Example programs in the TDS

INMOS Technical Note 56

Michael Poole

September 1988
72-TCH-056-00

Contents

1 Introduction

The INMOS Transputer Development System IMS D700D (TDS) is a package of software and associated documentation for the development of occam programs for transputers and networks of transputers.

The software supplied includes a variety of occam programs as examples which users can study, compile, run, or use as a basis for their own experiments. Most of these examples are not discussed at all in the documentation, and the purpose of this note is to describe these programs so that readers may see if any are likely to be of interest without needing to study all the occam source text.

All occam source text supplied in the TDS may be used, copied and adapted by users at their own risk. Some sources of tools which are part of the TDS system itself may include copyright notices, but permission to copy these sources will not normally be withheld if requested.

The TDS presents the user with a screen interface based on the concept of a folding editor, which is a tree structure or hierarchy (strictly a set of hierarchies) entered at a root and traversed by means of commands from the keyboard. Program source is held within this structure in folds, which may be opened to examine their contents. Some folds also correspond directly to files in the underlying file system.

Most of the examples are coded as programs to be executed within the TDS itself (EXEs). Such programs have the advantage that they can use the TDS as a run-time system for accessing a keyboard, screen and filing system and so users do not need to concern themselves with the more intimate details of these matters. There are also examples of PROGRAMS written to run on networks of transputers. Some of these need run-time support from an EXE running in the TDS or directly from a server running on a host computer. Others communicate solely through a terminal attached to an INMOS evaluation board. The lecture simulation examples described in section 5 below show how a program can evolve form a single EXE running within the TDS to a variety of network PROGRAMS, without the need to change the principal procedures of the example, or even to recompile them if the network is built from the same kind of processors.

A reference manual is supplied with the TDS. References to sections of this manual are given below in the form 'Refman 4.2'.

1.1 A pictorial representation for parallel processes

Some of the examples described in this note involve several processes running in parallel. As an aid to the appreciation of this parallelism, the coding of which is natural in occam, a pictorial representation has been designed. Each picture represents a snapshot of the execution of the program at a time when most of the principal processes are running in parallel. Each process is represented by a box, which is drawn as a rectangle with rounded corners, communicating with other boxes by channels represented by dotted or dashed lines.

Each box is named strictly by reference to the occam source code it describes. This is usually a procedure name, but in some cases is the name given to a process as a comment on the fold containing its source code. The arrows representing channels are drawn with different styles of dots and dashes for different protocols. The variety of protocols is infinite and so the actual choice of styles has been made on an arbitrary basis for the purpose of this technical note. An arrow is named with the name of the channel as declared in the enclosing procedure.

It is a feature of occam processes that they can contain within themselves nested structures of parallel processes, this process nesting is directly represented by the nesting of the boxes in the pictures.

1.2 Structure of this note

The rest of this note is written assuming that readers have an appreciation of the TDS folded file store so that they can locate the programs on their development machines and find their way around them. Newcomers can gain this appreciation by means of the TDS on-line tutorial.

Section 2 briefly discusses this tutorial and other introductory examples described in detail in the User Guide chapters of the Refman.

Section 3 describes a sequence of examples showing how simple input and output operations may be coded in occam by calling procedures from the input/output library. These programs all run as EXEs within the TDS.

Section 4 describes some introductory PROGRAMS for simple networks of transputers on INMOS evaluation boards. They start with single processor networks on boards with RS232 terminal ports, and extend to multiprocessor networks, supported at run time by the host in a variety of ways.

Section 5 describes a more extensive example which is supplied in a variety of different configurations, as an illustration of the ease of adaptation of occam programs to such different structures with minimal need for recompilation.

Finally in Section 6 a brief mention is made of various software tools and other programs which are supplied with the TDS as occam source. These may be used as a source of useful ideas, especially by more advanced users, who may be considering such exercises as rehosting the TDS. Some of these tools are more fully described in other technical notes.

Where subsections of this note refer to particular example programs, the subsection heading matches the fold comment on the example program source.

2 Tutorial examples

These are supplied in the directory `\TDS2\TUTOR`.

By means of the TDS tutorial (see Refman 4.3) a user is taken to the stage where a simple program sending a message to the screen and waiting for a response from the keyboard may be written, compiled and run, using the TDS as the run time support environment.

Also in this directory are the pipeline sorter examples mentioned in various places in the Refman, and the Debugger example mentioned in Refman 9.6.

These are both relatively complex programs involving several processes running in parallel. Newcomers may prefer to look at the input/output library examples mentioned below, before returning to these.

2.1 The pipeline sorter

This example is introduced by Pountain and May in their Tutorial introduction to occam programming, and is revisited in Refman 6.6. The example is fully described in those books. The example program is supplied in a variety of configurations for T4 transputers. If it is being compiled for T8 transputers it is desirable to replace all occurrences of T4 by T8 in the source before compilation. If a multi-processor version of the program is being built then it is possible to use a mixture of transputer types. This will need especial care. If the host transputer is a module (TRAM) on an IMS B008 or similar motherboard it may be necessary to change the link numbers used in some of the configurations.

Each version of this program reads lines of text from the keyboard (a line is terminated by a RETURN), and the program then sorts the characters into ascending order and sends the sorted line back to the screen. A line containing a single percent sign terminates the program.

2.2 The debugger example

The debugger example program is a simple example of a single processor program containing several processes running in parallel, cooperating on the simple task of computing factorials and summing their squares. The number of factorials computed, squared and summed is determined by a number read from the keyboard. If this exceeds 99, then an arithmetic overflow occurs and the debugger may be entered to locate the cause. The use of the debugger on this example is discussed in Refman 9.6.

