DESCRIPTION

Buffers are constructed from a linked list of one or more buffer segments. A buffer segment is a contiguous area of memory containing all or part of a message. Each buffer segment has an associated control block called a buffer descriptor; this contains the base address and size of the buffer segment, plus the base address and size of that section of the segment which contains valid data. Buffer descriptors need not be contiguous with their associated buffer segment and may be linked together to form a buffer list.

The format of a buffer descriptor and a buffer link are defined in the file `<ansa/ansa.h>` included by `"capsule.h"`.

```
typedef struct ansa_buffer_descriptor *ansa_BufferLink ;
typedef struct ansa_buffer_descriptor
    {
    ansa_BufferLink link ; /* NULL = end of buffer list     */
    unsigned long   type ; /* unused                        */
    unsigned long  *base ; /* word aligned segment address  */
    unsigned long   size ; /* number of bytes in segment    */
    char           *data ; /* byte address of used section  */
    unsigned long   used ; /* number of bytes used          */
    }
    ansa_BufferDescriptor;
```

The *buffer_init* function is called by each communications protocol stack as part of its startup sequence. *Buffer_init* accumulates the maximum header, data and trailer sizes in order to calculate the size of a standard buffer and leave enough room for the largest header and trailer in all buffers.

The *buffer_make* function allocates, and returns a pointer to, a single buffer descriptor with an attached data segment of at least the requested number of bytes plus the largest header and trailer used by the communication protocols loaded in the capsule. The buffer is initialized with a used data section of zero bytes starting after the largest buffer header. If insufficient memory was available to allocate the buffer, a NULL pointer is returned. If a buffer length of zero bytes is requested then a standard buffer, which will hold an unfragmented message for any loaded protocol, is supplied.

The *buffer_free* function disposes of a buffer (and any buffers that are linked to it).

The *buffer_reset* function re-initializes the buffer by setting the fields of the first buffer's descriptor to the values used by *buffer_make* and freeing any linked buffers.

The *buffer_swap* function exchanges the buffer segments attached to the old and new buffer descriptors and frees the new buffer, which now consists of the new descriptor and the old segment, leaving the new segment attached to the old descriptor.

FILES

ERRORS

SEE ALSO

INSTRUCT (XI).

USAGE

The buffer management routines are used by the communications protocols and automatically generated stubs.

FUTURE DIRECTIONS

Linked buffer support may be added to rex and the stub generator.

CHANGE HISTORY

Date	Release	Pages	Changes
1/04/88	Version 1.0		initial release
16/12/88	Version 2.0		function names changed according to module naming conventions argument types made consistent with `<ansa/ansa.h>` buffer segment functions replaced by *buffer_swap* *buffer_init* function added

NAME

Channel - binds plugs to sockets

PURPOSE

Manages the capsule's channel table, enabling the creation, updating and deletion of entries for both plugs and sockets.

SYNOPSIS

```
#include "capsule.h"
#include "channel.h"

void     channel_init       (void) ;

void     channel_select     (ChannelId     index) ;

Status   channel_selectType (ChannelId     index,
                             ChannelType   type) ;

Status   channel_make       (ChannelId     index) ;

Status   channel_socket     (Cardinal      concurrency,
                             Dispatch     *dispatcher) ;

Status   channel_plug       (ProtocolId    protocol,
                             EndPoint      destination) ;

Boolean  channel_decrement  (void) ;

void     channel_increment  (void) ;

void     channel_free       (void) ;

void     channel_cleanup    (void) ;
```

DESCRIPTION

The *channel_init* function creates and initializes the channel table with the dummy (zero) entry and one free channel. The dummy entry is used by *channel_make* to initialize new channels.

The *channel_select* function makes the specified channel table entry the current one by setting the global variables `channelIndex` to its index and `channelPtr` to its address.

The *channel_selectType* function makes the specified channel the current one only if it is the correct type. The type `DUPLEX` is equivalent to `PLUG` or `SOCKET`.

The *channel_make* function finds, initializes and selects a free channel, extending the channel table if necessary. If `index` is non-zero then that particular channel is selected; otherwise the first free channel is selected.

The *channel_socket* function fills in the currently selected channel as a socket. The concurrency argument limits the number of invocations which may be active on the socket; when this limit is reached additional invocations are ignored until one of the current invocations completes. When an invocation which would not exceed the concurrency is received, then the `dispatcher` function is called.

The *channel_plug* function fills in the currently selected channel as a plug, which is bound via the specified `protocol` to the capsule and socket specified in the `destination` argument.

The *channel_decrement* function checks if the concurrency limit has been reached for the current channel; if so it just returns `FALSE`, otherwise it decrements the available concurrency and returns `TRUE`.

The *channel_increment* function increments the available concurrency of the current channel.

The *channel_free* function frees the current channel for re-use and disconnects all the sessions which are using it.

The *channel_cleanup* function frees all the capsules's channels and disconnects all of its sessions.

FILES

ERRORS

SEE ALSO

INSTRUCT (XI), NUCLEUS (XI), SESSION (XI).

USAGE

Channel_init is called by the main program in the nucleus during capsule initialization. *Channel_make*, *channel_socket* and *channel_plug* are only called by *nucleus_socket* and *nucleus_plug*. *Channel_select* and *channel_selectType* are called by nucleus, schedule and session. *Channel_decrement* and *channel_increment* are only called by schedule before and after dispatching an invocation. *Channel_free* is only called by *nucleus_withdraw* and *nucleus_discard*. *Channel_cleanup* is only called by the *Terminate* and *Abort* instructions during capsule termination.

FUTURE DIRECTIONS

Export stamps will be added.

CHANGE HISTORY

Date	Release	Pages	Changes
1/04/88	Version 1.0		initial release
16/12/88	Version 2.0		Completely redesigned as an internal module with all external interfaces provided by the nucleus.

NAME

ECS - synchronization support with eventcounts and sequencers.

PURPOSE

Provides primitives for controlling the relative ordering of concurrent thread execution within a capsule.

SYNOPSIS

```
#include    "capsule.h"
#include        "ecs.h"

typedef struct  ecrec {
    Counter     tag ;
    ThreadId    que ;
} ECRec ;

typedef struct  sqrec {
    Counter     tag ;
} SqRec ;

typedef     ECRec   *EventCount ;
typedef     SqRec   *Sequencer  ;

EventCount  ecs_makeEventCount  (Cardinal   iv) ;
Status      ecs_freeEventCount  (EventCount ec) ;
Counter     ecs_read            (EventCount ec) ;
void        ecs_await           (EventCount ec  ,
                                 Counter     v) ;
void        ecs_advance         (EventCount ec) ;
Sequencer   ecs_makeSequencer   (Cardinal   iv) ;
Status      ecs_freeSequencer   (Sequencer  sq) ;
Counter     ecs_ticket          (Sequencer  sq) ;
Counter     ecs_castCounter     (Cardinal    c) ;

#define ecs_makeEventCount0 ecs_makeEventCount ((Cardinal)0)
#define ecs_makeSequencer0  ecs_makeSequencer  ((Cardinal)0)
```

DESCRIPTION

The *ecs__makeEventCount* function allocates and returns a pointer of type EventCount to an ECRec record. The tag field is initialized to the value of iv, and que is set to the dummy (zero) ThreadId. If insufficient memory is available to allocate the record, a NULL pointer is returned.

The subsidiary definition *ecs__makeEventCount0* operates identically, except that the tag value is initialized to zero.

The *ecs__freeEventCount* function disposes of an eventcount if its associated queue is empty and returns ok, otherwise resourceInUse is returned.

The *ecs__read* function returns the current value of the eventcount ec.

The *ecs__await* function suspends the calling thread if the value of the eventcount ec is less than the parameter v. Threads are set into the WAITING state and placed onto a queue associated with the eventcount prior to *scheduler()* being called.

The *ecs__advance* function increments the current value of the eventcount ec by one and reawakens one or more threads on the associated queue which are awaiting the value just attained.

The *ecs__makeSequencer* function allocates and returns a pointer of type Sequencer to an SqRec record. The tag field is initialized to the value of iv. If insufficient memory is available to allocate the record, a NULL pointer is returned.

The subsidiary definition *ecs__makeSequencer0* operates identically, except that the tag value is initialized to zero.

The *ecs__freeSequencer* function disposes of a sequencer and returns ok.

The *ecs__ticket* function returns the current value of the sequencer sq and increments its value by one.

The *ecs__castCounter* function returns the value of its argument of type Cardinal cast to type Counter.

FILES

ERRORS

SEE ALSO

SCHEDULE (XI), THREAD (XI).

USAGE

These synchronization primitives are to be used in generated stubs to provide support for declarative path expression interface constraints.

FUTURE DIRECTIONS

This design is predicated on the use of non-preemptive scheduling and relies on mutual exclusion guarantees given by the current single processor environments of the testbench. Ports to multi-processor machine environments will require explicit low-level mutual exclusion mechanisms.

CHANGE HISTORY

Date	Release	Pages	Changes
8/3/89	Version 2.5		initial release

NAME

Idcache - cache of InterfaceId to plug mappings

PURPOSE

These routines maintain a cache of InterfaceId to plug mappings. Entries are made to the cache whenever an interface is imported or if a remote invocation is made on an InterfaceRef which does not have an entry in the cache.

In actual fact, the cache is indexed by InterfaceId and operation number; such a structure permits multiple plugs per imported interface.

Future releases may provide additional situations where entries are made in the cache.

SYNOPSIS

```
#include <ansa/ansa.h>   /* or "capsule.h" */
#include "idcache.h"

void            idcache_init (void) ;

int             idcache_add  (InterfaceId     id,
                              Cardinal        op,
                              ansa_ChannelId  ch) ;

void            idcache_del  (InterfaceId     id) ;

ansa_ChannelId  idcache_map  (InterfaceId     id,
                              Cardinal        op) ;
```

DESCRIPTION

idcache_init() initializes the cache, and is invoked during capsule initialization.

idcache_add() adds an entry to the cache corresponding to the specified InterfaceId and operation number. If an entry currently exists for that (InterfaceId, op) pair, all entries corresponding to that particular InterfaceId are purged from the cache before making the addition.

idcache_del() purges all entries in the cache corresponding to the specified InterfaceId.

idcache_map() scans the cache for the entry corresponding to the specified InterfaceId and operation number. If successful, the corresponding channel is returned as the value of the function; if not in the cache, the value (ansa_ChannelId)0 is returned.

