Occam input and output procedures for the TDS

INMOS Technical Note 28

Michael Poole

May 1988
72-TCH-028

Contents

1 Introduction

Programming languages designed with a concise defining document as one of the principal design goals usually leave the design and implementation of a collection of procedures for input and output of text and numbers to the user. Occam is such a language. However it is sensible for an implementer to help the users by providing a basis on which they can build.

Accordingly, procedures for input and output have now been included in the software shipped as part of the transputer development system (TDS). The purpose of this note is to introduce these procedures and to explain some of the guiding principles which have gone into their design and implementation. The fact that users will have requirements not exactly met by these procedures is acknowledged and the occam source of the procedures is provided as a basis for enhancement where this is seen to be necessary.

Occam is defined in a reference manual [1]. The TDS is introduced in another technical note in this series [2]. Reference is also made below to INMOS product documentation for readers who have access to the product. Some details in this note apply particularly to the version of the TDS sold as IMS D700D, but as far as possible similar facilities are available in other versions of the TDS.

Many of the procedures would be suitable for use with any implementation of occam. However this note is particularly concerned with procedures designed to be used in programs developed and tested within the TDS itself. Such programs, when written using these procedures, may easily be converted to run on arbitrary transputer hardware configurations.

1.1 Input and output

In occam the terms input and output strictly apply to the low level communications between processes executing in parallel. These communications use occam channels, which may or may not correspond to physical hardware links, and are made up of bit streams organised as sequences of bytes corresponding to values represented in the occam type system.

In general usage the terms input and output apply more loosely to the transmission of values (text and/or binary numbers) between a program and its physical peripherals such as keyboards, screens, printers, or disks and other mass storage systems, usually abstracted by an operating system as a filing system.

Occam permits the abstraction of peripheral devices, possibly with their low level driving software, as processes connected to their environment by channels. This view enables the two levels of input/output to be merged.

Input procedures receive their input data along a channel, whose identity is passed to the procedure as a parameter. The values received are passed to the caller by means of element parameters (parameters whose specification permits their values to be changed).

Output procedures send their output data along a channel, whose identity is passed to the procedure as a parameter. The data are passed to the procedure for output as value parameters.

1.2 Packaging of the procedures

Pre-written procedures can be provided in various ways:

1. Predefined by the compiler and converted directly to in-line code
2. Predefined by the compiler and compiled into calls to the system library
3. Provided in a user library
4. Provided as separately compilable unit(s)
5. Provided as source code possibly including free variable references.

The input and output procedures supplied with the TDS are packaged in a group of user libraries, some of which use other libraries and also the predefined procedures provided by the compiler. The allocation of procedures to libraries is indicated in section 5.

Because of limitations with the library mechanism the names of all procedures used in a program from user and system libraries must be unique.

Alt the procedures in the input/output library are provided both precompiled and as source. If the user's requirements can be met by calling the recompiled procedures, then that is the preferred way to use them. However it is accepted that some users will have particular requirements which may best be met by adapting the procedures to meet these requirements.

1.3 Structure of this note

This note is structured as five main sections.

Section 1 is this introduction.

Section 2 discusses conventions for the use of channels in occam programs, with particular reference to the channels defined by the TDS itself.

Section 3 discusses the procedures provided with the TDS for calling in sequence in arbitrary occam programs. These procedures include some which are applicable in any occam program, some which assume sequential text-oriented devices, and some which are only meaningful in the presence of a folded file store as provided by the TDS. These procedures are collectively called the user procedures. An example using these procedures is included.

Section 4 discusses the procedures provided with the TDS for calling in parallel with applications using the user procedures. These enable programs to be easily adapted to support different implementations of sequential text files, including those found in conventional host text files and in the TDS folded file store. These procedures are called the interface procedures. An example of the use of these procedures is given.

Section 5, subdivided for the user procedures and the interface procedures, lists all the procedures and gives a brief statement of the function of each. The structure of these lists is related to the packaging of the procedures in library files.

