RH[illegible]

The Afserver V1.5

March 4, 1988

Contents

1 Release contents

This release contains the sources for V1.5 for the **afserver** program. The sources supplied are for the following versions of the program:

Host	**Operating System**	Transputer Board
IBM PC	MS-DOS 2.11 or above	IMS B004
DEC VAX	VMS V4.5	IMS B001/B002 (ROM loader)
SUN 3	SUN UNIX 3.2 or above	IMS B011
SUN 3	SUN UNIX 3.2 or above	Niche NT1000

The sources for different versions of the program are supplied in sub-directories on the distribution disk. Also supplied on this disk in a sub-directory is a test program (written in occam) that can be used to test the **afserver** interactively or otherwise. Note that this test software is contained in TDS format files and so can only be compiled using the TDS (IMS D700C or IMS D700D).

The structure of the distribution disk is as follows.

- The toplevel directory contains **afserver** source files that are common to all releases as well as a number of sub-directories containing the test software and version dependent source files for the **afserver** program.
- The sub-directory **ibmdos** contains the source files which are specific to the IBM/MS-DOS - B004 version of the **afserver**
- The sub-directory **vaxvms** contains the source files which are specific to the VAX/VMS - B001/B002 version of the **afserver**
- The sub-directory **sununix\b011** contains the source files which are specific to the SUN3/UNIX - B011 version of the **afserver**
- The sub-directory **sununix\niche** contains the source files which are specific to the SUN3/UNIX - NT1000 version of the **afserver**

2 Source file structure

The **afserver** source file names are the same for each implementation of the program (including the implementation specific sources). The files that make up the sources for the **afserver** program are as follows:

```
version.h
afserver.h
boot.c
linkio.c
version.c
afserver.c
```

Listed below is a brief description for each of the above files, describing their portability and contents.

version.h

This file is **afserver** implementation dependent and is one of the header files that is always included by the C source files. This header file contains all the implementation dependent constants. It also **include**s all the system header files that might be needed by the C source files.

afserver.h

This header file for the **afserver** is implementation independent and is also included by all the C source files. It conatins the definitions of the constants used by the **afserver** for the various protocol commands and tags. It also contains other constant definitions that are not dependent on the **afserver** implementation.

boot.c

This source file is implementation dependent. The file contains the routines that are used to load a transputer board with a boot file and also to determine the method in which the transputer board is reset (or not).

linkio.c

This source file is implementation dependent. The file contains all the routines that are used to send and receive date to and from the transputer board. Other functions included are routiness to initialise and reset the transputer board and a function to peek memory from the transputer board. Functions for interpreting the tag protocol of the **afserver** are also provided in this file.

version.c

This source file is implementation dependent. The files contains the functions that perform the **afserver** commands which are non-portable. For example, the IBM/DOS specific **afserver** commands. Version specific support routines used in the processing of the **afserver** commands are also included in this file.

afserver.c

This source file for the **afserver** is implementation independent. The file contains all the functions necessary to perform the **afserver** commands (which includes any support functions). The file also contains the main entry point for the program (i.e. the **main** function).

3 Afserver interface and protocol specification

The following sections describe the operation and protocol of the **afserver** (version **V1.5**) which is used to load standalone programs and to provide a file service on the host system for these programs.

3.1 afserver command syntax

The syntax for the command line of the **afserver** is as follows:

afserver *[command.line]*

Where *command.line* is defined as follows:

```
command.line       =  option
                   |  program.parameter
                   |  option command.line
                   |  program.parameter command.line

program.parameter  =  any argument that is not an option

option             =  -: options
                   |  /: options

options            =  b boot.file.name
                   |  o option.flag
                   |  s board.size
                   |  i
                   |  l link.address
                   |  x
                   |  n
                   |  e
                   |  c boot.file.name
                   |  a load.address
                   |  d

boot.file.name     =  standard host file name

option.flag        =  number

board.size         =  number

link.address       =  number

load.address       =  number

number             =  decimal value
                   |  #hexadecimal value
```

program.parameter is supplied to resident programs on request by the program by issuing a **ReadBlock.Cmd** command on standard input stream 1 (the parameter stream). Note, this can only be done after the stream has been opened for access.

3.2 Using the `afserver`

The **`afserver`** has the filer process incorporated into it. This is the only protocol that is used by the **`afserver`**. As a result, the **`afserver`** will not work with boot files previously used with other server programs.

3.2.1 `afserver` options

[illegible]

The `afserver` options are described below.

Normal boot transputer (`-:b` *boot.file.name*)

If the option **`-:b`** is used, the **`afserver`** will try and use the given file name after the **`-:b`** to boot the transputer. If the file name is not a valid file or the **`afserver`** is unable to boot the transputer with this file, an error message will be generated by the **`afserver`** and then will terminate.

If this option is not specified the **`afserver`** will try and communicate with a program that has been previously loaded onto the transputer board. If no program is loaded on the transputer, the **`afserver`** will be unable to detect this and so will not terminate. If this happens then breakout of the **`afserver`** using the appropriate break key.

The ability to ignore this option enables the user to load a program onto a board once and then to re-run the program any number of times without the need to re-load the transputer board every time the program is to be used.

Specify option flag (`-:o` *[#]option.flag*)

This option is used to pass values directly to a user program by using the **`afserver`** command **`RunTimeData.Cmd`** with an option number of zero. If a **#** is used as a prefix for the following number then the number is taken to be a hexadecimal number. If no number is specified an error will occur.

Note that if this option is not used the the value of the option flag defaults to zero. This option should only be used with implementations that specify how to make use of the option flag.

An example use of this option is to enable the user to change the mode in which a program runs. That is, whether the program runs with a separate stack and heap store or a combined stack and heap store. The former would primarily be used for trying to place the run-time stack in fast memory.

Specify board size (`-:s` *[#]board.size*)

This option is used to pass to the user program the size of the transputer board on which the program is running. The value given with this option is accessed using the **`afserver`** command **`RunTimeData.Cmd`** with an option number of one. If a **#** is used as a prefix of the following number then the number is taken to be a hexadecimal number. If no number is specified an error will occur.

Note that if this option is not used the the value for the board size defaults to zero.