FILES

ERRORS

SEE ALSO

USAGE

All uses of these routines in release 2.5 specify an operation number 0 in `idcache_add()` and `idcache_map()` calls.

FUTURE DIRECTIONS

CHANGE HISTORY

Date	Release	Pages	Changes
10/4/89	Version 2.5		initial release

NAME

Instruct - executes the interpreter instructions

PURPOSE

Provides (or re-packages) those instructions required by ANSA capsules but not provided by the host processor or operating system.

The distinction between instructions and operations is that instructions perform those functions that must be provided by the local capsule infrastructure and cannot be provided by separate capsules. Instructions are not usually referenced directly by source programs, but are executed by compiled or generated code.

On an ANSA processor, these instructions would be implemented in hardware or microcode. In an ANSA operating system they would be the system calls.

SYNOPSIS

```
#include <ansa/ansa.h>   /* or "capsule.h" */
#include "instruct.h"

ansa_Status    Call        (ansa_ChannelId   plug,
                             ansa_BufferLink  buffer) ;

ansa_Status    Cast        (ansa_ChannelId   plug,
                             ansa_BufferLink  buffer) ;

ansa_SessionId Request     (ansa_ChannelId   plug,
                             ansa_BufferLink  buffer) ;

ansa_Status    Collect     (ansa_SessionId   promise) ;

ansa_ThreadId  Spawn       (Dispatch        *dispatcher,
                             ansa_BufferLink  buffer) ;

ansa_ThreadId  Fork        (Dispatch        *dispatcher,
                             ansa_BufferLink  buffer) ;

ansa_Status    Join        (ansa_ThreadId    child) ;

ansa_ThreadId  Thread      (void) ;

void           Pause       (void) ;

void           Terminate   (String           message,
                             int              reason) ;

void           Abort       (String           module,
                             ansa_Status      reason) ;

typedef void  (Dispatch)   (ansa_ChannelId   socket,
                             ansa_BufferLink  buffer) ;
```

DESCRIPTION

The *Call* instruction transmits the request buffer to the capsule bound to the other end of the channel referenced by the plug. The calling thread is blocked until the corresponding response buffer is received. The request and response buffers both use the same buffer descriptor but the response buffer may have a different buffer segment.

The *Cast* instruction transmits the request buffer to the capsule bound to the other end of the channel referenced by the plug. The calling thread is only blocked until the request buffer has been processed.

The *Request* instruction transmits the request buffer to the capsule bound to the other end of the channel referenced by the plug. The calling thread is only blocked until the request buffer has been processed. A `promise` is returned which may later be used by the calling thread in a *Collect* instruction, which will block until the associated response is available. The request and response buffers both use the same buffer descriptor but the response buffer may have a different buffer segment. `Promises` may be collected in any order.

The *Spawn* instruction creates a child thread, which may be processed concurrently, and which the parent thread never intends to *Join*. The spawned thread will execute the `dispatcher` function and unmarshall its arguments from the `buffer`. When the `dispatcher` exits the spawned thread will terminate and the `buffer` will be freed.

The *Fork* instruction creates a child thread, which may be executed concurrently, and which the parent thread subsequently intends to *Join* in order to obtain its results. The forked thread will execute the `dispatcher` function and unmarshall its arguments from, and marshall its results into, the `buffer`. When the `dispatcher` exits, the forked thread will block until its parent joins with it and then terminate. The *Fork* instruction returns the thread identifier of the child thread which may be used by the parent thread in a *Join* instruction, which will block until the child thread has finished executing and its response buffer is available. The request and response buffers both use the same buffer descriptor but the response buffer may have a different buffer segment.

The *Thread* instruction returns the thread identifier of the calling thread.

The *Pause* instruction polls for incoming messages and schedules the calling task (and thread) on the end of the queue of tasks waiting to run.

The *Terminate* instruction outputs a message on `stderr` of the format:

```
:: capsule <id> terminated: <message> #<reason>
```

if the `message` argument is not `NULL`. It then performs the capsule cleanup operations and exits with `reason` as the status code.

The *Abort* instruction outputs a message on stderr of the format:

```
<module> :: capsule <id> aborted: status <reason>: <text>
```

where the `text` string describes the `reason`. If `reason` is a system error, it outputs a second line describing the system error. It then performs the capsule cleanup operations and exits with `reason` as the status code.

The *Dispatch* instruction is an up-call used by the interpreter to call a dispatch procedure to service incoming invocations. Outgoing response messages to incoming calls must use the same buffer descriptor as the incoming request, but the buffer segment may be changed. The outgoing response to a call is transmitted when the dispatch function exits. Responses are not sent for casts.

FILES

ERRORS

SEE ALSO

Support Environment (X), BUFFER (XI), NUCLEUS (XI).

USAGE

Before a *Call*, *Cast* or *Request* instruction can be executed, a plug must be created by *nucleus_plug*. Calls and casts performed by the same thread on the same plug are sequence preserving. Requests are not sequenced.

Forked and spawned threads may be serialized if insufficient tasks are provided (see *nucleus_tasks*) for their concurrent execution. In particular, forked threads will be executed on their parent thread's task if they have not been assigned a task by the time their parent executes the *Join* for them; and spawned threads may never be executed if there are no free tasks and none of the existing threads terminate.

Scheduling between threads (and the tasks executing them) is non pre-emptive. The *Pause* instruction should be used in long compute bound loops to allow the capsule to receive incoming messages and service the other tasks.

Before a *Dispatch* instruction can be executed by the interpreter a socket must have been created for it by *nucleus_socket*.

The *Abort* instruction should only be executed by the internal modules and, except for *Pause,* all other instructions should only be executed by code generated by the preproccessor or stub generator.

FUTURE DIRECTIONS

Support for object groups will be added.

CHANGE HISTORY

Date	Release	Pages	Changes
1/04/88	Version 1.0		initial release
16/12/88	Version 2.0		argument types made consistent with `<ansa/ansa.h>` non-blocking calls (*Request* and *Collect*) added multi-threading (*Spawn*, *Fork*, *Join*, *Thread* and *Pause*) added status code argument added to *Terminate*

NAME

MPS - a generic interface for message passing services

PURPOSE

Provides a standard interface between protocols such as REX and GEX and locally available message passing services. A map indicates which MPSs are loaded (aka the protocol stack). Each map entry contains the address of a table containing the entry points for each of the required MPS functions.

SYNOPSIS

```
#include "capsule.h"
#include "protocol.h"

typedef struct
{
   Status   (*mpsStartup)   (CapsuleAdr *self,
                             Cardinal   *extra);
   Status   (*mpsSendMsg)   (char       *pkt,
                             Cardinal   psize,
                             CapsuleAdr *remcap);
   Cardinal (*mpsReceiveMsg)(char       *pkt,
                             Cardinal   psize,
                             CapsuleAdr *remcap);
   void     (*mpsTick)      (void);
   void     (*MpsCleanup)   (void);
   Cardinal maxSendUnit ;
   Cardinal maxReceiveUnit ;
} MpsTable;

MpsTable *mpsMap[] ;
```

DESCRIPTION

mpsStartup initializes the message passing service. It is called during capsule initialization, it's principal function is to ensure that the MPS is functional and to acquire a unique address for this capsule/MPS combination. The *port number* may be specified in `self`. Any necessary local resources (e.g. sockets under BSD style networking) are also allocated.

On successful initialization two result values are provided, `self` holds the capsule address used by this MPS and `extra` is set to the number of bytes of header information which this MPS requires.

The port number is notified to the interpreter via the *protocol_open* operation.

The two message transfer routines take a pointer to a REX packet buffer (`pkt`) and a pointer to a capsule address structure (`remcap`) as parameters. Each also takes a buffer size parameter, for *mpsSendMsg* `psize` is the number of bytes to transmit, for *mpsReceiveMsg* it is the maximum size of the buffer. The current design relies upon REX to provide retry mechanisms to overcome transient errors and so any error codes which are signalled should reflect serious MPS failures.

mpsSendMsg attempts to send `psize` bytes of REX header and data to the nominated capsule and host. Returns `transmitFailure` or `ok`.

mpsReceiveMsg attempts to receive data waiting for this capsule and returns the number of bytes received (0 means there wasn't any) and the capsule address of the source. If the data available exceeds `psize` only the first `psize` bytes are copied into `pkt`, no indication is given that this has happened and the MPS quietly discards the excess.

mpsTick is called at irregular intervals to enable time related activities to be scheduled, the minimum inter-tick period is guaranteed by the interpreter.

mpsCleanup is called to allow the MPS to do any necessary housekeeping before the capsule is shutdown.

The values of *maxSendUnit* and *maxReceiveUnit* are used by Rex to determine how to fragment a message too large for a particular MPS.

FILES

ERRORS

SEE ALSO

MPSUDP(XI), MPSTCP(XI), PROTOCOL (XI), REX (XI).

USAGE

The functions for a particular MPS are called by an expression of the form:

```
        (*mpsMap[protocolIndex]->mpsTick)()
```

FUTURE DIRECTIONS

This MPS interface has been developed on the socket interface to the UDP/IP and TCP/IP protocols on various machines, other protocols and different networking styles streams are being considered so as to validate the general approach as well as the incorporation of Error signalling.

CHANGE HISTORY

Date	Release	Pages	Changes
1/04/88	Version 1.0		designed by E. Oskiewicz
16/12/88	Version 2.0		designed by D. Oliver and E. Oskiewicz
10/4/89	Version 2.5		addition of maxUnit fields to MPS Table entry

NAME

MPSIPC - a message passing service interface to UNIX system V named pipes.

PURPOSE

Implements the standard MPS interface between protocols such as REX and GEX and UNIX system V named pipes. This provides optimized transport for inter-capsule traffic when the capsules are on the same UNIX host system.

SYNOPSIS

```
#include "capsule.h"
#include "protocol.h"

Status     MPS_startup     (CapsuleAdr *locp,
                            Cardinal   *extra);

Status     MPS_send        (char       *bufp,
                            Cardinal    dlen,
                            CapsuleAdr *remp);

Cardinal   MPS_receive     (char       *bufp,
                            Cardinal    bmax,
                            CapsuleAdr *remp);

void       MPS_decay       (void);

void       MPS_cleanup     (void);
```

DESCRIPTION

MPS_startup initializes the IPC message passing service, tries to create a named pipe in the /usr/tmp directory, and opens a file descriptor to the named pipe. If successful, *protocol_open* is called with the pin id equivalent to this file descriptor.

The named pipe is given the name ansa.%d, where the %d is replaced by a formatted decimal number. If the port component of the capsule address locp is non-zero that value is used, otherwise the process id of the capsule is used. The remainder of locp is set to the ASCII hostname.