2 Conventions for the use of channels

It is desirable to be able to use the same input procedures for receiving sequences of characters from channels whose other end may be any kind of character source (keyboard, file, process) and to use the same output procedures for sending characters out along channels whose other end may be any kind of character sink (screen, printer, file, process). To achieve this it is necessary to adopt a set of rules and conventions which determine the representation of information sent along the channels.

The simplest communication paths are those where only the sender has control and the receiver must be prepared to receive everything sent, up to and including an agreed terminator. Input from a keyboard and output to a screen or printer can usually be handled satisfactorily in this way, and require a single occam channel from the sender to the receiver.

There are, however, situations where the receiving process needs to be able to control what the sender is sending. The simplest example is when the receiver wishes to tell the sender to stop sending; more complex cases arise when the receiver can influence which alternative the sender selects out of an available set of alternatives. These situations require a pair of occam channels, one in each direction. The sender sends data and control information and receives commands and possibly error indications. The receiver receives data and control information and sends commands and error indications. If a pair of processes is connected by a pair of channels then the identity of the sender and receiver could change from one to the other during the execution of the program.

The channel usage conventions adopted in the input and output procedures provided with the TDS are determined by the TDS itself. They are the conventions used by the `keyboard`, `screen` and `user.filer` channels which are passed by the TDS to an executable program (`EXE`) running in the TDS. `EXE`s are discussed further in [2].

They are, however, not restricted to use on these particular channels, and interface procedures are provided to perform such tasks as receiving input from a file as if from a keyboard, sending output to a file as if to a screen, merging screen streams, duplicating screen streams, etc.

2.1 Key stream conventions

A stream of characters deriving from a keyboard must be capable of being received by a receiver who never has any knowledge of what is coming. As arbitrary byte values may be possible the protocol is defined to be `INT` with non-negative values being valid data values, and negative values indicating control and error conditions. A particular negative value `ft.terminated` is used as a general terminator on a key stream. The range of possible positive values is determined by the hardware or software generating them, but will normally include at least the full ASCII character set.

A key stream channel may be used for any simple one way communication of a sequence of positive integers.

Some procedures take note of the control characters `'*c'` and `'*n'` (ASCII CR and LF), the normal rule being that the former is the line terminator, and the latter is ignored. The input procedures themselves have no knowledge of the more advanced features of the TDS keyboard interface, such as the encoding of function keys (or key sequences) as integers `>= 200`.

2.2 Screen stream conventions

The TDS screen interface is based on the requirements of programs such as editors and the problems deriving from the incompatible control features provided by different terminal types. The screen channel protocol is a tagged protocol, but for historical reasons cannot be described as such in the occam language (its invention was before the language was fully defined). Each communication consists of a one byte tag followed, according to the tag, by zero, one or more specific communications of bytes, integers or byte arrays.

For the purpose of the output procedures the significant feature of this protocol is its ability to package strings or lines of text into single communications. In some situations this will give a performance advantage over sending each character individually.

A special tag `tt.endstream` is defined to act as a stream terminator.

A screen stream channel may be used for any one-way communication of text, with the option to include screen control commands, if the ultimate destination has the ability to process them.

As the TDS screen requires lines of text to be separated by `"*c*n"` it is the convention that both of these control characters must be sent at the end of each line.

2.3 User filer channel conventions

The uses filer is a process in the TDS which provides an `EXE` with a means of communicating with a part of the folded file structure identified by the current cursor position at the time the `EXE` is called. Access to the user filer requires a pair of channels with a versatile tagged protocol permitting fully flexible access to hierarchically structured data in the TDS folded file store.

This protocol is discussed in full in the 'System interfaces' chapter in [3].

2.4 The other channels between the TDS and an EXE

An `EXE` has access to a set of implicit parameters provided to it by the TDS when it is called. These include a variety of channels between the TDS and the code of the `EXE`. These parameters are recognised by the compiler by their names. For further details see [2] and [3].

