AServer Programmers' Guide

May 1995

This product incorporates innovative techniques which were developed with support from the European Commission under the ESPRIT Projects:

- P2701 PUMA (Parallel Universal Message-passing Architectures)
- P5404 GPMIMD (General Purpose Multiple Instruction Multiple Data Machines).
- P7250 TMP (Transputer Macrocell Project).
- P7267 OMI/STANDARDS.
- P6290 HAMLET (High Performance Computing for Industrial Applications)

Document Number: 72 TDS 403 02

Contents

1 Introduction

The AServer (Asynchronous Server) system is a high performance interface system which allows multiple processes on a target device to communicate via a hardware serial link with multiple processes on some external device. The AServer software acts as a standard interface which is independent of the hardware used.

A simple example is shown in Figure 1.1, in which the external device is the host. A more complex example is shown in Figure 1.2. The AServer system supports the new generation of INMOS development tools such as the INQUEST interactive debugger.

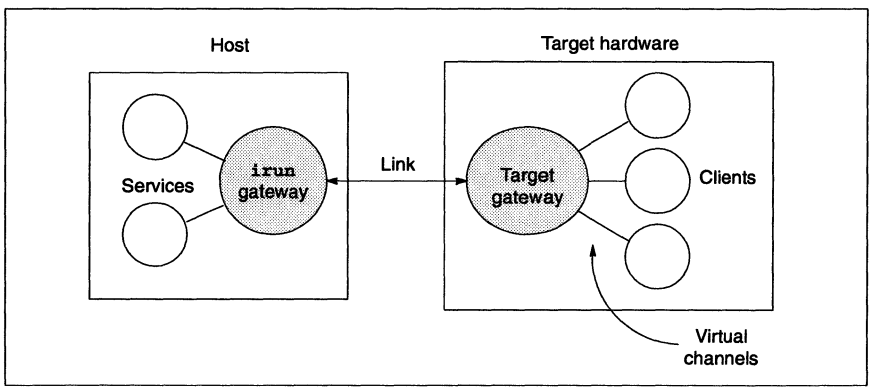

Figure 1.1 A simple software host-target interface

The AServer is a collection of programs, interface libraries and protocols that together create a system to enable applications running on target hardware to access external services in a way that is consistent, extensible and open. The software elements provided are:

- a target gateway which runs on the target (see Chapter 3);
- an `irun` gateway which runs on the host (see Chapter 2);
- an `iserver` service which runs on the host (see Chapter 4);
- an `iserver` converter which runs on the target (see Chapter 4);
- a library of interface routines for use by client and service processes (see Chapter 6);
- simple example services.

1.1 AServer features

This type of architecture offers a number of advantages:

1 The AServer handles multiple services.

A number of services may be available on one device. These are handled by a single gateway for a single link to the target hardware. For example, new services may be added without modifying a standard server.

2 The AServer handles multiple clients.

Any process on any processor in the target hardware may open a connection to any service. The process opening the connection is called the client. Using virtual channels, a number of clients may be directly connected to the appropriate gateway. Several clients may access the same service. The gateway will automatically start new services as they are requested.

3 Services are easy to extend.

The AServer enables users to extend the set of services that are available to a user's application. The AServer provides the core technology to allow users to create new services by providing new processes.

For example, the `iserver` provides terminal i/o, file access and system services. The AServer allows this to be expanded by adding new service processes tailored by the user for the particular application. The new services stand alone, so that the `iserver` code does not need to be modified.

4 AServer communications can be fast and efficient.

The communications over the link between gateways use mega-packets, which make efficient use of the available bandwidth (see Chapter 3).

Messages between the client and the service are divided into packets of up to 1 kbyte. The packets are bundled into mega-packets to send over the hardware serial link. Packets from different clients and services can be interleaved to reduce latency.

5 AServer communications are independent of hardware.

When an AServer connection has been established the process can send data messages of arbitrary length to the service it is connected to, receive data messages of arbitrary length and disconnect from the service. The gateways are responsible for building and dividing mega-packets and complying with hardware protocols.

1.2 Gateways

A pair of gateway processes, one at each end of a link interface, acts like a pair of telephone exchanges. Each gateway multiplexes outgoing communications to the link and demultiplexes incoming communications from the link.

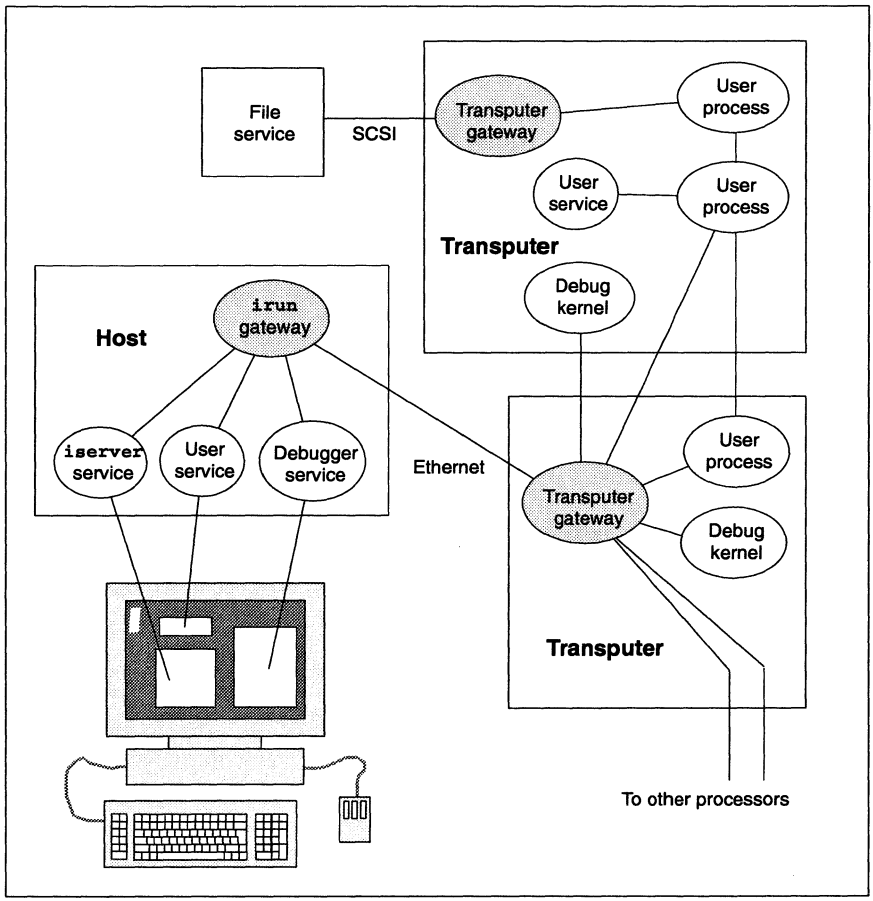

Figure 1.2 Example of AServer uses

Figure 1.2 shows an example of a transputer network that has a link to a host (in this case using Ethernet) and a link to a file service (in this case using SCSI). The links from the transputer network are interfaced via gateways.

A gateway has the role of routing AServer communications to the appropriate place, i.e. to a process or on to another gateway. Optimized gateways can be written to make efficient use of the particular link mechanism.

Processes running on the target can communicate with gateways using virtual channels. The virtual channels can be provided in software, using the new generation of INMOS configurers, or they can be provided by hardware, as in IMS T9000 networks

with asynchronous packet switches. On the host machine the communication to a gateway is provided by a host inter-process communication capability.

The gateway process which runs on a target is called the *target gateway* and is provided as part of the AServer software. The gateway process which runs on the host is called a *host gateway*. The standard host gateway provided with the AServer is called `irun`. The `irun` gateway provides target hardware booting and subsystem control similar to that found in the `iserver`. This enables `irun` to be used to boot the target and subsequently become a gateway.

1.3 Clients and services

A process using the AServer system may be acting as a client or as a service, or possibly both. A client process makes the initial request for a service. The AServer attempts to make a connection to a suitable service process and then handles the communications until the connection is closed by the client.

A single process may be able to service a number of clients. A list of services, how to start them and the maximum number of connections for each process is held in an *AServer database* file. The AServer keeps a count of how many clients are connected to each process. When a new service is requested, a connection will be made to an existing service process if one exists with spare connections.

When a client requires use of a remote service, it issues a connect request to the local gateway giving the name of the service to connect to. The connect request is sent to the remote gateway which decides how the request can be serviced. If a connect request is received by an `irun` gateway, the gateway will either:

1 find an existing process that can provide the service and can handle more connections, or

2 create a process to provide that service.

The process that is created handles the connect request and sends a connect reply back to the initiating process via the gateway. The reply contains a connection identifier that is used in subsequent communications between the client and the service that has been created.

A client may be connected to a local service without passing through gateways. This may be desirable to make the client or service more portable. In this case the service process cannot be created dynamically but both processes must be configured into the same program.

Clients and services may in principle be anywhere in the system. In this implementation, only the `irun` gateway can start services. This means that clients on a remote device cannot request services on target hardware.

Services provided by INMOS include:

- an `iserver` service (see Chapter 4)
- an INQUEST debugger
- an example print server
- an example file server

Examples of possible services which may be provided by the user include:

- a non-`iserver` host window
- a fast file server
- a graphics window server

1.4 Protocols

The gateways and processes described in sections 1.2 and 1.3 communicate by means of open standard protocols.

The gateways communicate between each other using a mega-packet protocol. This is designed for use over serial link interfaces to external hardware and uses large packets for high performance. The two protocols are illustrated in Figure 1.3.

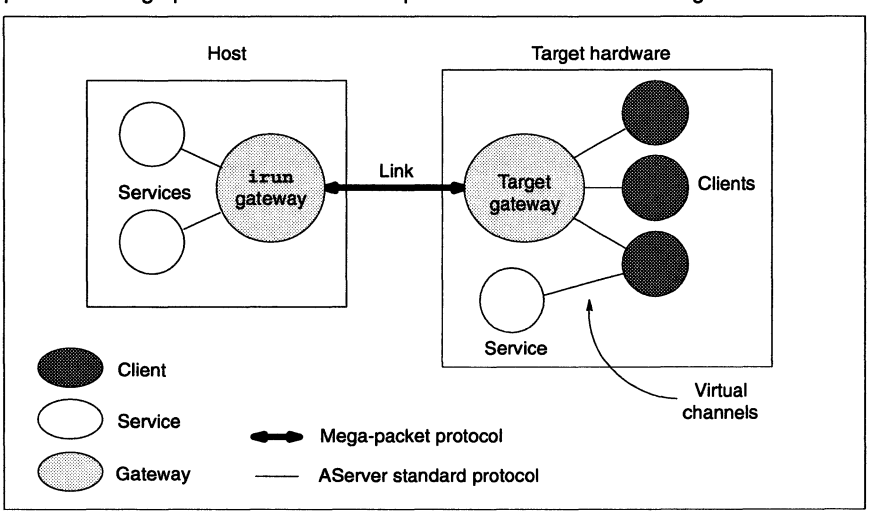

Figure 1.3 AServer protocols

Processes communicate with each other and with gateways using the AServer standard protocol, as described in Appendix B. The AServer protocol provides a standard

portable interface so that processes do not need to know where the process at the other end of the communication is located. The main use of the AServer protocol is for processes to communicate via gateways and links, but clients and services on the same target hardware may communicate using the AServer protocol to aid portability.

The maximum size of packets used by this protocol is 1 kbyte, for reasonable efficiency without unduly affecting latencies. For ease of use, this protocol is supported by the AServer libraries, as described in Chapter 6.

1.5 Access points

Each AServer process has one or more *access points*. An access point is a host-independent means of accessing other AServer processes. Each access point is connected either directly to another process access point or to a gateway and uses the AServer standard protocol. On a target, an access point consists of a pair of virtual channels, one input and one output.

The access points of one process are arranged into an array and the *access point number* is the index into the array. The access point number is used as a parameter to some AServer library functions.

The access points of a process must be initialized before use, using the library routine `as_apstart`, as described in Chapter 6.

1.6 Using the AServer with occam

This manual generally assumes that any customized AServer processes running on the target will be written in ANSI C. The AServer libraries are provided as ANSI C libraries. It is recommended that occam applications either use the `iserver` protocol or have ANSI C interface processes.

occam processes may call the AServer libraries by adding a shell to each library function to make it look like an occam procedure, as described in the *occam 2 Toolset User Guide*. An example of this is provided in the examples directory and described in Chapter 5.

2 irun

This chapter describes the `irun` program, its purpose, what it does and how the user controls it.

`irun` runs on the host and is the main control program for the AServer, performing a multiplexing and process creation role. It is a gateway process running on the host. Unlike the gateway running on the target hardware, `irun` starts host services dynamically at the request of the target hardware, using information in an *AServer database* file to determine how to execute what in order to provide the service. The user controls `irun` through the values of environment variables, through `irun`'s command line interface and through the AServer database file.

2.1 Running `irun`

2.1.1 Environment variables

The AServer uses up to three environment variables, which should have been set up during installation. On Suns these may be set using the `setenv` command, for example:

```
setenv TRANSPUTER target
```

On PCs, the environment variables must either be set globally from DOS (i.e. *not* from a DOS Window) or be set in a Windows environment file. The `irun` chapter of the *Toolset Reference Manual* describes how this is done.

When using Windows tools (i.e. `irun` or an INQUEST tool), variable values set in a Windows environment file will override an environment variable of the same name set from DOS.

In the rest of this document, the term *environment variable* will mean either a true environment variable or a variable set in a Windows environment file, whichever is appropriate.

The three environment variables used by the AServer are as follows:

- The `ISEARCH` variable specifies a path used by the INMOS tools to find files. It will normally name the toolset library directory, any include file directories and any user directories as required. If you are using toolset software then the toolset libraries directory will also have to be on the `ISEARCH` path. See your *Toolset Delivery Manual* for more information.
- The `ASERVDB` variable points to the AServer database file. This variable need not be specified if the AServer database file is on the `ISEARCH` path. The AServer database specifies resources such as user services, hardware interfaces and INQUEST, which may be requested by the application.

- The **TRANSPUTER** variable specifies which target in the AServer database is to be used. **irun** uses the **TRANSPUTER** environment variable to find the hardware serial link to serve and to which bootable files are to be copied. The **TRANSPUTER** variable may be overridden by the **-SL** option on the command line.

2.1.2 Starting AServer applications

On X-Windows systems, AServer applications can be started by a command line. On Microsoft Windows systems, AServer applications are run in the same way as other Windows applications.

The name of the command that starts AServer applications is **irun**.

▶ **irun** *bootable_file* {*options*}

where: *bootable_file* is the name of the application code bootable file,

options is a list of one or more options. A full list of options is given in your *Toolset Reference Manual.*

Running **irun** with no parameters causes **irun** to display its version number, build date and brief help information.

Each of **irun**'s command line parameters causes **irun** to perform an action. It executes these actions strictly in the order described in the *Toolset Reference Manual.*

2.2 The AServer database

The *AServer database* lists:

- the AServer host services provided by **irun** and
- the target hardware connections.

It is a text file containing the names of the services and connections and the information about them that **irun** needs to be able to respond to service requests. The pathname of the AServer database is held by the **ASERVDB** environment variable.

The AServer database is described in the *Toolset Reference Manual.*

AServer database lines beginning with the hash (#) character are comments and are ignored. Each other line of the AServer database contains information about one resource, which may be:

- an AServer service available on the host;
- a target hardware connection from the host.

Each resource line contains four fields, each field beginning with a bar (|) character.

2.2.1 AServer database service resources

An AServer service is a piece of software to which a client may request a connection. Normally a service is a separate process, but it may not be. Services listed in the AServer database run on the host. If an AServer database line is a service, the four fields are:

1 The name of the service, as used by the client in calls to `asc_connect`. Any leading or trailing spaces are ignored, but spaces are allowed in the service name.

2 The command to be executed when the service is requested. This is generally the name of the service executable.

3 The maximum number of connections that the service will accept. The service must be able to handle at least the number of connections specified here.

4 Any additional command line parameters for the service.

The command line used to start a service on the host is built up from:

1 the **Path** field of the service database,

2 the name of any bootable file given in the `irun` command line,

3 for the auto-iserver, all parameters to `irun`. For other services, the parameters are provided by the client in the connect request.

4 the **Extra Params** field of the service database.

The `iserver` converter process does not give any parameters in the connect request for the `iserv` service.

2.2.2 AServer database connection resources

A connection resource is a special case of an AServer service running on the host, used internally by the `irun` target booting software. The connection process 'knows' how to control links, and the parameters provided in the database line define the type and location of the link.

If an AServer database line is a hardware connection, the four fields are:

1 The name of the resource, such as `myb103`. If the resource is a connection to target hardware then the target may be used by:

- setting the environment variable `TRANSPUTER` to this name or
- using the name with the `irun` option `-sl`, such as:

```
irun -sl myb103 app.btl
```

If the resource is an AServer service then the name may be used by the client to request the service.

A description of the resource, such as `txcs tcp front@Apple`. The format of the description will depend upon the interface software being used, as described in your *Toolset Delivery Manual*.

2 The number of connections that may be made to the resource or to a single instance of the service. For target hardware connections this is always 1.

3 Any extra parameters that may be needed, as described in your *Interface Software User Manual*.

2.3 Implementation limit

There is an implementation limit on the number of host processes `irun` can handle. This limit depends upon the processes, but is normally about 6 or 7.

3 The target gateway

This chapter describes the supplied gateway process that runs on the target hardware to connect to a host. An AServer gateway process performs multiplexing and demultiplexing, allowing one or more clients or services access to a hardware link. It also converts the AServer packet protocol into a mega-packet protocol which makes more efficient use of the hardware link. The hardware link may connect to a host, as in Figure 3.1, or to some other device outside the target hardware.

The target gateway process is a gateway running on the target hardware giving access through a hardware serial link connection to a device such as the host. The `irun` process performs a similar gateway role on the host.

The target gateway provided can only handle client processes on the target hardware. It should be used only for client processes on the target hardware to communicate with external services, as shown in Figure 3.1.

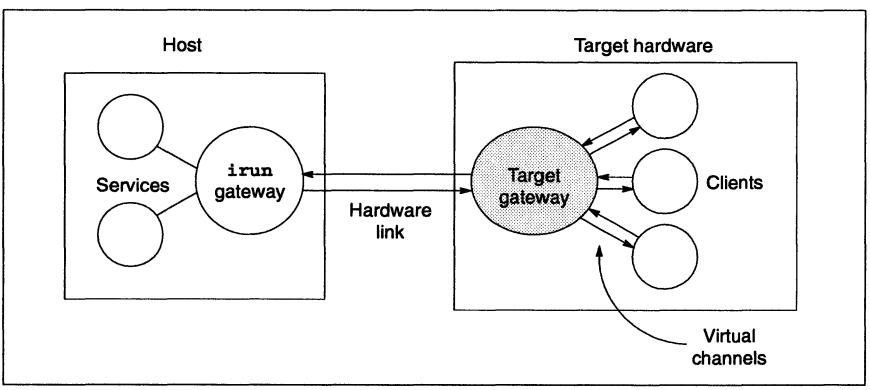

Figure 3.1 A software host-target interface

The target gateway process is provided as a configuration level process called `gateway`. It is a linked unit in the file `gateway.cax` for TA class transputers, `gateway.c0x` for T9000 transputers and `gateway.c6x` for T450s and ST20s. It should be configured as a target process running in parallel with the user's application code.

The `gateway` process takes five parameters, as listed in Table 3.1. The `ASPROT` protocol is defined in the file `gateway.inc`.

For example, using the C-style configurer, the gateway process should look similar to:

```
process(interface (input from_link,
                   output to_link,
                   input  from_processes[gateway_fan_in],
                   output to_processes[gateway_fan_in],
                   int max_mega_packet_size_to_host = 1040))
   gateway;
```

Parameter name	Parameter type	Purpose
`from_link`	input channel	The channel from the link.
`to_link`	output channel	The channel to the link.
`from_processes`	input channel array	The array of input channels from the application clients and services, one for each access point.
`to_processes`	output channel array	The array of output channels to the application clients and services, one for each access point.
`max_mega_packet_size_to_host`	integer constant	The maximum size of a mega-packet. It should be given the value 1040 in this implementation.

Table 3.1 `gateway` parameters

Using the occam-style configurer, the gateway process has the interface:

```
gateway(CHAN OF ANY from.link,
        CHAN OF ANY to.link,
        [gateway.fan.in]CHAN OF ASPROT from.processes,
        [gateway.fan.in]CHAN OF ASPROT to.processes,
        VAL INT max.mega.pkt.size)
```

In the configuration, the process should look similar to:

```
VAL max.mega.pkt.size IS 1040:
gateway(from.link,
        to.link,
        from.processes,
        to.processes,
        max.mega.pkt.size)
```

3.1 Configuration example

This example is similar to the configuration of the Print test example, described in section 5.9. The arrangement of processes is shown in figure 3.2. It shows how the gateway process is connected between the host edge of the target hardware and the AServer processes.

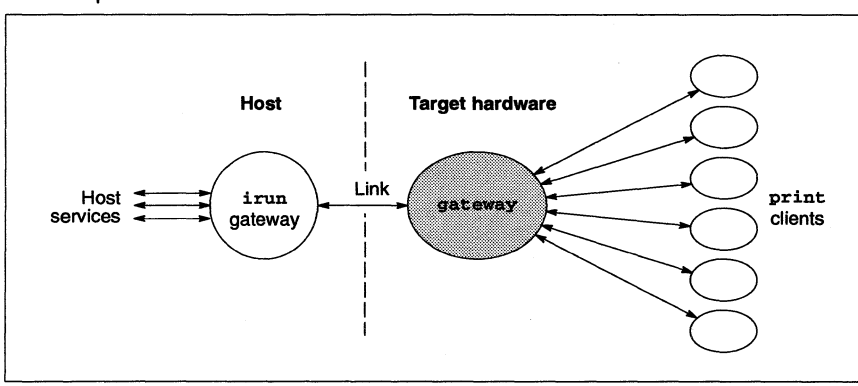

Figure 3.2 Print example

Notice that the gateway process **must** always be inserted between the host edge and the AServer processes, even if there is only one AServer process. This is because the gateway process is needed to convert from the AServer protocol to the mega-packet protocol. Mega-packets are described in section 3.2.

The following is the C-style version of the configuration:

```
/* hardware */

T800 (memory = 1M) board;

connect host to board.link[0];

/* software */

input from_host;
output to_host;

val num_prints 6;

val gateway_fan_in num_prints;

process(interface (input from_link,
                   output to_link,
                   input  from_processes[gateway_fan_in],
                   output to_processes[gateway_fan_in],
                   int max_mega_pkt_size_to_host = 1040),
        nodebug = true)
     gateway;

connect gateway.to_link to to_host;
connect from_host       to gateway.from_link;

process (interface (input as_in,
                    output as_out,
                    int hello_num),
         heapsize = 50000,
         stacksize = 20000)
    print[num_prints];

rep i = 0 for num_prints
{
  connect gateway.from_processes[i] to print[i].as_out[0];
  connect gateway.to_processes[i]   to print[i].as_in[0];
  print[i] (hello_num = i);
}

/* mapping */

place from_host on host;
place to_host on host;

place gateway on board;
use "gatyeway.lku" for gateway;

rep i = 0 for num_prints
{
  place print[i]   on board;
  use "print.cax" for print[i];
}
```

The following code calls `gateway` from an occam-style configuration:

```
-- Hardware description

#INCLUDE "hostio.inc"
#INCLUDE "occonf.inc"
#INCLUDE "gateway.inc"
NODE root :
ARC hostlink :
NETWORK board
  DO
    SET root (type, memsize := "T800", 1 * M)
    CONNECT root[link][0] TO HOST WITH hostlink
:

-- Software description

#USE "gateway.lku" for gateway
#USE "print.cax" for print

VAL num.prints        IS 6 :
VAL gateway.fan.in    IS num.prints:
VAL max.mega.pkt.size IS 1040 :

CONFIG
  CHAN OF ASPROT.OVEREDGE from.link, to.link  :
  PLACE from.link ON hostlink :
  PLACE to.link   ON hostlink :
  [gateway.fan.in]CHAN OF ASPROT gway.to.print, print.to.gway :
  PROCESSOR root
    PAR
      gateway(from.link,
              to.link,
              print.to.gway,
              gway.to.print,
              max.mega.pkt.size)
      PAR i=0 FOR num.prints
        print(gway.to.print[i], print.to.gway[i])
:
```

3.2 Mega-packets

To improve performance, the connection between the host and the target hardware uses bundles of packets called mega-packets. The gateway process is buffered and pipelined in each direction and the two directions of the gateway run in parallel. Mega-packets contain one or more AServer packets and are compatible with INMOS communications software.