On return, locp contains the complete capsule address and extra is the size of the extra header required by this MPS for message encapsulation (4 bytes).

MPS_send attempts to send dlen bytes of data in bufp via the named pipe associated with the destination capsule remp. It always returns ok, relying upon protocols such as REX or GEX above to handle failures.

MPS_receive performs a two-part read from the named pipe. First the MPSIPC header is read to determine the remote capsule's process id, which is placed in remp, and the length of the message. The message is then read into bufp and the number of bytes received returned. If the message length exceeds bmax the capsule is terminated, as this is a heinous protocol error.

MPS_decay is a null operation as MPSIPC has no relevant housekeeping requirements.

MPS_cleanup is called just before the capsule expires to permit the removal of the named pipe from /usr/tmp.

FILES

ERRORS

SEE ALSO

MPS TABLE (XI), REX (XI).

USAGE

FUTURE DIRECTIONS

CHANGE HISTORY

Date	Release	Pages	Changes
16/12/88	Version 2.0		initial release

NAME

MPSTCP - a message passing service interface to the ARPA Transmission Control Protocol.

PURPOSE

Implements the standard MPS interface between protocols such as REX and GEX and the Transmission Control Protocol (TCP) which provides a connection-oriented transport service, layered above the Internet Protocol (IP). This manual page describes MPSTCP for the Berkeley socket abstraction.

SYNOPSIS

```
#include "capsule.h"
#include "protocol.h"

Status    MPS_startup    (CapsuleAdr *locp,
                          Cardinal   *extra);

Status    MPS_send       (char       *bufp,
                          Cardinal    dlen,
                          CapsuleAdr *remp);

Cardinal  MPS_receive    (char       *bufp,
                          Cardinal    bmax,
                          CapsuleAdr *remp);

void      MPS_decay      (void);

void      MPS_cleanup    (void);
```

DESCRIPTION

MPS_startup initializes the TCP message passing service and tries to acquire a non-blocking socket with a queue for incoming connection requests. If successful, *protocol_open* is called with the pin id equivalent to this listening socket number.

If the port component of the capsule address locp is non-zero that value is used, otherwise a port will be allocated by the system. The remainder of locp is set to the IP host number of the machine.

On return, locp contains the complete capsule address and extra is the size of the extra header required by this MPS for message encapsulation.

MPS_send attempts to send dlen bytes of data in bufp via a socket connected to the remote capsule remp, if no connection exists and resources are available a connection will be made. Always returns ok and relies upon protocols such as REX or GEX above to handle failures.

MPS_receive's behaviour is determined by value of the global variable pinIndex. If equal to the listening socket, and resources are available, a request for connection is accepted and 0 returned.

Otherwise a connection exists and a two-part read is performed. First the MPSTCP header is read to determine the remote capsule address, which is placed in remp, and the length of the message. The message is then read into bufp and the number of bytes received returned. If the first read gets 0, then the connection is being shutdown by the remote capsule and resources are released before returning a value of 0. If the message length exceeds bmax the message is discarded, but the actual length is still returned.

MPS_decay is called at regular intervals by the session handler and initiates the close down of any connections which have not been used during the previous interval.

MPS_cleanup is called just before the capsule expires to initiate the close down of all connections and close the listening socket.

FILES

ERRORS

SEE ALSO

MPS TABLE (XI), REX (XI).

USAGE

FUTURE DIRECTIONS

The management of resources for all connection-oriented protocols may be moved above the MPS interface.

CHANGE HISTORY

Date	Release	Pages	Changes
16/12/88	Version 2.0		initial release

NAME

MPSUDP - a message passing service interface to the ARPA User Datagram Protocol.

PURPOSE

Implements the standard MPS interface between protocols such as REX and GEX and the User Datagram Protocol layered above the Internet Protocol (IP). This manual page describes MPSUDP for the Berkeley socket abstraction.

SYNOPSIS

```
#include "capsule.h"
#include "protocol.h"

Status    MPS_startup    (CapsuleAdr *self,
                          Cardinal   *extra);

Status    MPS_sendMsg    (char       *pkt,
                          Cardinal   psize,
                          CapsuleAdr *remcap);

Cardinal  MPS_receiveMsg (char       *pkt,
                          Cardinal   psize,
                          CapsuleAdr *remcap);

void      MPS_tick       (void);

void      MPS_cleanup    (void);
```

DESCRIPTION

MPS__startup tries to acquire a nonblocking datagram socket, if the user specifies a non-zero port in `self` that value is used; otherwise a system provided port will be used. The port number is cached for use with the other interface routines. The remainder of self is set to the IP host number of this machine; `Extra` is returned as zero, as MPSUDP needs no additional encapsulation header space in the buffers.

MPS__sendMsg attempts to send `psize` bytes of REX header and data to the nominated capsule and host. Returns `transmitFailure` or `ok`.

MPS__receiveMsg accepts any data waiting for this capsule, it returns the number of bytes received and the capsule address of the source.

MPS__tick and *MPS__cleanup* are null operations as MPSUDP has no relevant housekeeping requirements.

FILES

ERRORS

SEE ALSO

MPS (*XI*).

USAGE

FUTURE DIRECTIONS

Error signalling and the `MaxMsg` value need to be considered.

CHANGE HISTORY

Date	Release	Pages	Changes
1/04/88	Version 1.0		designed by E. Oskiewicz
16/12/88	Version 2.0		designed by E. Oskiewicz

NAME

Nucleus - resource allocation

PURPOSE

Provides an object-like interface to the host operating system and capsule resource allocation functions. The nucleus also contains the capsule's *main* function.

SYNOPSIS

```
#include <ansa/ansa.h>    /* or "capsule.h" */
#include "nucleus.h"

ansa_Status nucleus_tasks     (Cardinal         extraTasks,
                               Cardinal         stackSize);

ansa_Status nucleus_socket    (Cardinal         concurrency,
                               Dispatch        *dispatcher,
                               ansa_ChannelId  *socketPtr);

ansa_Status nucleus_withdraw (ansa_ChannelId   socket) ;

ansa_Status nucleus_plug      (ansa_ProtocolId  protocol,
                               ansa_EndPoint    destination,
                               ansa_CapsuleId   client,
                               ansa_ChannelId  *plugPtr);

ansa_Status nucleus_discard   (ansa_ChannelId   plug) ;

extern void body              (int argc,  char *argv[]) ;

ansa_CapsuleId  nucleus_createCapsule  (string   type,
                                        string   arguments,
                                        string   properties) ;

void            nucleus_deleteCapsule  (ansa_CapsuleId id) ;
```

DESCRIPTION

The *nucleus_tasks* function enables multi-tasking within the capsule and creates extraTasks tasks with stacks of size stackSize bytes. Tasks are anonymous and can be assigned by the interpreter to execute any thread; but once assigned will not be re-assigned until their thread terminates. Stacks must therefore be large enough to execute any thread in the capsule and (because of nested *Forks*) all of its offspring. All capsules start with one task, which uses the main stack, *nucleus_tasks* can then be called to increase the number of tasks. Because most systems only have mechanisms for automatically expanding or detecting overflow on the main stack, the sizes of any extra stacks should be generous. The nucleus imposes a system specific minimum stack size; see "`stack.h`".

The *nucleus_socket* function allocates and initializes a socket. If `socketPtr` addresses a zero index then a free socket is allocated and its index returned, otherwise the specified socket is used. The `dispatcher` argument specifies the address of the dispatch procedure which is to service incoming invocations on the socket, and the concurrency argument limits the number of invocations which may be active on the socket at the same time.

The *nucleus_plug* function allocates and initializes a plug. If `plugPtr` addresses a zero index then a free plug is allocated and its index returned, otherwise the specified plug is used. The `destination` argument contains the (protocol specific) capsule address and channel index of the socket, to which the new plug is to be bound, and the `protocol` argument specifies the protocol stack to be used for communicating with the socket. See "`options.h`" for those protocol stacks loaded in your capsule. The `client` argument must be zero or the calling capsule's ID; it is there to allow for a future out-of-capsule nucleus.

The *nucleus_withdraw* function cancels a prior *nucleus_socket* and the *nucleus_discard* function cancels a prior *nucleus_plug*.

The nucleus also contains the function *main* for the capsule. This performs capsule initialization and then calls the application function *body*, passing on the arguments supplied to *main* by the system. The capsule will not necessarily terminate when *body* exits, because other threads may not have terminated and sockets may still be active. The capsule will terminate silently with an exit status of zero when all threads have terminated and all sockets have been withdrawn. Premature termination with a message and specified exit status can be forced via the *Terminate* instruction.

The *nucleus_createCapsule* function creates a capsule from the template specified by the `type` parameter, passing it the argument string in the `arguments` parameter and initializing its environment to the `NAME=VALUE` pairs specified in the `properties` parameter. If successful, it returns the `CapsuleId` of the created capsule, otherwise it returns `(ansa_CapsuleId)0`.

The *nucleus_deleteCapsule* function causes the graceful termination of the specified capsule.

FILES

ERRORS

SEE ALSO

BINDER (XI), CHANNEL (XI), INSTRUCT (XI).

USAGE

All capsules must supply a *body* function instead of a *main* function. This will normally perform the application initialization, imports and exports (which will result in calls to *nucleus_plug* and *nucleus_socket*) and create any extra tasks and threads required. Server capsules would then normally exit the *body* function to await incoming invocations on their sockets, but this is only required if no extra tasks were created to service invocations concurrently with *body*. Spawned threads will not be executed until a task becomes free; in single-tasking capsules this means after *body* has exited.

FUTURE DIRECTIONS

CHANGE HISTORY

Date	Release	Pages	Changes
1/04/88	Version 1.0		initial release
16/12/88	Version 2.0		function names changed according to module naming conventions argument types made consistent with `<ansa/ansa.h>` *nucleus_tasks* and *nucleus_plug* added *Listen* removed *main* moved to nucleus and replaced by *body* in application
10/4/89	Version 2.5		addition of capsule creation and destruction routines

NAME

pin - local device control

PURPOSE

Provides a low level interface to the `select` system call for the purpose of controlling local devices.

SYNOPSIS

```
#include <ansa/ansa.h>   /* or "capsule.h" */
#include "protocol.h"
void   pin_open  (PinId  pin,  void (*handler)());
void   pin_close (PinId  pin);
```

DESCRIPTION

Pins are an abstraction of whatever local device identifiers are provided by the host operating system (file descripters, sockets etc.). They are not normally visible to applications programs and are intended to increase the portability of the interpreter. Because the interpreter polls or catches the interrupts of all local devices attached to the capsule, special arrangements must be made for applications which require direct control of local devices. Such application device drivers must have intimate knowledge of the device control mechanisms of the host operating system and are therefore not portable.