3 Examples showing the use of the input/output library

These examples are in the directory `\TDS2\EXAMPLES`.

This is a progressive set of simple example programs originally written to aid in testing the procedures in the input/output libraries. These libraries are defined in Refman 14.10 - 14.16 and are also discussed more fully in Technical Note 28 which covers the design principles and coding conventions used.

Their value is in showing these library procedures in use, and acting as a basis for simple programs written by the user who is starting to write occam programs for the first time.

All the examples are EXEs designed to be called from and supported by the TDS. When an EXE is called by the TDS, channels are passed to it in the same way as parameters are passed to an occam procedure. These channels connect the EXE with processes running in parallel with it inside the TDS, which in turn communicate with a multiplexing process which communicates across the hardware link to the host processor. On the host processor a server program provides access to the terminal and filing system of the host.

These channels include the keyboard channel (input to the EXE in key stream protocol), the screen channel (output from the EXE in screen stream protocol), and user filer channels. In the pictures these three kinds of channels are represented respectively by arrows with dashed lines, dotted lines and solid lines. The pictures do not show the TDS itself; arrows going off the edges of the pictures are channels to and from the TDS run-time system.

All these examples may be compiled with all compiler checks switched on, either for a T4 or for a T8. Note that code compiled for a T4 will run unchanged on a T8 (IMS T800) if and only if it does not do any floating point operations. Code compiled for a T8 cannot be run on a T4. The use

of the TDS compiler is introduced in Refman 5.4.

3.1 ex1 - Extreme numbers to screen

This example sends a few lines of output to the screen, using a variety of number output procedures from the library userio. The output display includes the values of the largest numbers in the various number types in occam.

To run, GET CODE the EXE and press RUN EXE.

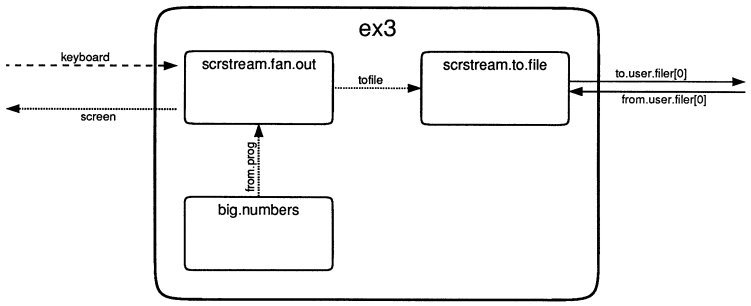
3.2 ex2 - Read a list of real numbers and display it

This example demonstrates the use of number input procedures, with particular reference to the need to read ahead for the first character of a number and the special precautions needed to handle numbers which might be out of range or invalid.

To run, GET CODE the EXE and press RUN EXE. Then type a sequence of real numbers in decimal or hexadecimal notation, terminated by 0.0, at the keyboard. Any character that cannot be part of such a number acts as a separator. The list will be tabulated in a standard form with invalid numbers noted as either `Inf` or `Nan` if necessary.

3.3 ex3 - Extreme numbers to screen and/or a file

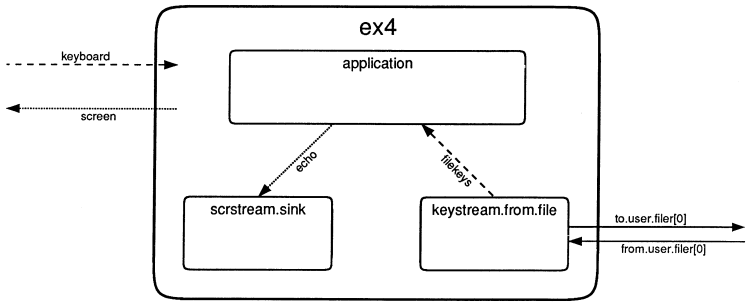
This example is derived directly from ex1. It shows how an output stream may be duplicated to be sent to a file as well as to the screen. This is achieved by calling the application process in parallel with the interface procedure `scrstream.fan.out` and a protocol conversion procedure `scrstream.to.file` which takes a stream in screen stream protocol and writes its contents into a file in the TDS folded file store.



To run, GET CODE the EXE and press RUN EXE with the cursor on an empty fold.

3.4 ex4 - Real numbers from a file

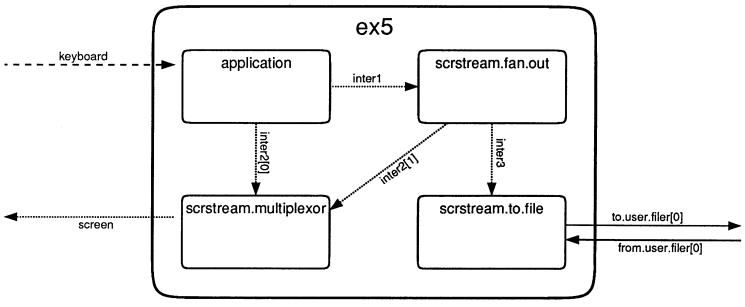
This example is derived directly from ex2. It shows how a filed fold in the TDS folded file store may be read by the procedure `keystream.from.file`, which generates a stream of character codes as integers in key stream protocol. This interface procedure is called in parallel with the application process which is written without any knowledge of whether a real or simulated keyboard is generating its input.



To run, GET CODE the EXE and press RUN EXE with the cursor on any fold containing a sequence of decimal or hexadecimal real numbers, terminated by a 0.0.