`afserver` information (`-:i`)

If this option is used the **`afserver`** will display a copyright message and its version date.

A typical message given is:

```
Afserver (IBM) V1.5 (25th February 1988)
Copyright INMOS Limited, 1985, 1986, 1987, 1988
```

Specify link address (`-:l` *[#]link.address*)

This option is only for use with the IBM/DOS version of the **`afserver`**.

The use of this option enables you to change the address which the **afserver** uses to communicate with the transputer board. If a # is used as a prefix of the following number then the number is taken to be a hexadecimal number. If no number is specified an error will occur.

The default link address used by the **afserver** when running on an IBM PC XT/AT is **#150** (hexadecimal 150). If the **afserver** is running on an IBM PC then the default link address used is **#300** (hexadecimal 300).

This option need only be used at present to change the base link address if a B004 board configured for the IBM PC XT/AT (link address **#150**) is used in an IBM PC and vice-versa. Therefore the link addresses which have to be specified are **#150** and **#300** respectively.

Boot in analyse mode (-:x)

This option is only for use with the IBM/DOS version of the **afserver**.

This option is only relevent if a transputer board is being loaded with a program. If this option is specified then the transputer being loaded is reset in analyse mode. The default is to reset the transputer in the normal way.

Also when this option is used the **afserver** performs a core dump of the first 16 Kbytes of the transputer's memory (i.e. starting from **MININT**) and stores the core in an internal buffer. This core dump can then be accessed by using the **afserver** command **ReadCoreDump.Cmd**.

No reset when booting (-:n)

This option is only for use with the IBM/DOS version of the **afserver**.

This option is also only relevent if a transputer board is being loaded with a program. If this option is specified then the transputer being loaded is **NOT** reset any way. This option overrides the option **-:x** (described above) and also prevents a core dump being preformed.

This option should only be used if the transputer being loaded has been previously loaded with a program that can understand the boot file being sent to it by the **afserver**.

Test error flag (-:e)

This option is only for use with the IBM/DOS version of the **afserver**.

If this option is used the **afserver** will test the error flag on the transputer board. If the error flag is detected the **afserver** will terminate. If this option is not specified the **afserver** defaults to not testing the transputer error flag.

If the error flag has been set and the **afserver** is not testing for it then the **afserver** will not be able to detect if the program on the transputer has halted (or not). It is therefore important to no what mode the program loaded has been compiled for. If it has been compiled so that if a run-time error occurs the error flag is set then this option should be used, otherwise it is not necessary for this option to be used.

Special boot transputer (-:c *boot.file.name*)

This option is only for use with the VAX/VMS version of the **afserver**.

This option is used in the same way as the **-:b** option described previously. The only difference between the use of this option and the **-:b** option is that the specified boot file used with this option must have been created using the standalone configurer. This option should not be used with boot files created by the standalone linker, they should be loaded instead using the **-:b** option.

Specify load address (-:a *[#]load.address*)

This option is only for use with the VAX/VMS version of the **afserver**.

This option allows the user to specify the start address at which the code is loaded from on the board being loaded. This option is only applicable for boards connected to a serial line, i.e. boards with their own built in loader (in ROM) and used in conjunction with the `-:b` option. This option has no effect if it is used with the `-:c` option as the load address is embedded into the code that is being loaded. If a # is used as a prefix of the following number then the number is taken to be a hexadecimal number. If no number is specified an error will occur.

The load address used by default is determined by the data at the start of the boot file which specifies the workspace requirements of the program to be loaded. This workspace requirement enables the **afserver** to calculate the minimum load address of the program, i.e. the code is loaded at a location which is calculated by adding the scalar workspace size to a default address which is the start of user memory on the transputer board. Note that this calculation takes into account the workspace requirement of the loader to ensure that the code being loaded does not overwrite the ROM loader's workspace area when this code is being loaded by the ROM loader.

Note, the use of this option is not to be recommended if the size for the workspace requirement of the program is unknown. The scalar workspace requirement of a linked program can be found by using the linker's option `/i`.

Serial boot debug (-:d)

This option is only for use with the VAX/VMS version of the **afserver**.

This option will cause the **afserver** to output debugging information to a file called **afserver.log**. The information contained within the file gives a representation of the states of the **afserver** during the booting process.

This option is only useful if the **afserver** fails to boot the board for some reason. If this happens, the file generated will give an indication of the state the **afserver** was at during the boot process when the error occured.

The IBM/DOS version of the **afserver** will reset transputer on the board that is being loaded and is able to load the transputer using the boot files created by the standalone linker and the standalone configurer using the `-:b` option.

The VAX/VMS version of the **afserver** is unable to reset the transputer on the board being loaded as it cannot access the reset pin of the transputer over the serial line it uses to load the board by. The VAX/VMS version is able to load the transputer using the boot files created by the standalone linker and the standalone configurer. To load a boot file created by the linker the `-:b` option has to be used, whereas if the file was created by the configurer the `-:c` option is used instead. Note that the boot file used with the `-:b` option must have been created using the standalone linker with the linker switch `/c`, (this switch prevents a loader being appended to the front of the code generated by the linker). The switch is used because a loader is already present in (ROM) on the board that is being loaded. Also note that the VAX/VMS version will only load transputer boards that are connected to a terminal's serial line.

3.3 afserver protocol

The **afserver** supports an interface to the host operating system, enabling user programs to access the filing system (among the operations available). The **afserver** also provides the service of loading a compiled and linked user program onto a transputer board for running.

3.3.1 Filer process

This section describes the set of operations that is supported by the "filer process" to enable user programs to access the host filing system.

The filer process supports a simple model of a filing system which allows the user to create, open, read and write files. A file is accessed by its name, which must be a legal host file name.

Files may be opened using one of two access methods:-

1 Binary byte-stream access: View the file as a byte stream.

2 Text byte-stream access: View the file as a byte stream, the filer process inserting carriage-returns to make the file a host format text file.

Files may be opened in read, write or update mode. Update mode allows both reading and writing.

The filer process also supports a number of standard input and output streams. Standard input stream 0 normally comes from the terminal, but may be redirected from a file by the user. The user may output to standard input stream 0, provided it is connected to the terminal, in which case output is sent to the terminal. Standard input stream 1, if it is used, consists of a sequence of characters supplied by the user (up to a maximum of 256 bytes) (eg from the command line). Standard output streams 0 and 1 normally go to the terminal, but may also be redirected to files by the user.