The pin interface provides the minimum facilities for sharing the device polling or interrupt mechanisms between the interpreter and the application. All the device setup and control code must be done by the application. The interpreter will merely notify the application when there is incoming data available, it will not read it or execute any other function on the pin.

The `pin_open` function requests the interpreter to call the supplied device `handler` whenever data is available for reading.

The `pin_close` function requests the interpreter to stop calling the device `handler`.

FILES

ERRORS

SEE ALSO

PROTOCOL (XI), PROTOCOL TABLE (XI).

USAGE

A device handler with the following signature must be supplied by the application:

```
void   handler   (PinId  pin);
```

This handler is run as part of the interpreter and as such has no thread or task dedicated to it. Therefore it must not use any of the interpreter functions that assume a thread and task. The only function that is safe to use is spawn.

If the handler wishes to use any other interpreter functions or be scheduled with the rest of the application threads it must perform a spawn function to acquire its own thread. If there are not sufficient tasks available to run the thread its execution may be delayed for some time. It is therefore advisable for the handler to close the pin before doing a spawn in order to prevent multiple notifications of the same event. The spawned thread must re-open the pin after it has finished processing.

The following code fragments illustrate the use of the pin handler to read single key presses from stdin.

```
void body()
{
     (void)system("stty raw");
     nucleus_tasks(1, 0);       /* because we spawn a thread */
     pin_open((PinId)0, handler);     /* open pin for stdin */

     .....

     pin_close((PinId)0);
     (void)system("stty cooked");
}

void handler(pin)
PinId pin;
{
     BufferLink buf;

                /* remove handler to prevent repeat firings */
     pin_close(pin);
```

```
          buf = buffer_make(0);
          buf->data[0] = (char)pin;
          buf->used = 1;

          (void)Spawn(key, buf);
     }

     void key(skt, buf)
     ChannelId skt;
     BufferLink buf;
     {
          char rbuf[128];

          (void)read(0, rbuf, 1);
          (void)write(1, rbuf, 1);
                                                  /* restore handler */
          pin_open((PinId)buf->data[0], handler);
     }
```

FUTURE DIRECTIONS

CHANGE HISTORY

Date	Release	Pages	Changes
20/03/89	Version 2.5		initial release

NAME
Protocol - protocol stack handler.

PURPOSE
Provides housekeeping functions across all loaded protocol stacks.

SYNOPSIS
```
#include "capsule.h"
#include "protocol.h"

void  protocol_init    (void) ;

void  protocol_open    (PinId new) ;

void  protocol_close   (PinId old) ;

void  protocol_tick    (void) ;

void  protocol_cleanup (void) ;
```

DESCRIPTION
The *protocol_init* function calls the *startup* function for each loaded protocol stack.

The *protocol_open* function adds the pin (UNIX socket, file descriptor or similar i/o identifier) to the set on which incoming messages can be read and processed.

The *protocol_close* function removes the pin from the set on which incoming messages can be read and processed.

The *protocol_tick* function calls the *MpsTick* function for each loaded MPS.

The *protocol_cleanup* function calls the *cleanup* function for each loaded protocol stack.

FILES

ERRORS

SEE ALSO
MPS TABLE (XI), PROTOCOL TABLE (XI), REX (XI).

USAGE

Protocol_init is only called by *main* in nucleus. *Protocol_open* and *protocol_close* are called by the various MPSs. *Protocol_tick* is called by *decaySession*. *Protocol_cleanup* is called by the *Terminate* and *Abort* instructions.

FUTURE DIRECTIONS

CHANGE HISTORY

Date	Release	Pages	Changes
16/12/88	Version 2.0		initial release

NAME

Protocol table - a generic interface for protocol handlers.

PURPOSE

The testbench is designed to operate on top of a wide variety of protocols, the protocols available may vary but all are required to provide the same basic service. A map indicates which protocols are loaded. Each map entry contains the address of a table containing the entry points for each of the required protocol functions.

SYNOPSIS

```
#include "capsule.h"
#include "protocol.h"

typedef struct
{
    Status  (*startup)     (void);
    Status  (*connect)     (void);
    Status  (*receive)     (void);
    Status  (*send)        (void);
    Status  (*reply)       (void);
    Status  (*cast)        (void);
    void    (*reject)      (void);
    void    (*disconnect)  (void);
    void    (*cleanup)     (void);
}   ProtocolTable;

ProtocolTable *protocolMap[];
```

DESCRIPTION

All operations are parameterless, the session and MPS to which the current operation refers being held in a global data structures indicated by session and protocol indices. In the case of the receive operation the session is acquired when a valid incoming message has arrived.

Startup is called to initialize the protocol and it's protocol stack; it is called once for each MPS in the protocol stack, no other protocol operation will be invoked until all calls to *startup* have been made. Each time it is called *startup* will notify the interpreter via *buffer_init* of the buffer sizes required by this protocol/MPS combination. An unsuccessful return code (= ok) indicates that this combination is not available.

Connect obsolete

Receive is called to accept the next buffer from the current MPS. In addition to the returned status code, a global variable `sessionIndex` is set to a non-zero value if the interpreter has some action to take; otherwise it is assumed that the protocol handler has swallowed any message which arrived.

Send transmits a `CALL` message to the remote endpoint for the current session. It does not block, the protocol handler will cope with retries and acknowledgements. The interpreter will call *receive* at some later time to collect the response.

Reply similar to *send* but transmits a `REPLY` message to the remote endpoint for the current session.

Cast transmits a `CAST` message to the remote endpoint for the current session.

Reject is used to make the protocol handler "forget" the last message, it is intended to be used to erase any record of the arrival of a CALL or CAST so that a retransmission can be processed as though it were seen for the first time.

Disconnect tidies up the session prior to being discarded by the interpreter

Cleanup is called once for each MPS immediately prior to the capsule being shut down. The protocol should take the opportunity to do any housekeeping required and to call the MPS cleanup routine.

FILES

ERRORS

SEE ALSO

MPS TABLE (XI), PROTOCOL (XI), REX (XI).

USAGE

The functions for a particular protocol are called by an expression of the form:

```
(*protocolMap[protocolIndex]->send)()
```

FUTURE DIRECTIONS

CHANGE HISTORY

Date	Release	Pages	Changes
16/12/88	Version 2.0		initial release designed by D. Oliver, E. Oskiewicz.

NAME

REX - an RPC interface to enable object interaction.

PURPOSE

Provides an RPC service which conforms to the standard interface for remote interaction, regardless of the underlying message passing service.

SYNOPSIS

```
Status  rex_startup    (void);
Status  rex_send       (void);
Status  rex_reply      (void);
Status  rex_cast       (void);
Status  rex_receive    (void);
Status  rex_connect    (void);
void    rex_disconnect (void);
void    rex_reject     (void);
void    rex_cleanup    (void);
void    rex_printstats (FILE *fd);
```

DESCRIPTION

These operations are used by the interpreter to invoke REX activity. With the exception of *rex_printstats*, they are invoked via the protocol table and are parameterless (because all pertinent details are recorded in global data structures).

Rex_startup ensures that REX and its message passing services are initialized and available for use. Each MPS allocates a unique id for this capsule and also indicate their maximum packet size and header requirements.

REX provides three message sending operations: *rex_send*, *rex_cast*, and *rex_reply*. These are all non-blocking, i.e. they return when the message has been sent or a locally detectable error occurs. Buffers which are too big to be sent in one packet are transparently sent as a stream of fragments. Some hardware architectures limit the maximum size of a standard contiguous region of memory and this will limit the sizes of buffer which may be sent or received.

Rex_send is used to send a call message, a reply is expected in return.

Rex_reply is used to send the reply to a call.

Rex_cast is used to send a cast message for which no acknowledgement or reply is expected. Although *rex_cast* returns the same error codes as *rex_send* this is intended as advice because no error recovery will be attempted by REX.

There is one message reception operation:

Rex_receive is used to accept and predigest incoming messages. It will quietly assemble fragmented buffers, discard erroneous messages and retransmissions and send acknowledgements.

The outcome of *rex_receive* depends on three reply values, a status code, a session index and the state of the session referenced by the session index. If the message requires further action by the interpreter then sessionIndex is non zero, otherwise there is nothing further for the interpreter to do.

The decision to call *rex_receive* is made by the interpreter which uses the *select* system call to poll the capsule's socket.

There are four housekeeping operations:

Rex_connect is a null operation, it always returns ok.

Rex_disconnect is called to to do housekeeping before a session is discarded, it's main actions are acknowledging unanswered messages and disarming unfired alarms.

Rex_reject is used by the interpreter to cause REX to forget casts or unacknowledged calls which have been received. This involves discarding buffers and resetting sequence numbers.

Rex_cleanup is called prior to capsule termination, this releases any resources acquired and calls the current MPS cleanup routine. It also calls *rex_printstats*.

There is an additional operation, intended for developers:

rex_printstats, if statistics gathering is enabled, causes REX to dump statistics about messages sent and received and errors detected; otherwise it has no effect.

The following timer operations are called when a REX initiated timer fires:

Rex__flow is used to control the rate of transmission of fragments, this timer is armed when the first fragment is sent. Each time *rex__flow* is called it sends the next fragment of the current buffer. When the last fragment has been sent it arms a flow probe timer, otherwise it re-arms the flow timer.

Rex__flowProbe is used to solicit a `fragnack` by retransmitting the first fragment (up to some threshold).

Rex__checkFragmentProgress is called when progress is not being made in a fragmented transfer. It constructs and sends a `fragnack`, if a threshold of unanswered `fragnacks` is exceeded, the incoming message is abandoned.

Rex__retry is called when a timeout occurs. This timer is armed whenever a reply is expected in response to a message sent. Each time *rex__retry* is called it resends the message and rearms the retry timer (subject to limits determined at compilation time).

The REX procedures make use of an interface to an idealized message passing service (see MPS TABLE (XI)) to insulate themselves from the protocols, topology and hardware used by lower layers.

FILES

ERRORS

SEE ALSO

INTERPRET (XI), MPS TABLES(XI), MPSUDP(XI), MPSTCP(XI).