Packets to the host are collected together in a mega-packet at the same time as the previous mega-packet is being sent to the host. Similarly, a mega-packet from the host is demultiplexed to the processes at the same time as the next one is being read from the host. Using this technique, mega-packets may be sent almost continuously.

Packets from different clients may be interleaved in the mega-packet protocol to reduce the latency of the communications. Similarly, mega-packets arriving from the link may contain interleaved packets for different clients.

4 iserver service

This chapter describes the iserver service, its purpose, what it does, and how the user controls it. Three examples of simple programs using the iserver service are provided at the end of the chapter. The sources of the examples are provided with the software and full listings are given in Appendix A.

The AServer provides iserver services to support the INMOS standard libraries for C and occam, which both use the iserver protocol. The iserver provides a standard service for applications or parts of applications which do not need high performance input and output. The iserver service provided by the AServer is fully compatible with the iserver protocol and provides the same facilities that the iserver provides.

The iserver service may be used either as an auto-iserver or with the iserver converter. These two methods are described in sections 4.1 and 4.1.1 respectively. In both cases the user application uses normal iserver protocol to communicate with the host. This means that the user application can be the same linked unit using either the iserver itself, the auto-iserver or the iserver converter.

Other AServer services can be used simultaneously with both these mechanisms. For example, the INQUEST debugger uses the auto-iserver capability to run a program which uses the iserver and at the same time the debugger itself runs as a service on the host to perform the debugging.

One iserver service is incorporated into irun, so it uses the same window as irun. If further iserver services are needed, they are spawned as separate processes and so normally run in separate windows. On X-Windows systems, this is done by using xterm with the -e option. By changing the path field in the AServer database file from xterm -e iserv to iserv, the iserver service can be run in the same window as irun.

4.1 Auto-iserver mode

The *auto-iserver* capability of irun allows application programs that run with the iserver to run unmodified under the AServer. This case is shown in Figure 4.2 (cf. Figure 4.1).

In this mode, the user application is built exactly as if the iserver were being used. In most cases a binary .btl file produced for use with the iserver may be used without recompiling or reconfiguring. To permit this configuration, the irun gateway can distinguish iserver protocol packets and interpret them appropriately.

In auto-iserver mode, each iserver packet must be no larger than the AServer packet size. This limits iserver packets to 1024 bytes rather than the iserver maximum of 1040 bytes. The INMOS ANSI C and occam run-time libraries always use iserver packets of less than 1024 bytes.

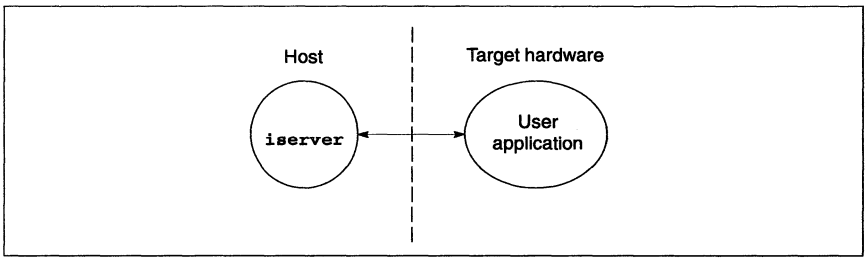

Figure 4.1 `iserver` and client

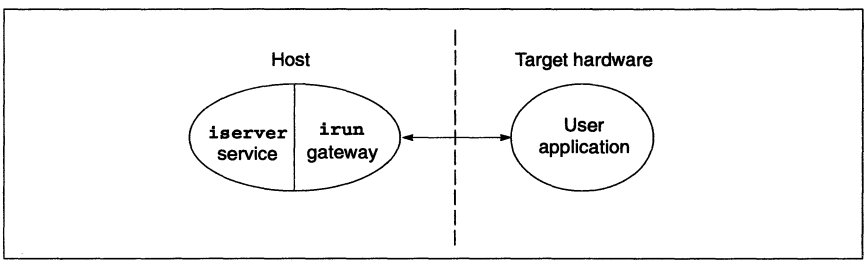

Figure 4.2 Auto-iserver and client

4.1.1 Use with the `iserver` converter

The *iserver converter* converts `iserver` requests from the program into AServer messages that are routed to an `iserver` service. It is described in section 4.2. The use of the `iserver` converter is shown in Figure 4.3. This arrangement allows multiple `iserver` clients and services at the same time.

To use this configuration, the target gateway, `gateway`, and the `iserver` converter, `isconv`, must be configured with the user application to run on the root processor.

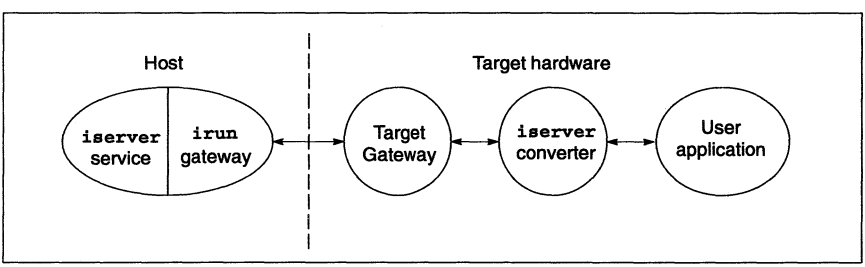

Figure 4.3 `iserver` service, `iserver` converter and client

When using the `iserver` converter, `iserver` packets can be any size up to 1040 bytes.

4.1.2 Parameters

If the irun 'built-in' iserver process is being used, then all the command line parameters given to irun are given to the iserver service in addition to those listed in the 'Extra Parameters' field.

For additional iserver processes, parameters given in the 'Extra Parameters' field of the iserv entry in the AServer database are given to the iserver service.

Parameters to the iserver service should **not** be given in the 'Path' field of the AServer database, or the command line may not be built correctly.

Possible parameters for the iserver service are listed in Table 4.1.

Parameter	Meaning
-sz	Provide debugging information about the iserver service's operation. (Not recommended for normal use)

Table 4.1 iserver service parameters

4.2 iserver converter

The iserver converter process, isconv, is used to convert iserver packets to AServer messages and vice versa. The converter connects to an iserver service, and passes iserver packets to the iserver service in AServer packets. The iserver converter processes requests until it has processed an iserver exit request, when it disconnects from the iserver service and terminates.

The iserver converter process is provided as a configuration level routine called isconv. Linked units are supplied for the target processors supported by your Interface Software. It should be configured as a process running in parallel with the user's application code.

The four parameters for isconv are all channels and are shown in Table 4.2. The ASPROT protocol is defined in the file gateway.inc.

Parameter type	occam type	Purpose
input channel	CHAN OF ASPROT	Input from gateway.
output channel	CHAN OF ASPROT	Output to gateway.
input channel	CHAN OF SP	Input from client. Connect to application ts channel.
output channel	CHAN OF SP	Output to client. Connect to application fs channel.

Table 4.2 isconv parameters

For example, the following code in a C-style configuration file calls isconv:

```
process(interface (input as_in,
                   output as_out,
                   input iserv_in,
                   output iserv_out),
        stacksize = 5K)
    isconv;
```

The following example code calls `isconv` from an occam-style configuration:

```
... other declarations
[gway.fanout]CHAN OF ASPROT gway.to.client, client.to.gway:
CHAN OF SP app.to.isconv, isconv.to.app:
PROCESSOR root
  PAR
    isconv(gway.to.client[0],
           client.to.gway[0],
           app.to.isconv,
           isconv.to.app)
    ... other processes
```

4.3 Hello example

This example demonstrates `iserver` compatibility with the AServer using the `iserver` converter process and the `iserver` service. Figure 4.4 shows the processes involved and the AServer communication. Hello uses the standard i/o routines which use `iserver` protocol, so an `iserver` converter is needed to convert to AServer protocol used by the gateways and the `iserver` service.

This program could be run using the `iserver` without the AServer, as shown in Figure 4.5. Generally, running applications with the AServer improves performance, enhances portability and allows easier future expansion. This example is too simple to illustrate these advantages.

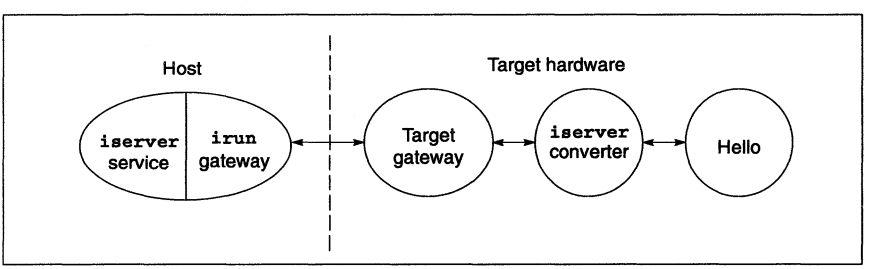

Figure 4.4 Hello example using the AServer

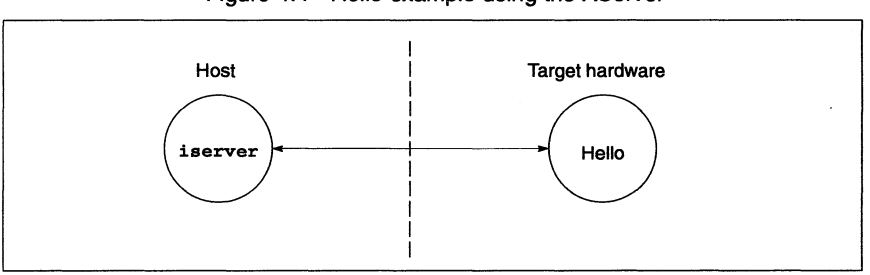

Figure 4.5 Hello example using the `iserver`

4.3.1 The Hello client

The Hello program prints one thousand times the message "`Hello world - via isconv, and aserv (n)`", then waits 10 seconds and terminates.

4.3.2 Configuring the Hello example

The program `hello.c` can be run using the `iserver` directly. The program can also run with the AServer using the `iserver` converter and `iserver` service. This is done by configuring the *same* compiled and linked unit with the `iserver` converter, `isconv`, and the target gateway, `gateway`. This is shown in the configuration file `hello.cfs`, which is outlined below:

```
/* hardware */

...

/* software */

... host edge

process(...) gateway;
... connections

process(...) isconv;
... connections

process (...) hello;
... connections

/* mapping */

...

place gateway on board;
place isconv  on board;
place hello   on board;

...
```

4.3.3 Building the Hello example

The makefiles provided are equivalent to the following, which builds the bootable file `hello.btl` that contains the Hello process, the `iserver` converter and the target gateway process:

```
hello.btl: hello.cfb
    icollect hello.cfb -o hello.btl

hello.cfb: hello.cfs hello.cax
    icconf hello.cfs -o hello.cfb

hello.cax: hello.lnk hello.tax
    ilink -f hello.lnk -ta -x -o hello.cax

hello.tax: hello.c
    icc hello.c -g -ta -o hello.tax
```

4.3.4 Running the Hello example

Once built, the Hello example can be run using a command line like:

```
irun -si hello.btl
```

The `SI` option tells `irun` to display extra information as it runs, and the bootable filename `hello.btl` tells `irun` to boot `hello.btl` onto the target hardware, serve it and monitor the error flag. When run, `hello.btl` will cause `irun` to try to use the built-in `iserver` service.

After displaying the message 1000 times, the program `hello.btl` will terminate, along with the `iserver` service, the `iserver` converter and the gateway `irun`.

4.4 Hello2 example

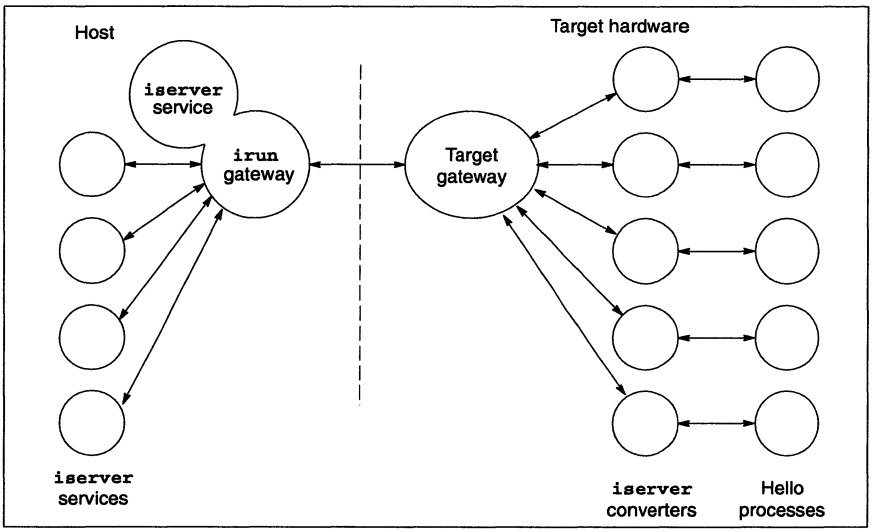

Figure 4.6 Hello2 example

This example is based upon the Hello example, but has five 'Hello world' processes running at once, each with its own `iserver` service running in a separate window. Figure 4.6 shows the processes involved and the AServer communication. Each Hello process is identical to the Hello process in the previous example and has its own `iserver` converter.

4.4.1 Configuring the Hello2 example

The configuration file `hello2.cfs` listed below is similar to `hello.cfs` but has replicated copies of the hello process and the `iserver` converter, `isconv`.

The number of 'Hello world' processes can be changed simply by changing the line

```
val num_hellos 5;
```

in the file `hello2.cfs`.

Note that there is an implementation limit for the number of host processes `irun` can handle. This limit is currently around 6 or 7.

```
/* hardware */

...

/* software */

...

val num_hellos 5;

process(...) gateway;
... connections

process(...)isconv[num_hellos];

rep i = 0 for num_hellos
{
  connect gateway.from_processes[i] to isconv[i].as_out;
  connect gateway.to_processes[i]   to isconv[i].as_in;
}

process (...) hello[num_hellos];

rep i = 0 for num_hellos
{
  connect isconv[i].iserv_in  to hello[i].ts;
  connect isconv[i].iserv_out to hello[i].fs;
}

/* mapping */

...

rep i = 0 for num_hellos
{
  place isconv[i]  on board;
  place hello[i]   on board;
  use "isconv.cax" for isconv[i];
  use "hello.cax" for hello[i];
}
```

4.4.2 Running the Hello2 example

Hello2 can be built in a similar way to Hello, this time using the makefile `hello2.mak`. It can be run using the command line:

```
irun -si hello2.btl
```

The first process uses the built-in `iserver` service. The other four `iserver` requests start `iserver` processes. For X users, these will appear in separate windows if the AServer database has `xterm -e` as the command.

4.5 Getkey Example

The Getkey example demonstrates the use of the standard library function `getkey()` to read keyboard input. Getkey starts three `iserver`-using processes, the source of which is in the file `getkey.c` (listed below). Figure 4.7 shows the processes involved and the AServer communication. Again each Getkey process has an `iserver` converter. One `iserver` service is the built-in service and the other two are separate processes.

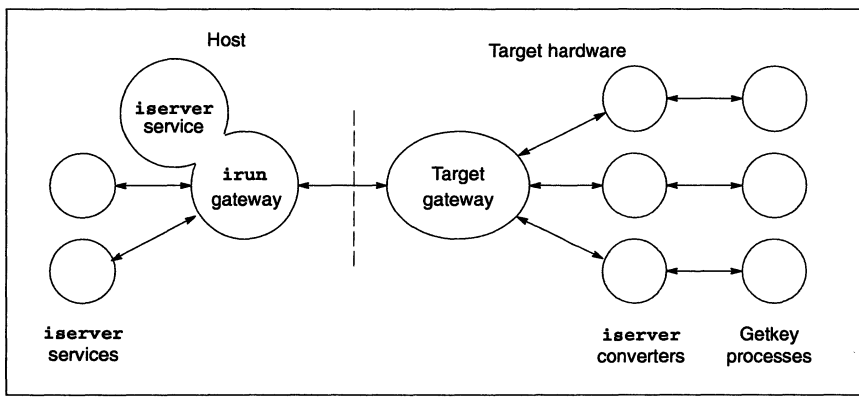

Figure 4.7 Getkey example

The following is an outline of a Getkey process.

```
int main()
{
  int key;
  ...
  key = ' ';

  while (key != 'q')
  {
    key = getkey();

    if (key == -1)
      printf("No key pressed\n");
    else
      printf("Key is \'%c\'\n", key);

    if (key == 'e')
      abort();
  }
...
}
```

Each Getkey process reads keys from the keyboard using the `getkey()` function. Each key is then displayed. If the key is a `q` then the process terminates; if the key is `e` then the process sets the error flag using the `abort()` function. The function call

```
set_abort_action(ABORT_HALT);
```

is used to make the `abort()` function set the error flag.

If the error flag is set, then all three `iserver` services will display the same message that the error flag is set, and then terminate.

Getkey is configured in a similar manner to Hello2, using three replicated copies of the Getkey process, three replicated copies of the `iserver` converter, `isconv`, and a target gateway, `gateway`.

5 Clients and services

5.1 Introduction

To gain full benefit from the AServer, the user may wish to write customized client or service processes. This chapter describes in outline how such processes may be constructed using the AServer library provided. Examples of clients and services are described in sections 5.7 and 5.9. An alphabetical list of AServer library functions, with parameters and other details, is given in Chapter 6.

Clients and services communicate using the AServer protocol, possibly through gateways. This protocol makes the processes portable. The AServer protocol is defined in Appendix B and is supported by the AServer library.

The full libraries are provided as ANSI C libraries for portability. It is expected that client processes running on target hardware will be written in INMOS ANSI C, and the examples in the body of the text of this chapter use ANSI C. occam users are recommended to use ANSI C interface processes between the occam application and the AServer protocol channels. Client processes can be written in occam if necessary, and an example is given in section 5.10.

5.2 Initializing data structures

The AServer library uses a static data structure in each process to hold the type of communications used and maintain the state of the access point. This must be initialized by calling the function in Table 5.1 before any other AServer library routine is called.

Function name	Purpose
`as_apstart`	Initialize an access point

Table 5.1 Library function to initialize access points

The parameters for `as_apstart` depend on the processor on which it is running. To initialize the access point in a host process on a PC under Windows, no parameters are needed, so the code would look like:

```
asrc result_code;
...
result_code = as_apstart ();
```

For Sun hosts, `as_apstart` takes two predefined file descriptors. The defaults are `AS_CHILD.IN_PIPE` and `AS_CHILD.OUT_PIPE`, which are defined in the header file `asslib.h`. To initialize the access point in a host process on a Sun the code would look like:

```
#include <osslib.h>
asrc result_code;
...
result_code = as_apstart (AS_CHILD.IN_PIPE,
                          AS_CHILD.OUT_PIPE);
```

Initializing the access points for an application process running on the target processor is slightly different, because the application uses channels which must be passed in from the context. These channels will often be configuration level channels connected to a gateway. In this case, the channels must be passed into the process code, so in ANSI C code they must be retrieved using `get_param`.

For example, for a single access point process:

```
#include <aslib.h>

int main()
{
  asrc result_code;
  Channel *chan_aserv_in  = (Channel *)get_param(3);
  Channel *chan_aserv_out = (Channel *)get_param(4);

  result_code = as_apstart(chan_aserv_in, chan_aserv_out);
  if (result_code != ASRC_SUCCESS)
    printf("Failed to initialize access point\n");
  ...
}
```

Connection table

A connection table is used by the AServer library to hold for each connection the access point number and the address of the process connected to the other end. The table must be declared as an array of type `conn_id`, with the number of elements being the maximum number of connections open at any one time. This array must be kept and not altered until all the connections are no longer needed. The connection table is initialized with the function `as_setconntable`, as shown in Table 5.2.

Function name	Purpose
`as_setconntable`	Set up a connection table for calling process

Table 5.2 Library function to initialize a connection table

For example:

```
#include <aslib.h>
#define MAX_CONNECTIONS 4
conn_id conn_table[MAX_CONNECTIONS];
asrc res;

res = as_setconntable(MAX_CONNECTIONS, conn_table);
if (res != ASRC_SUCCESS)
    printf("Failed to initialize connection table\n");
```

5.3 Waiting for packets to arrive

It is essential that services and clients are always ready to receive a packet, since the AServer contains no buffering. For example, processes must not wait for a key press using `getkey`, since a packet may arrive while the process is waiting.

The mechanism to wait for incoming packets without blocking other events depends on the platform being used. On the target processor in C, a channel `ProcAlt` function call can be used, and a macro is provided to return the channel. In occam an `ALT` can be used. On Sun systems, a 'select' system call can be used, and a macro is provided to return an input pipe/socket file descriptor. In Windows 3.1 the AServer callback function is used, as described in section 5.3.1.

This is summarized in Table 5.3 and further details are given in Chapter 6.

Platform	Mechanism	Macro or function
Target	Channel ALT	Macro `AS_AP_GIVE_TX_CHAN (`*ap_num*`)` returns a channel.
Sun	Pipe / socket 'select' call	Macro `AS_AP_GIVE_SUN_FD (`*ap_num*`)` returns an input pipe / socket file descriptor
Windows 3.1	AServer callback	Function `ass_set_cb(`*callback_function*`)` sets up the AServer callback function.

Table 5.3 Library macros to wait for packets

If necessary, the receiving process can use `as_read_ready` to poll for incoming packets, but this should be avoided if possible, as polling can be very wasteful of processor time.

5.3.1 AServer callback

For services running on a PC under Windows only, an AServer callback function should be set up using `ass_set_cb`, as shown in Table 5.4.

Function name	Purpose
`ass_set_cb`	Set up an AServer callback function

Table 5.4 Library function to initialize a connection table

For example, to set up a callback function called `AserverCallback`:

```
ass_set_cb (AserverCallback);
```

The AServer callback function must be of type `AserverProc`, defined by:

```
typedef void (*AserverProc) (short aserverEvent);
```

AServer callback is described in more detail in section 5.8.2.

5.4 Connecting and disconnecting

Before communication can take place, a connection must be made between a client and a service. The connection should be closed when the communication has ceased. The AServer library functions related to connecting and disconnecting are listed in Table 5.5.

Function name	Purpose
asc_connect	Request a connection to a service
ass_acceptconnect	Accept a connection request from a client
ass_processconnect	Process a connection request
as_decode_conn_req	Decode a connection request
asc_disconnect	Request a disconnection from a service
ass_sendabort	Request to abort a connection to a client
as_get_conn_info	Get information about a connection

Table 5.5 Library functions to connect and disconnect

First, the client must request a connection, using `asc_connect`. This function waits for a reply before returning. The service may be waiting to accept the connection, using `ass_acceptconnect`, which automatically processes the request and sends a reply.

Alternatively, the service may be reading packets from some other connection using `as_rec_packet` or an `as_readpktcb` callback function. In this case, the header of the packet is extracted using `as_translate_pkt` and the type is extracted from the header using `as_hdr_get_type`. The service then calls `ass_processconnect` to process the request and send a reply.