Channels to the terminal and user filer have been mentioned. The other channels between an `EXE` and the TDS are provided for special purposes. One pair of these channels needs to be mentioned here. This is the channel pair to and from the TDS server which may be used to perform accesses direct to the host filing system. Some procedures are provided which use these channels and further details are given in [3].

3 User procedures - to be called in sequence

3.1 An introductory example

An example, simplified from [3], shows some of the simple input and output procedures being used to read and write numbers and text. These procedures are called in sequence with the rest of the computation in the same way as the input and output procedures of other programming languages.

```
#USE uservals
#USE userio
SEQ
  newline (screen)
  write.text.line (screen,
   "Type a sequence of real numbers terminated by 0.0")
  newline (screen)

  REAL32 x:
  INT kchar:
  [1000]REAL32 ax:
  INT j:
  SEQ
    x := 1.0(REAL32)
    j := 0
    WHILE x <> 0.0(REAL32)
      SEQ
        write.char (screen, '>')
        read.echo.char (keyboard, screen, kchar)
        read.echo.real32 (keyboard, screen, x, kchar)
        IF
          kchar = ft.number.error
            write.char (screen, '!')
          TRUE
            SKIP
        ax[j] := x
        j := j + 1
    newline (screen)
    write.text.line (screen, "These are the numbers you typed")
    newline (screen)
    SEQ i = 0 FOR j
      SEQ
        write.real32 (screen, ax[i], 10, 10)
        newline (screen)

  write.full.string (screen, "Type ANY to return to TDS")
  INT any:
  read.char (keyboard, any)
```

The general style of the simple input and output procedures may be observed in the example. The first parameter of the procedure identifies the channel for communication. Subsequent parameters define the value to be communicated or the variable to receive the communication.

3.2 Number conversion procedures

A set of number conversion procedures as defined in [1] is provided. These procedures perform conversions between arrays of bytes (containing ASCII characters) and integer and real numbers in all the occam types. Integer numbers may be represented either in decimal or in hexadecimal notation. Real numbers may be either in fixed point representation or in floating point with a signed decimal exponent.

For completeness, procedures for boolean type are also provided.

The input conversion procedures each have three parameters: a boolean error flag (set if a legal conversion cannot be performed) the result and the string to be converted.

The output conversion procedures each have three or more parameters: an integer returned as the number of characters generated an array into which the characters are stored, the value to be converted and, when necessary, integers to define the format.

These procedures may be considered to be defined as part of the occam language, as they are totally independent of any channels provided by a particular run-time environment.

3.3 Simple input procedures

Two alternative groups of input procedures are provided. The first group reads strings from an input channel with BYTE protocol, using either a space or ' *c' as terminator. The second group uses a TDS key stream input channel and is suitable for reading arbitrary text with embedded numbers. This group includes variants of the procedures for use when input is from an interactive keyboard to which the input characters must be echoed and at which simple line editing (character delete) operations can be supported.

The style of coding for which the TDS input procedures has been designed is sequential reading of single characters, switching to an appropriate number input procedure when a digit or other significant character is encountered. All number input procedures have an integer parameter which is the value of this 'read ahead' character on input, and is the value of the character which terminated the number on exit.

3.4 Simple output procedures

The simple output procedures generate a TDS screen stream on their output channel. This channel can be connected either directly to the screen channel of an EXE or to the input of any process designed to accept such a stream. These processes may provide a route to a screen, a printer, a file or any other process which expects a sequence of textual input.

Two modes of working are supported: the first enables individual values to be converted into a sequence of characters which is immediately sent to the channel; the second allows lines of text to be accumulated in a buffer array before being sent to the output in a single operation.

A program using these procedures may be trivially converted to a different run-time environment, either by use of interface procedures as discussed below, or by recoding the procedure bodies to use the conventions of the new run-time environment.