3.5 ex5 - Screen multiplexor demonstration

This is a somewhat contrived example designed to show how messages from more than one parallel process may be multiplexed to a single screen. One of the streams in screen stream protocol is also duplicated to a filed fold.



To run, GET CODE the EXE and press RUN EXE with the cursor on an empty fold. Each line of input typed at the keyboard should consist of any

text, any integer and any more text, terminated by a IRETURNg these values are sent to the screen, where echoed input and generated output are identified by tags created by the multiplexor. The action repeats until the integer is a 0.

3.6 ex6 - Create a nested fold structure

Before studying this and the following examples the reader is recommended to read the description of the user filer interface in Refman 16.2. Although the interface is substantially hidden in the examples by the use of library procedures it is useful to have an appreciation of how the interface supports the concepts of folded streams.

The library userio includes a set of procedures (see Refman 14.3.6) which may be used to generate a nested fold structure in the TDS folded file store. This example shows how a simple sequence of calls to these procedures may be coded.

To run, GET CODE the EXE and press RUN EXE with the cursor on an empty fold.

3.7 ex7 - Display text from folded structure

This example demonstrates the use of folded stream input procedures to scan a fold from the TDS folded file store and, in this case, to display its contents as text on the screen. In particular it demonstrates the use of the `fsd.tags` as indicators of the type of the next element in the stream. The folded stream input library procedures used in this example are described in Refman 14.3.7.

To run, GET CODE the EXE and press RUN EXE with the cursor on any fold whose contents are to be displayed.

3.8 ex8 - Select folds of a particular kind

This more substantial example demonstrates some of the more advanced procedures for reading folded streams from the TDS fold structure. Folds whose contexts are not needed are skipped. It also demonstrates the ability simultaneously to read and write folds within a fold bundle.

To run, GET CODE the EXE and press RUN EXE with the cursor on a fold bundle within which is one or more nested structures, including some CODE EXE folds. A new fold will be created at the end of the bundle containing

copies of all CODE EXE folds which are not at the outermost nesting level.

3.9 ex11 - Diagnostic folded text display from nested structure

This example allows the user to explore an arbitrary TDS folded data structure. Any records (including numbers, fold headers, etc.) may be displayed as requested. Folds may be entered, skipped or repeated in response to commands from the keyboard.

To run, GET CODE the EXE and press RUN EXE with the cursor on a fold to be explored. The program will display a menu of possible operations which may be requested.

3.10 ex13 - Copy folded stream

This is a simple fold copying example which may be used as a basis for any program which reads a fold structure and outputs a derived one with the same, or a similar, pattern of nested folds.

To run, GET CODE the EXE and press RUN EXE with the cursor on a fold bundle, the first fold inside which will be copied to a new fold at the end of the bundle.

3.11 ex15 - User filer workout

This example allows the user to exercise most of the user filer commands described in Refman 16.2.5. The calls to this interface are coded using appropriate procedures from the library `ufiler`.

To run, GET CODE the EXE and press RUN EXE with the cursor on a fold bundle. A menu of user filer operations is then presented to the user, and these operations may be requested in turn.

3.12 ex19 - Tryout string procs

This example, written to test the procedures and functions of the library `strings` (Refman 14.12), demonstrates the use of all the routines in this library.

All output is to the screen, and consists of instructions for the performance of a sequence of editing and other exercises on a string entered by the user. These exercises cover string comparison, deletion, insertion, range tests, case

conversion and searching. Each question asked should be answered either by a string terminated by RETURN, or an integer which is interpreted as a position within the string or a count of bytes within the string as appropriate. If difficulties are experienced running this program, it is suggested that the user studies the program source.

3.13 ex20 - Elementary function demonstration

This example displays the values of π and $\sqrt{3}$ on the screen. The output first uses REAL32 arithmetic and then REAL64 arithmetic. It is the only example using the mathematical function libraries. See Refman 14.8 and 14.9.

Note that as supplied this example uses the libraries `snglmath` and `dblmath` which calculate elementary functions using floating point arithmetic, which is available on all processor types either in hardware or in software. Faster functions are available on the T4 using the library `t4math` which uses fixed point arithmetic.

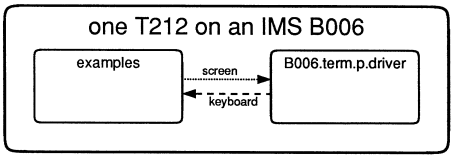
4 Simple transputer network examples

These examples are also in the directory `\TDS2\EXAMPLES`. They may be found in the fold labelled 'simple example PROGRAMS, some with supporting EXEs'. They are useful as a user's first attempts to compile, configure and load network PROGRAMS. See Refman 7.2.

4.1 One T212 on an IMS B006

An IMS B006 is an INMOS evaluation board containing one or more T212 transputers. The master T212 is supported by 64K bytes of memory and an RS232 terminal driver.

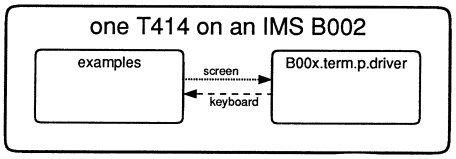
This example consists of examples ex20, ex1 and ex2 called in sequence. The whole lot is called in parallel with the RS232 UART driver `B006.term.p.driver`. See the notes on the separate examples above.



4.2 One T414 on an IMS B002

An IMS B002 is an INMOS evaluation board containing one T414 transputer supported by 2M bytes of memory and an RS232 terminal driver.

This example consists of examples ex20, ex1 and ex2 called in sequence. The whole lot is called in parallel with the RS232 UART driver `B00x.term.p.driver`. See the notes on the separate examples above.