For example:

```
int ap_num, c_num;
packet_hdr hdr;
pkt_type type;
unsigned char data[MAX_PACKET_SIZE];
asrc result;

result = as_rec_packet(ap_num, hdr, data);
result = as_translate_pkt(result, ap_num, hdr, data, NULL);
type = as_hdr_get_type(hdr);
if (result != ASRC_SUCCESS)
    printf("Error trying to read packet\n");
else if ((type & ASPH_TYPE_MASK) == ASPT_CONNECT_REQ)
    result = ass_processconnect(ap_num, hdr, data, &c_num,
                                ASRC_SUCCESS, NULL);
```

The `asc_connect` function returns a connection number to the client as one of its parameters. Similarly, `ass_acceptconnect` or `assprocess_connect` give a connection number to the service. This number is used when sending and receiving message or packets and when disconnecting.

After the communications are complete, the connection should be closed by the client calling `asc_disconnect`. Any further packets received on that connection will be discarded. If the service needs to close the connection for any reason, then it can do so by calling `ass_sendabort`.

5.5 Sending and receiving

Messages and packets may be sent and received by either the client process or the service process. The AServer functions to do this are listed in Table 5.6.

Function name	Purpose
`as_sendmessage`	Send an AServer message
`as_recmessage`	Receive an AServer message
`as_send_packet`	Send an AServer packet
`as_rec_packet`	Receive an AServer packet
`as_read_ready`	Poll whether a packet is waiting to be read
`as_translate_pkt`	Translate packet type into components

Table 5.6 Library functions to send and receive

The connection being used is identified by a connection number. The client should use the connection number given by `asc_connect`, while the service should use the connection number given by `ass_acceptconnect` or `assprocess_connect`.

5.5.1 Packet level communications

Support is provided for low-level packet communication. If the access point is receiving interleaved packets, then communications should be read as packets using `as_rec_packet`. In other cases, all communications should normally be performed by the message communication functions, which are much simpler to use.

Sending a packet

The sending process must provide a packet header for each packet sent. The header will normally be constant for any particular connection except possibly for the data length. Packet headers are built using the library functions in Table 5.7.

The packet header consists of five fields: data length, gateway index, connection number, packet type and protocol byte. The data length is given in bytes. The gateway index and connection number and can be set by `as_set_dest`.

Function name	Purpose
`as_hdr_set_len`	Set data length field in packet header
`as_set_dest`	Set the gateway index and connection number fields of a packet header
`as_hdr_set_gateway_i`	Set gateway index field in packet header
`as_hdr_set_conn_num`	Set connection number field in packet header
`as_hdr_set_type`	Set packet type field in packet header
`as_hdr_set_p_byte`	Set protocol byte field in packet header
`as_pack_hdr`	Set all fields of packet header

Table 5.7 Library functions to build packet headers

The packet type should normally be `ASPT_DATA`, except that if the current packet is the last packet of the message then `ASPT_DATA` should be or-ed with the constant `ASPH_EOM_MASK` to set the End-of-Message bit. Other packet types are listed in Chapter 6.

The protocol byte field is user-defined, for use as a tag for data message types.

Receiving a packet

The packet is received using `as_rec_packet`. The receiving process must then call `as_translate_pkt` to process the packet. This function automatically deals with disconnect and abort requests.

The received packet header may be decoded using the library functions listed in Table 5.8. The packet type needs to be further decoded by ANDing with `ASPH_EOM_MASK` to extract the type and using the macro `ASPH_IS_EOM (packet_type)` to extract whether the packet is the last of the message.

Function name	Purpose
`as_hdr_get_len`	Get data length field from packet header
`as_hdr_get_gateway_i`	Get gateway index field from packet header
`as_hdr_get_conn_num`	Get connection number field from packet header
`as_hdr_get_type`	Get packet type field from packet header
`as_hdr_get_p_byte`	Get protocol byte field from packet header
`as_unpack_hdr`	Get all fields from packet header

Table 5.8 Library functions to decode packet headers

5.6 Terminating data structures

For services running on a PC under Windows only, the service libraries and data structures should be closed down using `as_apfinish`, as shown in Table 5.9, before the service is terminated. For example:

```
result_code = as_apfinish();
```

Function name	Purpose
`as_apfinish`	Close down data service structures

Table 5.9 Library function to close down Windows service data structures

5.7 Echo example

In this example, the application process talks directly to a custom built Echo service, using the AServer gateways but avoiding the `iserver` protocol. This route is used for the high performance part of the example. It also uses the `iserver` service where appropriate.

A full listing of the Echo example is given in Appendix A.

`iserver` messages are at most 1040 bytes long, and each one is acknowledged before the client can continue. The AServer supports messages of arbitrary length and responses are not always necessary. The Echo example demonstrates the performance gains that come from using messages greater that one packet long. If this facility is being used then the `iserver` service is not suitable, so a custom-built service must be written, in this case the Echo service. This example also demonstrates how easy it is to add functionality to that provided by the standard `iserver` service.

The target code in the file `echo.c` acts as a client, sending messages to the specially written Echo service, which receives AServer messages and sends those same messages back to the client. Since this is all performed using AServer protocol, no `iserver` converter is needed for this communication.

However, the Echo process also uses standard i/o, using `iserver` protocol, to write messages on the screen. For this reason, an `iserver` service and `iserver` converter are needed as well. Figure 5.1 shows the processes involved and the AServer communication.

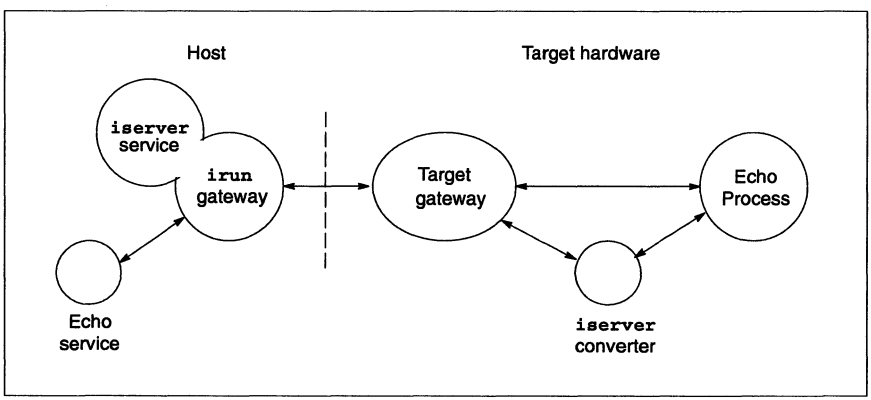

Figure 5.1 Echo example

5.7.1 Echo client

The Echo client has one AServer access point, which it uses to communicate directly with the gateway to send the large messages using the AServer library routines. The Echo client uses `printf` calls using the standard C run-time library, which uses the `iserver` service. This uses a pair of channels to the `iserver` converter which are only explicitly shown in the configuration code.

Connecting to the service

The Echo client initializes the AServer library with the line

```
res = as_apstart(chan_aserv_in, chan_aserv_out);
```

and then displays 'Hello' ten times.

If all is still well, Echo initializes its connection table:

```
res = as_setconntable(CONN_TABLE_SIZE, conn_table);
```

and connects to the echo service:

```
res = asc_connect("echo", "", 0, &conn_num, NULL);
```

The first argument of `asc_connect` is `"echo"` which is the name of the service requested and is the name which appears in the AServer database file. There are no extra parameters to pass to the echo service, so the second argument is an empty string (`""`). The connect request is to be sent from the access point 0, which is the only access point. It is connected to the gateway and thus can be used to reach the Echo service on the host. `conn_num` returns the connection number if the connection is successful. The final, `NULL`, parameter, causes `asc_connect` to discard any unexpected packets; there will not be any in this example.

Communicating with the service

The main code of Echo calls `echo_test()` a number of times, each with different message lengths and number of messages. The outline of echo_test is shown below:

```
asrc echo_test(int conn_num,
               unsigned char *buf,
               unsigned long msg_len,
               int num_msgs,
               const as_bool time_operation)
{
   int count = num_msgs;
   ...
   while ((count > 0) && (res == ASRC_SUCCESS))
   {
      ...
      res = as_sendmessage(conn_num, msg_len, buf, NULL);

      rec_size = msg_len;

      if (res == ASRC_SUCCESS)
         rec_res = as_recmessage(conn_num, &rec_size, buf);

      if ((res == ASRC_SUCCESS) && (rec_res != ASRC_SUCCESS))
         res = rec_res;

      if (res == ASRC_SUCCESS)
      {
         ...
      }
      count --;
   }
   ...
   return(res);
}
```

The `echo_test()` function times the duration of a loop being executed `num_msgs` times. Inside each loop, a message of length `msg_len` is sent and then received. The

total number of bytes both sent and received is displayed, along with the time taken to send and receive them and the number of bytes transferred per second.

The example shows how the number of bytes transferred per second dramatically increases as the message size increases. Typically, over Ethernet to an INMOS IMS B300, the transmission rate for 8kbyte messages is about four to five times greater than for 1kbyte messages.

Disconnecting from the service

After performing the Echo tests using the function `echo_test()`, Echo disconnects from the Echo service using:

```
(void) asc_disconnect(conn_num, NULL);
```

Echo ignores the result code returned because it does not make any further use of the AServer library.

5.7.2 Configuration of the Echo client

The configuration source file is `echo.cfs`. There are three processes running on the target:

```
/* software */
...
val gateway_fan_in 2;

process(...) gateway;
...
process(...) isconv;
...
process(...) echo;
...
```

5.7.3 Echo service

The Echo service runs on the host and simply reads and sends back the messages it receives from the `irun` gateway. The Sun version is called `echoserv`, while the PC version is called `wechosrv`.

Connecting to the client

The Echo service use of the AServer library starts with the same calls as Echo, to `as_apstart()` and `as_setconntable(MAX_ECHO_CONNECTS, conn_table)`.

The Echo service then waits for a connection request with the statement:

```
res = ass_acceptconnect(0, &conn_num, hdr, echo_buf, NULL);
```

The parameters to `ass_acceptconnect()` are the access point 0, `conn_num` which returns the connection number, `hdr` the header of the connect request packet, and

`echo_buf`, the data of the connect request packet. A host client or service only has one access point, so it is always number 0. The final, `NULL`, parameter causes `ass_acceptconnect` to discard any unexpected packets - there will not be any in this example.

Communicating with the client

Once a connection has been formed, the Echo service enters a loop receiving messages using:

```
res = as_recmessage(conn_num, &mess_len, echo_buf);
```

It sends back the messages using:

```
res = as_sendmessage(conn_num, mess_len, echo_buf, NULL);
```

If either `as_recmessage` or `as_sendmessage` returns a result which is not equal to `ASRC_SUCCESS` then the Echo service terminates.

Disconnecting from the client

When the client sends a disconnection request, `as_recmessage` recognises the request and returns `ASRC_GOT_DISCONNECT_REQ`. The Echo service then exits from the loop. No further action is needed.

5.8 Callback

Callback functions are used by AServer to deal with external asynchronous events which need immediate servicing. A function, called a *callback function*, is provided to deal with the event whenever it occurs. When the event occurs, the normal sequence of code is interrupted and the callback function is executed. When the callback function returns, the interrupted code continues. The callback function can leave some state which the main code can test and use.

There are two kinds of callback used by the AServer. A *read callback* is used on all hosts when a packet arrives during certain library function calls which may be sending messages. The read callback function reads the incoming packet and stores it in a buffer. Read callbacks are described in section 5.8.1.

An *AServer callback* is used by Windows code for services run on PC hosts. Windows code is essentially event-driven, so the code to handle an incoming packet or message is an event-handling routine called an AServer callback function. AServer callbacks are described in section 5.8.2.

5.8.1 Read callback

It is possible that a process may receive a packet while it is executing one of the sending functions `asc_connect`, `asc_disconnect`, `ass_processconnect`, `as_sendmessage`, `as_send_packet` or `as_translate_pkt`. In order not to lose such a

packet, a read callback can be used. Read callback means that the user provides a function to handle the packet when it arrives. The callback function is provided by passing a pointer as a parameter, so the callback function must be of type **as_readpktcb**, which is defined by:

```
typedef asrc (*as_readpktcb) (int apnum,
                              packet_hdr hdr,
                              unsigned char *pkt_data)
```

A function of type **as_readpktcb** is used by a client or a service as a parameter to a sending function. If any packets are received while sending, then the callback function is called by the sending function.

A **NULL** callback function pointer can be passed to a sending function, in which case the sending function will not attempt to read from the access point and thus may deadlock. A **NULL** pointer should be used with care, and a function should normally be used. Some implementations may have to read and discard the packet. In this case it returns an error.

The callback function must not make any calls to the AServer library apart from those that simply examine the packet (**as_hdr_get_***), so the packets should normally be buffered and processed later by the calling process.

After the sending function has returned, the calling process should check for buffered packets. Any buffered packets should first be processed by **as_translate_pkt** before processing the packet.

The Print example described in section 5.9 shows how callbacks and buffering can be used.

5.8.2 AServer callback

The AServer callback applies only to services to be run on PC hosts under Microsoft Windows.

The AServer callback function is set up by the function **ass_set_cb**. For example, to set up a callback function called **AserverCallback**:

```
ass_set_cb (AserverCallback)
```

The callback function is started by the AServer when certain events occur. The two types of event which trigger the AServer callback are:

- **ASCB_RECPKT** whenever a packet arrives for the service;
- **ASCB_SIGINT** whenever a terminate signal arrives from **irun**.

The AServer callback function must be of type **AserverProc**, which is defined by:

```
typedef void (*AserverProc) (short aserverEvent)
```

When the callback function is called, the `aserverEvent` should have a value of either `ASCB_RECPKT` or `ASCB_SIGINT`, indicating the type of event which has occured. If `ASCB_RECPKT` has occurred then the the incoming packet should be read, as described in section 5.5. If `ASCB_SIGINT` has occurred then the window should be closed.

An example of an AServer callback function and its use is given in the Print example, described in section 5.9.3. The normal structure of an AServer callback function is:

```
void AserverCallback(short aserverEvent)
{
  ...

  switch(aserverEvent)
  {
    case ASCB_RECPKT :
      ... deal with incoming packet
    case ASCB_SIGINT :
      ... close window
  }
}
```

When dealing with the incoming packet, it is usual to read and unpack the packet header first and then deal with any packets read by the read callback function:

```
case ASCB_RECPKT :
  /* A packet is ready for reading */

  /* Read it */
  res = as_rec_packet(apNum, hdr, data);

  do
  {
    /* Process packet */
    ... interpret packet header

    /* Keep processing packets from the list
     * until the packet list is empty.      */
    if (pktListLength > 0)
      ... remove packet from list

  } while (anotherPacket);
  break;
```

5.9 Print example

The Print example demonstrates:

1 an asynchronous service in which not all messages from the client have replies.

2 callbacks;

3 multiple connections to a service;

4 the building up of messages from possibly interleaved packets.

This Print example uses a custom print service, called `printer`. Figure 5.2 shows the processes involved and the AServer communication.

A full listing of the Print example is given in Appendix A.

5.9.1 The Print client

The Print client, `print.c`, initializes itself and then connects to the print service. The client then loops 1000 times. Inside the loop the function `pr_print` is called to print a message. `pr_print` takes a connection number as its first parameter and then a format string and values in the same manner as the standard `printf` function.

Printing function `pr_print`

`pr_print` is much faster than using the standard `printf` function. `printf` uses the `iserver` service, and thus each `printf` sends a string to the `iserver` and then waits for a reply. `pr_print` simply sends a string to the print service and then continues without awaiting a reply.

```
static asrc pr_print(int c_num, const char format[], ...)
{
  ...
  res = pr_sendmessage(c_num, 2L + (long) strlen(str), msg, NULL);
  return (res);
}
```

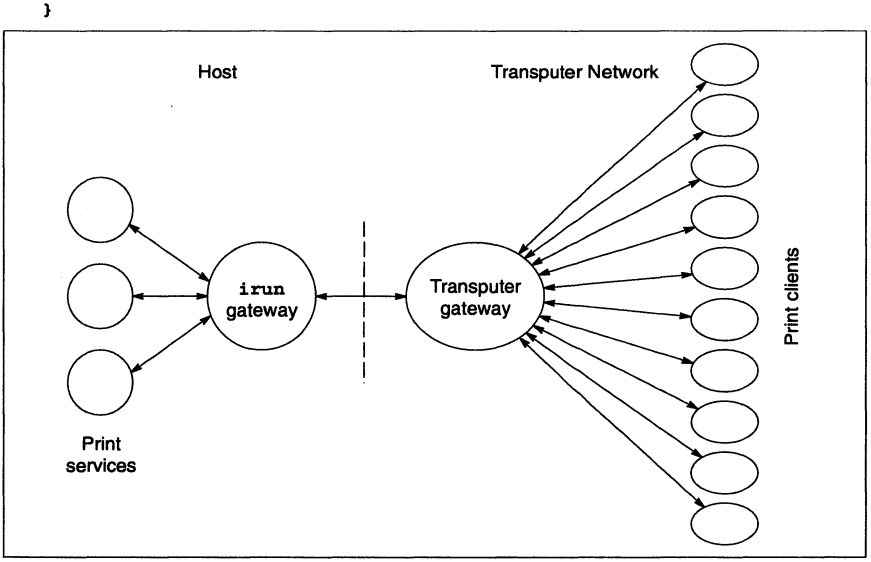

Figure 5.2 Print example

Sending messages function pr_sendmessage

The function pr_print uses the function pr_sendmessage instead of the standard as_sendmessage function. pr_sendmessage takes exactly the same parameters as as_sendmessage. It is different in that the maximum size of the packets sent can be adjusted by changing the value of the PR_SEND_PKT_SIZE macro. PR_SEND_PKT_SIZE must never be made greater than MAX_PACKET_SIZE.

In general, most strings will fit in a single packet of size MAX_PACKET_SIZE. If, however, PR_SEND_PKT_SIZE is set to a value of, say, 5, then each packet will have a data part of a most 5 bytes, and each message will usually require more than one packet. It also works very slowly if PR_SEND_PKT_SIZE is reduced to 1. This will reduce the performance but demonstrates the ability of the printer service to build up messages from interleaved packets, since the packets making up different messages from different clients will arrive interleaved.

```
static asrc pr_sendmessage(int c_num,
                           unsigned long data_size, unsigned char *data,
                           as_readpktcb read_fn)
{
  ...
  while ((res == ASRC_SUCCESS) &&
         ((len_to_go > 0UL) || (sent_a_pkt == AS_FALSE)))
    {
    data_len p_len;

    if (len_to_go > PR_SEND_PKT_SIZE)
    {
      p_len = (data_len) PR_SEND_PKT_SIZE;
    }
    else
    {
      p_len = (data_len) len_to_go;
      as_hdr_set_type(send_hdr, ASPT_DATA | ASPH_EOM_MASK);
    }

    as_hdr_set_len(send_hdr, p_len);
    res = as_send_packet(ap_num, send_hdr, data, read_fn);
    len_to_go -= (unsigned long) p_len;
    data += p_len;
    sent_a_pkt = AS_TRUE;
  }

  return(res);
}
```

Flushing function pr_flush

The purpose of the pr_flush function is to flush out all communications in progress. It makes the client wait until the service has caught up and all handshaking has completed. It is included in this example as an illustration.

Every 100 times round the loop pr_flush is called. pr_flush takes a connection number as its parameter. It sends a message to the service and then waits for a reply. The first byte of each message sent to the print service indicates the type of message.

The value `PR_STRING`, defined in `print.h`, indicates a string to be printed, while `PR_FLUSH` indicates a flush request.

```
static asrc pr_flush(int c_num)
{
  ...
  res = as_sendmessage(c_num, 1, msg, NULL);
  if (res == ASRC_SUCCESS) {
    unsigned long msg_len = 1UL;
    res = as_recmessage(c_num, &msg_len, msg);
  }
  return(res);
}
```

The main program

The main part of the program starts by setting up the connection to the print service:

```
res = as_apstart(chan_aserv_in, chan_aserv_out);
if (res == ASRC_SUCCESS)
  res = as_setconntable(CONN_TABLE_SIZE, conn_table);
if (res == ASRC_SUCCESS)
  res = asc_connect("print", "", 0, &conn_num, NULL);
if (res == ASRC_SUCCESS)
  connected = AS_TRUE;
```

The main loop does 1000 prints, provided there are no communication failures:

```
while ((res == ASRC_SUCCESS) && (i < 1000))
{
  res = pr_print(conn_num,
                 "Hello world (hello no. = %d) - via printer (i =
%d)\n",
                 *hello_num, i);
  if ((res == ASRC_SUCCESS) && ((i % 100) == 0))
    res = pr_flush(conn_num);
  i++;
}
```

5.9.2 Configuring the Print client

The configuration source file is `print.cfs`. It uses replication to connect the multiple print clients to the gateway.

```
/* software */
...
val num_prints 10;

val gateway_fan_in num_prints;

process (...) gateway;
...
process (...) print[num_prints];

rep i = 0 for num_prints
{
  connect gateway.from_processes[i] to print[i].as_out[0];
  connect gateway.to_processes[i]   to print[i].as_in[0];
  print[i] (hello_num = i);
}
```

The line

```
val num_prints 10;
```

sets the number of Print clients. Changing the number will change the number of clients. The source of the print service currently allows a maximum of 8 clients; this can also be changed.

5.9.3 The Print service

The Print service, `printer`, is the most complex of the example services. The source code of the service is in the file `printer.c` for Suns or `wprinter.c` for PCs, and is listed in Appendix A.

Initializing

On Suns the initializing is the same as for the the Echo service. On PCs, the AServer callback function must also be set up. For example, if the callback function is called `AserverCallback` then:

```
ass_set_cb(AserverCallback);
```

After this has been done, the AServer callback function is automatically called whenever a packet or terminate signal arrives.

Read callback

The print service could receive packets from one of the clients at any time, even while it is sending. It is therefore necessary that all the AServer library calls that could send packets (`as_translate_pkt`, `ass_processconnect` and `as_sendmessage`) all have a callback function, `add_to_pkt_list`, as their final parameter.

The callback function is called whenever the function receives a packet it cannot process because it is either sending a packet or waiting for a specific packet. The callback function is not allowed to use any AServer library calls, and thus should normally (as in this example) buffer the packet up for later processing.

```
static asrc add_to_pkt_list(int ap_num,
                  packet_hdr hdr, unsigned char *data)
{
   ... set up space for entry on packet list

   if (res == ASRC_SUCCESS)
   ... copy packet into linked list

   return (res);
}
```

Buffered packets are processed later. On Suns they are processed at the start of the main `while` loop. On Windows systems they are processed by the AServer callback function immediately after a packet has been read following the `ASCB_PKTREC` AServer callback event.

If there are any packets buffered up, a packet is removed from the list of packets, rather than read from the access point. Note that the `as_translate_packet` call is made after the packet has been removed from the list of buffered packets.

The main code

Each time round the main loop, the Sun version of the service either:

1 removes a packet from the buffer of waiting packets if there is one or else

2 reads a packet in.

```
if (pkt_list_len > 0)
{
   res = remove_from_pkt_list(&ap_num, hdr, data);
   ...
}
else
{
   ap_num = 0;
   res = as_rec_packet(ap_num, hdr, data);
}
```

The PC Windows version is structured differently, demonstrating the use of AServer callback. The AServer callback function `AserverCallback` is automatically called whenever the following AServer events occur:

- `ASCB_RECPKT` whenever a packet arrives;
- `ASCB_SIGINT` whenever a terminate signal arrives from `irun`.