Procedures for terminal-type dependent operations such as cursor control are also provided but are meaningful only when the receiving software can generate the character sequences appropriate for the type of terminal in use.

3.5 User filer procedures

This and the following sub-section assume knowledge of the TDS, and in particular its concept of a folded file store.

In order to give EXEs running in the TDS flexible access to the folded file structure the TDS includes a process called the user filer, which may be connected to an EXE by one or more pairs of channels obeying a bidirectional protocol. This protocol is defined in terms of command tags and their parameters, and corresponding reply tags and their parameters. Meaningful sequences of these communications are also specified in the TDS reference manual [3].

Procedures are provided which embody most of the frequently used communications across the user filer interface.

In particular, sets of procedures are provided which support sequential access to folded data streams, both for input and for output. These treat such streams as sequences of items, each item corresponding to a line on the screen in the editor's representation of the stream. There are therefore procedures corresponding to the input and output of top and bottom creases (filed and ordinary), record items and number items.

There are two groups of folded input procedures. The first group is designed for an exhaustive sequential pass through a folded input stream and returns the data of the current item and the tag defining the type of the following item. The second group splits these two components and so gives the user the option to decide to skip folds, or to repeat or prematurely exit from a fold.

3.6 Other procedures

The other procedures provided include:

1. string handling procedures and functions
2. procedures supporting the channels in an `EXE` which provide direct access to host files,
3. procedures supporting the alien filer interface protocol for programs running under the host file server,
4. procedures supporting access to peripherals of transputer evaluation boards.

4 Interface procedures - to be called in parallel

It is often desirable, when writing programs to read or write sequential text streams, to design the program to be independent of whether the input sources and output destinations are peripheral devices, files or processes. For this purpose a set of interface procedures is provided.

Calls, or instances, of one or more of these procedures are then suitable as processes to be run in parallel with an application process to obtain the effect required. The same application code (written as a separately compilable procedure) may be called in parallel with different combinations of interface procedures to take inputs from or direct outputs to a variety of sources and sinks.

The interface procedures are designed for use in programs which process streams of text to exhaustion. Input from a file or output to a file may require a pair of channels, but otherwise connections require a single channel each.

When building these procedures into a program it is important to ensure that every interface procedure will terminate. Interface procedures with a single input channel are terminated by sending a terminator on that channel. Multiplexors have a special stopper channel.

4.1 Protocol converters

Interface procedures are provided for reading key streams from host files and from TDS folded files in data mode (ignoring non-text folds and all the creases), and for writing screen streams to files of both these types.

Procedures are also provided for simple copying (buffering) of screen streams, for converting screen streams to simple byte streams for commonly used screen types, for converting from key stream to screen stream protocol, and for saving a screen stream in an array and subsequently regenerating it.

4.2 Multiplexors, etc.

Some of the interface procedures do not change the protocol but merely serve to join together various components of a program. Such procedures have different numbers of input and of output channels.

The screen multiplexor takes any number of input channels and merges screen stream protocol messages on these to a single output channel. For practical purposes it is probably desirable for the merged streams to be organised as sequences of complete lines of text, but the multiplexor does not enforce this mode of use.

The screen fan out procedure makes two copies of a screen stream input. It can therefore be used, for example, to file a copy of what is sent to the screen.

Key stream and screen stream sink procedures are provided for consuming streams which are no longer wanted, such as diagnostic output.

Examples of the use of interface procedures are given in [3].

4.3 An example calling interface procedures in parallel

This example shows the output from an application (arbitrarily called `big.numbers`) being duplicated by a call of `scrstream.fan.out`, and then one of these outputs being sent to a file.