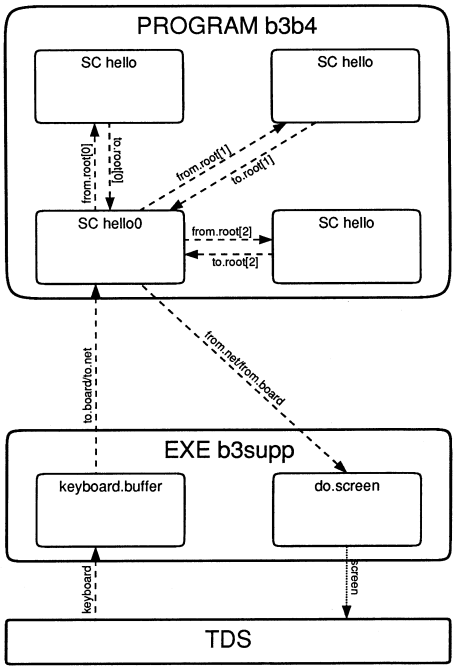
4.3 An IMS B004 and an attached IMS B003

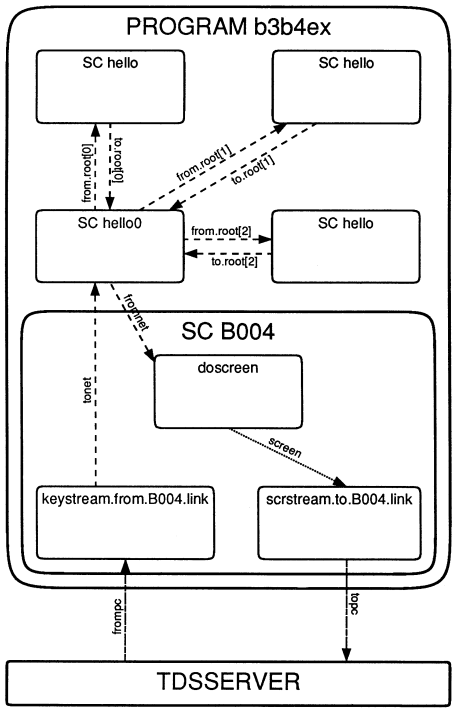
This example includes two variants of a simple 4 processor program suitable for running on an IMS B003. The IMS B003 has four T414 transputers each with 256K bytes of memory connected in a square configuration. In addition to the hard wired connections on the B003 another connection is required across the middle of the square. Each processor is loaded with a simple program which, on request, will send back its identity to one of the four acting as a master, which sends an identification message to the screen. The master in turn communicates with the host for keyboard and screen access.

The first variant runs an EXE on the host which talks to a 4 processor PROGRAM.

The second variant has all 5 processes in the PROGRAM and is loadable directly by the TDS server.

A valuable exercise would be to construct the corresponding program loadable by the host file server. This is most easily written using the library procedure `af.multiplexor` to multiplex keyboard and screen access to the server, replacing the procedures `keystream.from.B004.link` and `scrstream.to.B004.link`. Alternatively advantage could be taken of the known sequence of keyboard inputs and responses to write a simpler version using `read.key.wait` and `write.block`.


PROGRAM b3b4
SC hello
SC hello
from.root[0]
to.root[0]
from.root[1]
to.root[1]
SC hello0
from.root[2]
to.root[2]
SC hello
to.board/to.net
from.net/from.board
EXE b3supp
keyboard.buffer
do.screen
keyboard
screen
TDS


PROGRAM b3b4ex
SC hello
SC hello
from.root[0]
to.root[0]
from.root[1]
to.root[1]
SC hello0
from.root[2]
to.root[2]
SC hello
fromnet
SC B004
doscreen
tonet
screen
keystream.from.B004.link
scrstream.to.B004.link
frompc
topc
TDSSERVER

4.4 Subsystem error monitor

This simple example demonstrates the use of an occam PORT to access the IMS B004 subsystem control. A simple PROGRAM which sets error is also supplied. The subsystem control on an INMOS evaluation board is a mapping into transputer address space of the reset, analyse and error signals to and from an attached network. The connection is realised by a 4-wire reset cable, which must be connected to the network in addition to the link cable(s).

In order to run this example the PROGRAM must be loaded into the network, and then the EXE should be run. The PROGRAM will immediately set error and this condition will be detected by the EXE and reported to the user.

5 The lecture simulation examples

These examples are also in the directory `\TDS2\EXAMPLES`.

The origin of these examples was a need to have a fairly simple example which could be used to support an introductory lecture on occam to a branch meeting of the British Computer Society. The principal requirements were to include examples of use of all the constructs of the language and to be able to show parallel activity on the screen in a reasonably short execution time.

The program was originally written to run on a single processor. The possibility of distributing variants of the program over two or more processors was in mind when it was originally written, and it has since become a useful example program for different styles of program construction, running on transputers with or without run-time support from a file server. Sequential processes in the program have also been rewritten in FORTRAN and Pascal during exercises in building mixed language programs in various ways.

The program simulates a lecture given to a replicated class of students parameterised solely by independent speeds of note-taking. The lecturer reads a file of titles of his slides and writes these titles in a box on one side of the screen. The students write their (trivial) notes in boxes on the other side of the screen. There is a count-down clock whose value is displayed at one second intervals in another box on the screen. The lecture ends either when the clock reaches zero or when the file of titles is exhausted, and so running time may be reduced by using a shorter file of titles. A report on the lecture is written into an output file.