Standard streams are considered to be opened for text byte-stream access. Seek operations may or may not succeed on them, depending on what they are attached to.

Results from operations are returned as integers with the following conventions:

0	Indicates successful operation.
1	Indicates an end-of-file has occurred on this stream.
2	Higher positive integers are error conditions produced by the filer process for this operation.
-1	Negative integers are error conditions generated by the host operating system.

3.3.2 Filing system operations

Before any stream has been opened for reading or writing the following operations are supported.

Terminate

The terminate operation closes down the **`afserver`**. It returns a result. It is the responsibility of the user program process to close all open streams before issuing the terminate command to the **`afserver`**.

Input:	None
Output:	Result

In this implementation the terminate result never returns an error.

OpenFile

Given a file name, return a stream identifier for it. This stream identifier will be used in all subsequent operations on the file, until it is closed.

Input:	File name Access method Open mode Exist mode Record length
Output:	Stream identifier Result

The access method is one of {**Binary byte-stream**, **Text byte-stream** }. The open mode is one of {**Read**, **Write**, **Update**}. Update means both reading and writing is allowed. The exist mode is one of {**Old**, **New**}.

The record length is currently ignored; a number must always be supplied, however.

The possible error conditions detected by the `afserver` on `OpenFile.Cmd` are as follows:

1 Invalid file name length

2 No free channel

3 Invalid access method

4 Invalid open mode

5 Invalid exist mode

6 Operation failed

An `OpenFile.Cmd` specifying an **Old** file will only succeed if the file already exists in the host filing system. An `OpenFile.Cmd` specifying a **New** file will succeed if a file with that name can be made. Depending on the option specified at close time, the file name of a newly-created file may be added to the set of names permanently in the filing system, replacing any file of the same name which already exists. Exactly when the name of a newly-created file is added to the filing system is undefined, except that it will be no earlier than the open operation and no later than the close operation.

OpenTemp

Create a temporary file and return a stream identifier for it. The file will be opened for update. A temporary file is always deleted when it is closed.

Input:	Access method Record length
Output:	Stream identifier Result

The parameters and results are the same as for `OpenFile.Cmd`.

OpenInputStream

Open one of the standard input streams for read-only text byte-stream access.

Input:	Standard input stream number
Output:	Stream identifier Result

The error conditions are:

1 No free channel

2 Invalid input stream number

OpenOutputStream

Open one of the standard output streams for write-only text byte-stream access.

Input:	Standard output stream number
Output:	Stream identifier Result

The error conditions are:

1 No free channel

2 Invalid output stream number

AlienTerminate

The alien terminate operation preforms no action. All it does is to acknowledge the receipt of command. The reason for its inclusion in the protocol is for historical reasons.

Input:	None
Output:	Result

In this implementation the result never returns an error.

SetResult

Set the return value of the **afserver** when it terminates. This is useful if running the **afserver** within some environment that expects a result from the **afserver**.

Input:	Value to be returned by the **afserver**
Output:	Result

In this implementation the result never returns an error.

RunTimeData

The run time data operation asks the **afserver** the value of parameter that has been passed to it when executed. The choice of parameter is determined by specifying an index (or option number) that is used by the **afserver** to decide which option is passed back.

Input:	Option number
Output:	Option value Result

The only error condition is:

1 Invalid data option

At present the only values for the option number can be 0 or 1 (corresponding to the **afserver** options `-:o` and `-:s` respectively). If any other option number is used the operation fails.

ReadCoreDump

The read core dump operation is used to ask from the **afserver** a slice of memory that has been dumped from the transputer when the transputer was reset. Note that this operation will only succeed if the transputer was reset in analyse mode (by using the **afserver** option `-:x`). If this option is not used (or if another option has overridden the option) then the operation will fail.

Input:	Offset in core dump to read from Record length required
Output:	Record read Result

The slice required is specified by giving to the **afserver** the byte offset from **MININT** (of the transputer) from which the start of slice is to be taken and the length of the slice required. If the given offset or the slice length are out of range then an error is returned.

Note that if the combination of offset and slice length means that there is not enough core memory to fulfil the slice length requirement then the **afserver** returns the core memory that is available and a result of **EndOfFile**.

The error conditions are:

1 Invalid core offset

2 Invalid record length

3 Operation failed

In the current implementation the size of the core dump extracted by the **afserver** from the transputer is 16 Kbytes, starting at **MININT** in the transputer's memory.

ServerVersion

This command is used to find out from the **afserver** the host on which it isrunning, the date and version of the **afserver** and the state in which the **afserver** loaded the transputer (i.e. the method used for loading the transputer).

Input:	None
Output:	Version number Version date Version state Result

The *version number* is a 32 bit number which is divided up into four 8 bit fields. These are the following:

Bits	Value
0 – 7	Version sub-number
8 – 15	Version main number
16 – 23	Sub-host type
24 – 31	Host type

Presently the **afserver** host types are the following:

Value	Host
0	**IBMPC.Host** (includes XT and AT machines)
1	**NECPC.Host**
2	**VAXVMS.Host**
3	**SUN3.Host**

The sub-host type field is used to identify the different transputer systems that reside within a host; for instance the transputer board type. That is, as well as indicating the host on which the **afserver** is running, it will also be possible to identify the board family for which **afserver** was built for. At present this field is unused.

The version main number and sub-number fields of the *version number* contain the current **afserver** version. For example, if the verison for the **afserver** is *V1.5* then the version is encoded so that version main number has the value 1 and sub-number has the value 5.

The *version date* number is also a 32 bit number which is used to encode the date on which the **afserver** was produced. The date comprises of the day, month and year and is encoded into four 8 bit fields. These are the following:

Bits	Value
0 – 7	Year of century
8 – 15	Century
16 – 23	Month
24 – 31	Day of month

For example, the date 15th February 1988 will be represented (in hexadecimal) as the value **#0F021358**.