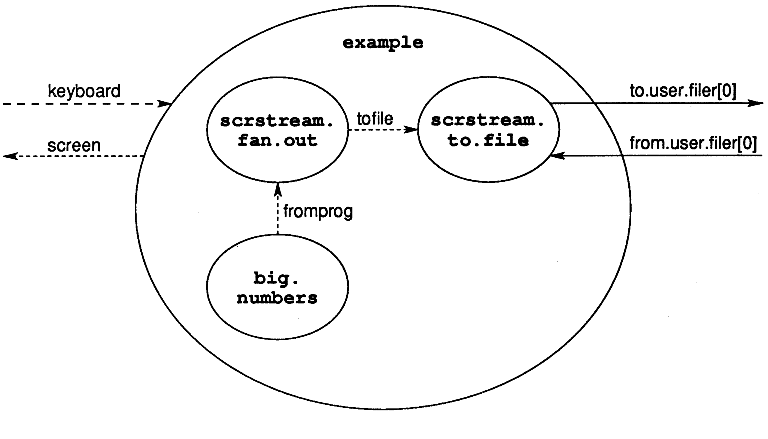
```
#USE userio
#USE interf
SEQ
  -- This example uses screen output with a copy sent to a file

  PROC big.numbers (CHAN OF ANY screen)
    ... any application code with a screen stream output
```

```
:

CHAN OF ANY fromprog, tofile:
INT foldnum, any:
PAR
  -----------------------------------------------------
  SEQ
    big.numbers (fromprog)
    write.endstream (fromprog)
  -----------------------------------------------------
  SEQ
    scrstream.fan.out (fromprog, tofile, screen)
    write.endstream (tofile)
  -----------------------------------------------------
  SEQ
    scrstream.to.file (tofile, from.user.filer[0],
          to.user.filer[0], "big.numbers", foldnum, error)
  -----------------------------------------------------
write.full.string (screen, "Type ANY to return to TDS")
read.char (keyboard, any)
```

The process structure of this program may be represented by the diagram below. In this diagram channels with different protocols are represented by lines drawn in different styles.



5 Table of procedures in the D700D libraries

Each of the library files provided in the IMS 07000 software package contains one or more separately compiled groups of procedures. Within the description of a library the groups of procedures are indicated. The occam compilation system permits only those groups of procedures which are required in a program to be included in the object code generated.

There are some interdependencies between the various libraries themselves. Programmers only need to mention those libraries they use explicitly. The majority of programs will only use procedures from the library `userio` and any necessary interface procedures.

The description here is purposely brief, as it is intended that serious users should study the detailed documentation [3], or the occam source of the procedures themselves. Note that libraries are named according to the host file in which they are found.

5.1 User procedures

Library `ioconv` - number/string conversions

These procedures for simple number to string conversions (and vice versa) are defined in [1]. They are used by the simple input output procedures in `userio`.

`INTTOSTRING`	Convert integer to decimal string
`STRINGTOINT`	Convert decimal string to integer
`HEXTOSTRING`	Convert integer to hexadecimal string
`STRINGTOHEX`	Convert hexadecimal string to integer
`BOOLTOSTRING`	Convert boolean to 'TRUE' or 'FALSE'
`STRINGTOBOOL`	Convert 'TRUE' or 'FALSE' to boolean

Library `extrio` - more number/string conversions

These extend the previous group for the extra integer and real types.

`INT16TOSTRING`	Convert 16-bit integer to decimal string
`INT32TOSTRING`	Convert 32-bit integer to decimal string
`INT64TOSTRING`	Convert 64-bit integer to decimal string
`STRINGTOINT16`	Convert decimal string to 16-bit integer
`STRINGTOINT32`	Convert decimal string to 32-bit integer
`STRINGTOINT64`	Convert decimal string to 64-bit integer
`HEX16TOSTRING`	Convert 16-bit integer to hexadecimal string
`HEX32TOSTRING`	Convert 32-bit integer to hexadecimal string
`HEX64TOSTRING`	Convert 64-bit integer to hexadecimal string
`STRINGTOHEX16`	Convert hexadecimal string to 16-bit integer
`STRINGTOHEX32`	Convert hexadecimal string to 32-bit integer
`STRINGTOHEX64`	Convert hexadecimal string to 64-bit integer
`STRINGTOREAL32`	Convert decimal real string to real32 value
`STRINGTOREAL64`	Convert decimal real string to real64 value
`REAL32TOSTRING`	Convert real32 value to decimal real string
`REAL64TOSTRING`	Convert real64 value to decimal real string