There is a screen handler which is a simple sequential command driven process. The simulation itself is mapped directly on to an appropriate nested set of parallel processes. The simulation is essentially clock driven, with a small degree of indeterminism introduced by means of random numbers determining the time intervals.

In the following sections the various versions of the program are described. Each description includes a diagram showing the principal processes and the channels connecting them. The different protocols used on different channels are represented by arrow lines drawn with different dot/dash patterns. It is instructive to note how the same procedures recur in the various versions of the program. One copy of each such recurring procedure is needed for each processor type for which it is used. The names used on the arrows representing channels are the names of the actual parameters used in the calls of the procedures.

Complete compilation and running instructions are provided with the source text of each example. Users will only be able to try out those versions for which they have the appropriate hardware. The production of alternative versions for different combinations of evaluation boards, etc., is a valuable training exercise.

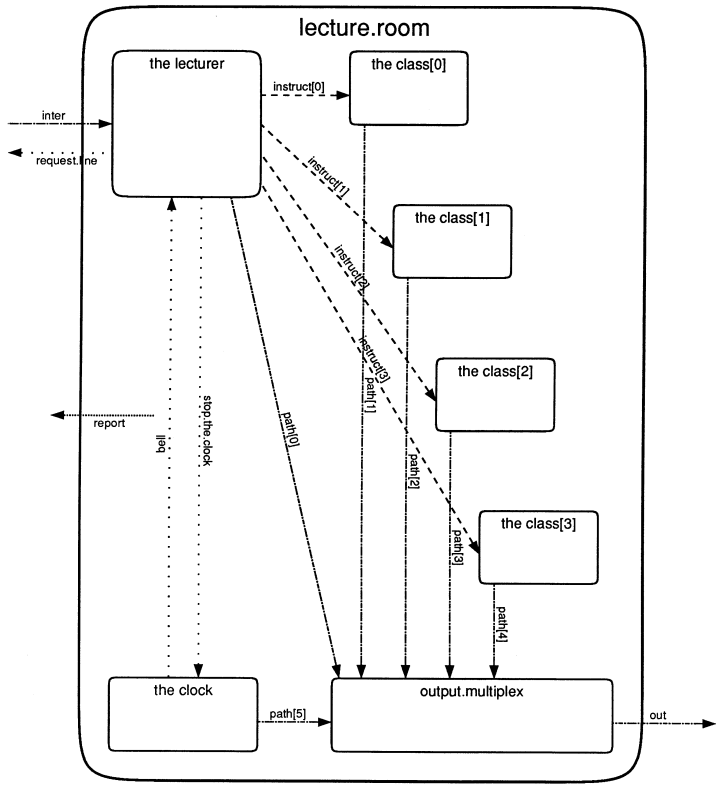
Nine different configurations of this program are included as examples in the TDS. These all use a pair of common processes which are managed as a library (`simlect.tsr`) for ease of code sharing. Other processes common to some of the configurations are also kept in the library. The hardware addresses of the transputer links and a set of protocols used on the channels between the principal processes are also held in libraries (`hardlink.tsr` and `lectprot.tsr`).

The separate compilation units in the library `simlect.tsr` include two copies of each of the central procedures `lecture.room` and `display.as.requested` and the procedure `file.simulator`. These have been written in such a way that they will work on both 16-bit and 32-bit processors without changing the source code.

The two copies of each procedure share access to a common source file. This has been achieved by attaching the file to two distinct folds. This technique should be used with extreme care as accidental deletion of any one of these folds will delete the file attached to the other one also. However the technique is valuable if it is genuinely desired to ensure that the source stays identical in the two folds, as is the case here. Users intending to build versions of this program for networks of different types of transputers should compile each of these procedures for each transputer type to be used. If any of the existing folds are not to be used then the text file should be detached before being deleted. The procedure `filed.fold.reader` is suitable only

for a T4 or T8 running the TDS and so need not be compiled for T2.

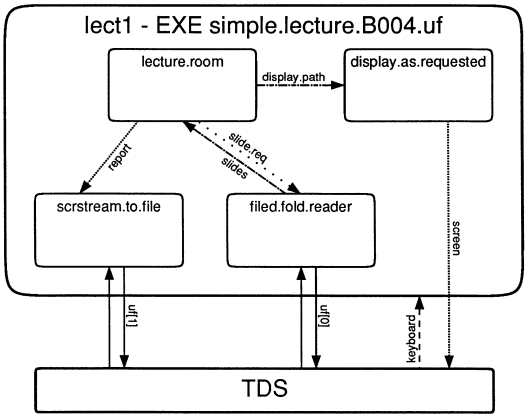
This picture shows the processes inside `lecture.room` which are active while the simulation is under way. The lecturer, clock and class processes all generate commands for the screen handler and these commands are multiplexed by the procedure `output.multiplex` on to a single channel out which goes to the procedure `display.as.requested`. The initiative to terminate the simulation can arise either within the clock process if time runs out, or within the lecturer if his slides run out. These processes are therefore able to terminate each other. The class processes are terminated by messages along the channels `instruct` from the lecturer. The report stream is written by the main body of `lecture.room` before the internal processes start and after they have finished.