If the event was a packet arriving then the `AServerCallback` function handles the incoming packets by:

1 reading in a packet;

2 removing packets from the buffer of waiting packets until there are none left.

```
/* Read packet */
res = as_rec_packet(apNum, hdr, data);

/* Process new packet, and keep processing packets *
 * until the packet list is empty.                 */
do
{
... deal with packet waiting in buffer
} while (anotherPacket);
break;
```

The packet handling is the same in the two versions except that in the Sun version it is part of the main loop while in the PC version it is part of the AServer callback function. In both versions, each packet is decoded to distinguish connect requests and disconnect requests from data:

```
res = as_translate_pkt(res, ap_num, hdr, data, add_to_pkt_list);
type = as_hdr_get_type(hdr);
if (res == ASRC_SUCCESS)
{
   switch(type & ASPH_TYPE_MASK)
   {
      case ASPT_CONNECT_REQ:
         ... deal with connect request
      case ASPT_DATA:
         ... deal with data
   }
}
else if (res == ASRC_GOT_DISCONNECT_REQ)
   ... deal with disconnect request
```

Each connect request packet is processed using the `ass_processconnect` function, and each data packet is added to a buffer. Each connection has a buffer, and incoming packets are added to it. This allows interleaved packets from different connections.

The data packets from each connection are built up into messages:

```
int c_num = (int) as_hdr_get_conn_num(hdr);
int len = (int) as_hdr_get_len(hdr);

memcpy(msg_buffers[c_num].data + msg_buffers[c_num].len, data, len);
msg_buffers[c_num].len += len;
```

When a complete message has been built, the message is processed. If the first byte is `PR_STRING`, the rest of the message is a string to be printed. If the first byte is `PR_FLUSH`, then a zero length message is returned to the client. Note that because the message is zero length, a `NULL` data pointer can be used.

```
if (ASPH_IS_EOM(type))
{
   switch(msg_buffers[c_num].data[0])
   {
      case PR_STRING:
         printf("got string ...", ...);
         break;
      case PR_FLUSH:
         printf("got flush ...", ...);
         res = as_sendmessage(c_num, 0, NULL, add_to_pkt_list);
         printf("sent flush reply\n");
         break;
   }
   msg_buffers[c_num].len = 0;
}
```

5.9.4 The AServer database

The default PC AServer database file contains the line:

```
| print          | wprinter.exe                                    | 4  |
```

Similarly the Sun AServer database file contains the line:

```
| print          | xterm -e printer                                | 4  |
```

In each case, the number `4` in the last field is the maximum number of client connections that `irun` allows each instance of the print service to handle. This number must be greater than or equal to 1, and no more the maximum this service is programmed handle. In the printer example, the maximum number of connections the service code can handle is given by `MAX_PRINT_CONNECTS`, which is set to 8.

With 10 clients, `irun` will start three print services, two with four clients, and one with two. Using the `SI` option will show this in operation. Changing the 4 to 5 in the AServer database file will cause only two print services to be started, each with five clients.

An implementation limit means that currently ten services cannot be supported at once, so changing the 4 to 1 is not possible, at least without changing `num_prints`.

5.10 occam clients

An example of an occam client is provided in the examples directory in `oecho.occ` and listed in Appendix A. This code is the occam equivalent of the Echo example in section 5.7.1. It uses occam versions of some of the library routines. The occam versions are procedures, since occam functions cannot perform channel i/o.

The occam versions of the libraries use modified versions of the standard ANSI C AServer libraries, as the C must not return a value. Some examples are given in the examples directory in `oaslib.c`. For example, `as_apstart` is modified by the addition of an interface as follows, to make it into `oc_as_apstart`:

```
#include <stdio.h>
#include <aslib.h>
#include <misc.h>


void oc_as_apstart(Channel *in, Channel *out, asrc *result)
{
  asrc res;

  set_abort_action(ABORT_HALT);
  res = as_apstart(in, out);
  *result = res;
}
```

Each library procedure requires a **TRANSLATE** pragma and an **EXTERNAL** pragma to translate the name and define the interface, as described in the *occam 2 Toolset User Guide*. For example:

```
#PRAGMA TRANSLATE as.apstart "oc_as_apstart"
#PRAGMA EXTERNAL "PROC as.apstart(VAL INT GSB,
                                  CHAN OF ASPROT in, out,
                                  INT result) = 2048"
```

The structure of the outer level of the program is as follows. Stack and heap areas must be set up for the C library routines. The body of the process is in the procedure `main`.

```
... external file references
... TRANSLATE and EXTERNAL pragmas
... constants

PROC echo(CHAN OF SP fs, ts, CHAN OF ASPROT as.in, as.out)
  ... PROC definitions
  INT GSB:
  ... declare heap and stack areas

  SEQ
    ... set up heap and stack
    main(GSB, fs, ts, as.in, as.out)
    ... close down
:
```

The procedure `main` is the body of the process. It sets up the communications and then runs the procedure `echo.test` a number of times with different parameters.

```
PROC main(INT GSB, CHAN OF SP fs, ts, CHAN OF ASPROT as.in, as.out)
  INT res :
  VAL INT CONN.TABLE.SIZE IS 1 :
  [CONN.TABLE.SIZE][3]INT32 conn.table :
  [MAX.ECHO.BUF.SIZE]BYTE buf :
  INT conn.num :
  ...
  SEQ
    ...  initialize
    as.apstart(GSB, as.in, as.out, res)
    IF
      res <> ASRC.SUCCESS
        ...  display message
      TRUE
        SEQ
          ...  display messages
          as.setconntable(GSB, conn.table, res)
          ...  display message
          asc.connect(GSB, "echo", "", 0, conn.num, res)
          ...  display message
           -- Test code
          ...  fill buffer with 1s
          ...  set up message lengths and numbers
          SEQ i = 0 FOR number.runs
            echo.test(GSB, fs, ts, conn.num, buf,
                                        msg.len[i], num.msgs[i])

         IF
            res = ASRC.SUCCESS
              asc.disconnect(GSB, conn.num, res)
            TRUE
              SKIP
:
```

The procedure `echo.test` sends the messages and receives the replies.

```
PROC echo.test(VAL INT GSB, CHAN OF SP fs, ts,
               INT conn.num, []BYTE buf,
               VAL INT32 msg.len, VAL INT num.msgs)
  INT count, res, rec.res :
  TIMER clock :

  SEQ
    ...  initialize and display messages
    WHILE (count > 0) AND (res = ASRC.SUCCESS)
      SEQ
        as.sendmessage(GSB, conn.num, buf, msg.len, res)
        IF
          res = ASRC.SUCCESS
            as.recmessage(GSB, conn.num, buf, rec.size, rec.res)
          TRUE
            ...  display message
        ...  check response
        count := count - 1

    ...  finish off
:
```

6 AServer library

This chapter describes the AServer library. The library consists of C functions for use when sending and receiving using the AServer protocol. They may be called either in client code or in service code except where the function description states otherwise. Where appropriate, they are provided compiled for both the host and the target. Each client and service on the host has only one access point. Processes on a target may have multiple access points.

A description of using these library functions to write clients and servers is given in Chapter 5.

6.1 Restrictions

The library functions provide no flow control or buffering other than that provided indirectly by the host transport mechanism, for example in UNIX pipes. This means that if a packet could arrive then the programmer must ensure that the program is ready.

The receive message function cannot handle receiving interleaved packets from different messages from different connections.

6.2 Function prototype and constant files

Function prototypes and constants for the AServer library are referenced by including the header file `aslib.h`. `aslib.h` includes `asconst.h` and `aspack.h`. `asmega.h` should also be included by user code that directly processes mega-packets instead of using the library functions.

The filenames of the binary libraries are shown in Table 6.1. One of these files should be linked in with user code, depending on the platform, i.e. the processor on which the code will run.

Platform	Software	Library binary filename
target	ANSI C Toolset	`ast.lib`
PC with MS-DOS	Watcom compiler	`aslib.lib`
PC with MS-DOS	Microsoft compiler	`aslib_ms.lib`
Sun		`libash.lib`

Table 6.1 Library binary files

6.3 Data types and macros

Table 6.2 lists the defined types used by the AServer libraries. These types are defined in the file `asconst.h` except for `as_readpktcb` which is defined in `aslib.h`.

Defined type	Underlying type	Description
AS_UINT16	unsigned short	16-bit unsigned integer
data_len	AS_UINT16	Length of the data in an AServer packet
gateway_index	AS_UINT16	Gateway Index
conn_num	AS_UINT16	Connection Number
pkt_type	unsigned char	AServer packet type
protocol_byte	unsigned char	Protocol byte (not used by the AServer libraries, can be used by the user)
packet_hdr	unsigned char [ASPH_SIZE]	AServer packet header
asrc	int	AServer library result code (see Appendix C).
as_readpktcb	see section 6.5	Read callback function parameter for sending functions.
AserverProc	see section 6.5	Aserver callback function parameter for Windows.

Table 6.2 AServer defined types

The size of an AServer packet header is ASPH_SIZE bytes, which has the value 8 in this implementation and is defined in asconst.h. In calculations, the constant name or the expression sizeof(packet_hdr) should be used instead of the literal number.

The as_bool boolean type is defined in asconst.h. It can take the values AS_FALSE and AS_TRUE. The macro AS_BOOL_TO_INT takes an as_bool as a parameter, and returns an int which is non-zero for AS_TRUE, and zero for AS_FALSE.

On the target, the macro AS_AP_GIVE_TX_CHAN(ap_num) takes an access point number as a parameter and returns a pointer to a Channel, enabling the use of the various ProcAlt function calls to wait for either input from an access point or from other channels. This macro is defined in astlib.h.

On Sun hosts, the macro AS_AP_GIVE_SUN_FD(ap_num) takes an access point number as a parameter and returns the file descriptor of the input pipe/socket. This facility enables the use of the 'select' system call to wait for either input from an access point, or one or more file descriptors. This macro is defined in asslib.h.

When using AS_AP_GIVE_SUN_FD(ap_num) or AS_AP_GIVE_TX_CHAN(ap_num), either as_rec_packet or as_recmessage should be called when the access point is ready for a read operation.

The function as_read_ready is also available to poll to see if a packet is ready for reading from an access point (see section 6.6.31 for details).

6.4 Constants and limits

The maximum size of an AServer packet is defined by the constant MAX_PACKET_SIZE. The value is currently 1024. The maximum length of a service name is MAX_SERVICE_NAME, set to 64.

These constants are in the header file asconst.h.

6.5 Callback function type definition

6.5.1 `AserverProc`

Name `AserverProc`

Purpose A function of type `AserverProc` is used as an AServer callback routine. This callback function type is used by an AServer service to monitor AServer events.

Used by An AServer service running under Windows.

Description This callback is called when an AServer event occurs. The following events are supported:

- `ASCB_RECPKT` – A packet has been received and is ready to read.
- `ASCB_SIGINT` – Interrupt. The `irun` gateway is attempting to close this service down. Depending on the application, the application should terminate when this is received.

Interface

```
typedef void (*AserverProc) (short aserverEvent)
```

Header file `aswlib.h`

Parameters

Name	Type	In/Out	Description
`aserverEvent`	`short`	In	The ID of the AServer event being raised.

Precondition None.

Return None.

Side effects None.

6.5.2 as_readpktcb

Name `as_readpktcb`

Purpose A function of type `as_readpktcb` is used by a client or a service as a parameter to a sending function (e.g. `as_send_packet`). If any packets are received while sending, then the the callback function is called by the sending function.

Used by Any process.

Description A function of this type is called by the sending process.

A `NULL` callback function pointer can be passed into a sending function. The sending function will not attempt to read from the access point, and thus may deadlock. A `NULL` pointer should be used with care, and a function should normally be used. Communications may fail if a `NULL` callback function means that packets are discarded.

A typical callback function will buffer any packets received, and then the caller of the sending function will read the packets from the buffer.

The callback must not make any calls to the AServer library apart from those that simply examine the packet (`as_hdr_get_*`). The packets should normally be buffered and processed later. `as_translate_pkt` should be called after the packet has been removed from the buffer. Callback functions are described in section 5.8 of Chapter 5, and the Print example in section 5.9 shows how callbacks and buffering can be used.

Interface

```
typedef asrc (*as_readpktcb) (int apnum,
                              packet_hdr hdr,
                              unsigned char *pkt_data)
```

Header file `aslib.h`

Parameters

Name	Type	In/Out	Description
`ap_num`	`int`	In	The number of the access point from which the packet was received.
`hdr`	`packet_hdr`	In	The header of the packet received.
`data`	`unsigned char *`	In	The data of the packet received.

Precondition None.

Return AServer result code.

Result **Return value:** `ASRC_SUCCESS`.

The send should continue.

Return value: any result other than `ASRC_SUCCESS`.

The send should be aborted and return the same result code this function returns.

Side effects None.

6.6 Functions

The use of the following functions and some examples are described in Chapter 5. Functions beginning with `asc_` may only be used by a client. Functions beginning with `ass_` may only be used by a service. Functions beginning with `as_` may be used by clients or services.

6.6.1 `asc_connect`

Name `asc_connect`

Purpose To make a connection to a service.

Used by An AServer client.

Description Attempts to connect to a service. A connect request is sent through the access point, and a connect reply from the service or from the AServer indicates whether the connection was successful or not.

Interface

```
asrc asc_connect(const char service_name[],
                 const char service_params[],
                 int ap_num, int *c_num,
                 as_readpktcb read_fn)
```

Header file `aslib.h`

Parameters

Name	Type	I/O	Description
`service_name`	`const char []`	In	The name of the service to connect to.
`service_params`	`const char []`	In	Additional command line parameters for the service. If there are no additional command line parameters the empty string "" should be used.
`ap_num`	`int`	In	The number of the access point through which the connection is to be established.
`c_num`	`int *`	Out	A pointer to the variable to hold the connection number for this connection.
`read_fn`	`as_readpktcb`	In	A callback function called if any packets arrive that the function cannot handle. If `NULL`, such packets are ignored.

Precondition There must be at least one free entry in the connection table.

Return AServer result code.

Result **Return value:** `ASRC_SUCCESS`.

The connection has been successful, and the variable pointed to by `c_num` contains the connection number to be used when using the connection.

Return value: any result other than `ASRC_SUCCESS`.

The connection request failed, with the result code indicating the reason (see Appendix C).

Side effects If the connection is successful, an entry in the connection table is allocated to the connection.

6.6.2 asc_disconnect

Name asc_disconnect

Purpose To disconnect from a service.

Used by An AServer client.

Description Disconnects from a service.

Interface
```
asrc asc_disconnect(int c_num, as_readpktcb read_fn)
```

Header file aslib.h

Parameters

Name	Type	I/O	Description
c_num	int	In	The connection number returned by the connection request.
read_fn	as_readpktcb	In	A callback function called if any packets arrive that the function cannot handle. If NULL, packets for other connections are discarded. Packets for this connection are always discarded.

Precondition c_num must be a valid connection number.

Return AServer result code.

Result **Return value:** ASRC_SUCCESS.

The disconnect has been successful.

Return value: any result other than ASRC_SUCCESS.

An error occurred sending the disconnect request or receiving the reply, and the connection is closed. The result code indicates the reason (see Appendix C).

Side effects An entry in the connection table is freed.

6.6.3 ass_acceptconnect

Name ass_acceptconnect

Purpose To wait for a connect request from a client.

Used by An AServer service.

Description Wait for an AServer connect request from any access point. When a connect request arrives, allocate a connection number, return it in c_num, and send a connect reply to the client.

This function should be used only if a connect request packet is expected. If any other packet could arrive, packets should be received using as_rec_packet, and then ass_processconnect should be called on each connect request.

Interface

```
asrc ass_acceptconnect(int ap_num, int *c_num,
                       packet_hdr hdr,
                       unsigned char *data)
```

Header file aslib.h

Parameters

Name	Type	In/Out	Description
ap_num	int	In	The access point to await a connect request from.
c_num	int *	Out	The connection number of the connection from which the connection request was received.
hdr	packet_hdr	Out	The packet header of the received connect request. Note that connect request messages are only one packet long.
data	unsigned char *	Out	The data of the connect request message. See section B.3.1 for information on the connect request format.

Precondition None.

Result

Return value: ASRC_SUCCESS.

A connect request was received successfully.

Return value: ASRC_NOT_CONNREQ.

A packet other than a connect request was received.

Return value: any result other than ASRC_SUCCESS or ASRC_NOT_CONNREQ.

An error occurred receiving the connect request. The result code indicates the reason (see Appendix C).

Side effects None.

6.6.4 ass_processconnect

Name ass_processconnect

Purpose To process a connect request received by as_rec_packet or by the as_readpktcb callback function.

Used by Used by an AServer service.

Description Process a connect request received by as_rec_packet or by the as_readpktcb callback function. Return the connection number in c_num for use in subsequent operations on the connection.

The connection is only made if the result returned is ASRC_SUCCESS and the result parameter is ASRC_SUCCESS. The connection request can be refused by passing a result parameter other than ASRC_SUCCESS (normally ASRC_FAILED) to the function.

Interface

```
asrc ass_processconnect(int ap_num, packet_hdr hdr,
                        unsigned char *data,
                        int *c_num, asrc result,
                        as_readpktcb read_fn)
```

Header file aslib.h

Parameters

Name	Type	I/O	Description
ap_num	int	In	The access point number of the access point from which the request came.
hdr	packet_hdr	In	The header of the connect request packet.
data	unsigned char *	In	The data of the connect request packet.
c_num	int *	Out	If successful, the connection number of the newly formed connection.
result	asrc	In	The result to be returned to the requesting client. If this code is anything other than ASRC_SUCCESS then the connection is closed, and the connection number released.
read_fn	as_readpktcb	In	A function to be called if any packets are received while sending the connect reply. If NULL, ass_processconnect may deadlock if there is a packet waiting to be read.

Return AServer result code.

Result **Return value:** ASRC_SUCCESS.

The connect reply was successfully sent.

Return value: any result other than ASRC_SUCCESS.

An error occurred sending the connect reply. The result code indicates the reason (see Appendix C).

6.6.5 ass_sendabort

Name ass_sendabort

Purpose To abort a connection to a client.

Used by An AServer service.

Description Disconnect from a client.

Interface asrc ass_sendabort(int c_num)

Header file aslib.h

Parameters

Name	Type	In/Out	Description
c_num	int	In	The connection number returned by ass_acceptconnect.

Result **Return value:** ASRC_SUCCESS.

The abort was successful.

Return value: any result other than ASRC_SUCCESS.

An error occurred sending the abort request or receiving the reply, and the connection is closed. The result code indicates the reason (see Appendix C).

Side effects An entry in the connection table is freed.

6.6.6 ass_set_cb

Name ass_set_cb

Purpose Sets the AServer service callback routine.

Used by An AServer host service running under Windows.

Interface `void ass_set_cb(AserverProc asCallback)`

Header file aswlib.h

Parameters

Name	Type	I/O	Description
asCallback	AserverProc	In	The AServer callback function. If NULL then the AServer callback is ignored.

Result None.

Side effects The callback function is set up.

6.6.7 as_apfinish

Name as_apfinish

Purpose To close down service structures.

Used by An AServer host service running under Windows.

Description This function should be used to clear library data structures and close down libraries before terminating a Windows service.

Interface

```
asrc as_apfinish(void)
```

Header file aswlib.h

Parameters None.

Result **Return value:** ASRC_SUCCESS.

The structures have been removed.

Return value: any result other than ASRC_SUCCESS.

The removal failed, with the result code indicating the reason (see Appendix C).

Side effects The structures are cleared.

6.6.8 as_apstart

Name as_apstart

Purpose To initialize a single access point.

Used by Any process.

Description Initialize a single access point for a process. as_apstart must be called before any other AServer library functions. The target version of this function has channel parameters for ease of use in configuration and occam. The versions for other platforms have parameters which depend on the platform.

Interface On targets:

```
asrc as_apstart(Channel *in, Channel *out)
```

On PC Windows hosts:

```
asrc as_apstart()
```

On Sun hosts:

```
asrc as_apstart(int in, int out)
```

Header file aslib.h. The Sun default values are defined in asslib.h.

Parameters For targets, see table:

Name	Type	In/Out	Description
in	Channel *	In/Out	Access point input channel.
out	Channel *	In/Out	Access point output channel.

For Sun hosts, see table:

Name	Type	Default value	Description
in	int	AS_CHILD.IN_PIPE	Access point file descriptor for input.
out	int	AS_CHILD.OUT_PIPE	Access point file descriptor for output.

Precondition This function should be called before any other AServer library calls.

Result **Return value:** ASRC_SUCCESS.

The access point has been correctly initialized.

Return value: any result other than ASRC_SUCCESS.

The initialization failed, with the result code indicating the reason (see Appendix C).

Side effects Set the values of static variables in the AServer library that contain information about the access point that belongs to the calling process.

6.6.9 as_asrc_to_str

Name as_asrc_to_str

Used by Any process.

Description Converts an asrc value into a string, typically for debugging.

Interface char *as_asrc_to_str(asrc res)

Header file aslib.h

Parameters

Name	Type	In/Out	Description
res	asrc	In	The value to be converted to a string.

Precondition None.

Result **Return value:** A pointer to a null-terminated string.

If res is a valid asrc the pointer is set to a string containing the name of the value.

If res is not a valid asrc the pointer is set to a string containing "Unknown ASRC value".

Side effects None.

6.6.10 as_bool_to_str

Name as_bool_to_str

Used by Any process

Description Converts an as_bool value into a string, typically for debugging.

Interface char *as_bool_to_str(as_bool value)

Header file aslib.h

Parameters

Name	Type	In/Out	Description
value	as_bool	In	The as_bool boolean value to be converted to a string.

Precondition None.

Result **Return value:** A pointer to a null-terminated string.

If value is AS_TRUE then the string returned is "TRUE".

If value is AS_FALSE then the string returned is "FALSE".

Side effects None.

6.6.11 as_decode_conn_req

Name as_decode_conn_req

Used by An AServer service.

Description Decode a connect request, returning pointers to the service name and parameters given by the client.

Interface

```
void as_decode_conn_req(unsigned char *data,
                        char **service,
                        char **params)
```

Header file aslib.h

Parameters

Name	Type	In/Out	Description
data	unsigned char *	In	The data part of a connect request.
service	char **	Out	Points to a pointer to the first character of a null-terminated string containing the name of the service requested by the client.
params	char **	Out	Points to a pointer to the first character of a null-terminated string containing any parameter given by the client.

Precondition None.

Side effects None.

6.6.12 as_get_conn_info

Name as_get_conn_info

Purpose To obtain information about a connection, typically for debugging.

Used by An AServer client or service.

Description Provide information about a connection.

as_get_conn_info takes a connection number and returns the access point to reach the other end of the connection, the destination gateway index, and the destination connection number.

Use as_set_dest to set the destination of a packet from a connection number.

Interface

```
void as_get_conn_info(int c_num, int *ap_num,
                      gateway_index *dest_gateway_i,
                      conn_num *dest_conn_num)
```

Header file aslib.h

Parameters

Name	Type	I/O	Description
c_num	int	In	The number of the connection for which information is required.
ap_num	int *	Out	A pointer to the access point number for the connection. If the value is INVALID_AP_NUM, then the connection is not in use.
dest_gateway_i	gateway_index *	Out	A pointer to the gateway index of the other end of the connection.
dest_conn_num	conn_num *	Out	A pointer to the connection number of the other end of the connection.

Precondition None.

Side effects None.

6.6.13 as_give_ap_ptr

Name as_give_ap_ptr

Purpose To find an access point.

Used by An AServer client or service.

Description This returns a pointer to access point whose number is given as a parameter.

Interface

```
apoint *as_give_ap_ptr (int ap_num)
```

Header file aslib.h

Parameters

Name	Type	I/O	Description
ap_num	int	In	The access point number.

Precondition The access point number, ap_num, must lie in the range 0 up to (num_aps - 1) inclusive.

Return A pointer to the specified access point.

Side effects None.

6.6.14 as_hdr_get_conn_num

Name `as_hdr_get_conn_num`

Purpose To decode the connection number field from an AServer packet header.

Description Unpack a packet header connection number and return that value. `as_hdr_set_conn_num` should have been used to pack the connection number.

Used by Any process.