Library `strings` - string handling procedures and functions

Character manipulation

`is.in.range`	Checks if a byte is within a range
`is.upper`	Checks if a byte is an ASCII upper case letter
`is.lower`	Checks if a byte is an ASCII lower case letter
`is.digit`	Checks if a byte is an ASCII digit
`is.hex.digit`	Checks if a byte is an ASCII hexadecimal digit
`is.id.char`	Checks if a byte is valid in an occam identifier
`to.upper.case`	Converts all letters in string to upper case
`to.lower.case`	Converts all letters in string to lower case

String handling

`compare.strings`	Compares strings lexicographically
`egstr`	Check strings for equality
`str.shift`	Moves a sub-array within an array of bytes
`delete.string`	Deletes bytes from a string
`insert.string`	Inserts a string within a string
`string.pos`	Finds a match of a string in a string
`char.pos`	Finds a match of a byte in a string
`search.match`	Looks for a match of one in a set
`search.no.match`	Looks for a match of one not in a set

Appending text and numbers to text lines. These procedures allow the cumulation of text into a line buffer.

`append.char`	Append byte to a line
`append.text`	Append text to a line
`append.int`	Append decimal integer to a line
`append.int64`	Append decimal integer to a line
`append.hex.int`	Append hexadecimal integer to a line
`append.hex.int64`	Append hexadecimal integer to a line
`append.real32`	Append decimal real number to a line
`append.real64`	Append decimal real number to a line

Library `userio` - user input and output procedures

Simple input procedures

These procedures support input from a key stream, with and without echo.

`read.echo.char`	Read and echo one byte
`read.char`	Read one byte
`read.echo.text.line`	Read and echo a line of text
`read.text.line`	Read a line of text
`read.echo.int`	Read and echo a decimal integer
`read.int`	Read a decimal integer
`read.echo.hex.int`	Read and echo a hexadecimal integer
`read.hex.int`	Read a hexadecimal integer

Simple output procedures

These procedures support output to a screen stream.

`write.char`	Output one byte
`write.int`	Output decimal integer as characters
`write.hex.int`	Output hexadecimal integer as characters
`write.len.string`	Output string with computed length
`write.full.string`	Output fixed length string
`newline`	Output ”*c*n”
`write.text.line`	Output a complete line of text
`write.endstream`	Terminate a stream in screen protocol

Procedures for the extra types

These procedures support input and output of values in the extended types. This group is incomplete and can be extended by the user if necessary.

`write.int64`	Output decimal integer as characters
`write.hex.int64`	Output hexadecimal integer as characters
`write.real32`	Output a 32-bit real value in decimal
`write.real64`	Output a 64-bit real value in decimal
`read.int64`	Read a 64-bit integer number
`read.echo.int64`	Read and echo a 64-bit integer number
`read.hex.int64`	Read a 64-bit hexadecimal integer
`read.echo.hex.int64`	Read and echo a 64-bit hexadecimal integer
`read.echo.real32`	Read and echo a 32-bit real number
`read.reai32`	Read a 32-bit real number
`read.echo.real64`	Read and echo a 64-bit real number
`read.real64`	Read a 64-bit real number

Control codes to a screen stream

`goto.xy`	Move cursor to absolute screen position
`clear.eol`	Clear to end of line
`clear.eos`	Clear to end of screen
`beep`	Send BELL character
`up`	Move cursor up
`down`	Move cursor down
`left`	Move cursor left
`right`	Move cursor right
`insert.char`	Insert char at cursor
`delete.chl`	Delete char to left of cursor
`delete.chr`	Delete char at the cursor
`ins.line`	Insert blank line
`del.line`	Delete line

Folded stream output

These procedures support straightforward output to the folded file store of the TDS.