5.1 lect1 - An EXE using the user filer interface

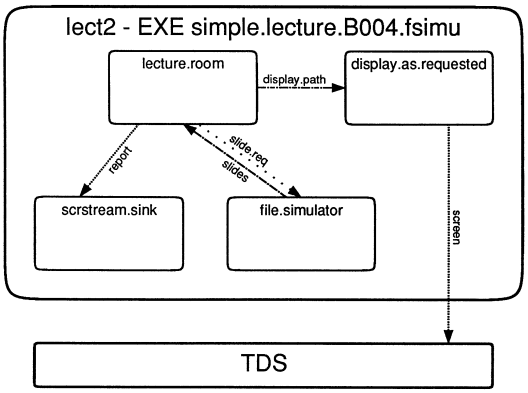
This is the basic version of the simulation. It takes a list of titles in a TDS filed fold and displays the simulation on the screen using TDS screen stream protocol. It also stores a report on the lecture in another filed fold. The procedure `filed.fold.reader` reads the input file from the TDS folded

file store and passes it to the simulation in the procedure `lecture.room` along the channel `slides` using the protocol `LINES`. As in all versions the commands from the simulation to the screen handler are passed along the channel `display.path` using the protocol `DISP.REQ`.



5.2 lect2 - An EXE using a process to simulate the user filer

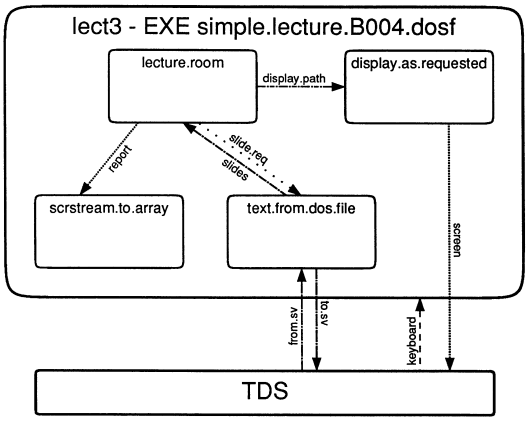
This is a simplified version which does not make any filing system access. The input file is simulated by a call of the procedure `file.simulator` which generates lines of text internally and communicates it using the protocol `LINES`. The report is consumed by a call of the library procedure `scrstream.sink` which generates no output.

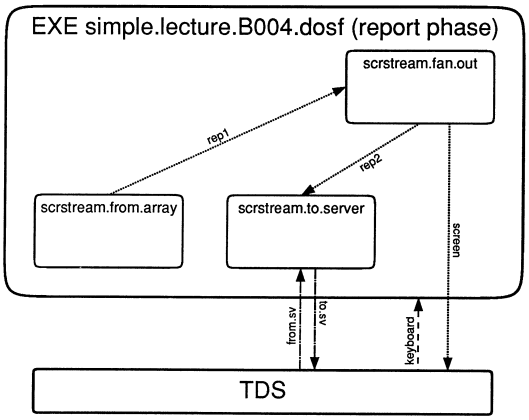


5.3 lect3 - An EXE using the kernel filer interface to DOS files

This version, still running as an EXE in the TDS, uses DOS text files both as input file and as output file. The user is asked for the name of the input file; the output file is called `REPORT.LIS`. The library procedures `keystream.from.server` and `scrstream.to.server` are used for these accesses. As the kernel filer interface only allows one DOS file to be opened at any one time, the output file is buffered in an internal array as it is generated, and then the contents of this array are streamed into the output file when the simulation has finished. The library procedures `scrstream.to.array` and `scrstream.from.array` are used for this purpose. The two pictures show the process structure during the simulation and during the output of the report.

It is recommended that applications requiring access to several host files in parallel are not coded in this way as EXEs but as PROGRAMS loaded by and served by the host file server (see lect5 below).

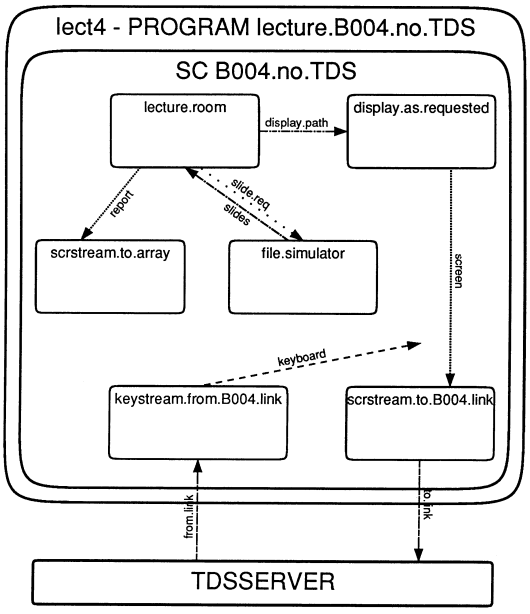




5.4 lect4 - A PROGRAM bootabte directly by the TDS server

A user program may be loaded directly by the TDS server, and must then communicate with the server using the protocols defined in Refman 16.4. This is a simple example of such a program using the interface only for keyboard and screen communications. The filing system accesses could be added, with some difficulty as a complex multipexor is needed, such as the one used in the TDS itself. Such a multiplexor is included in the source of the TDS system loader (see section 6.2 below). In this example the input file is simulated by calling `file.simulator` and the output file is buffered during the simulation and then displayed on the screen.