The *version state* is another 32 bit number which currently uses only the bottom (i.e. least significant) 8 bits of the number. These bits are used to describe the method used by the **afserver** to load the transputer. The **afserver**'s loading states are currently defined as the following:

Value	Host	**afserver** option used
0	**NoBootFile.State**	No **-:b**
1	**ResetBootFile.State**	**-:b**
2	**NoResetBootFile.State**	**-:b** with **-:n**
3	**AnalyseBootFile.State**	**-:b** with **-:x**

In this implementation the result never returns an error.

RunCommand

Run the given command line on the host system. This has exactly the same effect of using the C library call **system (char *)**. (On completion of the command, control is returned to the **afserver**.)

Input:	Command line to be executed on the host system
Output:	Result

The error conditions are:

1 Invalid record length

2 Operation failed

RenameFile

Rename the given file name using the given replacement file name. Only succeeds if the original file name exists, if not, an error is returned.

Input:	Original file name Replacement file name
Output:	Result

The error conditions are:

1 Invalid file name length

2 Operation failed

ReadTime

Get the current time. Value returned is the number of seconds that have elapsed since 00:00:00 GMT, January 1, 1970 according to the host system clock.

Input: None

Output: Number of seconds that have elapsed
Result

The only error condition is:

1 Operation failed

ReadKey

Get the value of the first unread key pressed on the host's keyboard that is available in the host's keyboard buffer. If no key has been pressed an error value is returned as a result. The value that is returned is the **ascii** value of the key, if pressed. If a key is pressed then it will not be echoed to the screen.

Note if a key is available then that key will be removed from the host's keyboard buffer. Also, it is advised that when reading input from the keyboard only one method should be used, either using the **ReadKey.Cmd** (or **ReadKeyWait.Cmd**) command or using the **ReadBlock.Cmd** command on standard input stream 0. If this is not adhered to, unpredictable results may occur.

Input: None

Output: Value of last key pressed that has not been read
Result

The only error condition is:

1 Operation failed

ReadKeyWait

Wait and get the value of the first unread key pressed on the host's keyboard. The value that is returned is the **ascii** value of the key, if pressed. The key pressed will not be echoed to the screen.

Note will be removed from the host's keyboard buffer. Also, it is advised that when reading input from the keyboard only one method should be used, either using the **ReadKey.Cmd** (or **ReadKeyWait.Cmd**) command or using the **ReadBlock.Cmd** command on standard input stream 0. If this is not adhered to, unpredictable results may occur.

Input: None

Output: Value of last key pressed that has not been read
Result

The only error condition is:

1 Operation failed

ReadEnvironment

The read environment operation passes to the **afserver** a string representing the environment variable whose value is wanted. If the variable exists in the environment, the **afserver** passes back the value of the variable as defined in the host system. The value passed back is always a string. If the variable does not exist the operation fails. This operation is equivalent to using the C library function **getenv (char *)**.

Input: Environment variable

Output: Value of variable
Result

The error conditions are:

1 Invalid record length

2 Operation failed

3.3.3 Stream operations

Once a stream has been successfully opened, the operations described in this section may be applied to it. These are a set of status requests which return information on the current status of the stream, and a close operation.

StreamAccess

Return the access method used to open this stream.

Input:	Stream identifier
Output:	Access method Result

The only error condition is:

1 Invalid stream identifier

StreamStatus

Return 0 to indicate OK, 1 to indicate end-of-file on this stream or a negative number which is a host-specific error value.

Input:	Stream identifier
Output:	Result

The only error condition is:

1 Invalid stream identifier

StreamFile

Return the file name associated with this stream (if any), possibly a full hierarchical name.

Input:	Stream identifier
Output:	File name Result

The only error condition is:

1 Invalid stream identifier

StreamLength

Return the current length in bytes of this stream.

Input:	Stream identifier
Output:	Stream length Result

The error conditions are:

1 Invalid stream identifier

2 Operation failed

StreamConnect

Return the device to which this stream is connected. The connection is one of {**Screen**, **Keyboard**, **File**, **Temporary file**, **Parameter**}.

Input:	Stream identifier
Output:	Device connected Result

The error conditions are:

1 Invalid stream identifier

2 Operation failed

CloseStream

Close this stream.

Input:	Stream identifier Close option
Output:	Result

The close option is one of {**Close**, **Close-Delete**}. Of the options available at close time, if **Close-Delete** is selected, the file is closed and the meaning of the name used to open the file becomes undefined. If **Close** is selected, the file is closed and its name is added to the set of names permanently in the filing system, replacing any file of the same name which already exists. If the file has been opened with an **`OpenTemp.Cmd`** operation then either option may be specified, but have no effect on the file name system (a temporary file is always deleted).

The error conditions are:

1 Invalid stream identifier

2 Invalid close option

3 Operation failed

3.3.4 File operations

The remaining operations are the read and write operations for text byte-stream and binary byte-stream access. Read operations may only be used when the file has been opened for read or update. Write operations may only be used when the file has been opened for write or update.

A file opened for text or binary byte-stream access is seen as a sequence of bytes. There is a stream position which is an offset in bytes from the start of the file. When a file is opened the stream position is 0.

If accessing a file that has been opened in **Update** mode, then when changing from writing to the file to reading from it (and vice-versa), a seek operation must be performed between the change of operation.

The following operations may be applied to a stream opened in either of the byte-stream access modes :-

ReadBlock

Given a length of data to be read, read up to that length of data from the byte stream, starting with the byte indicated by the current stream position. Move the stream position beyond the last byte of data read. The end of file condition becomes true when a read is made that attempts to read past the last byte in the file.

Input:	Stream identifier Record length required
Output:	Record read Result

The error conditions are:

1 Invalid stream identifier

2 Invalid record length

3 Operation failed

WriteBlock

Write the given block of data to the byte stream, starting with the position indicated by the current stream position. Move the stream position beyond the last byte of data written. If the current stream position is beyond the old end of the file, then the data between the old end of the file and the first byte of data written is undefined.

Input:	Stream identifier Record to be written
Output:	Record length written Result

The error conditions are:

1 Invalid stream identifier

2 Invalid record length

3 Operation failed

Seek

Move the stream position to the byte offset indicated, from the start of the file.