`create.new.fold`	Create empty fold for writing
`write.record.item`	Write a record to fold stream
`write.fold.top.crease`	Write top crease to fold stream
`write.filed.top.crease`	Write filed top crease to fold stream
`write.bottom.crease`	Write bottom crease to fold stream
`write.number.item`	Write a number item to fold stream
`finish.folded.stream`	Finish a newly written fold stream

Folded stream input

These procedures support input from the folded file store of the TDS. The read. procedures read ahead the tag of the following item, the input. procedures do not.

`read.fold.heading`	Read fold header and attributes
`read.file.name`	Read file name on fold
`open.folded.stream`	Open folded stream for reading
`read.record.item`	Read record and type of next item
`read.fold.top.crease`	Read top crease and type of item within
`read.filed.top.crease`	Read filed top crease and type of item within
`read.bottom.crease`	Read bottom crease and type of next item
`read.number.item`	Read number item and type of next item
`read.error.item`	Read error item
`close.folded.stream`	Terminate reading of folded stream
`input.record.item`	Input a record item
`input.number.item`	Input a number item
`input.top.crease`	input a top crease item
`exit.fold`	Exit fold and return to enclosing fold
`repeat.fold`	Return to start of current fold
`skip.item`	Move to next item
`enter.fold`	Move to first item within fold

Library `slice` - block transfer procedures

These procedures may be used for communicating blocks of bytes.

`assign.bslice`	Copy an array of bytes
`output.len.bslice`	Output a length and a block
`input.len.bslice`	Input a length and a block

Library `ufiler` - user filer procedures

This group of procedures supports the user filer interface at a more intimate level than the folded stream procedures in `userio`.

`get.stream.result`	Read result of user filer command
`clean.string`	Make string suitable for file name
`truncate.file.Id`	Remove filename extension
`write.fold.string`	Write fold comment text
`create.fold`	Add new fold to end of fold bundle
`send.command`	Send user filer command mode command
`make.filed`	Make a fold into a filed fold
`open.stream`	Open a folded stream
`read.data.record`	Read a record from a data stream
`read.fold.string`	Read the fold comment text
`read.fold.attr`	Read the fold attributes
`number.of.folds`	Count the folds in the bundle
`open.data.stream`	Open a fold stream in data mode
`close.stream`	Close a fold stream

Library `msdos` - DOS file via TDS server procedures

These procedures are used by the interface procedures which provide access to host files outside the TDS folded file store.

`test.exists`	Test for existence of host file
`make.id`	Make file identity
`file.lock`	Lock host file against multiple access
`file.release`	Release file lock
`open.tkf.file`	Open access to host file
`close.tkf.file`	Close access to host file
`write.block`	Write a block to a host file
`read.block`	Read a block to a host file
`read.line`	Read a line of text from a block

Library `derivio` - derived number inputs

These procedures are also described in [3] and support simple input and output using channels with `BYTE` protocol.

`GETSTRING`	Read a line or word from a CHAN OF INT
`INTREAD`	Read an integer from a CHAN OF INT
`INT16READ`	Read a 16-bit integer from a CHAN OF INT
`INT32READ`	Read a 32-bit integer from a CHAN OF INT
`INT64READ`	Read a 64-bit integer from a CHAN OF INT
`HEXREAD`	Read a hexadecimal integer from a CHAN OF INT
`REAL32READ`	Read a real32 value from a CHAN OF INT
`REAL64READ`	Read a real64 value from a CHAN OF INT
`INTWRITE`	Send an integer value to a CHAN OF BYTE
`INT16WRITE`	Send a 16-bit integer to a CHAN OF BYTE
`INT32WRITE`	Send a 32-bit integer to a CHAN OF BYTE
`INT64WRITE`	Send a 64-bit integer to a CHAN OF BYTE
`HEXWRITE`	Send an integer value in hexadecimal to a CHAN OF BYTE
`REAL32WRITE`	Send a real32 value in decimal to a CHAN OF BYTE
`REAL64WRITE`	Send a real64 value in decimal to a CHAN OF BYTE

