AN INTRODUCTION TO TRANSPUTERS AND OCCAM

R.W. DOBINSON, D.R.N. JEFFERY AND I.M. WILLERS
ECP DIVISION, CERN, 1211 GENEVA 23, SWITZERLAND

Aims And Non Aims

The aim of these lectures is to teach students how to write programs in the Occan language developed by INMOS to support parallel processing on Transputers.

The approach taken will be to first introduce sequential features of Occam, comparing periodically with Fortran. We then go on to describe the parallel processing constructs contained in the language, explaining some of the things you definitely cannot do in Fortran but can do with Occam.

These lectures combined with the practical exercises should allow participants to progress from a simple mono-processor program to quite complex multi-Transputer programs over the period of the school.

What is a Transputer?

A Transputer Fig (1) is a family of micro-computers which have the following components packaged on a single chip:

* An integer processor.

* 2 or 4 kbytes of fast static RAM.

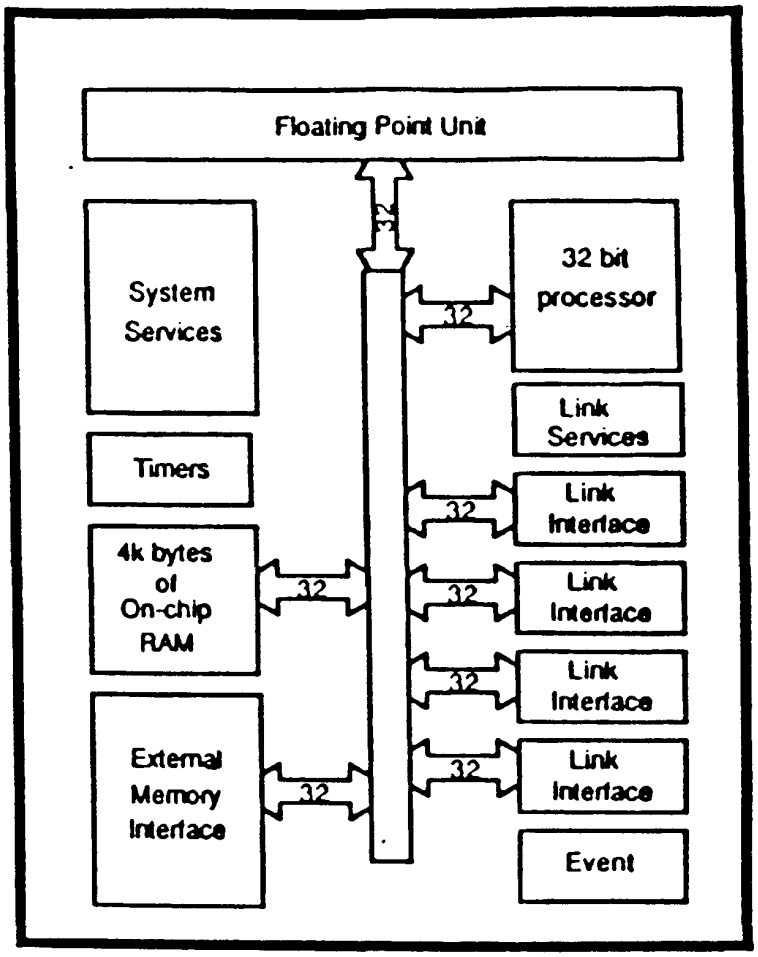
* Four serial links which allow easy connection to other Transputers, Fig (2).

* An interface to external memory.

* A real time kernel for process scheduling.

* A timer.

* An on-chip floating point processor (on some Transputers).


Floating Point Unit
32
System Services
32 bit processor
32
Link Services
Timers
Link Interface
32
4k bytes of On-chip RAM
32
Link Interface
32
Link Interface
External Memory Interface
32
Link Interface
32
Event

FIG(1)

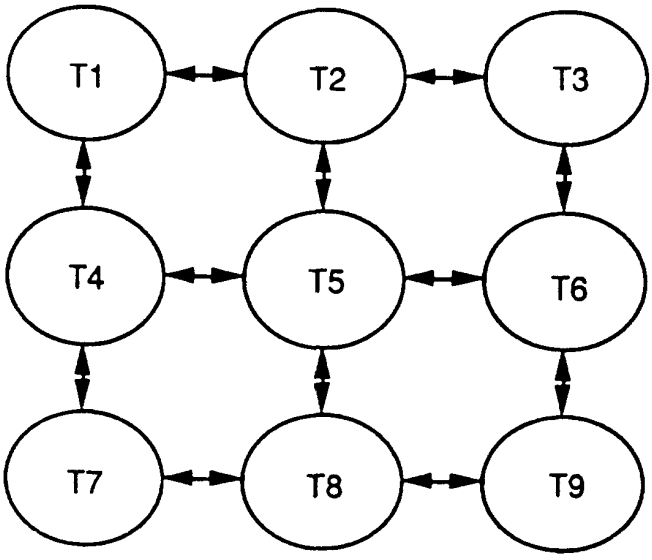

T1
T2
T3
T4
T5
T6
T7
T8
T9

FIG (2)

Performance

The Transputer that you will use is the 20 MHz T800 chip it has a computational power of 1.5 to 3 Vax 11/780 equivalents. It has a low context switching overhead, a few microseconds, and process to process link transfer rates of 1.7 Mbytes/s between two Transputers. Only a few microseconds setup time is required for a link transfer.