USAGE

FUTURE DIRECTIONS

CHANGE HISTORY

Date	Release	Pages	Changes
1/04/88	Version 1.0		written by E. Oskiewicz
16/12/88	Version 2.0		written by E. Oskiewicz

NAME

Schedule - controls task, thread, message and alarm scheduling

PURPOSE

Provides a comprehensive scheduling service for a capsule. Waiting messages are read and processed, alarms processed, threads woken or dispatched, threads assigned to tasks and tasks executed or resumed.

SYNOPSIS

```
#include "capsule.h"
#include "schedule.h"

void schedule      (void) ;

void schedule_poll (void) ;
```

DESCRIPTION

The *schedule* function assigns free tasks to execute queued threads, resumes unblocked tasks, receives incoming messages and processes alarms. If the calling thread/task is ever resumed, schedule will return. If the calling thread has finished its execution, both the thread and task should be freed before calling *schedule*.

The *schedule_poll* function reads any queued messages and queues (wakes or dispatches) a thread to process each one, before returning.

FILES

ERRORS

SEE ALSO

INSTRUCT (XI), NUCLEUS (XI), REX (XI).

USAGE

The *schedule* function is called by those instructions that block the calling thread, or by the nucleus when the execution of the calling thread is complete. The *schedule_poll* function is only used by the *Pause* instruction before waking its calling task and calling *schedule*.

FUTURE DIRECTIONS

CHANGE HISTORY

Date	Release	Pages	Changes
1/04/88	Version 1.0		initially released as part of INTERPRET
16/12/88	Version 2.0		*schedule* split off into its own module *schedule_poll* function added to support the *Pause* instruction

NAME

Session - manages the capsule's session table.

PURPOSE

Provides the storage and management of communications session data. A session is a temporary association between two communications endpoints whose lifetime is sufficiently long that all the required guarantees and constraints can be maintained. Session table entries are purged after they have been dormant long enough to guarantee that there are no messages still in transit. (Strictly speaking each capsule only manages its own half-session.)

SYNOPSIS

```
#include "capsule.h"
#include "session.h"

void    session_init           (void) ;

Status  session_invalid        (SessionId  index) ;

void    session_select         (SessionId  index) ;

Status  session_selectOutgoing (ChannelId  sourceChannel,
                                Cardinal   sequencing) ;

Status  session_selectIncoming (ChannelId    destChannel,
                                SessionId    destSession,
                                EndPoint    *sourceEndPtr,
                                SessionId    sourceSession
                                InterfaceId  ifID) ;

void    session_setState       (SessionSt  new) ;

Boolean session_testFlag       (Cardinal   flag) ;

void    session_setFlag        (Cardinal   flag) ;

void    session_clearFlag      (Cardinal   flag) ;

void    session_decrement      (void) ;

void    session_attachBuffer   (BufferLink buffer) ;

void    session_detachBuffer   (void) ;

void    session_setAlarm       (MilliSecs  delay,
                                Action     action) ;

void    session_clearAlarm     (void) ;

Time    session_readAlarm      (void) ;

Action  session_triggerAlarm   (void) ;
```

```
void    session_disconnect    (void) ;
```

DESCRIPTION

The *session_init* function creates and initializes the session table. This is later expanded by *session_selectOutgoing* and *session_selectIncoming* as required.

The *session_invalid* function checks if the session index is out of range.

The *session_select* function checks if the session index is in range and then selects the indexed entry as the current one by setting the global variables `sessionIndex` to its index and `sessionPtr` to its address.

The *session_selectOutgoing* function selects the session table entry for the specified channel and current thread; either by locating an existing one or by creating a new one. If the `sequencing` argument is `SEQUENCED` then invocations of the same channel by the same thread are sequence preserving; otherwise if it is `UNSEQUENCED` then there are no sequencing guarantees which need to be preserved.

The *session_selectIncoming* function selects the session table entry for the message just received; either by locating the specified session or, if it has been re-used, by creating a new one.

The *session_setState* function changes the current session's state and if the new state is `IDLE` sets the decay alarm.

The *session_testFlag*, *session_setFlag* and *session_clearFlag* functions perform the obvious operations on one of the current session's flags without disturbing the others.

The *session_decrement* function decrements the current session's sequence number after a message has been rejected so as to avoid rejection of a re-transmission as a duplicate.

The *session_attachBuffer* and *session_detachBuffer* functions attach and detach a buffer to the current session, freeing or swapping any existing buffers according to the session state.

The *session_setAlarm*, *session_clearAlarm*, *session_readAlarm* and *session_triggerAlarm* functions control the alarm for the current session. *Session_setAlarm* aborts if an alarm is already set, but *session_clearAlarm* will re-clear a cleared alarm. *Session_readAlarm* non-destructively reads thew first alarm set. *Session_triggerAlarm* clears the first alarm that has fired and returns a pointer to the action function specified to *session_setAlarm*.

The *session_disconnect* function arranges for the disconnect function of the current session's protocol to be called when the session state becomes IDLE and to free the session.

FILES

ERRORS

SEE ALSO

INSTRUCT (XI), NUCLEUS (XI), REX (XI).

USAGE

Session_init is only called by the *main* function in the nucleus. *Session_selectOutgoing* is called by those instructions that implement inter-capsule communications. *Session_selectIncoming* is called by the protocol handlers as soon as they receive a message. *Session_disconnect* is only called by *channel_cleanup*.

FUTURE DIRECTIONS

Currently sessions are only robust in the face of passive communication failures. To survive malicious attacks, challenge/response functions within a secure protocol are required for the re-establishment of dormant sessions.

CHANGE HISTORY

Date	Release	Pages	Changes
1/04/88	Version 1.0		initially released as part of INTERPRET
16/12/88	Version 2.0		completely redesigned as a separate module
10/04/89	Version 2.5		addition of ifID to selectIncoming()

NAME

stack - stack handler.

PURPOSE

Provides register dumping and stack switching functions to support multi-tasking. The implementation of multi-tasking is designed to use the standard C library functions *setjmp* and *longjmp* whenever possible so as to increase portability by avoiding assembler code for register manipulation. The actual switching of stacks is done by altering the saved values of the stack front and stack frame registers in the RegDump before calling *stack_switch*. Unfortunately, some implementations of *longjmp* check that *longjmp* is being called by a more deeply nested function than *setjmp*. For these systems, equivalent assembler routines must be written; otherwise the stack functions are just macro definitions of *setjmp* and *longjmp*.

SYNOPSIS

```
#include "stack.h"
int       stack_dump          (RegDump env) ;
void      stack_switch        (RegDump env, int val) ;
```

DESCRIPTION

See *setjmp* and *longjmp* in the UNIX manual.

FILES

ERRORS

SEE ALSO

SCHEDULE (XI), TASK (XI), THREAD (XI).

USAGE

Stack_dump is only used to define the *task_dump* macro. *Stack_switch* is only used by *task_switch* and *task_dispatch*.

FUTURE DIRECTIONS

CHANGE HISTORY

Date	Release	Pages	Changes
16/12/88	Version 2.0		initial release

NAME

System - interfaces to the host operating system.

PURPOSE

Provides indirect access to the necessary functionality of the host operating system. The only other modules which call the host operating system directly are the various MPS modules.

SYNOPSIS

```
#include "capsule.h"
#include "system.h"

void     system_init     (void) ;

Cardinal *system_allocate (Cardinal     bytes) ;

Cardinal *system_extend   (Cardinal    *pointer,
                           Cardinal     bytes) ;

void     system_free     (Cardinal    *pointer) ;

Set      system_wait     (Time         alarm) ;

void     system_HtoN     (Cardinal     type,
                          CapsuleAdr  *ca) ;

void     system_NtoH     (Cardinal     type,
                          CapsuleAdr  *ca) ;

int      system_ipc_heq  (u_char      *a1,
                          Cardinal     l1,
                          u_char      *a2,
                          Cardinal     l2) ;

void     system_genIfID  (InterfaceId  ifID) ;

int      system_cmpIfID  (InterfaceId  if1,
                          InterfaceId  if2) ;

void     system_copyIfID (InterfaceId  to,
                          InterfaceId  from) ;

void     system_putIfID  (char        *bf,
                          InterfaceId  ifID) ;

void     system_getIfID  (char        *bf,
                          InterfaceId  ifID) ;

int      system_allocateRef (InterfaceId   id,
                             AddressHint   ah,
                             InterfaceRef *ref) ;

void     system_freeRef  (InterfaceRef *ref) ;
```

```
Cardinal  system_time    (void) ;
```

DESCRIPTION

The *system_init* function performs system specific initialization.

The *system_allocate* function allocates an area of memory which is at least the requested number of bytes long. The *system_extend* function extends a previously allocated area to the requested size; if extension in-situ is not possible, a new allocation will be made, the contents of the old area will be copied to the start of the new one and the old area freed. If successful both functions return a (word-aligned) pointer to the area; otherwise they return NULL. Neither function initializes the newly allocated or extended memory.

The *system_free* function de-allocates a previously allocated or extended area of memory.

The *system_wait* function performs a timed-out read of the capsule's i/o pins (UNIX sockets, file descriptors, TSAPs etc.). The exact nature of a specific pin is known only by the system and MPS modules. The protocol module keeps track of which pins are in use and which protocol stack is servicing each pin. *System_wait* returns the set of pins for which there are messages waiting to be accepted; if the alarm triggered, this set will be empty.

The *system_HtoN* function and the *system_NtoH* function cause any structured items in a capsule address to be converted from host to network format and network to host format, respectively. The *system_ipc_heq* function compares two IPC addresses, returning 0 if they are the same, non-zero if they are different.

The *system_genIfID* function generates a new InterfaceId. The *system_cmpIfID* function compares two InterfaceId's, returning 0 if they are the same. The *system_copyIfID* function copies the value of from to to. The *system_putIfID* function creates a string representation for the InterfaceId in the buffer provided, while the *system_getIfID* function creates an InterfaceId from the string representation in bf.

The *system_allocateRef* function generates an InterfaceRef from the InterfaceId and AddressHint provided, returning a pointer to the allocated InterfaceRef. The return value of the function is 1 if successful, 0 if not. The *system_freeRef* function returns storage allocated in a previous call to *system_allocateRef*.

The *system_time* function returns a representation of the current time as a Cardinal. The units of this value is deemed to be seconds.

FILES

MASTER_FILE	(defined in options.h, but normally /usr/local/etc/ansa/masteraddress)
TRADER_FILE	(defined in options.h, but normally /usr/local/etc/ansa/traderaddress)

If MASTER_FILE exists, it is assumed to contain the address of the master trader to be used by capsules on this host, overriding the one in options.h. If TRADER_FILE exits, it is assumed to contain the address of the default trader to be used by capsules on this host, overriding the one in options.h. A trader address is an ASCII string in the same (protocol specific) format as the definition of TRADER_ADDRESS in options.h, but without the enclosing double quotes.