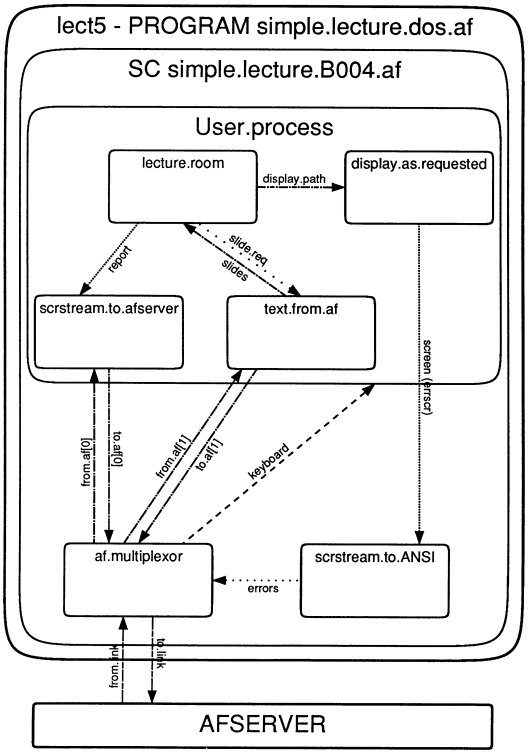
The advantage of this approach over that used in lect3 is that the TDS code and workspace do not consume some of the transputer board's memory. However use of the TDS server in the host for supporting such programs is not usually appropriate in practice, and a simpler server such as the host file server (Refman 16.3), or an even simpler one designed specifically for the application is often to be preferred. See lect5 below.



5.5 lect5 - A PROGRAM bootabte by the host file server

This is the preferred style of program loadable from the host, and with run time access to the terminal and filing system of the host via the host file server. The program asks the user for the name of a DOS file to be used

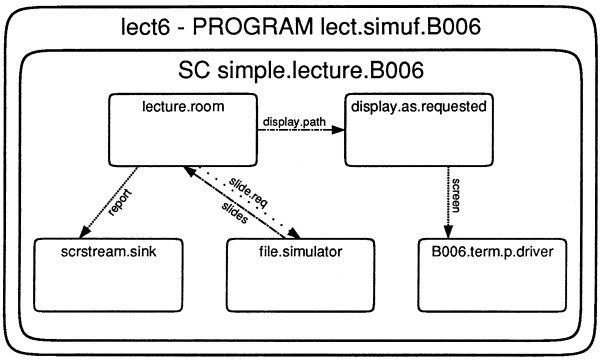
as input file and creates an output file called `AFREPORT.LIS`. Terminal and filing system accesses to the host file server are multiplexed using the library procedure `af.multiplexor`.



5.6 lect6 - A PROGRAM loadable from the TDS into a B006 (T2)

This is an example of a single processor network program designed to run on an IMS B006 evaluation board. This has an IMS T212 transputer and support for a terminal connected by an RS232 port. The adaptation of this program for an IMS B002 (with a T4 transputer) is straightforward and has been supplied as example lect6a.

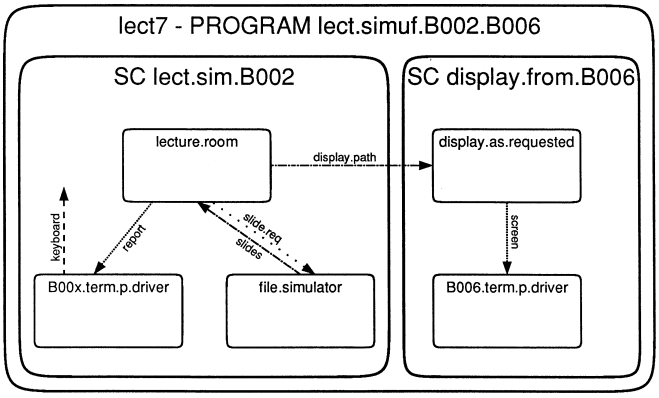
This version uses the `file.simulator` and is derived from lect2 by adding a call of the RS232 UART handler `B006.term.p.driver` to drive the screen in parallel with everything else.



5.7 lect7 - A PROGRAM loadable from the TDS into a B006 and a B002

This is an example of a two processor program, with the two central procedures on separate processors. The example is set up for an IMS B006 and an IMS B002 but could easily be adapted for any other combination of boards, with terminal ports on each. Note that, in order to be independent of transputer type, the protocol `DISP.REQ` on the channel `display.path` between the processors is word length independent; in particular it does not use the type `INT` whose meaning depends on the word length.

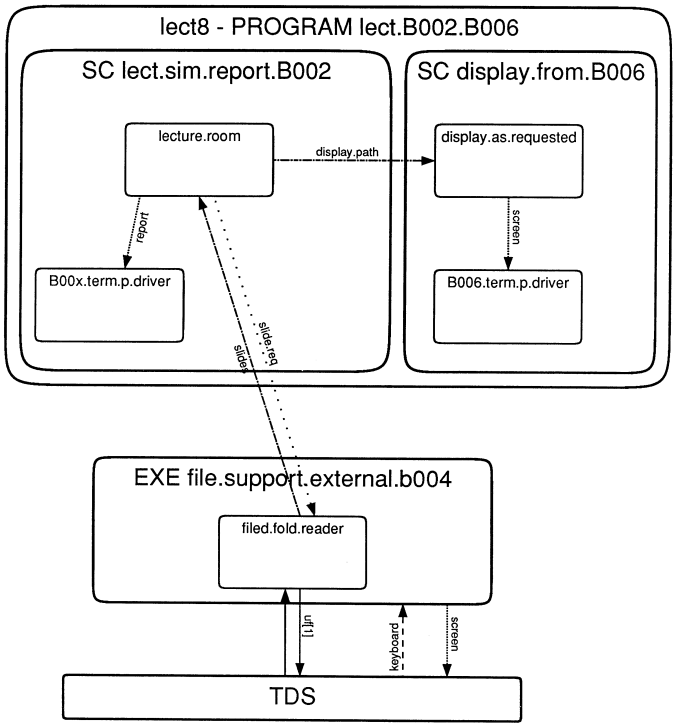
This version of the program uses `file.simulator` for the input file and sends the report to the screen connected to the IMS B002. The output of the simulation goes to the screen connected to the IMS B006. The configuration code defines the connections between the boards and the load path from the host, which is connected in this case to the IMS B006, demonstrating that T4 code can be loaded through a 12 processor.