Input:	Stream identifier Offset
Output:	Result

The error conditions are:

1 Invalid stream identifier

2 Seek not possible on this stream

3 Invalid seek offset

4 Operation failed

3.3.5 IBM PC/DOS extensions

The following protocol commands are only applicable to the IBM PC running under DOS as they allow the user to gain full access to the machine's hardware and software. These commands should not be used if the program communicating with the **afserver** if it is not to be resident on an IBM PC/DOS system (e.g. the VAX/VMS system). If this is the case, then the commands described in the previous sections should only be used. This will enable the user to maintain maximum portability of programs between different hardware and operating system combinations.

If the following commands are used on a system that does not support them (i.e. not an IBM PC/DOS system) then they will be accepted but no action will be performed and the command will fail.

ReceiveBlock

Receive a block of data from the host system (via the **afserver**), the size of which is specified in bytes. The location of the start of this block in the host system is specified as a host dependent address.

Input: Location of block of data to be sent
Size of block of data to be sent

Output: Block of data received
Result

The error conditions are:

1 Invalid record length

2 Not implemented

When implemented on an IBM PC running under DOS this address will be a 32 bit quantity, the most significant 16 bits indicating the memory segment and the least significant 16 bits being the offset in the given segment.

SendBlock

Send a block of data to the host system starting at a specified location in the host's memory, the value of which is host dependent.

Input: Storage location of block of data
Block of data to be stored

Output: Number of bytes stored
Result

The error conditions are:

1 Invalid record length

2 Not implemented

The type of address used, when running on the IBM PC under DOS, will be the same as that used for the **ReceiveBlock.Cmd** command.

CallInterrupt

Invoke an interrupt call on the host system with the host processor registers appropriately set up. On return from the interrupt receive the registers contents and the value of any flags.

The values for the registers are sent over as a block of bytes, which is then split up by the **afserver** into register values and then assigned to the appropriate registers. Each register value occupies 4 bytes of the block. The order in which the register values appear in this block (at 4 byte intervals) is host architecture dependent. At present only one flag value is returned.

Input:	Interrupt number Values of the host processor's registers before the interrupt
Output:	Value of flag after interrupt Values of the host processor's registers after the interrupt Result

The error conditions are:

1 Invalid record length

2 Not implemented

This condition only occurs if the amount of register data sent over is not adequate.

On the IBM PC the 2 most significant bytes are ignored as this machine has only 2 byte registers (16 bit registers). The order in which the registers appear in the block of data passed over is defined as the following:

REGISTER	Start position in block **l.s. byte**	End position in block **m.s. byte**
`ax`	0	3
`bx`	4	7
`cx`	8	11
`dx`	12	15
`di`	16	19
`si`	20	23
`cs`	24	27
`ds`	28	31
`es`	32	35
`ss`	36	39

The flag that is returned is the value of the carry flag on the IBM PC after the interrupt has been done.

Note that the segment registers are available. (The registers `cs, ds, es, ss`). Therefore you will need to use the `ReadRegs.Cmd` command to read the values of these segment registers before assigning values to them so as to ensure that they retain their values before the interrupt call. This only has to be done if you do not need to reassign these registers.

The current implementation does not use the registers `cs` and `ss`, as these registers cannot be assigned to using the current `afserver`. The registers `ds` and `es` are restored to their original values once the interrupt has been done. This means that it is not possible to change the segment registers.

Also note that if registers are left unspecified in the block the values that these registers take will be whatever random data that is in the relevant positions in the block.

ReadRegs

Inquire the about the state of the registers on the host machine. On return the values of the host's registers are given.

The values of the registers are returned as a block of bytes, each register occupying 4 bytes of the block. The order in which the registers appear in the block (at 4 byte intervals) is host dependent.

Input: None

Output: Values of the host processor's registers
Result

The only error condition is:

1 Not implemented

On the IBM PC the 2 most significant bytes are ignored as this machine has only 2 byte registers (16 bit registers). The order in which the registers appear in the block of data passed over is defined as the following:

REGISTER	Start position in block **l.s. byte**	End position in block **m.s. byte**
cs	0	3
ds	4	7
es	8	11
ss	12	15

PortRead

Returns the value at the port specified, using the given port address, at the moment the port is read by the **afserver**. No check is made to ensure that the value received from the port is valid, it is the users responsibility to ensure that the value returned is valid.

Input: Address of the port

Output: Value at the port when read
Result

The only error condition is:

1 Not implemented

On the IBM PC the 2 most significant bytes of the port address are ignored as the port address can only be represented as a 2 byte number (i.e. a 16 bit number).

PortWrite

Writes the given value at the port specified using the given port address. No check is made to ensure that the value written to the port has been correctly read by the device connected to the port (if any), as for **PortRead.Cmd**, it is upto the user to do this.

Input: Address of the port
Value to be written to the port

Output: Result

The only error condition is:

1 Not implemented

On the IBM PC the 2 most significant bytes of the port address are ignored as the port address can only be represented as a 2 byte number (i.e. a 16 bit number).

3.3.6 **afserver** termination

To terminate the **afserver**, **Terminate.Cmd** has to be given. This should not be given from within a user's program. The command should only be given after the user's program has terminated (within a harness). The harness should be the only separately code to terminate the **afserver** using **Terminate.Cmd**. If this is not done and the **afserver** is terminated from within the user's program then it may not be guaranteed that code outside the user's program (for instance in the harness) will not be waiting to communicate with the **afserver**.

The **AlienTerminate.Cmd** was originally introduced into the protocol to overcome the above problem in previous releases of compilers and their run-time libraries. The command has been left in so that the **afserver** remains compatible with these compilers and their run time libraries.

When the terminate command is received by the **afserver** is terminated, so ensure that there is no further code to execute after the terminate command that relies on communication with the **afserver**.

3.3.7 **afserver** protocol

The **afserver** has an interface to an user program process consisting of two channels. The channel **to.filer** is used to send commands and data from the user program process to the **afserver**. The channel **from.filer** is used to send data from the **afserver** to the user program process.