ERRORS

SEE ALSO

MPS (XI), PROTOCOL (XI).

USAGE

System_init is only called by the *main* function in the nucleus and must be the first initialization function called. *System_wait* is only called by *schedule*. *System_allocate*, *system_extend* and *system_free* are a repackaging of the C library routines *malloc*, *realloc* and *free* to use word aligned pointers and provide a single monitor point for memory usage.

FUTURE DIRECTIONS

CHANGE HISTORY

Date	Release	Pages	Changes
16/12/88	Version 2.0		initial release
10/4/89	Version 2.5		addition of InterfaceId, InterfaceRef and time routines

NAME

Task - multi-tasking support.

PURPOSE

Provides the task management functions required to execute more than one thread at a time. A thread is an independent execution path which can be executed concurrently with other threads. While a thread is being executed, it requires a stack to store local variables and function return links; when a thread is blocked, it requires a register dump area to save the processor state so that the processor can execute another thread.

A thread only needs a register dump and stack while executing; these are provided by a task so that an executing thread may be blocked to allow other tasks to execute their threads. Tasks are significant resources and their unrestricted creation can easily exhaust the available memory. Therefore the number of tasks in a capsule is controlled by the application via the *nucleus_tasks* function.

This restriction on the number of tasks is dealt with by serializing the execution of any excess threads. A deadlock may occur if an executing thread blocks waiting for a thread which has not started executing due to resource limitations. This is avoided by nesting the execution of the thread being waited for, on the task of the waiting thread. In the current implementation, this can only occur if a parent thread tries to join with a child thread which has not started executing. Because of this thread nesting, the stack sizes should be calculated on the basis of executing all potential offspring on the same task.

SYNOPSIS

```
#include "capsule.h"
#include "task.h"

void      task_init       (void) ;

void      task_setup      (void) ;

Status    task_make       (Cardinal  extraTasks,
                           Cardinal  stackSize) ;

void      task_select     (TaskId    index) ;

void      task_wake       (TaskId    index) ;

void      task_setThread  (void) ;
```

```
int     task_dump      (void) ;
void    task_switch    (void) ;
void    task_dispatch  (void) ;
void    task_free      (void) ;
```

DESCRIPTION

The *task_init* function creates and initializes the task table to contain the dummy (zero) entry and an entry for the capsule's initial task which uses the main stack. The *task_setup* function sets up the initial stack dump for task 1 by copying the dump for the dummy entry and saving the stack front and stack frame pointers.

The *task_make* function creates and initializes extra tasks.

The *task_select* function makes the specified task table entry the current one by setting the global variables `taskIndex` to its index and `taskPtr` to its address.

The *task_wake* function puts the specified task onto the end of the queue of tasks that are eligible for execution.

The *task_setThread* function assigns the current thread to the current task.

The *task_dump* function dumps the registers in the current task's dump area. The exit from the initial call of *task_dump* will return a zero result; subsequent exits from *task_dump* caused by *task_switch* will return a non-zero result.

The *task_switch* function dequeues the first task that is eligible for execution, selects it and switches to its stack, resuming execution by restoring the registers and simulating another exit with a non-zero result from the last *task_dump* executed on that stack.

The *task_dispatch* function dequeues the first task from the pool of free tasks and starts the execution of its assigned thread by selecting it and its thread, re-initializing its register dump by copying the dump from the dummy task, resetting the stack front and stack frame registers in the dump and then simulating a *task_switch* to the original *task_dump* in *main*, which has a *thread_execute* call following its non-zero return branch.

The *task_free* function puts the current task into the pool of free tasks and then checks if there is any more work for the capsule to do.

If there are no active tasks, threads queued for execution or open sockets, then the capsule is terminated.

FILES

ERRORS

SEE ALSO

NUCLEUS (XI), SCHEDULE (XI), STACK (XI), THREAD (XI).

USAGE

Task_init and *task_setup* are only called by the *main* function in the nucleus. They must be separated only by a call of *task_dump*. This is done so that the register dump can be taken at the top level. *Task_make* is only used to implement *nucleus_tasks*. *Task_dump* is only used in *main* and *schedule*. *Task_dispatch* and *task_switch* are only used in *schedule*.

FUTURE DIRECTIONS

CHANGE HISTORY

Date	Release	Pages	Changes
16/12/88	Version 2.0		initial release

NAME

Thread - multi-threading support.

PURPOSE

Provides the thread management functions required to support the execution of more than one thread at a time. A thread identifies an independent execution path which can be executed concurrently with other threads. Threads are created as a result of *Fork* and *Spawn* instructions or incoming invocations (*Dispatch*). There is no limit imposed on the number of threads that can be created in a capsule, but the number of active threads allowed on each socket is specified by the application when the socket is created.

SYNOPSIS

```
#include "capsule.h"
#include "thread.h"

void      thread_init       (void) ;

void      thread_select     (ThreadId    index) ;

Status    thread_invalid    (ThreadId    thread) ;

ThreadId  thread_parent     (ThreadId    thread) ;

ThreadSt  thread_state      (ThreadId    thread) ;

void      thread_setState   (ThreadSt    new) ;

void      thread_setSession (void) ;

void      thread_queue      (ThreadId   *head,
                             ThreadId    thread) ;

ThreadId  thread_dequeue    (ThreadId   *head) ;

void      thread_wake       (ThreadId    thread) ;

ThreadId  thread_dispatch   (ThreadId    parent,
                             Dispatch   *dispatcher,
                             ChannelId   socket,
                             BufferLink  buffer,
                             void       (*epilogue)()) ;

void      thread_execute    (void) ;

void      thread_nest       (ThreadId    child) ;

void      thread_free       (void) ;
```

DESCRIPTION

The *thread_init* function creates and initializes the thread table to contain the dummy (zero) entry and an entry for the capsule's initial thread.

The *thread_select* function makes the specified thread table entry the current one by setting the global variables `threadIndex` to its index and `threadPtr` to its address.

The *thread_invalid* function checks if the thread index is out of range.

The *thread_parent* function returns the parent of the specified thread.

The *thread_state* function returns the state of the specified thread.

The *thread_setState* function changes the current thread's state.

The *thread_setSession* function assigns the current session to the current thread.

The *thread_queue* function puts the specified thread onto the end of the specified queue.

The *thread_dequeue* function dequeues and returns the first thread from the front of the specified queue. If the queue was empty then a zero index is returned.

The *thread_wake* function wakes up the task that the specified thread is assigned to.

The *thread_dispatch* function creates a new thread, initializes it and queues it waiting for a free task to execute it. The `socket` and `buffer` arguments are passed to the `dispatcher` function and the `epilogue` function is called after the `dispatcher` has returned in order to perform any required termination operations, such as waiting for its parent to join it or sending a reply.

The *thread_execute* function executes the current thread on the current task by calling its dispatcher and epilogue functions.

The *thread_nest* function takes the child thread off the thread queue, executes it on the current task and then returns to the parent thread.

The *thread_free* function puts the specified thread into the pool of free threads.

FILES

ERRORS

SEE ALSO

INSTRUCT (XI), SCHEDULE (XI), TASK (XI).

USAGE

FUTURE DIRECTIONS

CHANGE HISTORY

Date	Release	Pages	Changes
16/12/88	Version 2.0		initial release

Part XI
Chapter 12
Installation

This chapter describes the installation procedures for systems to which the ANSA Testbench has been ported.

12.1 UNIX

12.1.1 The testbench distribution tape

The testbench software is delivered on a `tar` format cartridge tape and has been checked on a SUN 3/110 running SunOS[†] and a Hewlett Packard 9000 model 350 running HP-UX[‡].

The tape contains C language sources for the interpreter, the nucleus, REX, the trader, the stub compiler, and the distributed processing language pre-processor. An installation script is provided to enable these files to be unpacked and compiled. Some test programs are also provided to ensure that the installation was successful.

To run the script and do a successful installation of the testbench, the following programs need to be available:

```
/bin/sh, /bin/csh (Bourne and C shells), ar, as, awk, cat, cc,
cp, date, echo, ln, make, mkdir, mv, pwd, ranlib, rm, tee, yacc
```

The testbench also requires the UDP protocol, and/or the TCP protocol, and/or named pipes in order for REX to work.

12.1.2 Preparing to read the tape

In order to successfully complete the installation process you must:

1) decide where to place the testbench source, include and library files
2) determine which host on your network will be running the trader
3) determine your system type (currently one of **sun**, **hpux**, or **m6000**)
4) determine which MPS modules you wish supported

[†]SunOS is a trademark of Sun Microsystems
[‡]HP-UX is a trademark of Hewlett-Packard Company

5) determine the device which contains the installation tape.

These options are discussed below.

12.1.2.1 The testbench logical root directory

This directory will be used to contain all the testbench sub-directories and used as a source of libraries and header files by application programmers and developers. We recommend that you use:

```
/usr/local
```

Your chosen directory must be your current working directory when you read the distribution tape. Whichever directory you choose must be readable by all users who need to access the testbench files and will have the following sub-directories created if they do not already exist:

```
src, lib, include, etc, bin
```

12.1.2.2 The trader host

The host machine chosen to support the trading service must be decided before the installation process begins. The host chosen must be accessible by all machines running the testbench and must be suitable for supporting a pivotal network service, i.e. it should be a reasonably stable machine which is not unduly subject to crashes or reboots. You must provide the host's name during the installation process and it must be the same for all machines on a given network.

N.B. the installation script is unable to detect if you provide a bogus host name. The name given must be a legitimate host name which matches the entry in /etc/hosts exactly, case is significant. If you provide an unsuitable name the installation process will complete but the resulting library will be useless.

12.1.2.3 The host system

The testbench must be built afresh for each system type. Currently the installation should work without modification for SUN and HP-UX systems. The system type chosen is used to generate appropriate `Makefile's` for the build process (the difference is in the libraries needed to compile the testbench source files).

12.1.2.4 The MPS modules

This release of the testbench supports REX's use of multiple MPS modules for remote interactions. Optimized calls between capsules on the same host can be achieved if the IPC MPS is included. MPS's based upon UDP and TCP are also provided for conveying messages between capsules on different hosts. While both UDP and TCP can be included in the capsules, the current system will always choose UDP preferentially. As a result, the default set of MPS's is "IPC UDP".