Library `afiler` - alien filer procedures

These correspond exactly to the operations with the same names described in the section on the 'Host file server' in [3]. They are provided so that occam programs may do all the host operations available to the scientific languages.

`read.key`	Read a character from the keyboard
`read.key.wait`	Wait for a character from the keyboard
`open.file`	Obtain access to file
`read.block`	Read a block from a stream
`write.block`	Write a block to a stream
`seek`	Move to a position in a stream
`close.stream`	Close a stream
`open.temp`	Create temporary file
`open.input.stream`	Open a standard input stream
`open.output.stream`	Open a standard output stream
`terminate.filer`	Close down simple filer interface
`set.result`	Return result to server
`rename.file`	Change name of host file
`run.command`	Run host command line
`read.time`	Read host's clock time
`receive.block`	Receive data block from host
`send.block`	Send data block to host
`read.core.dump`	Read block from core dump file
`server.version`	Determine host type and server version

5.2 Interface procedures

These procedures are listed separately solely to match the structure of the main part of this note. They are organised in groups in a similar manner to the user procedures.

Instances of these procedures are suitable for calling in parallel with application code using the user procedures listed above. Each procedure has channel parameters which must be 'joined' to others using a matching protocol.

Library `interf` - interface procedures

These procedures handle the standard TDS protocols used by the procedures in `userio`.

Terminal to file protocol converters

These procedures allow files accessed sequentially to be treated in the same way as terminals.

`keystream.from.server`	Convert from host text file to keyboard protocol
`keystream.from.file`	Convert from user filer to keyboard protocol
`scrstream.to.server`	Convert from screen protocol to host text file
`scrstream.to.file`	Convert from screen to user filer protocol
`scrstream.to.ANSI`	Convert screen stream to ANSI byte stream
`scrstream.to.TVI920`	Convert screen stream to TVI920 byte stream
`keystream.to.screen`	Convert from integer characters to screen protocol

Multiplexors, stream sinks, etc.

These procedures split and join screen streams and act as dummies to consume streams which are no longer wanted.

`scrstream.multipiexor`	Merge inputs to one output (screen protocol)
`scrstream.fan.out`	Duplicate stream in screen protocol
`scrstream.to.array`	Save a screen stream for later regeneration
`scrstream.from.array`	Regenerate a saved screen stream
`scrstream.copy`	Suffer a screen stream
`scrstream.sink`	Consume stream in screen protocol
`keystream.sink`	Consume stream in keyboard protocol

Library `afinterf` - alien filer interface procedures

`keystream.from.afserver`	Generate key stream from AF file
`scrstream.to.afserver`	Send screen steream to AF file
`af.buffer`	Buffer an AF channel pair
`af.multiplexor`	Multiplex AF channel pairs

Library `t4board, t2board` - transputer board procedures

These procedures support terminal access on INMOS evaluation boards.

`B00x.term.p.driver`	Keyboard and screen handler for B001 and B002 via RS232 port
`B006.term.p.driver`	Keyboard and screen handler for B006 via RS232 port
`scrstream.to.B004.link`	Screen handler for B004 via PC link
`keystream.from.B004.link`	Keyboard handler for B004 via PC link
`terminate.server`	Termination routine for the TDS server

References

[1] Occam 2 reference manual, Prentice Hall, London 1988

[2] Occam program development using the IMS D701 transputer development system, Technical note 16, INMOS Ltd, Bristol 1988

[3] Transputer development system, Prentice Hall, London 1988