The basic protocol between the processes consists of a tagged protocol; the tags will indicate the type of the value following immediately. The following is the basic tagged protocol:-

```
bool.value       IS BYTE 0:
byte.value       IS BYTE 1:
int.value        IS BYTE 2:
int16.value      IS BYTE 3:
int32.value      IS BYTE 4:
int64.value      IS BYTE 5:
real32.value     IS BYTE 6:
real64.value     IS BYTE 7:
nilrecord.value  IS BYTE 8:
record8.value    IS BYTE 9:
record.value     IS BYTE 10:
record16.value   IS BYTE 11:
record32.value   IS BYTE 12:
```

The **afserver** protocol is implemented on top of this basic data protocol.

The following description of the protocol of the **afserver** uses a modified BNF notation. The terms *integer* and *record* are made up the basic tagged variants described above, as follows:-

```
integer  =  int32.value; INT32

record   =  nilrecord.value
         |  record32.value; INT32::[]BYTE
```

The meaning of which is as follows:

> To send an integer value to the **afserver** the tag **int32.value** is sent (indicating that a 4 byte integer value is about to be sent) followed by the integer value. The value of **int32.value** is represented by a byte of value of 4. To receive an integer value first **int32.value** is received followed by the integer.
>
> To send a record of bytes to the **afserver** either the tag **nilrecord.value** or **record32.value** is sent. If **nilrecord.value** is sent then no sequence of bytes following the tag is sent. This tag is sent if a zero length record is to be sent to the **afserver**, the value of which is represented by a byte value of 8. If **record32.value** is sent, then a sequence of bytes follows it (which are preceded by an integer value indicating the number of bytes to be sent, which will be 4 bytes in length). The value of **record32.value** is represented by a byte value of 12. To receive a record first **record32.value** is received followed by the record length and then the record itself. If **nilrecord.value** is received then no record will follow it.

The **nilrecord.value** variant must always be sent for a record of length 0.

Commands to the **afserver** are listed individually by name. Each of these is an *integer* value. The actual values corresponding to these, and to the other quantities which have a limited range of values, are defined in the next section.

First we define the types for each of the kinds of parameters:-

flag	=	*integer*
date	=	*integer*
state	=	*integer*
value	=	*integer*
result	=	*integer*
offset	=	*integer*
version	=	*integer*
option.no	=	*integer*
record.no	=	*integer*
interrupt	=	*integer*
stream.id	=	*integer*
open.mode	=	*integer*
exist.mode	=	*integer*
close.option	=	*integer*
std.stream.no	=	*integer*
access.method	=	*integer*
record.length	=	*integer*
stream.length	=	*integer*
port.location	=	*integer*
stream.connect	=	*integer*
source.location	=	*integer*
destination.location	=	*integer*
variable	=	*record*
filename	=	*record*
command.line	=	*record*
old.file.name	=	*record*
new.file.name	=	*record*
register.block	=	*record*

Now we define the protocol going to and from the **afserver** for each operation. A program using this protocol should output all the parameters on the **to.filer** channel, and then input all the results on the **from.filer** channel.

Terminate command

to.filer.protocol	=	*terminate.cmd*
from.filer.protocol	=	*result*

OpenFile command

to.filer.protocol	=	*openfile.cmd filename access.method open.mode exist.mode record.length*
from.filer.protocol	=	*stream.id result*

OpenTemp command

to.filer.protocol	=	*opentemp.cmd access.method record.length*
from.filer.protocol	=	*stream.id result*

OpenInputStream command

to.filer.protocol	=	*openinputstream.cmd std.stream.no*
from.filer.protocol	=	*stream.id result*

OpenOutputStream command

to.filer.protocol	=	*openoutputstream.cmd std.stream.no*
from.filer.protocol	=	*stream.id result*

StreamAccess command

to.filer.protocol	=	*streamaccess.cmd stream.id*
from.filer.protocol	=	*access.method result*

StreamStatus command

to.filer.protocol	=	*streamstatus.cmd stream.id*
from.filer.protocol	=	*result*

StreamFile command

to.filer.protocol	=	*streamfile.cmd stream.id*
from.filer.protocol	=	*filename result*

StreamLength command

to.filer.protocol	=	*streamlength.cmd stream.id*
from.filer.protocol	=	*stream.length result*

StreamConnect command

to.filer.protocol	=	*streamconnect.cmd stream.id*
from.filer.protocol	=	*stream.connect result*

CloseStream command

to.filer.protocol	=	*closestream.cmd stream.id close.option*
from.filer.protocol	=	*result*

ReadBlock command

to.filer.protocol	=	*readblock.cmd stream.id record.length*
from.filer.protocol	=	*record result*

WriteBlock command

to.filer.protocol	=	*writeblock.cmd stream.id record*
from.filer.protocol	=	*record.length result*

Seek command

to.filer.protocol	=	*seek.cmd stream.id offset*
from.filer.protocol	=	*result*

AlienTerminate command

to.filer.protocol	=	*alienterminate.cmd*
from.filer.protocol	=	*result*

SetResult command

to.filer.protocol	=	*setresult.cmd value*
from.filer.protocol	=	*result*

RunTimeData command

to.filer.protocol	=	*runtimedata.cmd option.no*
from.filer.protocol	=	*value result*

ReadCoreDump command

to.filer.protocol	=	*readcoredump.cmd offset record.length*
from.filer.protocol	=	*record result*

ServerVersion command

to.filer.protocol	=	*serverversion.cmd*
from.filer.protocol	=	*version date state result*

RunCommand command

to.filer.protocol	=	*runcommand.cmd command.line*
from.filer.protocol	=	*result*

RenameFile command

to.filer.protocol	=	*renamefile.cmd old.file.name new.file.name*
from.filer.protocol	=	*result*

ReadTime command

to.filer.protocol	=	*readtime.cmd*
from.filer.protocol	=	*value result*

ReadKey command

to.filer.protocol	=	*readkey.cmd*
from.filer.protocol	=	*value result*

ReadKeyWait command

to.filer.protocol	=	*readkeywait.cmd*
from.filer.protocol	=	*value result*

ReadEnvironment command

to.filer.protocol	=	*readenvironment.cmd variable*
from.filer.protocol	=	*record result*

The following describes the syntax for the protocol commands that can only be used on the IBM PC running under DOS.