Note that if one system is built with UDP as the sole remote MPS and another is built with TCP as the sole remote MPS then capsules on these systems will not be able to interact.

12.1.3 Installing the testbench

N.B. all example commands shown in this appendix assume that the user is running in the C shell. Set your current working directory to your chosen testbench logical root directory, load the tape in a cartridge drive and type the following:

```
tar xvbf 1 /dev/yourTapeDevice
```

Assuming that you have write access to this directory and its sub-directories, `tar` will now unpack all the installation files. At the end of the process (a couple of minutes at most) you will find the installation script (`InstallANSA`) in the directory. You run this script (and capture the output in a logfile) by typing:

```
InstallANSA |& tee Install.log
```

The script begins by asking five questions. You may abort the installation by typing the seven characters `*ABORT*` in response to any of the first four questions. The script provides a lot of detail about the actions you are being asked to undertake.

First you are asked to provide the testbench logical root directory (you must type an absolute pathname, e.g. `/foo/bar`), the default is your current working directory.

Second you are asked to enter the trader host name. This name must exist in `/etc/hosts` and case is significant. The default is the current hostname.

Third, you are asked to provide the list of MPS modules desired for this system - the list is simply a blank-separated sequence of the words IPC, UDP, and TCP. The default list is: IPC UDP.

Fourth you are asked to enter the system type (one of `sun`, `hpux`, or `m6000`).

Finally, you are asked if you wish to delete object and other intermediate files generated during the installation. The default action is **not** to delete the intermediate files.

The script takes 10 to 15 minutes to run, much of which is spent compiling. In the event of errors, use the log file to determine what the problem is.

12.1.3.1 Non-standard port numbers

The testbench installation assumes two UDP/TCP port numbers that are fixed. One of these (**port number 1100**) is for the trader and should not be changed without good reason, the other (**port number 1060**) is for a test program used for pre-trader checkout and may be changed if necessary. To change either of these values after the installation, proceed as follows - it is assumed that the starting directory is the testbench logical root denoted as **$R**:

To change the port number for the test program:

```
cd $R/src/ansa/test
#    edit both callServer.c and callStub.h to change
#    "#define callCapsule 1060" to your chosen value
make
cd $R
```

Changing the port number for the trader should not be undertaken lightly and involves a rebuild of the trader AND modifications to a file on each of the other hosts in the system. If it must be done, proceed as follows:

```
cd $R/include/ansa/capsule
#    edit options.h to change
#    "#define TRADER_UDP_PORT 1100"
#    "#define TRADER_TCP_PORT 1100"
#    to your chosen value
cd $R/src/ansa/trader
make trader; make tinstall
```

Now you must create the file $R/etc/ansa/traderaddress on each non-trader host. This file contains a single line; this line has the same format as the TRADER_ADDRESS symbol in $R/include/ansa/capsule/options.h, minus the quotation marks. For example, if the options.h file on a particular host had the following definition:

```
#define TRADER_ADDRESS "1:1.2.3.4:1100:2"
```

and you have just changed TRADER_UDP_PORT to 4321, then $R/etc/ansa/traderaddress should contain the following line:

```
1:1.2.3.4:4321:2
```

12.1.4 Installation tests

The installation script builds a number of test programs; the most basic one tests out the interpreter and REX in isolation. To run this do the following (from the test bench logical root directory)

```
src/ansa/test/callserver &
src/ansa/test/callclient -i100
```

This will repeatedly issue REX calls and collect the replies reporting the elapsed time and buffer sizes every 100 calls (about twice a second on a SUN). You can change the number of iterations by altering the number in the `callclient` line and can even nominate a different host, e.g.

```
src/ansa/test/callclient -i1000 SomeOtherHost
```

By default, the callclient test runs endlessly. Use the

```
-p<# of passes>
```

option to cause it to terminate after <# of passes>

By default, callclient uses UDP for its interactions. Use the `-mipc` flag to cause it to use the IPC MPS (the server had better be on the same host) or `-mtcp` to cause it to use TCP. Obviously, if you specify an MPS for which support was not included during the build, you will get a bind failure.

The `callserver` must be terminated by using the `kill` command.

12.1.4.1 Starting the trader and exercising it

Before any of the other test programs can be started, it is necessary that the trader be started on the host which was nominated during the build procedure. Assuming that the current working directory is the testbench logical root, the following command starts the trader:

```
etc/ansa/trader >&etc/ansa/trader.log &
```

Depending upon the MPS support selected, you should see one (or more) of the following (after invoking `netstat -a` and `ls /usr/tmp`):

- UDP port 1100 exists
- TCP port 1100 has a LISTEN outstanding
- the fifo `/usr/tmp/ansa.1` exists

Three tests which use the trader can now be performed.

1. When the trader starts up, it registers itself as the exporter of two additional services: *TrCtxt* and *TrType*. The following commands should yield information on these two service offers:

   ```
   src/ansa/trader/trlook TrCtxt /
   src/ansa/trader/trlook TrType /
   ```

2. The `trsearch` command causes information on all offers matching the type specified in the nominated trading scope to be returned.

```
        src/ansa/trader/trsearch ansa /
```

causes information about all known offers to be displayed (since all types are sub-types of type ansa).

3. Next launch a number of Lookup requests at the trader, keeping track of the elapsed time. This is done as follows:

```
        src/ansa/trader/trtest
```

This program will request the offer information on the *TrCtxt* offer 1000 times, and prints summary statistics of the performance upon completion. For those who truly want to exercise the trader, trtest takes an optional argument which is the number of times the 1000 requests should be attempted.

When you are satisfied that the trader works correctly, the startup command line should be added to /etc/rc; this guarantees that the trader is started everytime the system boots up. Obviously this should follow any commands which are necessary to enable the network!

12.1.4.2 The Echo service

Now that the trader is working, all other services are provided by normal capsules. The simplest of the example services provided with the distribution tape is an *Echo* service. To start the *Echo* server, issue the following command line:

```
    src/ansa/Echo/server >&src/ansa/Echo/server.log &
```

The following command line shows that the server actually posted its offer to the trader:

```
    src/ansa/trader/trlook Echo /
```

Three client programs are provided which use the *Echo* service:

1. client simply reads each line from its standard input, invokes the Echo function of the *Echo* server, and prints the echoed line on the standard output.

2. techo invokes the Echo function of the *Echo* server with a fixed size buffer of characters a given number of times. Upon completion, summary statistics about the transfer rate are displayed.

3. tsink invokes the Sink function of the *Echo* server with a fixed size buffer of characters a given number of times. Upon completion, summary statistics about the transfer rate are displayed.

The following command lines will exercise the *Echo* server started previously:

```
src/ansa/Echo/client <InstallANSA | diff - InstallANSA
src/ansa/Echo/techo -b1 -n1000
src/ansa/Echo/techo -b10 -n1000
src/ansa/Echo/techo -b100 -n1000
src/ansa/Echo/techo -b1000 -n1000
src/ansa/Echo/tsink -b1 -n1000
src/ansa/Echo/tsink -b10 -n1000
src/ansa/Echo/tsink -b100 -n1000
src/ansa/Echo/tsink -b1000 -n1000
```

For both techo and tsink, you may optionally specify the name of another host upon which the service is active with a command of the form:

```
src/ansa/Echo/techo -b1 -n1000 OtherHostName
```

12.1.4.3 The Sample service

The *Sample* service is provided to illustrate an interface specification with multiple operations. One of the operations returns multiple results, another takes no arguments, while the third returns no results. Starting the server requires the command:

```
src/ansa/Sample/server >&src/ansa/Sample/server.log &
```

The client simply invokes each operation 1000 times, printing the results from the two operations which return results on the standard output. After checking that the service offer was registered with the following command:

```
src/ansa/trader/trlook Sample /
```

the following command is suggested for exercising the client:

```
src/ansa/Sample/client | wc
```

12.1.4.4 The Netinfo service

The *Netinfo* service is provided as an example of a non-trivial application of the ANSA testbench. Each UNIX system which supports the socket abstraction for interfacing to the TCP/UDP/IP family of protocols provides a set of library routines which permit the programmer to determine information about hosts, networks, protocols, and services. The usual implementation of these library routines is to read the appropriate information from a set of distinguished files which are stored in /etc, returning to the caller a pointer to some statically stored data structures which have been loaded with the appropriate information. One complication of this particular implementation style is the necessity to keep multiple copies of the distinguished files consistent. The *Netinfo* service is a simple implementation of a name service which eliminates this consistency requirement.

Netinfo provides an RPC interface to the routines gethostbyname(), getnetbyname(), getprotobyname(), and getservbyname(). The return results from the remote procedure calls are discriminated unions (specified as CHOICE's) in which the full data structure is returned only if the lookup on the server host was successful. The specification and the client and server programs can be consulted for more information. To start up the server, use the following command:

```
src/ansa/Netinfo/server >&src/ansa/Netinfo/server.log &
```

Four client programs are provided, each one exercising a different operation in the interface. The following command lines should result in useful output from the client programs (note the backquotes in the second line):

```
src/ansa/trader/trlook Netinfo /
src/ansa/Netinfo/hclient `hostname`
src/ansa/Netinfo/nclient arpa
src/ansa/Netinfo/pclient udp
src/ansa/Netinfo/sclient tcp ftp
```

12.1.4.5 The test1 service

This service is exactly the one described in section 3.4 of this manual. To start up the server, use the following command:

```
src/ansa/test1/server >&src/ansa/test1/server.log &
```

The client program can be invoked as:

```
src/ansa/test1/client 1 2 3 4 5
```

12.1.5 Support and further developments

The ANSA testbench software is being made available in source form to enable recipients to experiment with it and port it. It is not guaranteed to be bug free although it has been extensively tested with all the facilities which the ANSA testbench implementation team have available.

The ANSA testbench is not supported as a warrantied software product and any or all parts of the testbench may change in future releases. ANSA will continue to evolve and these changes will be reflected in the design of the testbench.

If you experience any trouble with porting or using the testbench please contact the ANSA team. The ANSA team wish to be informed of all problems and difficulties encountered and within the constraints of the project, will give advice on how to deal with them.

The ANSA team will be pleased to receive extensions, improvements, new ports or additional services for incorporation in future releases of the testbench. The most convenient form for supplying code changes to us is as conditionally compilable sources in one of the following machine readable forms, listed in order of preference:

HP ¼" cartridge tape	- tar format
SUN ¼" cartridge tape	- tar format
electronic mail	- see cover sheet for addresses
5¼" floppy	- PC-DOS format ASCII file
½" magnetic tape	- tar format

ANSA cannot accept any code with IPR or license restrictions or fees. The originators copyright will be attributed on all contributed code.