Interface `conn_num as_hdr_get_conn_num(packet_hdr hdr)`

Header file `aspack.h`

Parameters

Name	Type	In/Out	Description
`hdr`	`packet_hdr`	In	An AServer packet header

Precondition None.

Return The unpacked connection number field from an AServer packet header `hdr`.

Side effects None.

6.6.15 as_hdr_get_gateway_i

Name as_hdr_get_gateway_i

Purpose To decode the gateway index field from an AServer packet header.

Used by Any process.

Description Unpack the gateway index value from an AServer packet header and return that value. as_hdr_set_gateway_i should have been used to pack the gateway index.

Interface

```
gateway_index as_hdr_get_gateway_i (packet_hdr hdr)
```

Header file aspack.h

Parameters

Name	Type	In/Out	Description
hdr	packet_hdr	In	An AServer packet header

Precondition None.

Return The unpacked gateway index field from the AServer packet header hdr.

Side effects None.

6.6.16 as_hdr_get_len

Name as_hdr_get_len

Purpose To decode the data length field from an AServer packet header.

Used by Any process.

Description Unpack a data length value from an AServer packet header and return the value. as_hdr_set_len should have been used to pack the data length.

Interface

```
data_len as_hdr_get_len(packet_hdr hdr)
```

Header file aspack.h

Parameters

Name	Type	In/Out	Description
hdr	packet_hdr	In	An AServer packet header

Precondition None.

Return The function returns the unpacked data length field from the AServer packet header hdr.

Side effects None.

6.6.17 as_hdr_get_p_byte

Name as_hdr_get_p_byte

Purpose To decode the protocol byte from an AServer packet header.

Used by Any process.

Description Unpack the protocol byte from an AServer packet header, and return that value. as_hdr_set_p_byte should have been used to pack the protocol byte.

Interface

```
pkt_type as_hdr_get_p_byte(packet_hdr hdr)
```

Header file aspack.h

Parameters

Name	Type	In/Out	Description
hdr	packet_hdr	In	An AServer packet header

Precondition None.

Return The unpacked protocol byte from the AServer packet header hdr.

Side effects None.

6.6.18 as_hdr_get_type

Name `as_hdr_get_type`

Purpose To decode the AServer packet type from an AServer packet header.

Used by Any process.

Description Unpack the AServer packet type from an AServer packet header, and return that value. `as_hdr_set_type` should have been used to pack the AServer packet type.

Bits 0 to 6 hold the type and bit 7 holds the end-of-message tag (see section B.2). Mask the result with `ASPH_TYPE_MASK` to get the packet type and use the `ASPH_IS_EOM()` macro to test for the end-of-message tag.

Interface

```
pkt_type as_hdr_get_type(packet_hdr hdr)
```

Header file `aspack.h`

Parameters

Name	Type	In/Out	Description
`hdr`	`packet_hdr`	In	An AServer packet header

Precondition None.

Return The unpacked AServer packet type from the AServer packet header `hdr`.

Side effects None.

6.6.19 as_hdr_set_conn_num

Name as_hdr_set_conn_num

Purpose To set the connection number field in an AServer packet header.

Used by Any process.

Description Pack a connection number into an AServer packet header.

as_hdr_get_conn_num should be used to unpack the connection number.

Interface

```
void as_hdr_set_conn_num(packet_hdr hdr, conn_num c_num)
```

Header file aspack.h

Parameters

Name	Type	In/Out	Description
hdr	packet_hdr	In/Out	An AServer packet header
c_num	conn_num	In	The value to be written into the connection number field of hdr

Precondition None.

Result The connection number field of the AServer packet header hdr is set to c_num.

Side effects None.

6.6.20 as_hdr_set_gateway_i

Name `as_hdr_set_gateway_i`

Purpose To set the gateway index field in an AServer packet header.

Used by Any process.

Description Pack a gateway index value into an AServer packet header.

`as_hdr_get_gateway_i` should be used to unpack the gateway index.

Interface

```
void as_hdr_set_gateway_i(packet_hdr hdr,
                          gateway_index gi)
```

Header file `aspack.h`

Parameters

Name	Type	In/Out	Description
`hdr`	`packet_hdr`	In/Out	An AServer packet header
`gi`	`gateway_index`	In	The value to be written into the gateway index field of `hdr`

Precondition None.

Result The gateway index field of the AServer packet header `hdr` is set to `gi`.

Side effects None.

6.6.21 as_hdr_set_len

Name as_hdr_set_len

Purpose To set the data length field in an AServer packet header.

Used by Any process.

Description Pack a data length value into an AServer packet header.

as_hdr_get_len should be used to unpack the data length.

Interface

```
void as_hdr_set_len(packet_hdr hdr,
                    data_len len)
```

Header file aspack.h

Parameters

Name	Type	In/Out	Description
hdr	packet_hdr	In/Out	An AServer packet header
len	data_len	In	The value to be written into the data length field of hdr

Precondition None.

Result The data length field of the AServer packet header (hdr) is set to len.

Side effects None.

6.6.22 as_hdr_set_p_byte

Name as_hdr_set_p_byte

Purpose To set the protocol byte field in an AServer packet header.

Used by Any process.

Description Packs a protocol byte into an AServer packet header.

as_hdr_get_p_byte should be used to unpack the protocol byte.

Interface

```
void as_hdr_set_p_byte(packet_hdr hdr,
                       protocol_byte p_byte)
```

Header file aspack.h

Parameters

Name	Type	In/Out	Description
hdr	packet_hdr	In/Out	An AServer packet header
p_byte	protocol_byte	In	The value to be written into the protocol byte field of hdr

Precondition None.

Result The protocol byte field of the AServer packet header hdr is set to p_byte.

Side effects None.

6.6.23 as_hdr_set_type

Name as_hdr_set_type

Purpose To set the packet type field in an AServer packet header.

Used by Any process.

Description Packs a packet type into an AServer packet header.

as_hdr_get_type should be used to unpack the packet type.

Bits 0 to 6 hold the type, and bit 7 holds the end-of-message tag (see section B.2).

Interface

```
void as_hdr_set_type(packet_hdr hdr, pkt_type type)
```

Header file aspack.h

Parameters

Name	Type	In/Out	Description
hdr	packet_hdr	In/Out	An AServer packet header
type	pkt_type	In	The value to be written into the packet type field of hdr

Precondition None.

Result The packet type field of the AServer packet header (hdr) is set to type.

Side effects None.

6.6.24 as_hdr_to_str

Name as_hdr_to_str

Used by Any process.

Description Converts an AServer packet header value into a string, typically for debugging.

Interface `char *as_asrc_to_str(char *str, packet_hdr hdr)`

Header file aslib.h

Parameters

Name	Type	In/Out	Description
str	char *	Out	A character string buffer.
hdr	packet	In	The AServer packet header to be converted to a string.

Precondition String str must be long enough to hold the header.

Return A pointer to a null-terminated string.

Side effects The string str is used as a buffer.

6.6.25 as_numaps

Name as_numaps

Purpose To determine how many access points the calling process has.

Used by Any process.

Description Returns the number of access point that are available to the calling process.

Interface

```
asrc as_numaps(int *num_aps)
```

Header file aslib.h

Parameters

Name	Type	In/Out	Description
num_aps	int *	Out	The number of access points

Precondition The function will not work until the access points have been initialized by as_apstart.

Return Result code.

Result **Return value:** ASRC_SUCCESS

The integer pointed to by num_aps is set to the number of access points that the system has.

Return value: ASRC_FAILED

The access point table has not been initialized.

Side effects None.

6.6.26 as_pack_hdr

Name as_pack_hdr

Purpose To pack values into all the fields of an AServer packet header.

Used by Any process.

Description Pack the len, gi, c_num, type, and p_byte values into an AServer packet header. as_unpack_hdr should be used to unpack the whole header, or the various as_hdr_get_* functions should be used to unpack parts of the header.

Interface
```
void as_pack_hdr(packet_hdr hdr, data_len len,
                 gateway_index gi, conn_num c_num,
                 pkt_type type, protocol_byte p_byte)
```

Header file aspack.h

Parameters

Name	Type	I/O	Description
hdr	packet_hdr	Out	An AServer packet header
len	data_len	In	The value to be written into the data length field of hdr
gi	gateway_index	In	The value to be written into the gateway index field of hdr
c_num	conn_num	In	The value to be written into the connection number field of hdr
type	pkt_type	In	The value to be written into the packet type field of hdr
p_byte	protocol_byte	In	The value to be written into the protocol byte field of hdr

Precondition None.

Result The fields of the AServer packet header hdr are set to len, gi, c_num, type, and p_byte.

Side effects None.

6.6.27 as_pack_int32

Name as_pack_int32

Purpose To pack a signed 32-bit integer into a buffer of unsigned chars.

Used by Any process.

Description Pack a signed 32-bit integer into a buffer of unsigned chars.

as_unpack_int32 should be used to unpack the 32-bit integer.

Interface

```
void as_pack_int32(unsigned char *buf, long num)
```

Header file aspack.h

Parameters

Name	Type	I/O	Description
buf	unsigned char *	Out	Address to write the signed integer as four unsigned chars, in little-endian format
num	AS_UINT16	In	Signed 32-bit integer to write into buffer

Precondition buf must be a valid pointer into a buffer of unsigned chars, there must be at least three unsigned chars after that pointed to by buf.

Result The signed 32-bit integer is stored in the buffer of unsigned chars.

Side effects None.

6.6.28 as_pack_uint16

Name as_pack_uint16

Purpose To pack an unsigned 16-bit integer into a buffer of unsigned chars.

Used by Any process.

Description Pack an unsigned 16-bit integer into a buffer of unsigned chars.

as_unpack_uint16 should be used to unpack the 16-bit integer.

Interface

```
void as_pack_uint16(unsigned char *buf, AS_UINT16 num)
```

Header file aspack.h

Parameters

Name	Type	In/Out	Description
buf	unsigned char *	Out	Address to write the unsigned int as two unsigned chars, in little-endian format
num	AS_UINT16	In	Unsigned 16-bit integer to write into buffer

Precondition buf must be a valid pointer into a buffer of unsigned chars, there must be at least one unsigned char after that pointed to by buf.

Result The unsigned 16-bit integer is stored in the buffer of unsigned chars.

Side effects None.

6.6.29 as_pack_uint32

Name as_pack_uint32

Purpose To pack an unsigned 32-bit integer into a buffer of unsigned chars.

Used by Any process.

Description Packs an unsigned 32-bit integer into a buffer of unsigned chars.

as_unpack_uint32 should be used to unpack the 32-bit integer.

Interface

```
void as_pack_uint32(unsigned char *buf, unsigned long num)
```

Header file aspack.h

Parameters

Name	Type	In/Out	Description
buf	unsigned char *	In	Address to write the unsigned int as four unsigned chars, in little-endian format
num	AS_UINT16	In	Unsigned 32-bit integer to write into buffer

Precondition buf must be a valid pointer into a buffer of unsigned chars, there must be at least three unsigned chars after that pointed to by buf.

Result The unsigned 32-bit integer is stored in the buffer of unsigned chars.

Side effects None.

6.6.30 as_ptype_to_str

Name as_ptype_to_str

Purpose To convert a pkt_type value into a string, typically for debugging.

Used by Any process.

Interface char *as_ptype_to_str(pkt_type type)

Header file aslib.h

Parameters

Name	Type	In/Out	Description
type	pkt_type	In	The packet type to be converted to a string. The value should be masked with ASPH_TYPE_MASK if necessary to remove any end-of-message tag.

Precondition None.

Result **Return value:** A pointer to a null-terminated string.

If type is a valid pkt_type the pointer is set to a string containing the name of the packet type.

If type is not a valid pkt_type the pointer is set to a string containing "Unknown packet type".

Side effects None.

6.6.31 as_read_ready

Name `as_read_ready`

Purpose To poll to see whether a packet is ready to be read from an access point.

Used by Any process.

Interface `as_bool as_read_ready(int ap_num)`

Header file `aslib.h`

Parameters

Name	Type	In/Out	Description
`ap_num`	`int`	In	The access point number of the access point to be checked.

Return Boolean value.

Result **Return value:** `AS_TRUE`

A packet is available to be read.

Return value: `AS_FALSE`

No packet is available.

Side effects None.

6.6.32 as_recmessage

Name as_recmessage

Purpose To receive an AServer message.

Used by Any process.

Description as_recmessage cannot cope with interleaved packets from different connections to the access point. If packets making up messages from different connections to an access point could arrive interleaved, then the packets should be read individually, and built up into messages by the receiving process. For an example see the source of the Print example provided in section 5.9.

Interface

```
asrc as_recmessage(int c_num,
                   unsigned long *data_size,
                   unsigned char *data)
```

Header file aslib.h

Parameters

Name	Type	In/Out	Description
c_num	int	In	The connection number of the connection from which the message is to be received.
data_size	unsigned long *	In/Out	On entry, *data_size is set to the maximum size that the incoming message may be. On exit, *data_size is set to the actual size of the message.
data	unsigned char *	Out	The data to be read from the message.

Precondition *data_size is set to the maximum size message that the caller wants.

Return AServer result code.

Result **Return value:** ASRC_SUCCESS.

The receive was successful.

Return value: ASRC_MSGTOOBIG

The message was longer than the *data_size value. The excess data was discarded and the data in the buffer should not be relied upon.

Return value: any result other than ASRC_SUCCESS or ASRC_MSGTOOBIG.

An error occurred receiving the message, with the result code indicating the reason (see Appendix C).

6.6.33 as_rec_packet

Name `as_rec_packet`

Purpose To receive an AServer packet from an access point. It is strongly recommended that `as_recmessage` should be used instead if possible.

Used by Any process.

Description Receives a packet from the access point whose access point number is `ap_num`. If a packet is not immediately available, then the function waits until one arrives, or an error occurs.

To perform a 'select' or 'alt' type operation, waiting for one of a number of access points and (possibly) host specific communications channels (e.g. Sun sockets/pipes, or target channels) to become ready for a read operation, the `AS_AP_GIVE_SUN_FD` and `AS_AP_GIVE_TX_CHAN` macros should be used.

Interface

```
asrc as_rec_packet(int ap_num,
                   packet_hdr hdr,
                   unsigned char *data)
```

Header file `aslib.h`

Parameters

Name	Type	In/Out	Description
`ap_num`	`int`	In	The access point number of the access point to read the packet from.
`hdr`	`packet_hdr`	Out	The header of the received packet.
`data`	`unsigned char *`	Out	The data of the received packet. The length of the data received is determined using the `as_hdr_get_len` function on `hdr`

Return AServer result code.

Result **Return value:** `ASRC_SUCCESS`.

The receive was successful.

Return value: any result other than `ASRC_SUCCESS`.

An error occurred receiving the packet, and `hdr` and `data` are invalid. The result code indicates the reason (see Appendix C).

An error occurred sending the packet.

Side effects None.

6.6.34 as_sendmessage

Name as_sendmessage

Purpose To send an AServer message to a connection.

Used by Any process.

Interface

```
asrc as_sendmessage(int c_num,
                    unsigned long data_size,
                    unsigned char *data,
                    as_readpktcb read_fn)
```

Header file aslib.h

Parameters

Name	Type	In/Out	Description
c_num	int	In	The connection number to send the message to.
data_size	unsigned long	In	The length of the data to be sent in the message.
data	unsigned char *	In/Out	The data to be sent in the message.
read_fn	as_readpktcb	In	A function to be called if any packets are received while sending. If NULL, then as_sendmessage may block if there is a packet to be read, thus causing deadlock.

Return AServer result code.

Result **Return value:** ASRC_SUCCESS.

The send was successful.

Return value: any result other than ASRC_SUCCESS.

An error occurred sending the message, with the result code indicating the reason (see Appendix C).

Side effects None.

6.6.35 as_send_packet

Name as_send_packet

Purpose Used by a client or a service to send an AServer packet. It is strongly recommended that as_sendmessage should be used instead if possible.

Used by Any process.

Description To send a packet to a connection, given a connection number, as_set_dest should be used to set the gateway index and connection number fields in the packet header and to determine which access point to send the packet to.

For general use, sending packets between clients and services, the packet type should be ASPT_DATA.

Interface

```
asrc as_send_packet(int ap_num,
                    packet_hdr shdr,
                    unsigned char *sdata,
                    as_readpktcb read_fn)
```

Header file aslib.h

Parameters

Name	Type	In/Out	Description
ap_num	int	In	The number of the access point to send the packet to.
shdr	packet_hdr	In	The header of the packet to send.
sdata	unsigned char *	In	The data of the packet to be sent.

Precondition The packet header must be correctly initialized (usually using as_set_dest).

Return AServer result code.

Result **Return value:** ASRC_SUCCESS.

The send was successful.

Return value: any result other than ASRC_SUCCESS.

An error occurred sending the packet, with the result code indicating the reason (see Appendix C).

Side effects None.

6.6.36 as_setconntable

Name as_setconntable

Purpose To set up the connection table needed by the AServer library.

Used by Any process.

Description Each entry in the table allows one connection to be made (either as a client or a service). Static variables in the library hold details of the connection table, and each entry in the table is initialized.

Interface

```
asrc as_setconntable(int max_connects,
                     conn_id conn_table[])
```

Header file aslib.h

Parameters

Name	Type	In/Out	Description
max_connects	int	In	The maximum number of connections that the calling process wishes to use.
conn_table	conn_id conn_table[]	In	The connection table to be used by the library.

Precondition The table should have max_connects elements. If the table is larger memory is wasted, if smaller then errors will occur.

The memory use by the table must not be de-allocated until the library has been completely finished with.

Return AServer result code.

Result **Return value:** ASRC_SUCCESS.

The connection table has been correctly initialized.

Return value: ASRC_FAILED.

The connection table has not been initialized because conn_table was a NULL pointer.

Side effects Set the values of static variables in the AServer library to hold a pointer to the connection table and to hold the maximum number of connections.

6.6.37 as_set_dest

Name as_set_dest

Purpose To set the destination fields (gateway index and connection number) of a packet header when sending a packet.

Used by Any process.

Interface

```
int as_set_dest(packet_hdr hdr, int c_num)
```

Header file aslib.h

Parameters

Name	Type	In/Out	Description
hdr	packet_hdr	In/Out	The packet header which is to have its destination fields set.
c_num	int *	Out	The number of the connection for which information is required.

Return The access point number of the access point used to access the other end of the connection (e.g. as a parameter to as_rec_packet), or, if the connection is not in use, INVALID_AP_NUM.

Side effects None.

6.6.38 as_translate_pkt

Name as_translate_pkt

Purpose To translate certain packet types into components.

Description If the value of the parameter prev_res is not ASRC_SUCCESS, then as_translate_pkt immediately returns the value of prev_res. Otherwise, as_translate_pkt examines the packet and, for certain packet types and contents, translates the packet into a result code. as_translate_pkt returns ASRC_SUCCESS if no translation was performed. as_translate_pkt also processes disconnect and abort requests.

If received packets are buffered up by a callback function, then as_translate_pkt should be called for each buffered packet after it has been removed from the buffer, rather than when it is added to it.

Any packet received as a packet (e.g. using as_rec_packet) must be processed with as_translate_pkt.

Used by Any process.

Interface

```
asrc as_translate_pkt(asrcPrev_res, int ap_num,
                      packet_hdr hdr,
                      unsigned char *data,
                      as_readpktcb read_fn)
```

Header file aslib.h

Parameters

Name	Type	In/Out	Description
prev_res	asrc	In	A Previous result code. If Prev_res is not ASRC_SUCCESS then as_translate_pkt returns Prev_res. This enables as_translate_pkt to be called immediately after, for example, as_rec_packet.
ap_num	int	In	The access point number of the access point that the packet came from.
hdr	packet_hdr	In/Out	The header of the received packet.
data	unsigned char *	In/Out	The data of the received packet. The length of the data received is determined by using the as_hdr_get_len function on hdr
read_fn	as_readpktcb	In	A callback function called if any packets arrive that the function cannot handle. If NULL, then the packets are discarded.

Precondition None.

Return AServer result code.

Result **Return value:** `ASRC_SUCCESS`.

The packet did not need translating.

Return value: `ASRC_GOT_DISCONNECT_REQ`

A disconnect request was received, and the connection has been closed.

Return value: `ASRC_GOT_ABORT_REQ`

An abort request was received, and the connection has been closed.

Return value: any other result

Either an error occurred, or the packet was translated into a result code (e.g. `ASRC_TX_ERROR` or `ASRC_TERMINATE`).

Side effects If the packet is a disconnect or abort request then `as_translate_pkt` will close the connection.

6.6.39 as_unpack_hdr

Name as_unpack_hdr

Purpose To unpack values from all the fields of an AServer packet header.

Used by Any process.

Description Unpack the len, gi, c_num, type, and p_byte values from an AServer packet header. as_pack_hdr should have been used to pack the whole header, or the various as_hdr_set_* functions should be used to pack parts of the header.

Interface

```
void as_unpack_hdr(packet_hdr hdr,
                   data_len *len,
                   gateway_index *gi,
                   conn_num *c_num,
                   pkt_type *type,
                   protocol_byte *p_byte)
```

Header file aspack.h

Parameters

Name	Type	I/O	Description
hdr	packet_hdr	In	An AServer packet header
len	data_len *	Out	A pointer to the data length unpacked from hdr
gi	gateway_index *	Out	A pointer to the gateway index unpacked from hdr
c_num	conn_num *	Out	A pointer to the connection number unpacked from hdr
type	pkt_type *	Out	A pointer to the packet type unpacked from hdr
p_byte	protocol_byte *	Out	A pointer to the protocol byte unpacked from hdr

Precondition None.

Result The variables pointed to by len, gi, c_num, type, and p_byte are set to the various fields of the packet header.

Side effects None.

6.6.40 as_unpack_int32

Name as_unpack_int32

Purpose To unpack signed 32-bit integers from a buffer of unsigned chars.

Used by Any process.

Description Unpack an signed 32-bit integer from a buffer of unsigned chars. as_unpack_int32 should be used to unpack integers packed using as_pack_int32.

Interface
```
long as_unpack_int32(unsigned char *buf)
```

Header file aspack.h

Parameters

Name	Type	In/Out	Description
buf	unsigned char *	In	Pointer to the character buffer. The integer is stored as four unsigned chars, in little-endian format

Precondition The parameter buf must be a valid pointer into a buffer of unsigned chars. There must be at least three unsigned chars after that pointed to by buf. buf should point to a number packed using as_pack_int32.

Return The unpacked 32-bit integer.

Side effects None.

6.6.41 as_unpack_uint16

Name as_unpack_uint16

Purpose Used for unpacking unsigned 16-bit integers from a buffer of unsigned chars.

Used by Any process.

Description Unpacks an unsigned 16-bit integer from a buffer of unsigned chars. as_unpack_uint16 should be used to unpack integers packed using as_pack_uint16.

Interface AS_UINT16 as_unpack_uint16(unsigned char *buf)

Header file aspack.h

Parameters

Name	Type	In/Out	Description
buf	unsigned char *	In	Pointer to the character buffer. The integer is stored as two unsigned chars, in little-endian format

Precondition buf must be a valid pointer into a buffer of unsigned chars, there must be at least one unsigned char after that pointed to by buf. buf should point to a number packed using as_pack_uint16.

Result The unpacked 16-bit integer.

Side effects None.

6.6.42 as_unpack_uint32

Name `as_unpack_uint32`

Purpose Used for unpacking unsigned 32-bit integers from a buffer of unsigned chars.

Used by Any process.

Description Unpacks an unsigned 32-bit integer from a buffer of unsigned chars. `as_unpack_uint32` should be used to unpack integers packed using `as_pack_uint32`.

Interface `unsigned long as_unpack_uint32(unsigned char *buf)`

Header file `aspack.h`

Parameters

Name	Type	In/Out	Description
`buf`	`unsigned char *`	In	Address to read the unsigned int from. The int is stored as four unsigned chars, in little-endian format

Precondition `buf` must be a valid pointer into a buffer of unsigned chars, there must be at least three unsigned chars after that pointed to by `buf`. `buf` should point to a number packed using `as_pack_uint32`.