ReceiveBlock command

to.filer.protocol	=	*receiveblock.cmd source.location record.length*
from.filer.protocol	=	*record result*

SendBlock command

to.filer.protocol	=	*sendblock.cmd destination.location record*
from.filer.protocol	=	*record.length result*

CallInterrupt command

to.filer.protocol	=	*callinterrupt.cmd interrupt register.block*
from.filer.protocol	=	*flag register.block result*

ReadRegs command

to.filer.protocol	=	*readregs.cmd*
from.filer.protocol	=	*register.block result*

PortRead command

to.filer.protocol	=	*portread.cmd port.location*
from.filer.protocol	=	*value result*

PortWrite command

to.filer.protocol	=	*portwrite.cmd port.location value*
from.filer.protocol	=	*result*

3.3.8 Constant values

This section lists the constant values defined for the required operations. Gaps in number sequences of constants are reserved for future use by INMOS.

Commands

```
AlienTerminate.Cmd    IS 0:
OpenFile.Cmd          IS 1:
OpenTemp.Cmd          IS 2:
OpenInputStream.Cmd   IS 3:
OpenOutputStream.Cmd  IS 4:
StreamAccess.Cmd      IS 5:
StreamStatus.Cmd      IS 6:
StreamFile.Cmd        IS 7:
StreamLength.Cmd      IS 8:

CloseStream.Cmd       IS 11:
ReadBlock.Cmd         IS 12:
WriteBlock.Cmd        IS 13:
Seek.Cmd              IS 14:

StreamConnect.Cmd     IS 23:
Terminate.Cmd         IS 24:
SetResult.Cmd         IS 25:
RunCommand.Cmd        IS 26:
RenameFile.Cmd        IS 27:
ReadTime.Cmd          IS 28:
ReadKey.Cmd           IS 29:
ReceiveBlock.Cmd      IS 30:
SendBlock.Cmd         IS 31:
CallInterrupt.Cmd     IS 32:
ReadRegs.Cmd          IS 33:
RunTimeData.Cmd       IS 34:
ReadEnvironment.Cmd   IS 35:
PortRead.Cmd          IS 36:
PortWrite.Cmd         IS 37:
ReadKeyWait.Cmd       IS 38:
ReadCoreDump.Cmd      IS 39:
ServerVersion.Cmd     IS 40:
```

Access methods

```
BinaryByteStream.Access IS 0:
TextByteStream.Access   IS 1:
```

Open modes

```
Read.Mode   IS 0:
Write.Mode  IS 1:
Update.Mode IS 2:
```

Exist modes

```
Old.File   IS 0:
New.File   IS 1:
```

Connections

```
Screen.Use    IS 0:
Keyboard.Use  IS 1:
File.Use      IS 2:
Temp.Use      IS 3:
Parameter.Use IS 4:
```

Close options

```
Close.Option      IS 0:
CloseDel.Option   IS 1:
```

Standard input

```
Keyboard.Input  IS 0:
Parameter.Input IS 1:
```

Standard output

```
Standard.Output IS 0:
Error.Output    IS 1:
```

Results

```
OperationOk                 IS 0:
EndOfFile                   IS 1:

InvalidFileNameLength.Err IS 2:
InvalidAccessMethod.Err   IS 3:
InvalidOpenMode.Err       IS 4:
InvalidExistMode.Err      IS 5:
InvalidRecordLength.Err   IS 6:
InvalidStdStream.Err      IS 7:
InvalidStreamId.Err       IS 8:
InvalidCloseOption.Err    IS 9:
NoSeekPossible.Err        IS 10:
InvalidRecordNumber.Err   IS 11:

OperationFailed.Err       IS 99:
NoFreeChannel.Err         IS 100:
NoSuchFile.Err            IS 101:
FileAlreadyOpen.Err       IS 102:
ReadOpenFail.Err          IS 103:
NotImplemented.Err        IS 104:
InvalidSeekOffset.Err     IS 105:
InvalidCoreOffset.Err     IS 106:
InvalidDataOption.Err     IS 107:
```

A **afserver** error messages

The following gives a list of the possible error messages that can be generated by the **afserver**. All errors generated by the **afserver** are fatal, i.e. the **afserver** will immediately terminate.

1 **Failed to open boot file :** *name*
Failed to close boot file : *name*

These are generated when the **afserver** cannot open or close the boot file *name* (as specified with the **-:b** (or **-:c**) option). This generally occurs when the file name given to the **afserver** does not exist in the host file system or because the **afserver** has insufficient privilege to read from it.

2 **Failed to read boot file :** *name*

This is generated when the **afserver** cannot read the contents of the boot file *name*. This may only occur if the contents of the boot file has been corrupted in some way.

3 **Failed to send boot file :** *name*

This is generated when the **afserver** cannot send the contents of the boot file *name* to the transputer board it is trying to load. This may happen when there is a hardware problem with the board which prevents any more code being sent to it or the loader on the board has decided that there is no more code to load where in fact there is.

4 **Failed to open I/O channel :** *name*

This is generated when the VAX/VMS version of the **afserver** cannot open the logical channel *name* it uses to communicate with the transputer board. This generally occurs if the logical channel has not been set up, i.e. the logical name **SYS$TRANSPUTER** has not been assigned to a terminal device.

5 **Failed to open debug file :** *name*
Failed to close debug file : *name*

These are generated when the VAX/VMS version of the **afserver** cannot open or close the debug file *name* that is used to record the communication dialog between the **afserver** and transputer board when loading the transputer board. This generally occurs when the debug file name **afserver.log** cannot be written to because the **afserver** has insufficient privilege or because the debug file is being written to at an illegal position in the file system.

6 **Unknown protocol in boot file :** *name*

This is generated when the VAX/VMS version of the **afserver** is used with the **-:c** option and the boot file *name*, that it is loading, has data within it that the **afserver** does not recognise. This may occur if the **-:c** option has been used when the **-:b** option should have been used.

7 **Missing boot file name parameter**

This is generated when the file option **-:b** (and **-:c** in the VAX/VMS version of the **afserver**) does not have specified with it the boot file name to be loaded onto a transputer board.

8 **Missing option flag parameter**

This is generated when the option **-:o** does not have specified with it the option flag value that may (or may not) be used by the program the **afserver** is communicating with (or loading).

9 **Missing board size parameter**

This is generated when the option **-:s** does not have specified with it the value that is to be used as the size of the transputer board with which the **afserver** is communicating with (or loading).