12.2 MSDOS

The ANSA testbench has been successfully ported to the IBM[†] PC running MSDOS[‡]. This appendix describes the requisite hardware and software, the build procedure, test procedures, and limitations to the PC port.

12.2.1 Requisite hardware and software

Hardware:

- IBM PC (or PC clone) with 640K RAM
- high density 5¼" floppy drive (1.2 Mbyte) and hard disk with at least 3.5Mbytes of free space
- network interface (see below)

Software:

- MSDOS v3.0 or later
- Microsoft C compiler, v4.0 or later
- PC/TCP v2.02 for each target system
- PC/TCP Development Kit v2.02 for a system upon which the testbench programs will be built

The choice of network hardware is dictated by those interfaces supported by the PC/TCP package. They are shown in Table 12-1 below.

The network software and documentation may be obtained by contacting

[†]IBM is a trademark of International Business Machines
[‡]MSDOS is a trademark of Microsoft, Inc.

Table 12-1: PC/TCP supported network interfaces

Part No.	Interface	Part No.	Interface
PC-2XX	Site License	PC-211	Scope DDN Microgateway
PC-201	ProNET-10 p1300	PC-212	Excelan EXOS-205/205T
PC-202	3COM 3C500/3C501	PC-213	National Semiconductor DP839EB
PC-203	MICOM-Interlan NI5010	PC-215	Banyan Vines Ethernet software
PC-204	3COM 3C505	PC-218	MICOM-Interlan NI5210
PC-205	Slip	PC-219	IBM Token Ring adapter
PC-207	BICC 4117	PC-220	3COM IE-6 (3C503)
PC-208	ProNET-4 p1340/p1344	PC-221	Western Digital StarCard PLUS or EtherCard PLUS
PC-209	Ungermann-Bass NIC	PC-222	Proteon interrupt driver
PC-210	Generic Ethernet driver	PC-224	3COM IE-MC (3C523)

FTP Software, Inc.
P.O. Box 150
Kendall Square Branch
Boston, MA 02142
UNITED STATES OF AMERICA

12.2.2 Preparation for installation

The following build procedure has worked successfully on the PC system at ANSA. It assumes that you have booted MSDOS from the C: disk, and that you have a high density $5\frac{1}{4}$" floppy drive (1.2 MByte) for device A:. Any other hardware configuration will necessitate major changes to all batch files and makefiles.

When installing the C compiler and the PC/TCP Development Kit, be sure to install the Large Memory Model libraries, as the testbench build procedure references these.

The build of the system at ANSA occurred under the following environment:

```
C:\> SET
COMSPEC=C:\COMMAND.COM
PATH=.;C:\PCTCP;C:\BIN;C:\UTILS
TEMP=C:\TEMP
PROMPT=$p$g
INCLUDE=C:\DEVKIT\INCLUDE;C:\INCLUDE
LIB=C:\DEVKIT\NETMSC4.0;C:\LIB
TMP=C:\
C:\>
```

Of these settings, four are critical to the successful building of the software:

- having \BIN in the search path is necessary for the Microsoft C utilities to be successfully executed; the \UTILS directory is where the standard MSDOS utilities (like COPY) are found on our system
- the value of INCLUDE guarantees that the C compiler will search for `#include` files in the include directory provided with the PC/TCP development kit and the include directories provided with the Microsoft C compiler
- the value for LIB guarantees that the requisite libraries needed to build the applications will be found
- the value for TMP is needed by the C compiler for building temporary files

The parsers for the preprocessor and stub compiler are provided in C form; the yacc grammars have already been processed using a public-domain parser generator named `bison`[†]. These parsers include `<stdio.h>`. If the user code provided with the grammar also includes `<stdio.h>`, a compile error will occur. Since both of these grammars will exhibit the above error, it is necessary that the distributed `<stdio.h>` be modified as follows:

```
#ifndef STDIO
#define STDIO
<distributed version of <stdio.h>>
#endif
```

12.2.3 Installation procedure

To build the testbench, execute the following steps:

1. Load the floppy disk into the A: drive
2. `CD \`
3. `COPY A:INSTALL.BAT .`
4. `INSTALL`
5. `CD \ANSA`
6. `ANSABLD <TradingHostName> [y/n]`

You must provide the name of the trading host as the first parameter of the invocation to ANSABLD. The optional second parameter determines whether intermediate files are deleted as the procedure proceeds. The completed system consumes 3.4 Mbytes of disk space if the intermediate files are not deleted, 2.6 Mbytes if they are.

[†]bison is publicly-available from the Free Software Foundation (GNU)

12.2.4 Installation tests

The installation script builds a number of test programs. All of the examples below assume that your current working directory is \ANSA and that you have started the trader and relevant server program on a UNIX host with which the PC can communicate. Note that the PC implementation uses UDP only for conveying REX packets, and assumes the standard port numbers as described in section A.3 above.

12.2.4.1 REX and interpreter test

The most basic test exercises the interpreter and REX in isolation. To run this, do the following:

```
src\test\callclient -i100 OtherHost
```

This will repeatedly issue REX calls and collect the replies reporting the elapsed time and buffer sizes every 100 calls; the calls are sent to the server running on OtherHost. You can change the number of iterations by altering the number in the -i flag above.

By default, the callclient test runs endlessly. Use the

```
-p<# of passes>
```

option to cause it to terminate after <# of passes>.

12.2.4.2 Exercising the trader

Three tests which use the trader can be performed.

1. When the trader starts up, it registers itself as the exporter of two additional services: *TrCtxt* and *TrType*. The following commands should yield information on these two service offers:

   ```
   src\trader\trlook TrCtxt /
   src\trader\trlook TrType /
   ```

2. The trsearch command causes information on all offers matching the type specified in the nominated trading scope to be returned.

   ```
   src\trader\trsearch ansa /
   ```

   will cause information about all known offers to be displayed (since all types are sub-types of type ansa).

3. Next launch a number of Lookup requests at the trader, keeping track of the elapsed time. This is done as follows:

   ```
   src\trader\trtest
   ```

   This program will request the offer information on the *TrCtxt* offer 1000 times, and prints summary statistics of the performance upon completion. For those who truly want to exercise the trader, trtest takes an optional argument which is the number of times the 1000 requests should be attempted.

12.2.4.3 The Echo service

The simplest of the example services provided with the distribution is an *Echo* service. This service is provided with three client programs:

`client`	simply reads each line from its standard input, invokes the Echo function of the *Echo* server, and prints the echoed line on the standard output.
`techo`	invokes the Echo function of the *Echo* server with a fixed size buffer of characters a given number of times. Upon completion, summary statistics about the transfer rate are displayed.
`tsink`	invokes the Sink function of the *Echo* server with a fixed size buffer of characters a given number of times. Upon completion, summary statistics about the transfer rate are displayed.

The following command lines will exercise the *Echo* server started previously:

```
src\Echo\client <ANSABLD.BAT  >t.out
fc t.out ANSABLD.BAT
del t.out
src\Echo\techo -b1 -n100
src\Echo\techo -b10 -n100
src\Echo\techo -b100 -n100
src\Echo\techo -b1000 -n100
src\Echo\tsink -b1 -n100
src\Echo\tsink -b10 -n100
src\Echo\tsink -b100 -n100
src\Echo\tsink -b1000 -n100
```

For both `techo` and `tsink`, you may optionally specify the name of a specific host upon which the service is active with a command of the form:

```
src\Echo\techo -b1 -n100 HostName
```

12.2.4.4 The Sample service

The *Sample* service is provided to illustrate an interface specification with multiple operations. One of the operations returns multiple results, another takes no arguments, while the third returns no results. The client simply invokes each operation 1000 times, printing the results from the two operations which return results on the standard output. The following commands are suggested for exercising the client:

```
src\Sample\client >t.out
more <t.out
del t.out
```

12.2.4.5 The Netinfo service

The *Netinfo* service is provided as an example of a non-trivial application of the ANSA testbench. Each UNIX system which supports the socket abstraction for interfacing to the TCP/UDP/IP family of protocols provides a set of library routines which permit the programmer to determine information about hosts, networks, protocols, and services. The usual implementation of these library routines is to read the appropriate information from a set of distinguished files which are stored in /etc, returning to the caller a pointer to some volatile data structures which have been loaded with the appropriate information. One complication of this particular implementation style is the necessity to keep multiple copies of the distinguished files consistent. The *Netinfo* service is a simple implementation of a name service which eliminates this consistency requirement.

Netinfo provides an RPC interface to the routines `gethostbyname()`, `getnetbyname()`, `getprotobyname()`, and `getservbyname()`. The return results from the remote procedure calls are discriminated unions (specified as `CHOICE`'s) in which the full data structure is returned only if the lookup on the server host was successful. The specification and the client programs can be consulted for more information. Four client programs are provided, each one exercising a different operation in the interface. The following command lines should result in useful output from the client programs:

```
src\Netinfo\hclient SomeHostName
src\Netinfo\nclient arpa
src\Netinfo\pclient udp
src\Netinfo\sclient tcp ftp
```

12.2.4.6 The test1 service

This service is exactly the one described in section 12.1.3.4 of this manual. The client program can be invoked as:

```
src\test1\client 1 2 3 4 5
```

12.2.5 PC limitations

The PC makes a reasonable ANSA client machine. Due to the masking of ^C interrupts when a program is using the network software, it does not make a particularly good server machine. You will often have to reboot the system to terminate a server program, with the net effect of leaving a stale export offer in the trader's database. This stale offer can be removed the next time the system is booted if the following command is added to the AUTOEXEC.BAT file after the network software has been started:

```
C:\ANSA\SRC\TRADER\DOSPURGE
```

Because of the hardware architecture of the IBM PC, fragmented buffers are limited to a maximum size of 64Kbytes. Attempts to allocate larger buffers when sending or receiving will fail. This limit is imposed because there is no portable or standard way of addressing larger regions of contiguous memory on an IBM PC. Users of the PC testbench who have a solution to this problem which is portable - i.e. will not cause inefficiency or obscure the source code for other users - and is also standard i.e. will work on many ANSI compilers, should submit it to ANSA for consideration as a part of a future release.

This port was completed to give collaborators an opportunity to build ANSA clients in the PC environment. There may be some incipient bugs remaining in the PC release, and we will attempt to fix any brought to our attention if the bug report is accompanied by test programs which easily reproduce the problem.