5.8 lect8 - A PROGRAM loadable into a B006 and a B002 with a support EXE

This version is an extension of lect7 making access to the TDS folded file store for its input file. This is done by an EXE using `filed.fold.reader` which is run in parallel with the rest of the PROGRAM running on the external transputer boards.

In order to run this combination of programs it is necessary to LOAD the PROGRAM into the network and then RUN EXE the EXE with the cursor on a fold containing the filed fold of titles to be read.

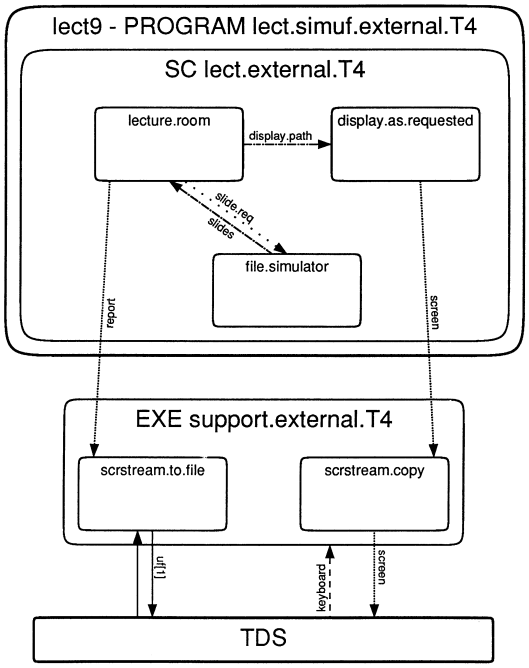


5.9 lect9 - A PROGRAM loadable into any T4 and supported by an EXE

This version runs the simulation on a single external T4 with no access to its own terminal. This could conveniently be a second TRAM module on an IMS B008 motherboard, or one processor of an IMS B003 connected externally. The processor type may be changed to T8 if necessary. The PROGRAM is supported by an EXE on the host transputer to which it is connected by two link cables, one for the terminal screen output and one for the report which is filed in the folded file store. The support EXE is

very straightforward, consisting of calls of `scrstream.copy` for the screen output, and `scrstream.to.file` for the report, in parallel.

As written, the input file is generated by `file.simulator`, but an interesting extension would be to use a third link cable, or a multiplexed link cable, to get read access to the TDS folded file store.



6 Library code and software tools supplied as source

The input/output and mathematical libraries, and many of the TDS tools, are also supplied as source, and are available to be studied, recompiled or modified as the user wishes.

These will be found in the directories:

```
\TDS2\IOLIBS\SRC
\TDS2\MATBLIBS\SRC
\TDS2\TOOLS\SRC
```

6.1 Library sources

The libraries are listed in a table at the beginning of Refman 14. Sources are available of all libraries named in this table except for `reinit` and `blockcrc`.

One reason for supplying these sources is to avoid the need to supply compiled libraries for all the combinations of target processor type and stopping mode. Another reason is to allow users with particularly tight space constraints to build libraries from which all unused code has been removed. The third reason is that the procedures supplied are not an exhaustive set covering all conceivable situations. There are many situations when users will want to write their own library procedures, using the supplied ones as models.

6.2 Tools sources

The sources supplied in this directory include those of most of the tools in the toolkit fold. The only such tools not supplied as source are the memory interface program and the transputer network tester. Tools available as source are:

Selective lister	see Refman 4.9.1
Unlister	see Refman 4.9.1
Link transfer program	see Refman 4.10
EPROM hex program	see Refman 15.4
hex to programmer program	see Refman 15.5

There is a 'Simple worm' program, which is an early version of the transputer network tester, and the following additional programs:

INMOS EPROM monitor	For a load-from-ROM board such as B002
EPROM network loader	For loading networks from ROM
TDS extractor and network loader	TDS loader as an EXE
TDS network memory browser	This is the low level part of the debugger
Worm converter/preamble adder	Adds loading preamble to a CODE SC
TDS loader	Loaded by the TDS server to load and run the TDS (see Refman 16.4.4)
Analyse worms	For the network memory browser
Loader worm	Simple boot from link loader
Disassembles	To look at compiled code

Descriptions of all these tools are provided with the source. Some of the tools are discussed in greater detail in INMOS Technical Notes (see the Bibliography).

Many of the programs in both these groups are substantial programs which may also be treated as coding examples. Users are recommended to browse around looking for code which interests them.

References

[1] A tutorial introduction to occam programming, Dick Pountain and David May, BSP Professional Books 1987.

[2] Transputer development system, INMOS Limited, Prentice Hall 1988.

[3] Occam program development using the IMS D700D transputer development system, M D Poole,
INMOS Technical Note 16. 72-TCH-016

[4] Occam input and output procedures for the TDS, M D Poole,
INMOS Technical Note 28. 72-TCH-028

[5] Analysing transputer networks, J M Wilson,
INMOS Technical Note 33. 72-TCH-033

[6] Loading transputer networks, J M Wilson,
INMOS Technical Note 34. 72-TCH-034

[7] Implementing data structures and recursion in occam, S Redfern,
INMOS Technical Note 38. 72-TCH-038

[8] Module motherboard architecture, T Watson,
INMOS Technical Note 49. 72-TCH-049