Return Unpacked 32-bit integer.

Side effects None.

Appendices

A AServer example code

This appendix gives full listings of the example programs using the AServer. These examples range from using single `iserver` services, through multiple `iserver` services, to purpose built services on the host. The first three examples show how to use the standard software processes provided and are described in Chapter 4. The last two examples show how to write a customized service and are described in Chapter 5.

Where the code is to run on the host, versions of the examples are given for a PC Windows host and for a Sun host.

The sources for all the examples can be found in the examples directory. The six examples are listed below.

- Hello displays the message "`Hello world`" 1000 times using the provided `iserver` converter and `iserver` service.
- Hello2 uses similar code to the Hello example, to have five "Hello" processes each displaying through an `iserver` converter and its own `iserver` service.
- Getkey reads keys from several different windows. This is done with multiple copies of a Getkey process each communicating through an `iserver` converter to its own `iserver` service.
- Echo echoes a message back to the client, to demonstrate how the communications bandwidth increases as the message size increases. It uses both an `iserver` service and a custom-built service.
- Print displays strings sent by a client using a custom-built service. The service uses packet level communications and read callback.

A.1 Running the examples

A.1.1 AServer database

The examples assume that the example AServer database file `aservdb` is being used, as provided with the examples in the same directory. The contents of file `aservdb` are shown in Figure A.1 for PC Windows users and in Figure A.2 for X-Windows users.

```
| inquest      | inquest.exe          | 1 |
| iserv        | iserv.exe            | 1 |
| autoiserver  | iserv.exe            | 1 |
| echo         | wecho.exe            | 1 |
| print        | wprinter.exe         | 4 |
```

Figure A.1 Example service database for PC Windows

```
# Service Name | Path              | Max Conns  | Extra Params
#--------------------------------------------------------------
| autoiserver  | iserv             | 1          |
| iserv        | xterm -e iserv    | 1          |
| inquest      | inquest_irun      | 1          |
| rspy         | rspy_irun         | 1          |
| iex          | iex_irun          | 1          |
| echo         | xterm -e echoserv | 1          |
| print        | xterm -e printer  | 4          |
```

Figure A.2 Example service database for X-Windows

A.1.2 Target processor

Each of the examples requires a single processor. The processor type given in the example configuration source (`.cfs`) files is IMS T805, but this can be changed by changing the line:

```
T805 (memory = 1M) board;
```

For example, when using an IMS T425, this would become:

```
T425 (memory = 1M) board;
```

The host is assumed to be connected to link 0 of the root processor. This can be changed by changing the `0` in the following line of the configuration source file to the number of the correct link.

```
connect host to board.link[0];
```

All the examples should run in 1 Mbyte of memory.

A.1.3 Environment

The environment variable `TRANSPUTER` should contain the name of the link to the target on which the examples will be run, as given in the AServer database.

The path of the directory containing the AServer header files and libraries must be added to the `ISEARCH` environment variable.

A.1.4 Building the examples

To build the examples, the ANSI C Toolset is required.

In addition, a host ANSI C compiler (e.g. `gcc` on a Sun or Watcom C/386 version 9 on a PC) is needed to build host services other than the `iserver` service, either those in the last two examples supplied or code written by the user. The compiler must be on the path and correctly set up.

A.2 Hello2 example

For a description and explanation of this example see Chapter 4, section 4.3. The source of this example is supplied in the examples directory.

A.2.1 The Hello client

The following is a listing of the main program source in the file `hello.c`.

```
#include <misc.h>
#include <stdlib.h>
#include <time.h>
#include <process.h>
#include <stdio.h>

int main()
{
  int i;

  for (i = 0; i < 1000; i++)
    printf("Hello world - via isconv, and aserv (%d)\n", i);

  ProcWait(CLOCKS_PER_SEC * 10);
  exit_terminate(EXIT_SUCCESS);
  return(0);
}
```

A.2.2 Configuring the Hello example

The following is the configuration source in the file `hello.cfs`.

```
/* hardware */

T805 (memory = 1M) board;

connect host to board.link[0];

/* software */

input from_host;
output to_host;

val gateway_fan_in 1;

process(interface (input from_link, output to_link,
                   input  from_processes[gateway_fan_in],
                   output to_processes[gateway_fan_in],
                   int max_mega_pkt_size_to_host = 1040),
        nodebug = true) gateway;

connect gateway.to_link to to_host;
connect from_host       to gateway.from_link;

process(interface (input as_in, output as_in,
                   input iserv_in, output iserv_out),
        stacksize = 5K) isconv;

connect gateway.from_processes[0] to isconv.as_out[0];
connect gateway.to_processes[0]   to isconv.as_in[0];
```

```
process (interface (input fs, output ts),
         heapsize = 10K, stacksize = 1K) hello;

connect isconv.iserv_in  to hello.ts;
connect isconv.iserv_out to hello.fs;/* mapping */

place from_host on host;
place to_host on host;

place gateway on board;
place isconv  on board;
place hello   on board;

use "gateway.cah" for gateway;
use "isconv.cax" for isconv;
use "hello.cax" for hello;
```

A.2.3 Building the Hello example

The makefile listed below is equivalent to the Sun `hello.mak`. A similar makefile `makefile.tx` is used in MS-DOS.

```
hello.btl: hello.cfb
     icollect hello.cfb -o hello.btl

hello.cfb: hello.cfs hello.cax
     icconf hello.cfs -o hello.cfb

hello.cax: hello.lnk hello.tax
     ilink -f hello.lnk -ta -x -o hello.cax

hello.tax: hello.c
     icc hello.c -g -ta -o hello.tax
```

A.2.4 Running the Hello example

Once built, the Hello example can be run using a command line like:

```
irun -si -sb hello.btl
```

A.3 Hello2 example

For a description and explanation of this example see Chapter 4, section 4.4. The source of this example is supplied in the examples directory.

A.3.1 Configuring the Hello2 example

The following is the configuration source in the file `hello2.cfs`.

```
/* hardware */

T800 (memory = 1M) board;

connect host to board.link[0];

/* software */

input from_host;
output to_host;

val num_hellos 5;

val gateway_fan_in num_hellos;

process(interface (input from_link, output to_link,
                   input  from_processes[gateway_fan_in],
                   output to_processes[gateway_fan_in],
                   int max_mega_pkt_size_to_host = 1040),
        nodebug = true) gateway;

connect gateway.to_link to to_host;
connect from_host       to gateway.from_link;

process(interface (input as_in, output as_out,
                   input iserv_in, output iserv_out),
        stacksize = 5K)
  isconv[num_hellos];

rep i = 0 for num_hellos
{
  connect gateway.from_processes[i] to isconv[i].as_out;
  connect gateway.to_processes[i]   to isconv[i].as_in;
}

process (interface (input fs, output ts),
         heapsize = 10K, stacksize = 1K) hello[num_hellos];

rep i = 0 for num_hellos
{
  connect isconv[i].iserv_in  to hello[i].ts;
  connect isconv[i].iserv_out to hello[i].fs;
}

/* mapping */

place from_host on host;
place to_host on host;

place gateway on board;
use "gateway.cah" for gateway;

rep i = 0 for num_hellos
{
  place isconv[i]  on board;
  place hello[i]   on board;
  use "isconv.cax" for isconv[i];
  use "hello.cax" for hello[i];
}
```

A.4 Getkey Example

For a description and explanation of this example see Chapter 4, section 4.5. The source of this example is supplied in the examples directory.

A.4.1 The main program

The following is a listing of the main program source in the file `getkey.c`.

```
#include <misc.h>
#include <stdlib.h>
#include <time.h>
#include <process.h>
#include <stdio.h>
#include <iocntrl.h>

int main()
{
  int key;

  set_abort_action(ABORT_HALT);

  printf("Hello world - via isconv, and aserv\n");
  printf("Type keys, and I\'ll echo them ('q' to quit,
                                        'e' to set error)\n");

  key = ' ';

  while (key != 'q')
  {
    key = getkey();

    if (key == -1)
      printf("No key pressed\n");
    else
      printf("Key is \'%c\'\n", key);

    if (key == 'e')
      abort();
  }

  ProcWait(CLOCKS_PER_SEC * 10);
  exit_terminate(EXIT_SUCCESS);
  return(0);
}
```

A.4.2 Configuring the getkey example

The following is the configuration source in the file `getkey.cfs`.

```
/* hardware */

T800 (memory = 1M) board;

connect host to board.link[0];

/* software */

input from_host;
output to_host;

val num_getkeys 3;

val gateway_fan_in num_getkeys;

process(interface (input from_link, output to_link,
                   input  from_processes[gateway_fan_in],
                   output to_processes[gateway_fan_in],
                   int max_mega_pkt_size_to_host = 1040),
        nodebug = true) gateway;

connect gateway.to_link to to_host;
connect from_host       to gateway.from_link;

process(interface (input as_in, output as_out,
                   input iserv_in, output iserv_out),
        stacksize = 5K) isconv[num_getkeys];

rep i = 0 for num_getkeys
{
  connect gateway.from_processes[i] to isconv[i].as_out[0];
  connect gateway.to_processes[i]   to isconv[i].as_in[0];
}

process (interface (input fs, output ts),
         heapsize = 50000, stacksize = 20000) getkey[num_getkeys];

rep i = 0 for num_getkeys
{
  connect isconv[i].iserv_in  to getkey[i].ts;
  connect isconv[i].iserv_out to getkey[i].fs;
}

/* mapping */

place from_host on host;
place to_host on host;

place gateway on board;
use "gateway.cah" for gateway;

rep i = 0 for num_getkeys
{
  place isconv[i]  on board;
  place getkey[i]   on board;
  use "isconv.cax" for isconv[i];
  use "getkey.cax" for getkey[i];
}
```

A.5 Echo example

The echo example is described in Chapter 5 in section 5.7. The source is supplied in the examples directory.

A.5.1 Echo client in ANSI C

The following is a listing of the ANSI C version of the Echo client, which is in the file `echo.c`:

```
/* Copyright SGS-THOMSON Microelectronics Limited 1992, 1993 */

/* @(#) Module: echo.c, revision 1.9 of 10/17/93 */

#include <misc.h>
#include <stdlib.h>
#include <time.h>
#include <process.h>
#include <stdio.h>
#include <aslib.h>

#define CONN_TABLE_SIZE      1
#define MAX_ECHO_BUF_SIZE   (32 * 1024)

asrc echo_test(int conn_num, unsigned char *buf,
               unsigned long msg_len, int num_msgs,
               const as_bool time_operation)
{
  int count = num_msgs;
  asrc res = ASRC_SUCCESS;
  asrc rec_res;
  unsigned long rec_size;
  int i, start_time, end_time;
  unsigned char *bufp;

  printf("echo_test: message len %ld, num. messages %d\n",
         msg_len, num_msgs);

  start_time = ProcTime();

  while ((count > 0) && (res == ASRC_SUCCESS))
  {
    if (AS_BOOL_TO_INT(time_operation) == AS_FALSE)
    {
      for (i = 0, bufp = buf; i < MAX_ECHO_BUF_SIZE / 2; i++, bufp += 2)
        as_pack_uint16(bufp, i);
    }

    res = as_sendmessage(conn_num, msg_len, buf, NULL);

    rec_size = msg_len;

    if (res == ASRC_SUCCESS)
      rec_res = as_recmessage(conn_num, &rec_size, buf);

    if ((res == ASRC_SUCCESS) && (rec_res != ASRC_SUCCESS))
      res = rec_res;

    if (res == ASRC_SUCCESS)
    {
      if (rec_size != msg_len)
      {
        printf("received message of different length\n");
        res = ASRC_FAILED;
      }
    }

    count --;
  }
```

```
  end_time = ProcTime();

  printf("echo_test: result = %s\n", as_asrc_to_str(res));

  if (res == ASRC_SUCCESS)
  {
    float time_taken = ((float) ProcTimeMinus(end_time, start_time)) /
                       CLOCKS_PER_SEC;
    long num_bytes = msg_len * num_msgs;

    printf("%ld bytes echoed in %f seconds\n", num_bytes, time_taken);
    printf("average send/receive speed is %f bytes/sec\n\n",
           (float) num_bytes * 2.0 / time_taken);
  }

  return(res);
}

int main()
{
  conn_id conn_table[CONN_TABLE_SIZE];
  asrc res;
  as_bool connected = AS_FALSE;
  int conn_num, i;
  unsigned char *buf = malloc(MAX_ECHO_BUF_SIZE);
  Channel *chan_aserv_in  = (Channel *)get_param(3);
  Channel *chan_aserv_out = (Channel *)get_param(4);

  res = as_apstart(chan_aserv_in, chan_aserv_out);

  for (i = 0; i < 10; i++)
    printf("Echo saying hello - via isconv\n");

  if (res != ASRC_SUCCESS)
    printf("Failed to initialise access points\n");
  else if (buf == NULL)
    printf("Failed to malloc echo buffer\n");
  else
  {
    res = as_setconntable(CONN_TABLE_SIZE, conn_table);

    printf("echo: as_setconntable returned %s\n", as_asrc_to_str(res));

    res = asc_connect("echo", "", 0, &conn_num, NULL);

    printf("echo: as_connect returned %s (%d)\n", as_asrc_to_str(res), res);

    if (res == ASRC_SUCCESS)
      connected = AS_TRUE;

    (void) echo_test(conn_num, buf,       512, 200, AS_TRUE);
    (void) echo_test(conn_num, buf,      1024, 100, AS_TRUE);
    (void) echo_test(conn_num, buf,      1500, 100, AS_TRUE);
    (void) echo_test(conn_num, buf,  2 * 1024,  50, AS_TRUE);
    (void) echo_test(conn_num, buf,  4 * 1024,  50, AS_TRUE);
    (void) echo_test(conn_num, buf,  8 * 1024,  50, AS_TRUE);

    if (AS_BOOL_TO_INT(connected))
      (void) asc_disconnect(conn_num, NULL);

  }

  if (buf != NULL)
    free(buf);

  printf("About to terminate\n");
  ProcWait(CLOCKS_PER_SEC * 10);
  exit_terminate(EXIT_SUCCESS);
  return(0);
}
```

A.5.2 Echo client in occam

The following is a listing of the occam version of the Echo client, which is in the file `oecho.occ`:

```
#INCLUDE "hostio.inc"
#INCLUDE "gateway.inc"
#USE "hostio.lib"
#USE "callc.lib"

#PRAGMA TRANSLATE as.apstart "oc_as_apstart"
#PRAGMA TRANSLATE as.setconntable "oc_as_setconntable"
#PRAGMA TRANSLATE asc.connect "oc_asc_connect"
#PRAGMA TRANSLATE asc.disconnect "oc_asc_disconnect"
#PRAGMA TRANSLATE as.sendmessage "oc_as_sendmessage"
#PRAGMA TRANSLATE as.recmessage  "oc_as_recmessage"
#PRAGMA TRANSLATE as.pack.uint16 "oc_as_pack_uint16"

#PRAGMA EXTERNAL "PROC as.apstart(VAL INT GSB, CHAN OF ASPROT in, out, INT result) = 2048"
#PRAGMA EXTERNAL "PROC as.setconntable(VAL INT GSB, [][3]INT32 conn.table,
                                                        INT result) = 2048"
#PRAGMA EXTERNAL "PROC asc.connect(VAL INT GSB, VAL []BYTE service.name,
                    VAL []BYTE service.params, VAL INT ap.num, INT c.num, result) = 2048"
#PRAGMA EXTERNAL "PROC asc.disconnect(VAL INT GSB, c.num, result) = 2048"
#PRAGMA EXTERNAL "PROC as.recmessage(VAL INT GSB, VAL INT c.num, []BYTE data,
                                                    INT32 data.size, INT result) = 40960"
#PRAGMA EXTERNAL "PROC as.sendmessage(VAL INT GSB, VAL INT c.num, []BYTE data,
                                                  VAL INT32 data.size, INT result) = 40960"
#PRAGMA EXTERNAL "PROC as.pack.uint16(VAL INT GSB, []BYTE buf, INT32 num) = 2048"

VAL INT    ASRC.SUCCESS       IS 0 :
VAL INT    ASRC.FAILED        IS 128 :
VAL INT    CONN.TABLE.SIZE    IS 1 :
VAL INT    MAX.ECHO.BUF.SIZE IS (32 * 1024) :
VAL REAL32 CLOCKS.PER.SEC     IS 15625.0(REAL32) :

PROC echo(CHAN OF SP fs, ts, CHAN OF ASPROT as.in, as.out)

  PROC delay(VAL INT interval)
    TIMER clock :
    INT timenow :
    SEQ
      clock ? timenow
      clock ? AFTER timenow PLUS interval
  :
```

```
PROC echo.test(VAL INT GSB, CHAN OF SP fs, ts, INT conn.num, []BYTE buf,
               VAL INT32 msg.len, VAL INT num.msgs)
  INT start.time, end.time :
  INT count, res, rec.res :
  INT32 rec.size :
  REAL32 time.taken :
  INT32 num.bytes :
  TIMER clock :

  SEQ
    count := num.msgs
    so.write.string(fs, ts, "echo test: message len ")
    so.write.int32(fs, ts, msg.len, 4)
    so.write.string(fs, ts, ", num. messages ")
    so.write.int(fs, ts, num.msgs, 4)
    so.write.nl(fs, ts)

    clock ? start.time

    res := ASRC.SUCCESS
    WHILE (count > 0) AND (res = ASRC.SUCCESS)
      SEQ
        as.sendmessage(GSB, conn.num, buf, msg.len, res)
        rec.size := (INT32 msg.len)
        IF
          res = ASRC.SUCCESS
            as.recmessage(GSB, conn.num, buf, rec.size, rec.res)
          TRUE
            so.write.string(fs, ts, "as.sendmessage failed*n")
        -- END IF
        IF
          (res = ASRC.SUCCESS) AND (rec.res <> ASRC.SUCCESS)
            res := rec.res
          TRUE
            SKIP
        -- END IF
        IF
          (res = ASRC.SUCCESS) AND (rec.size <> msg.len)
            SEQ
              so.write.string(fs, ts, "received message of different lengths*n")
              res := ASRC.FAILED
            -- END SEQ
          TRUE
            SKIP
        -- END IF
        count := count - 1
      -- END SEQ
    -- END WHILE

    clock ? end.time

    IF
      res = ASRC.SUCCESS
        SEQ
          time.taken := (REAL32 ROUND (end.time MINUS start.time)) / CLOCKS.PER.SEC
          num.bytes  := msg.len * (INT32 num.msgs)
          so.write.int32(fs, ts, num.bytes, 6)
          so.write.string(fs, ts, " bytes echoed in ")
          so.write.real32(fs, ts, time.taken, 6, 6)
          so.write.string(fs, ts, " seconds*n")
          so.write.string(fs, ts, "average send/receive speed is ")
          so.write.real32(fs, ts, ((REAL32 ROUND (num.bytes)) * 2.0(REAL32)) /
                                                                time.taken, 6, 6)
          so.write.string(fs, ts, " bytes/sec*n")
        -- END SEQ
      TRUE
        SKIP
    -- END IF

  -- END SEQ

:
```

```
PROC main(INT GSB, CHAN OF SP fs, ts, CHAN OF ASPROT as.in, as.out)
  INT res :
  VAL INT CONN.TABLE.SIZE IS 1 :
  [CONN.TABLE.SIZE][3]INT32 conn.table :
  [MAX.ECHO.BUF.SIZE]BYTE buf :
  INT conn.num :
  BOOL connected :
  SEQ
    connected := FALSE

    so.write.string(fs, ts, "Occam Echo*n")
    as.apstart(GSB, as.in, as.out, res)
    IF
      res <> ASRC.SUCCESS
        so.write.string(fs, ts, "Failed to initialise access points*n")
      TRUE
        SEQ
          SEQ i = 0 FOR 10
            so.write.string(fs, ts, "Echo saying hello - via isconv*n")

          as.setconntable(GSB, conn.table, res)
          so.write.string(fs, ts, "echo: as.setconntable returned ")
          so.write.int(fs, ts, res, 4)
          so.write.nl(fs, ts)

          asc.connect(GSB, "echo", "", 0, conn.num, res)
          so.write.string(fs, ts, "echo: as. returned ")
          so.write.int(fs, ts, res, 4)
          so.write.nl(fs, ts)

          IF
            res = 0
              connected := TRUE
            TRUE
              SKIP
          -- END IF

          SEQ i = 0 FOR SIZE buf
            buf[i] := 255(BYTE)
          -- Test code
          echo.test(GSB, fs, ts, conn.num, buf,              512(INT32), 200)
          echo.test(GSB, fs, ts, conn.num, buf,             1024(INT32), 100)
          echo.test(GSB, fs, ts, conn.num, buf,             1025(INT32), 100)
          echo.test(GSB, fs, ts, conn.num, buf,             1500(INT32), 100)
          echo.test(GSB, fs, ts, conn.num, buf, 2(INT32) * 1024(INT32), 100)
          echo.test(GSB, fs, ts, conn.num, buf, 4(INT32) * 1024(INT32), 100)
          echo.test(GSB, fs, ts, conn.num, buf, 8(INT32) * 1024(INT32), 100)

          IF
            connected
              asc.disconnect(GSB, conn.num, res)
            TRUE
              SKIP
          -- END IF
        -- END SEQ
    -- END IF

  -- END SEQ
:
```

```
  INT GSB, required.size :
  VAL static.size IS 4096 :
  VAL heap.size   IS 4096 :
  [static.size]INT static.area :
  [heap.size]INT   heap.area :

  SEQ
    init.static(static.area, required.size, GSB)
    IF
      required.size > static.size
        so.write.string(fs, ts, "Error initialising static*n")
      TRUE
        SEQ
          init.heap(GSB, heap.area)
          main(GSB, fs, ts, as.in, as.out)

        -- END SEQ

    -- END IF
    terminate.heap.use(GSB)
    terminate.static.use(GSB)
    so.write.string(fs, ts, "About to terminate*n")
    delay(100000)
    so.exit(fs, ts, sps.success)
  -- END SEQ
:
```

A.5.3 Configuration of the Echo client

The following is a listing of the configuration source file `echo.cfs`:

```
/* hardware */

T800 (memory = 1M) board;

connect host to board.link[0];

/* software */

input from_host;
output to_host;

val gateway_fan_in 2;

process(interface (input from_link, output to_link,
                   input  from_processes[gateway_fan_in],
                   output to_processes[gateway_fan_in],
                   int max_mega_pkt_size_to_host = 1040),
        nodebug = true) gateway;

connect gateway.to_link to to_host;
connect from_host       to gateway.from_link;

process(interface (input as_in, output as_out,
                   input iserv_in, output iserv_out),
        stacksize = 5K) isconv;

connect gateway.from_processes[0] to isconv.as_out[0];
connect gateway.to_processes[0]   to isconv.as_in[0];

process (interface (input fs, output ts,
                    input as_in, output as_out),
         heapsize = 50000, stacksize = 20000) echo;

connect isconv.iserv_in  to echo.ts;
connect isconv.iserv_out to echo.fs;

connect gateway.from_processes[1] to echo.as_out[0];
connect gateway.to_processes[1]   to echo.as_in[0];

/* mapping */

place from_host on host;
place to_host on host;

place gateway on board;
place isconv  on board;
place echo    on board;

use "gateway.cah" for gateway;
use "isconv.cax" for isconv;
use "echo.cax" for echo;
```

A.5.4 Building the Echo target code

On a PC, the Echo ANSI C target code can be built by:

- running the batch file `mkecho.bat` or
- by using a make utility with the makefile `echo.mak`.

The Echo occam target code can be built by:

- running the batch file `mkoecho.bat` or

- by using a make utility with the makefile `oecho.mak`.