10 **Missing link address parameter**

This is generated when the option `-:l` does not have specified with it the base IBM port address that is used by the IBM/DOS version of the **afserver** to communicate with a transputer board.

11 **`Missing load address parameter`**

This is generated when the option `-:a` does not have specified with it the value that is to be used as the base address address from which code is loaded using the VAX/VMS version of the **afserver**. (It can only be used in conjunction with the `-:b` option.)

12 **`Server terminated: bad protocol when expecting INT32`**

This is generated when the **afserver** is expecting to be sent a word from the transputer board it is communicating with but is in fact sent something else. That is, the **afserver** is expecting to receive the tag **int32.value** but has been sent something else.

13 **`Server terminated: bad protocol when expecting record`**

This is generated when the **afserver** is expecting to be sent a record from the transputer board it is communicating with but is in fact sent something else. That is, the **afserver** is expecting to receive the tags **record32.value** or **nilrecord.value** but has been sent something else.

14 **`Server terminated: illegal filer command received`**

This is generated when the **afserver** is sent a filer command from the transputer board it is communicating with that it does not recognise.

15 **`Server terminated : aborted by user`**

This is generated when the **afserver** is terminated by using the break key. Typical break keys are control-C and control-Y.

16 **`Server terminated: cannot initialise host`**

This is generated when the **afserver** is unable for some reason to set up the host machine, e.g. unable to open the i/o channels used to communicate with the transputer board.

17 **`Server terminated: cannot boot root transputer`**

This is generated when the **afserver** is unable to load the code from the specified boot file (using the option `-:b` or `-:c`) onto the target transputer board. This will be caused by any error that has occured prior to the **afserver** communicating with the target transputer board, e.g. unable to open the boot file etc.

18 **`Server terminated: bad command line`**

This is generated when the **afserver** cannot correctly parse the command line that it has been given. This will in general be caused when an option is used which expects a parameter and none is given, e.g. the option `-:b` etc.

19 **`Server terminated: error in transputer system`**

This is generated when the IBM/DOS version of the **afserver** is used with the option `-:e` and the **afserver** detects that the error flag on the transputer board it is communicating with is set.

B Making the IBM/DOS loader

This appendix covers how to re-compile the loader program (**afserver.exe**). For this to be possible it will be necessary to have access to the Microsoft C compiler and associated software. If this is not possible then the supplied source may have to be modified in order for it to compile using a non-Microsoft C compiler.

To make things simple a **make** file is supplied (**makefile**) and a file containing the responses for the linker (**afserver.rsp**). If the **make** program exists then all that is necessary to re-make the loader is to issue the following command:

```
> make makefile
```

If the **make** program does not exist then the following sequence of commands can be executed instead:

```
> msc /Ot /Ze boot.c;

> msc /Ot /Ze linkio.c;

> msc /Ot /Ze version.c;

> msc /Ot /Ze afserver.c;

> masm quickio.asm;

> link @afserver.rsp
```

To make the NEC version of the loader it is necessary to undefine the flag **IBMserver** in the header file **version.h** and instead define a flag called **NECserver**.

Also, if a version is to be made which accommodates Rev A T414's then the flag **REV_A_FIX** in the files **version.h** and **quickio.asm** will have to be defined. Note that if this is done then changes will have to be made to programs which communicate with the loader to take into account the loader's accommodation of Rev A T414's. The main differences between the version of the loader compiled to take into account Rev A T414's and a version which has not are:

- If a single byte value is to be transferred across the link to or from the loader then it is expanded to a four byte transfer.
- If a single byte string or record is to be transferred across the link to or from the loader then it is expanded to a two byte transfer.

These differences ensure that no single byte transfers are performed by the loader and the program it is communicating with.

C Making the VAX/VMS loader

This appendix covers how to re-compile the loader (**afserver.exe**). For this to be possible it will be necessary to have access to the DEC VAX/VMS C compiler and associated software. If this is not possible then the supplied source may have to be modified in order for it to compile using a non-DEC VAX/VMS C compiler.

To make the loader the the following sequence of commands should be executed:

```
$ cc boot.c

$ cc linkio.c

$ cc version.c

$ cc afserver.c

$ link afserver+version+linkio+boot /exe=afserver.exe
```

Also, if a version is to be made which accommodates Rev A T414's then the flag **REV_A_FIX** in the file **version.h** will have to be defined. Note that if this is done then changes will have to be made to programs which communicate with the loader to take into account the loader's accommodation of Rev A T414's. The main differences between the version of the loader compiled to take into account Rev A T414's and a version which has not are:

- If a single byte value is to be transferred across the link to or from the loader then it is expanded to a four byte transfer.
- If a single byte string or record is to be transferred across the link to or from the loader then it is expanded to a two byte transfer.

These differences ensure that no single byte transfers are performed by the loader and the program it is communicating with.

D Making the SUN/UNIX loaders

This appendix covers how to re-compile the loader program (**afserver**). For this to be possible it will be necessary to have access to the SUN/UNIX C compiler and associated software. If this is not possible then the supplied source may have to be modified in order for it to compile using a non-SUN/UNIX C compiler.

To make things simple a **make** file is supplied (**Makefile**), If the **make** program exists then all that is necessary to re-make the loader is to issue the following command:

```
$ Make Makefile
```

If the **make** program does not exist then the following sequence of commands can be executed instead:

```
$ cc -O -c boot.c

$ cc -O -c linkio.c

$ cc -O -c version.c

$ cc -O -c afserver.c

$ cc -O afserver.o version.o linkio.o boot.o -o afserver
```

Also, if a version is to be made which accommodates Rev A T414's then the flag **REV_A_FIX** in the file **version.h** will have to be defined. Note that if this is done then changes will have to be made to programs which communicate with the loader to take into account the loader's accommodation of Rev A T414's. The main differences between the version of the loader compiled to take into account Rev A T414's and a version which has not are:

- If a single byte value is to be transferred across the link to or from the loader then it is expanded to a four byte transfer.
- If a single byte string or record is to be transferred across the link to or from the loader then it is expanded to a two byte transfer.

These differences ensure that no single byte transfers are performed by the loader and the program it is communicating with.