A.5.5 Echo service for PC host

The following is a listing of the Echo service for Windows on a PC host. The source is in file `wechosrv.c`:

```
/*********** Copyright SGS-THOMSON Microelectronics Limited 1994 *************/

static char *rcsid = "@(#) $RCSfile: wechosrv.c,v $ $Revision: 1.1 $ of $Date: 1994/08/03
                13:30:08 $ Copyright SGS-THOMSON Microelectronics Limited 1994";

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <stdarg.h>
#include <aslib.h>

/*****************************************************************************
 *
 *  Definitions
 *
 *****************************************************************************/

static char applicationName[32] = "EchoServer";
#define MAIN_WINDOW_TITLE         applicationName
#define MAX_ECHO_SIZE             32768
#define MAX_ECHO_CONNECTS         1

#ifdef __WINDOWS__386__
#define _EXPORT
#else
#define _EXPORT __export
#endif

/*****************************************************************************
 *
 *  Global Data
 *
 *****************************************************************************/

static HANDLE      instanceHandle;
static HWND        windowHandle;

/*****************************************************************************
 *
 *  Code
 *
 *****************************************************************************/

extern LONG _EXPORT FAR PASCAL WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam);

static void Debug(const char *fmt, ...)

{
  static char b[1024];
  static char b2[1024];
  va_list args;

  va_start(args, fmt);
  vsprintf(b, fmt, args);
  sprintf(b2, "[%x] - %s", instanceHandle, b);
  OutputDebugString(b2);
  va_end(args);
}
```

```
void Error(const char *fmt, ...)
{
  char b[1024];
  va_list args;

  va_start(args, fmt);
  vsprintf(b, fmt, args);
  MessageBox(windowHandle, b, applicationName, MB_ICONEXCLAMATION);
  va_end(args);
  DestroyWindow(windowHandle);
}

LONG _EXPORT FAR PASCAL WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{
  switch (message) {

    case WM_CREATE :
      break;

    case WM_DESTROY :
      as_apfinish(); /* Close down the aserver library before we finish */
      PostQuitMessage(0);
      break;

    default :
      return (DefWindowProc(hWnd, message, wParam, lParam));
  }

  return(NULL);
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)

{
  WNDCLASS        wc;
  MSG             msg;
  asrc            res = ASRC_SUCCESS;
  int             connNum;
  conn_id         connTable[MAX_ECHO_CONNECTS];
  HGLOBAL         hEchoBuff;
  unsigned char *pEchoBuff;
  packet_hdr      hdr;
  unsigned long   messLen;

#ifdef __WINDOWS_386__
  sprintf(applicationName, "%s%d", applicationName, hInstance);
#else
  if(!hPrevInstance) {
#endif
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC)WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(hInstance, "GenericIcon");
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName =  NULL;
    wc.lpszClassName = applicationName;

    if(!RegisterClass(&wc))
      return FALSE;
#ifndef __WINDOWS_386__
  }
#endif

  instanceHandle = hInstance;
  windowHandle =
    CreateWindow(applicationName, MAIN_WINDOW_TITLE, WS_OVERLAPPEDWINDOW,
                 CW_USEDEFAULT, CW_USEDEFAULT, 350, 350,
                 NULL, NULL, hInstance, NULL);
```

```
  if(windowHandle) {
    ShowWindow(windowHandle, nCmdShow);
    UpdateWindow(windowHandle);
  } else {
    return FALSE;
  }

  res = as_apstart();
  if (res == ASRC_SUCCESS)
  {
    hEchoBuff = LocalAlloc(GHND, MAX_ECHO_SIZE);
    if (hEchoBuff == NULL) {
      Error("Failed to allocate an echo buffer of size %d bytes\n",
            MAX_ECHO_SIZE);
      res = ASRC_FAILED;
    }
  }
  else
    Error("Failed to initialise access points\n");

  if (res == ASRC_SUCCESS)
    res = as_setconntable(MAX_ECHO_CONNECTS, connTable);

  if (res != ASRC_SUCCESS)
    Error("Failed to as_setconntable - res = %s\n", as_asrc_to_str(res));
  else {
    pEchoBuff = (unsigned char *)LocalLock(hEchoBuff);
    res = ass_acceptconnect(0, &connNum, hdr, pEchoBuff);
    LocalUnlock(hEchoBuff);
  }

  /* Main Windows message loop */
  while(GetMessage(&msg, NULL, NULL, NULL)) {

    TranslateMessage(&msg);
    DispatchMessage(&msg);

    if(res == ASRC_SUCCESS)
    {
      /* Read message and echo it */
      messLen = MAX_ECHO_SIZE;
      pEchoBuff = (unsigned char *)LocalLock(hEchoBuff);
      res = as_recmessage(connNum, &messLen, pEchoBuff);
      if (res == ASRC_SUCCESS)
        res = as_sendmessage(connNum, messLen, pEchoBuff, NULL);
      LocalUnlock(hEchoBuff);
      if (res != ASRC_SUCCESS) {
        DestroyWindow(windowHandle);
      }
    }
  }

  return (msg.wParam);
}
```

A.5.6 Building the Echo host code

On a PC, the Echo and Print host code can be built by using a make utility:

`nmake -f ms.mak` (Microsoft make)

`wmake -f wat.mak` (Watcom make)

A.5.7 Echo service for Sun host

The following is a listing of the Echo service for X-Windows on a Sun host. The source is in file `echoserv.c`.

```
/* Copyright SGS-THOMSON Microelectronics Limited 1992, 1993 */

/* @(#) Module: echoserv.c, revision 1.7 of 10/17/93 */

static const char prog_name[]    = "echo";
static const char prog_version[] = "echo: version 1.1";

#include <stdio.h>
#include <aslib.h>

#define MAX_ECHO_SIZE       (128 * 1024)

#define MAX_ECHO_CONNECTS   1


int main()
{
  asrc res = ASRC_SUCCESS;
  int conn_num;
  conn_id conn_table[MAX_ECHO_CONNECTS];
  unsigned char *echo_buf = NULL;

  printf("The Echo Service (%s)....\n", prog_version);

  res = as_apstart();

  if (res == ASRC_SUCCESS)
  {
    echo_buf = (unsigned char *) malloc(MAX_ECHO_SIZE);

    if (echo_buf == NULL)
    {
      printf("Failed to malloc a echo buffer of size %d bytes\n",
             MAX_ECHO_SIZE);
      res = ASRC_FAILED;
    }
  }
  else
    printf("Failed to initialise access pointes\n");

  if (res == ASRC_SUCCESS)
    res = as_setconntable(MAX_ECHO_CONNECTS, conn_table);

  if (res != ASRC_SUCCESS)
    printf("Failed to as_setconntable - res = %s\n", as_asrc_to_str(res));
  else
  {
    packet_hdr hdr;

    res = ass_acceptconnect(0, &conn_num, hdr, echo_buf);
    /*printf("ass_acceptconnect res = %s\n", as_asrc_to_str(res));*/
  }

  while (res == ASRC_SUCCESS)
  {
    unsigned long mess_len = MAX_ECHO_SIZE;

    res = as_recmessage(conn_num, &mess_len, echo_buf);

    /*printf("as_recmessage returned %s\n", as_asrc_to_str(res));*/

    if (res == ASRC_SUCCESS)
    {
      /*printf("Echoing message of length %ld\n", mess_len);*/
      res= as_sendmessage(conn_num, mess_len, echo_buf, NULL);
    }
  }

  printf("echoserv finishing\n");

  printf("Zzzzzzz....\n");
  sleep(10);

  return(0);
}
```

A.6 Print example

The print example is described in Chapter 5 in section 5.9. The source is supplied in the examples directory.

A.6.1 The Print client

The following is a listing of the Print client which is in the file `print.c`:

```
#include <misc.h>
#include <stdlib.h>
#include <stdarg.h>
#include <time.h>
#include <process.h>
#include <stdio.h>
#include <string.h>
#include <aslib.h>
#include "print.h"

#define CONN_TABLE_SIZE   1
#define PR_SEND_PKT_SIZE  (MAX_PACKET_SIZE)

static asrc pr_sendmessage(int c_num,
                           unsigned long data_size, unsigned char *data,
                           as_readpktcb read_fn);
static asrc pr_print(int c_num, const char format[], ...);
static asrc pr_flush(int c_num);

static asrc pr_sendmessage(int c_num,
                           unsigned long data_size, unsigned char *data,
                           as_readpktcb read_fn)
{
  packet_hdr send_hdr;
  unsigned long len_to_go = data_size;
  asrc res = ASRC_SUCCESS;
  int ap_num = as_set_dest(send_hdr, c_num);
  as_bool sent_a_pkt = AS_FALSE;   /* allows sending 0 length messages */

  as_hdr_set_type(send_hdr, ASPT_DATA);
  as_hdr_set_p_byte(send_hdr, (protocol_byte) 0);

  while ((res == ASRC_SUCCESS) &&
         ((len_to_go > 0UL) || (sent_a_pkt == AS_FALSE))) {
    data_len p_len;

    if (len_to_go > PR_SEND_PKT_SIZE) {
      p_len = (data_len) PR_SEND_PKT_SIZE;
    } else {
      p_len = (data_len) len_to_go;
      as_hdr_set_type(send_hdr, ASPT_DATA | ASPH_EOM_MASK);
    }

    as_hdr_set_len(send_hdr, p_len);
    res = as_send_packet(ap_num, send_hdr, data, read_fn);
    len_to_go -= (unsigned long) p_len;
    data += p_len;
    sent_a_pkt = AS_TRUE;
  }

  return(res);
}
```

A.6 Print example

```
static asrc pr_print(int c_num, const char format[], ...)
{
  asrc res;
  unsigned char msg[MAX_PRINT_MSG_LEN];
  char *str = (char *) (msg + 1);
  va_list argp;

  msg[0] = PR_STRING;
  va_start(argp, format);
  vsprintf(str, format, argp);
  va_end(argp);
  res = pr_sendmessage(c_num, 2L + (long) strlen(str), msg, NULL);
  return (res);
}

static asrc pr_flush(int c_num)
{
  asrc res;
  unsigned char msg[1];
  msg[0] = PR_FLUSH;
  res = as_sendmessage(c_num, 1, msg, NULL);
  if (res == ASRC_SUCCESS) {
    unsigned long msg_len = 1UL;
    res = as_recmessage(c_num, &msg_len, msg);
  }
  return(res);
}

int main(void)
{
  conn_id conn_table[CONN_TABLE_SIZE];
  asrc res;
  as_bool connected = AS_FALSE;
  int conn_num;
  int i = 0;
  Channel *chan_aserv_in  = (Channel *)get_param(1);
  Channel *chan_aserv_out = (Channel *)get_param(2);
  int *hello_num = get_param(3);

  res = as_apstart(chan_aserv_in, chan_aserv_out);
  if (res == ASRC_SUCCESS)
    res = as_setconntable(CONN_TABLE_SIZE, conn_table);
  if (res == ASRC_SUCCESS)
    res = asc_connect("print", "", 0, &conn_num, NULL);
  if (res == ASRC_SUCCESS)
    connected = AS_TRUE;

  while ((res == ASRC_SUCCESS) && (i < 1000)) {
    res = pr_print(conn_num,
                   "Hello world (hello no. = %d) - via printer (i = %d)\n",
                   *hello_num, i);
    if ((res == ASRC_SUCCESS) && ((i % 100) == 0))
      res = pr_flush(conn_num);
    i++;
  }

  if ((res == ASRC_SUCCESS) && AS_BOOL_TO_INT(connected))
    (void) asc_disconnect(conn_num, NULL);

  return(0);
}
```

A.6.2 Configuring the Print client

The following is a listing of the configuration source file, `print.cfs`.

```
/* hardware */

T800 (memory = 1M) board;

connect host to board.link[0];

/* software */

input from_host;
output to_host;

val num_prints 10;

val gateway_fan_in num_prints;

process(interface (input from_link, output to_link,
                   input  from_processes[gateway_fan_in],
                   output to_processes[gateway_fan_in],
                   int max_mega_pkt_size_to_host = 1040),
        nodebug = true) gateway;

connect gateway.to_link to to_host;
connect from_host       to gateway.from_link;

process (interface (input as_in, output as_out, int hello_num),
         heapsize = 50000, stacksize = 20000) print[num_prints];

rep i = 0 for num_prints
{
  connect gateway.from_processes[i] to print[i].as_out[0];
  connect gateway.to_processes[i]   to print[i].as_in[0];
  print[i] (hello_num = i);
}

/* mapping */

place from_host on host;
place to_host on host;

place gateway on board;
use "gateway.cah" for gateway;

rep i = 0 for num_prints
{
  place print[i]   on board;
  use "print.cax" for print[i];
}
```

A.6.3 Building the Print target code

On a PC the Print target code can be built by:

- running the batch file `mkprint.bat` or
- using a make utility with the makefile `print.mak`.

A.6.4 The Print service for PC hosts

The following is a listing of the Print service for Windows on a PC host, which is in the file `wprinter.c`:

```
/************ Copyright SGS-THOMSON Microelectronics Limited 1994 ****************/

static char *rcsid = "@(#) $RCSfile: printer.c,v $ $Revision: 1.1.1.1 $ of $Date:
                1994/07/19 08:10:00 $ Copyright SGS-THOMSON Microelectronics Limited 1994";
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <stdarg.h>
#include <aslib.h>
#include <assert.h>
#include <string.h>
#include "print.h"

/*****************************************************************************
 *
 *  Definitions
 *
 *****************************************************************************/

#define WM_USER_INIT (WM_USER + 1)

static char applicationName[32] = "wPrinter";
#define MAIN_WINDOW_TITLE applicationName

#define MAX_PRINT_CONNECTS   8

#ifdef __WINDOWS_386__
#define _EXPORT
#else
#define _EXPORT __export
#endif

/*****************************************************************************
 *
 *  Structures
 *
 *****************************************************************************/

typedef struct {
  int             len;
  unsigned char   data[MAX_PRINT_MSG_LEN];
}               msg_buf;

struct pkt_list_st;
typedef struct pkt_list_st pkt_list_t;

struct pkt_list_st {
  int             ap_num;
  packet_hdr      hdr;
  unsigned char  *data;
  pkt_list_t     *next;
};

/*****************************************************************************
 *
 *  Global Data
 *
 *****************************************************************************/

static msg_buf     msgBuffers[MAX_PRINT_CONNECTS];

static pkt_list_t *pktListFirst, *pktListLast;
static int         pktListLength;

conn_id            connTable[MAX_PRINT_CONNECTS];
int                totalConnects = 0;

long               connectRequests = 0;
long               flushCount[MAX_PRINT_CONNECTS];
long               disconnectRequests = 0;
long               stringCount[MAX_PRINT_CONNECTS];
```

```
static HANDLE       instanceHandle;
static HWND         windowHandle;

/*****************************************************************************
 *
 *  Packet List Services
 *
 *****************************************************************************/

static asrc
add_to_pkt_list(int ap_num,
                packet_hdr hdr, unsigned char *data)
{
  asrc              res = ASRC_SUCCESS;
  int               len = (int) as_hdr_get_len(hdr);
  pkt_list_t      *entry = (pkt_list_t *) malloc(sizeof(pkt_list_t));

  assert(ap_num == 0);
  if (entry == NULL)
    res = ASRC_OUT_OF_MEMORY;
  else {
    entry->data = (unsigned char *) malloc(len);
    if (entry->data == NULL) {
      res = ASRC_OUT_OF_MEMORY;
      abort();
      free(entry);
    }
  }

  if (res == ASRC_SUCCESS) {
    entry->ap_num = ap_num;
    memcpy(entry->hdr, hdr, sizeof(packet_hdr));
    memcpy(entry->data, data, len);
    entry->next = NULL;
    if (pktListFirst == NULL)
      pktListFirst = entry;
    else
      pktListLast->next = entry;
    pktListLast = entry;
    pktListLength++;
  }
  return (res);
}

static asrc
remove_from_pkt_list(int *ap_num,
                     packet_hdr hdr, unsigned char *data)
{
  asrc              res = ASRC_SUCCESS;

  if (pktListFirst == NULL) {
    res = ASRC_FAILED;
  } else {
    pkt_list_t      *entry = pktListFirst;
    *ap_num = entry->ap_num;
    memcpy(hdr, entry->hdr, sizeof(packet_hdr));
    memcpy(data, entry->data, (int) as_hdr_get_len(hdr));
    pktListFirst = entry->next;
    free(entry->data);
    free(entry);
    pktListLength--;
    assert(*ap_num == 0);
  }
  return (res);
}
```

A.6 Print example

```
/*****************************************************************************
 *
 *  Windows Code
 *
 *****************************************************************************/

extern LONG _EXPORT FAR PASCAL WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam);

static void Debug(const char *fmt, ...)

{
  static char b[1024];
  static char b2[1024];
  va_list args;

  va_start(args, fmt);
  vsprintf(b, fmt, args);
  sprintf(b2, "[%x] - %s", instanceHandle, b);
  OutputDebugString(b2);
  va_end(args);
}

void Error(const char *fmt, ...)
{
  char b[1024];
  va_list args;

  va_start(args, fmt);
  vsprintf(b, fmt, args);
  MessageBox(windowHandle, b, applicationName, MB_ICONEXCLAMATION);
  va_end(args);
  DestroyWindow(windowHandle);
}

BOOL FirstInstance(HANDLE hInstance)

{
  WNDCLASS  wc;

  wc.style = CS_HREDRAW | CS_VREDRAW;
  wc.lpfnWndProc = (WNDPROC)WndProc;
  wc.cbClsExtra = 0;
  wc.cbWndExtra = 0;
  wc.hInstance = hInstance;
  wc.hIcon = LoadIcon(hInstance, "GenericIcon");
  wc.hCursor = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = GetStockObject(WHITE_BRUSH);
  wc.lpszMenuName =  NULL;
  wc.lpszClassName = applicationName;

  return(RegisterClass(&wc));
}

BOOL AnyInstance(HANDLE hInstance, int nCmdShow, LPSTR lpCmdLine)

{
  HWND hWnd;
  BOOL result = FALSE;

  hWnd = CreateWindow(applicationName, MAIN_WINDOW_TITLE, WS_OVERLAPPEDWINDOW,
                      CW_USEDEFAULT, CW_USEDEFAULT, 350, 350,
                      NULL, NULL, hInstance, NULL);

  if(hWnd) {
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    result = TRUE;
  }

  windowHandle = hWnd;
  instanceHandle = hInstance;

  return(result);
}
```

```
/*****************************************************************************
 *
 *  Main Code
 *
 *****************************************************************************/

void AserverCallback(short aserverEvent)
{
  BOOL            anotherPacket;
  int             apNum, connNumber;
  packet_hdr      hdr;
  pkt_type        type;
  unsigned char   data[MAX_PACKET_SIZE];
  asrc            res;

  switch(aserverEvent) {
    case ASCB_RECPKT :
      /* A packet is ready for reading */
      apNum = 0;

      /* Read it */
      res = as_rec_packet(apNum, hdr, data);

      /* Process new packet, and keep processing packets *
       * until the packet list is empty.                 */
      do {
        anotherPacket = FALSE;
        res = as_translate_pkt(res, apNum, hdr, data, add_to_pkt_list);
        type = as_hdr_get_type(hdr);
        if (res == ASRC_SUCCESS) {
          switch (type & ASPH_TYPE_MASK) {
            case ASPT_CONNECT_REQ:
              res = ass_processconnect(apNum, hdr, data, &connNumber,
                                       ASRC_SUCCESS, add_to_pkt_list);
              totalConnects++;
              connectRequests++;
              InvalidateRect(windowHandle, NULL, FALSE);
              break;

            case ASPT_DATA : {
              int             connNumber = (int) as_hdr_get_conn_num(hdr);
              int             len = (int) as_hdr_get_len(hdr);

              memcpy(msgBuffers[connNumber].data + msgBuffers[connNumber].len, data, len);
              msgBuffers[connNumber].len += len;

              if (ASPH_IS_EOM(type)) {
                switch (msgBuffers[connNumber].data[0]) {

                  case PR_STRING:
                    stringCount[connNumber]++;
                    InvalidateRect(windowHandle, NULL, FALSE);
                    break;
```

```
                case PR_FLUSH:
                  res = as_sendmessage(connNumber, 0, NULL, add_to_pkt_list);
                  flushCount[connNumber]++;
                  InvalidateRect(windowHandle, NULL, FALSE);
                  break;
              }
              msgBuffers[connNumber].len = 0;
            }
            break;
          }
        }
      } else if (res == ASRC_GOT_DISCONNECT_REQ) {
        totalConnects--;
        disconnectRequests++;
        InvalidateRect(windowHandle, NULL, FALSE);
        if(totalConnects == 0)
          DestroyWindow(windowHandle);
      }

      if (pktListLength > 0) {
        res = remove_from_pkt_list(&apNum, hdr, data);
        anotherPacket = TRUE;
        assert(apNum == 0);
      }

    } while (anotherPacket);
    break;

  case ASCB_SIGINT :
    /* "SIGINT" signal received.  Destroy application. */
    DestroyWindow(windowHandle);
    break;
  }
}

LONG _EXPORT FAR PASCAL WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{
  HDC hdc;
  PAINTSTRUCT ps;
  asrc res;
  int i;
  char b[100];

  switch (message) {

    case WM_CREATE :
      break;

    case WM_USER_INIT :
      /* Initialise everything. */
      pktListFirst = NULL;
      pktListLength = 0;

      for (i = 0; i < MAX_PRINT_CONNECTS; i++) {
        msgBuffers[i].len = 0;
        stringCount[i] = 0;
        flushCount[i] = 0;
      }

      /* Register our aserver callback routine with the aserver library */
      ass_set_cb(AserverCallback);

      /* Initialise access points */
      res = as_apstart();
      if(res != ASRC_SUCCESS)
        Error("Failed to initialise access points.");

      /* Set up the connection table */
      if (res == ASRC_SUCCESS) {
        res = as_setconntable(MAX_PRINT_CONNECTS, connTable);
        if (res != ASRC_SUCCESS)
          Error("Failed to as_setconntable - res = $s\n", as_asrc_to_str(res));
      }
      break;
```

```
    case WM_PAINT :
      hdc = BeginPaint(hWnd, &ps);
      /* TextOut(hdc, 1, 1, lpszStatus, strlen(lpszStatus)); */
      wsprintf(b, "Connect Requests [%ld]    ", connectRequests);
      TextOut(hdc, 1, 1, b, strlen(b));

      wsprintf(b, "Disconnect Requests [%ld]    ", disconnectRequests);
      TextOut(hdc, 1, 31, b, strlen(b));

      wsprintf(b, "Connections [%d]    ", totalConnects);
      TextOut(hdc, 1, 61, b, strlen(b));

      for(i = 0; i < connectRequests; i++) {
        wsprintf(b, "Conn %d flushes [%ld],  strings [%ld]", i, flushCount[i],
                                                          stringCount[i]);
        TextOut(hdc, 1, 91 + (i * 30), b, strlen(b));
      }

      EndPaint(hWnd, &ps);
      break;

    case WM_DESTROY :
      as_apfinish();
      PostQuitMessage(0);
      break;

    default :
      return (DefWindowProc(hWnd, message, wParam, lParam));
  }
  return(NULL);
}

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)

{
  MSG msg;

#ifdef __WINDOWS_386__
  sprintf(applicationName, "%s%d", applicationName, hInstance);
#else
  if(!hPrevInstance)
#endif
    if(!FirstInstance(hInstance))
      return (FALSE);

  if(!AnyInstance(hInstance, nCmdShow, lpCmdLine))
    return (FALSE);

  /* Initialise program */
  PostMessage(windowHandle, WM_USER_INIT, 0, 0L);

  /* Main Windows message loop */
  while(GetMessage(&msg, NULL, NULL, NULL)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }

  return (msg.wParam);
}
```

A.6.5 Building the Print host code

On a PC, the Echo and Print host code can be built by using a make utility:

```
nmake -f ms.mak          (Microsoft make)

wmake -f wat.mak         (Watcom make)
```

A.6.6 The Print service for Sun hosts

The following is a listing of the Print service for X-Windows on a Sun host, which is in the file `printer.c`:

```
/* Copyright SGS-THOMSON Microelectronics Limited 1992 */

static char SccsId[] = "@(#) Module: printer.c, revision 1.6 of 2/19/93";

/* #define DEBUG_MODULE */

static const char prog_name[]    = "printer";
static const char prog_version[] = "printer: version 1.2";

#include <stdio.h>
#include <assert.h>
#include <aslib.h>
#include "print.h"

#ifdef VMS
#include <unixio.h>
#include <file.h>

#include descrip
#include ssdef
#include prcdef
#include iodef
#include psldef

static short int InputChan;
static struct dsc$descriptor_s ttydesc;
#endif

#define MAX_PRINT_CONNECTS   8

typedef struct {
  int len;
  unsigned char data[MAX_PRINT_MSG_LEN];
} msg_buf;

static msg_buf msg_buffers[MAX_PRINT_CONNECTS];

struct pkt_list_st;
typedef struct pkt_list_st pkt_list_t;

struct pkt_list_st {
  int ap_num;
  packet_hdr hdr;
  unsigned char *data;
  pkt_list_t *next;
};

static pkt_list_t *pkt_list_first, *pkt_list_last;
static int pkt_list_len;
```

```
static asrc add_to_pkt_list(int ap_num,
                            packet_hdr hdr, unsigned char *data)
{
  asrc res = ASRC_SUCCESS;
  int len = (int) as_hdr_get_len(hdr);
  pkt_list_t *entry = (pkt_list_t *) malloc(sizeof(pkt_list_t));

  assert(ap_num == 0);
  if (entry == NULL)
    res = ASRC_OUT_OF_MEMORY;
  else {
    entry->data = (unsigned char *) malloc(len);
    if (entry->data == NULL) {
      res = ASRC_OUT_OF_MEMORY;
      abort();
      free(entry);
    }
  }

  if (res == ASRC_SUCCESS) {
    entry->ap_num = ap_num;
    memcpy(entry->hdr,  hdr,  sizeof(packet_hdr));
    memcpy(entry->data, data, len);
    entry->next = NULL;
    if (pkt_list_first == NULL)
      pkt_list_first = entry;
    else
      pkt_list_last->next = entry;
    pkt_list_last = entry;
    pkt_list_len++;
  }

  return (res);
}

static asrc remove_from_pkt_list(int *ap_num,
                                 packet_hdr hdr, unsigned char *data)
{
  asrc res = ASRC_SUCCESS;

  if (pkt_list_first == NULL) {
    res = ASRC_FAILED;
  } else {
    pkt_list_t *entry = pkt_list_first;
    *ap_num = entry->ap_num;
    memcpy(hdr,  entry->hdr,  sizeof(packet_hdr));
    memcpy(data, entry->data, (int) as_hdr_get_len(hdr));
    pkt_list_first = entry->next;
    free(entry->data);
    free(entry);
    pkt_list_len--;
/*    printf("Removed packet from list, len = %d\n", pkt_list_len);*/
    assert(*ap_num == 0);
  }
  return(res);
}

int main()
{
  asrc res = ASRC_SUCCESS;
  conn_id conn_table[MAX_PRINT_CONNECTS];
  int total_connects = 0;
  int i;
  as_bool has_been_used = AS_FALSE;
```

```
#ifdef VMS
#ifdef SEPARATE_TERMINAL
  char device_name[50];
  short device_length;
#else
  char* device_name = "TT:";
  short device_length = 3;
#endif /* SEPARATE_TERMINAL */
  int status;
#endif

#ifdef VMS
#ifdef SEPARATE_TERMINAL
  status = DECwTermPort(0,0,0,device_name, &device_length);
  device_name[device_length] = '\0';
  if (status != SS$_NORMAL)
    LIB$STOP(status);
  if (freopen(device_name,"a+",stdin,"rfm=udf") == NULL) {
    printf("freopen stdin failed\n");
  }
  if (freopen(device_name,"a+",stdout,"rfm=udf") == NULL) {
    printf("freopen stdout failed\n");
  }
  if (freopen(device_name,"a+",stderr,"rfm=udf") == NULL) {
    printf("freopen stdin failed\n");
  }
#endif /* SEPARATE_TERMINAL */

  ttydesc.dsc$w_length = device_length;
  ttydesc.dsc$b_dtype = DSC$K_DTYPE_T;
  ttydesc.dsc$b_class = DSC$K_CLASS_S;
  ttydesc.dsc$a_pointer = device_name;
  status = SYS$ASSIGN(&ttydesc, &InputChan, PSL$C_USER, 0);
  if (status != SS$_NORMAL)
    LIB$STOP(status);
#endif

  printf("The Printer Service (%s)....\n", prog_version);
  pkt_list_first = NULL;
  pkt_list_len = 0;

  for (i = 0; i < MAX_PRINT_CONNECTS; i++)
    msg_buffers[i].len = 0;

  res = as_apstart(AS_CHILD.IN_PIPE, AS_CHILD.OUT_PIPE);

  if (res != ASRC_SUCCESS)
    printf("Failed to initialise access points\n");


if (res == ASRC_SUCCESS) {
    res = as_setconntable(MAX_PRINT_CONNECTS, conn_table);

    if (res != ASRC_SUCCESS)
      printf("Failed to as_setconntable - res = %s\n",as_asrc_to_str(res));
  }

  while ((res == ASRC_SUCCESS) &&
         ((total_connects != 0) || (has_been_used == AS_FALSE))) {
    int ap_num, c_num;
    packet_hdr hdr;
    pkt_type type;
    unsigned char data[MAX_PACKET_SIZE];


    if (pkt_list_len > 0) {
      res = remove_from_pkt_list(&ap_num, hdr, data);
      assert(ap_num == 0);
    } else {
      ap_num = 0;
      res = as_rec_packet(ap_num, hdr, data);
    }
    res = as_translate_pkt(res, ap_num, hdr, data, add_to_pkt_list);
    type = as_hdr_get_type(hdr);
```

```
    if (res == ASRC_SUCCESS) {
      switch(type & ASPH_TYPE_MASK) {
        case ASPT_CONNECT_REQ:
          printf("got connect request\n");
          res = ass_processconnect(ap_num, hdr, data, &c_num,
                                   ASRC_SUCCESS, add_to_pkt_list);
          printf("ass_processconnect returned res = %s, c_num = %d\n",
                 as_asrc_to_str(res), c_num);
          total_connects ++;
          has_been_used = AS_TRUE;
          break;
        case ASPT_DATA: {
          int c_num = (int) as_hdr_get_conn_num(hdr);
          int len = (int) as_hdr_get_len(hdr);

          memcpy(msg_buffers[c_num].data +
                                      msg_buffers[c_num].len,data,len);
          msg_buffers[c_num].len += len;

          if (ASPH_IS_EOM(type)) {
            switch(msg_buffers[c_num].data[0]) {
              case PR_STRING:
                printf("got string (connection %d): %s", c_num,
                       msg_buffers[c_num].data + 1);
                break;
              case PR_FLUSH:
                printf("got flush (connection %d)\n", c_num);
                res = as_sendmessage(c_num, 0, NULL, add_to_pkt_list);
                printf("sent flush reply\n");
                break;
            }
            msg_buffers[c_num].len = 0;
          }
          break;
        }
      }
    } else if (res == ASRC_GOT_DISCONNECT_REQ) {
      total_connects --;
      printf("connection %d disconnected (num connects now %d)\n",
             (int) as_hdr_get_conn_num(hdr), total_connects);
      res = ASRC_SUCCESS;
    }
  }

  printf("printer finishing\nZzzzzzz....\n");
  sleep(10);

  return(0);
}
```

B AServer protocols

B.1 Introduction

This appendix describes the *AServer protocol* and the *Mega-packet protocol.* The AServer protocol is the protocol between a process and a gateway on the same device, or between a client and a service if no gateway is used. This protocol is described in sections B.2 to B.3. The Mega-packet protocol is the protocol used on hardware serial links between gateways. This protocol is described in section B.4.

The AServer allows processes on targets to communicate with external devices such as a host. Communication using the AServer is asynchronous.

Processes communicate by sending messages. Each message is made up of one or more packets. Each packet has a header describing the packet, and a data section containing information being carried by the packet. A packet header is a fixed size of 8 bytes, while the data section can be any length from 0 to 1024 bytes.

B.2 Packets

A *packet* is the basic unit of data between clients and servers. A packet is made up of a header and a data section. A packet has a type, indicated by the header, such as a connect request or data. All the packets in a single message have the same packet type.

The header is `ASPH_SIZE` bytes long, where `ASPH_SIZE` is currently set to 8. The header consists of the fields listed in Table B.1, where `AS_UINT16` is an unsigned, little-endian, 16-bit integer, and `BYTE` is an unsigned 8-bit number.

The header includes the access point number and the connection number, which together form an address of the destination of the packet and control the routing of the packet to its destination.

Byte	Type	Description
0–1	`AS_UINT16`	Length of the data in bytes.
2–3	`AS_UINT16`	Access point number.
4–5	`AS_UINT16`	Connection number.
6	`BYTE`	Packet Type (bits 0-6), and End-of-Message tag (bit 7).
7	`BYTE`	Protocol byte, a user defined byte not used by the AServer itself.

Table B.1 Packet header fields

The maximum length of the data is `MAX_PACKET_SIZE` bytes, which is currently set to 1024.

If the End-of-Message tag bit is 1 then this packet is the last packet in the message; if it is 0, then there is at least one more packet in this message. This allows multi-packet messages to be built up.

To set the End-of-Message tag, the packet type should be or-ed with the constant ASPH_EOM_MASK:

```
Type_byte = packet_type | ASPH_EOM_MASK;
```

To obtain the packet type, the packet type byte should be and-ed with ASPH_TYPE_MASK:

```
packet_type = Type_byte & ASPH_TYPE_MASK;
```

To determine whether a packet header has the End-of-Message tag set, use the macro ASPH_IS_EOM(x) where x is the packet type byte:

```
EOM_is_set = ASPH_IS_EOM(Type_byte);
```

ASPH is an abbreviation of AServer Packet Header.

The defined packet types are listed in Table B.2. The packet types correspond to the types of message defined in section B.3. Other values of the packet type are reserved by INMOS for future expansion.

Type	Value	Meaning
ASPT_CONNECT_REQ	0	Connect request
ASPT_CONNECT_REPLY	1	Connect reply
ASPT_DISCONNECT_REQ	2	Disconnect request
ASPT_DISCONNECT_REPLY	3	Disconnect reply
ASPT_DATA	4	This packet carries data
ASPT_CONTROL	5	This packet carries control information
ASPT_ABORT_REQ	6	Abort service request

Table B.2 Packet types

ASPT is an abbreviation of AServer Packet Type.

ASPT_CONTROL packets are hidden from the user in the sense that they are translated into result codes by as_translate_pkt, and are only used in some implementations of the AServer. They are used internally in the AServer and the service/client libraries. They carry information such as the transputer error flag being set. The user of service/client libraries sees events such as the transputer error flag being set as a return code to a library call, enabling the implementation of the communication of such events to vary.

The protocol byte is not used by the AServer libraries, and thus is available to software layered above these libraries, for example for distinguishing different types of messages.

B.3 Messages

A *message* may be of arbitrary length, and is made up of one or more *packets*. Certain services may require that messages have a specified maximum length.

Each packet in a message must have the same packet type. The message type is the type of each packet in the message. The last packet in a message has its End-of-Message bit set; other packets do not.

The communication mechanism used to carry the packets ensures that packets and messages arrive in the same order as they were sent.

If a process sends a disconnect request message or an abort message to the other end of the connection, then that process should discard all data packets received from the connection before the disconnect reply is received.

B.3.1 Connect request messages

A connect request message is used by a client to initiate a connection to a service. The client will receive a connect reply that indicates whether or not the connect request was successful or not. The data in the message has the format shown in Table B.3. The data should always be extracted using `as_decode_conn_req`.

Bytes	Type	Meaning
0:1	`AS_UINT16`	Gateway index of client.
2:3	`AS_UINT16`	Connection number at client end.
4	`char`	Unsigned length *n* of name of requested service including null terminator.
5 : (4+*n*)	`char[]`	Name of requested service including null terminator.
5+*n*	`char`	Unsigned length *m* of service specific parameters including null terminator.
(6+*n*) : (6+*n*+*m*)	`char[]`	Service specific parameters (if any) including null terminator.

Table B.3 Connect Request message format

The array contains two null-terminated strings, which may have zero length. The length count is thus at least one, as it always contains the null termination byte.

Connect request messages always fit in a single packet. This makes the implementation easier.

The gateway index of the client is set by the gateway that the client is connected to; the client should simply initialize it to 0. If the client is directly connected to the service, then the gateway index does not get used.

B.3.2 Connect reply messages

A connect reply message is used by a service or the AServer to give the result of a connect request to the client. If the AServer is unable to start the requested service then it will return an error to the client in a connect reply message.

The data in the message has the format shown in Table B.4.

Type	Meaning
BYTE	Result code of connect
AS_UINT16	Gateway index at service end
AS_UINT16	Connection number at service end

Table B.4 Connect Reply message format

If the result code is not ASRC_SUCCESS, then the connect failed.

The gateway index and connection number returned in the connect reply are put into the header of any packet sent to the service subsequently.

B.3.3 Disconnect request messages

A disconnect request message is sent by a client to a service that the client no longer wishes to use.

The client will always receive either a disconnect reply or an abort request. The latter can happen if the service happens to send an abort after the disconnect request is sent and before it is received by the service.

This message contains no data.

B.3.4 Disconnect reply messages

A disconnect reply message is sent by a service to a client that has sent a disconnect request message.

This message contains no data.

B.3.5 Data messages

Data messages are used to carry service specific data from the client to the service and from the service to the client.

The contents of the data in the message is completely service specific.

B.3.6 Control messages

Control messages are used to carry information about the state of the AServer and the target. For example, if the transputer error flag becomes set on an IMS Txxx transputer network, then the processes (servers and/or clients) running on the host are sent a control message to inform them that the error flag has been set.

Control messages are not processed by the user; they are converted into result codes by `as_translate_pkt`.

The data in the message has the following format:

Type	Meaning
BYTE	Control message code

The following is a list of control messages codes:

Type	Meaning
CONMSG_TX_ERROR	Transputer error flag set; receiving function should return `ASRC_TX_ERROR`
CONMSG_TERMINATE	Receiving function should return `ASRC_TERMINATE`
CONMSG_RESTART	Receiving function should return `ASRC_RESTART`

B.3.7 Abort request messages

Abort request messages are used by services to terminate connections to clients in 'abnormal' conditions. Normally, the client will disconnect from the service when it chooses.

If a service decides that it no longer wishes to communicate with a client (for example, because the client is trying to do something invalid), then the service may send an abort request message to the client.

The client will reply with a disconnect reply. The service will ignore any (data) message from the client until the disconnect reply is received.

This message contains no data.

B.3.8 Result codes

The result codes are listed in Appendix C.

B.4 Mega-packet protocol

A mega-packet is used for transporting one or more AServer packets across a link that connects a gateway to another gateway, or `irun` to a gateway.

Transporting AServer packets in mega-packets has the advantage that a number of small AServer packets can be grouped together and sent as one larger packet. This reduces network overheads (e.g. on ethernet using an IMS B300 ethernet gateway), and/or host computer overheads (i.e. reduced number of device driver accesses).

The use of mega-packets is currently also used to maintain compatibility with the current Linkops format packet. Linkops is the INMOS link interface software. By making a mega-packet look like a Linkops packet it can be carried through current software (e.g. the IMS B300 Ethernet Gateway, and the host Linkops software). To maintain Linkops compatibility, the maximum size of a mega-packet is generally limited to the maximum size of a Linkops packet, which is 1040 bytes.

Note that the size of mega-packets that travel from one transputer or ST20 processor directly to another transputer or ST20 processor is not dependent upon the maximum size of a Linkops packet, as those mega-packets are not transported by Linkops.

By using moderately intelligent buffering techniques, the grouping of the packets can be done so that is does not delay any individual AServer packet significantly.

For example, in a gateway process, packet processing can be double buffered. AServer packets are read into a buffer from clients and services at the same time as a mega-packet is sent to irun (or the other gateway). As soon as the mega-packet has been sent, the buffers are swapped around, and the newly formed mega-packet sent.

Alternatively, in `irun`, AServer packets can be put into a buffer ready to be sent to the target. As soon as the buffer is full, or the last packet of a message is put in the mega-packet, the mega-packet is sent.

The idea is to get as much into a mega-packet without actually delaying waiting for more AServer packets to put in it – as soon as the process building the mega-packet is idle, the packet is sent.

A mega-packet has a header that is followed by one or more complete AServer packets. The format of a mega-packet header is shown in Table B.5.

Field Size	Field
2 bytes	mega-packet length
1 byte	Linkops passthrough tag
1 byte	not used (contains 0)

Table B.5 Mega-packet header format

Mega-packets from the target to the host contain the passthrough tag `LOPS_PASSTHROUGH_TO_HOST`. Mega-packets from the host to the target contain the passthrough tag `LOPS_PASSTHROUGH_TO_TX`.

The passthough tags are defined like this in the file `asmega.h`

```
#define LOPS_PASSTHROUGH_TO_HOST    (102)
#define LOPS_PASSTHROUGH_TO_TX      (140)
```

The passthrough tag is not used by the receiving gateway. However, `irun`, if it detects any other tag in a packet it receives, starts an auto-iserver to process any such packets.

An auto-iserver is an iserver service to which any packets that do not contain the passthrough tag are routed, and replies from that iserver service are sent directly to the link and not converted into AServer packets or mega-packets. Note that there is currently an implementation limit which limits packets to/from an auto-iserver to the size of an AServer packet, rather than that of a larger `iserver` packet.

C AServer result codes

The following AServer library result codes (or `asrc` values) are defined in the header file `asconst.h`:

```
#define ASRC_SUCCESS              (0)

#define ASRC_FAILED               (128)
#define ASRC_BROKEN               (129)
#define ASRC_TX_ERROR             (130)
#define ASRC_MSGTOOBIG            (131)
#define ASRC_TERMINATE            (132)
#define ASRC_RESTART              (133)
#define ASRC_INVALID_APOINT       (134)
#define ASRC_STR_TOO_LONG         (135)
#define ASRC_DATA_TOO_BIG         (136)
#define ASRC_NO_REC_BUFFER        (137)
#define ASRC_CONN_TABLE_FULL      (138)
#define ASRC_NOT_CONNREQ          (139)
#define ASRC_SERVICE_BUSY         (140)
#define ASRC_OUT_OF_MEMORY        (141)
#define ASRC_UNEXP_CONN_NUM       (142)
#define ASRC_OUT_OF_RANGE         (143)
#define ASRC_CANT_OPEN_FILE       (145)
#define ASRC_GOT_DISCONNECT_REQ   (146)
#define ASRC_GOT_ABORT_REQ        (147)
#define ASRC_NOT_IMPLEMENTED      (149)
#define ASRC_NO_SUCH_SERVICE      (150)
#define ASRC_BOOT_FAILED          (151)
#define ASRC_UNEXPECTED_READ      (154)
```

The meanings of these constants are listed in Table C.1.

Result Code	Value	Meaning
ASRC_SUCCESS	(0)	Function succeeded.
ASRC_FAILED	(128)	Function failed (general failure).
ASRC_BROKEN	(129)	The connection has broken.
ASRC_TX_ERROR	(130)	The transputer error flag is set.
ASRC_MSGTOOBIG	(131)	The message received was too large for the buffer provided.
ASRC_TERMINATE	(132)	The calling process should now terminate.
ASRC_RESTART	(133)	The calling process should now restart or terminate.
ASRC_INVALID_APOINT	(134)	The access point parameter was invalid.
ASRC_STR_TOO_LONG	(135)	A string parameter was too long.
ASRC_DATA_TOO_BIG	(136)	The data part of a packet was too long.
ASRC_NO_REC_BUFFER	(137)	No buffer was available to receive packet.
ASRC_CONN_TABLE_FULL	(138)	The connection table was full.
ASRC_NOT_CONNREQ	(139)	A connect request was expected but a packet other than a connect request was received.
ASRC_SERVICE_BUSY	(140)	The service was busy, and the connect failed.
ASRC_OUT_OF_MEMORY	(141)	A dynamic allocation of memory failed.
ASRC_UNEXP_CONN_NUM	(142)	A packet from an unexpected connection number was received.
ASRC_OUT_OF_RANGE	(143)	A parameter was out of range.
ASRC_CANT_OPEN_FILE	(145)	A file could not be opened.
ASRC_GOT_DISCONNECT_REQ	(146)	A disconnect request was received, and the connection is now closed.
ASRC_GOT_ABORT_REQ	(147)	An abort request was received, and the connection is now closed.
ASRC_NOT_IMPLEMENTED	(149)	The function is not implemented.
ASRC_NO_SUCH_SERVICE	(150)	The service requested does not exist.
ASRC_BOOT_FAILED	(151)	A boot operation failed.
ASRC_UNEXPECTED_READ	(154)	A packet was read by a write function with a null callback.

Table C.1 Result code meanings

D Glossary

access point

A software communication port using *AServer* protocol.

access point number

The index of an *access point* in the array of access points for an *AServer process.*

AServer

A system for portable communication between processes, generally between processes on targets and processes on devices external to the target. It consists of protocols, libraries, gateways, clients and services.

AServer callback

A *callback* used by Windows services to handle incoming packets and terminate signals.

AServer database

A list of *services* and target hardware connections available to a host *gateway.*

AServer process

A section of code run in parallel with the rest of the program which communicates using the *AServer* protocol.

AServer protocol

The *protocol* used by *AServer clients* and *services* to communicate with each other.

auto-iserver

The use of the `iserver` *service* to mimic the action of the `iserver` without the use of converter or interface processes running on the target.

callback

A function called like an interrupt routine when an event occurs. When the callback is completed the interrupted function resumes. Callbacks may be *read callbacks* or *AServer callbacks.*

client

The *AServer process* which opens a connection using the *AServer* protocol.

connect

To initialize an *AServer* connection.

connection

A means of *AServer* communication between two *access points* on different *processes*. A connection does not exist until it has been initialized.

gateway

An interface between multiple *AServer* communications on one hand and an external hardware interface using a *link* on the other hand. A gateway multiplexes outgoing communications and demultiplexes incoming ones and may change the protocol.

host

A programmable device capable of resetting and debugging the target.

host gateway

A *gateway* which runs on a *host*.

`irun`

A host *gateway*.

`iserver`

An INMOS/SGS-THOMSON server program which runs on a *host*, providing a host interface for the application software, using the *iserver protocol*. It boots and loads a program onto a target then provides host services for the target on demand from a *link*.

iserver protocol

The protocol used by the i/o libraries for communication with the host.

link

A hardware serial communication connection between a transputer or ST20 and another device using the on-chip link port and any necessary interfacing.

mega-packet

A large *packet* containing one or more *AServer* packets designed to maximize efficiency over hardware connections to remote devices.

mega-packet protocol

The *protocol* used between *gateways* to transmit *mega-packets*.

packet

The data sent in a single transmission.

process

A sequential section of code with its own memory and resources running in parallel with the rest of the program. One task in a multi-task program.

An AServer process is a section of code run in parallel with the rest of the program which communicates using the *AServer* protocol.

protocol

The format of possible communications.

read callback

A *callback* used when an incoming packet arrives while another routine is sending.

service

A process to which a *client* can open an *AServer connection*.

service database

A list of *services* available to a *gateway*.

target gateway

A *gateway* running on a target processor.

toolset

A collection of tools for building application programs.

transputer link

A serial communication port provided on-chip on transputers and ST20s.

Index

A

C

D

E

F

G

H

I

L

M

O

P

R

S

T

V

W