Some Simple Ideas About Processes

An application can be made up of a number of communicating processes. For the moment we loosely use process to mean a part of the overall program. There are various different ways of mapping processes onto one or more processors, see figs 3,4,5. All processes could reside in a single processor or the program may be partitioned, most likely for performance reasons, over several processors. Three processes are shown in fig.3 connected via channels . Fig 4 shows all the three processes running on one processor while fig 5 shows three processes running on three separate processors. We shall see that Transputers, their links and Occam make it relatively easy to spread an application over multiple processors. You might like to consider how you would do this with orthodox microprocessor chips and languages!

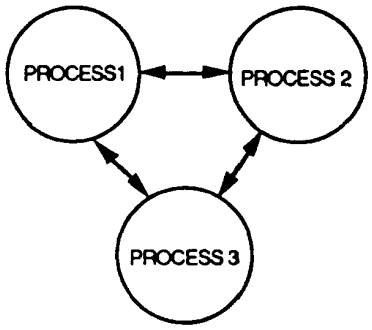
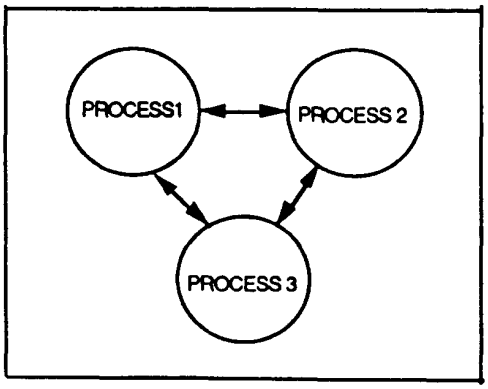


FIG (3)



PROCESSOR A
FIG (4)

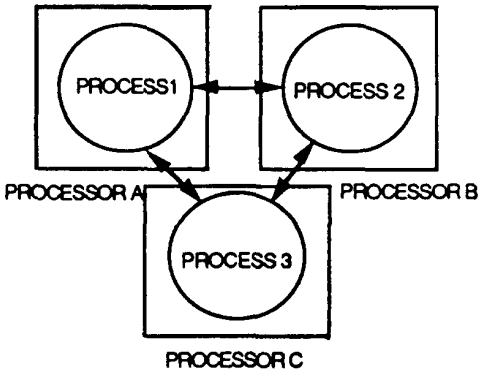


FIG (5)

Names Used in Occam

Names of objects in Occam, like variables, procedures, etc. must begin with an alphabetic character. They are a sequence of alpha-numeric characters and dots. There is no length restriction. Occam is sensitive to the case of characters. Fred is different from FRED.

There are a number of reserved keywords which always use capital letters,

```
e.g.  BOOL, ANY, ALT, PAR, SKIP, TIMER
```

We recommend you use lower case names in your programs. Mixing upper and lower case can be confusing.

* Some examples of legal and illegal names.

Legal:

```
dire.straights
charlie
input.buffer
flag1
```

Legal but discouraged in this course:

```
BIGONE
OutChar
WRITE.MT.RECORD
```

Illegal and flagged as such by the Occam compiler:

```
data_buffer             -- illegal character _
1stnumber               -- does not start with a letter
last word               -- contains a space
pointer$                -- illegal character $
FUNCTION                -- reserved  keyword
```

Variables and Declarations

Unlike Fortran, Occam requires all variables to be declared before use. Occam recognises the following data types.

BOOL	Boolean values true or false.
BYTE	Integer value 0 to 255. Used for characters.
INT	Generic signed integer (32 bits for the T800 we shall be using, 16 bits for 16 bit Transputers such as T222)
INT16	16 bit signed integer.
INT32	32 bit signed integer.
INT64	64 bit signed integer.
REAL32	32 bit IEEE floating point number.
REAL64	64 bit IEEE floating point number.

Examples of declaring variables.

```
INT fred :
BYTE  input.character :
REAL32  floating1, floating2 :
INT16 junk, morejunk,
          still.more.junk:
```

A declaration is terminated by a colon. Variable names in a multiple declaration are separated by commas. A line break can occur after a comma.

Constants

Constants can be defined in the following way using the keywords VAL and IS:

```
VAL leap.year IS 366 (INT) :
VAL character IS  'z'  (BYTE):
VAL pi IS 3.141592 (REAL32) :
VAL calibration.cons IS 6.1E-5(REAL64):
VAL end IS FALSE :
VAL start IS TRUE :
VAL string IS "Hello world":
```

Note TRUE and FALSE are reserved keywords

Arrays in Occam

An array in Occam is a group of objects of the same type joined into a single object with a name. An object in an array can be individually referred to by a number stating its position in the array. Example of declaring an array:

```
[5] INT x:
```

A one dimensional integer array which has its 5 components labelled 0,1,2,3,4, this is unlike Fortran which would label the array 1,2,3,4,5.

Another example

```
[4] [5] INT x:
```

This declares a two dimensional integer array. An array with four components each of type [5] INT. Components of array variables behave just like ordinary variables of the same type and can be used anywhere an ordinary variable could be used.

Assignment

An assignment in Occam is of the form

```
x := y+2
```

i.e. variable := expression

cf Fortran

```
X = Y+2
```

Note the use of := instead of just =

The value of the expression must have the same data type as the variable to which it is assigned i.e. the Occam compiler would say " real.number := integer.number " is not valid (later we will see how to convert).

An assignment is an example of what is called in Occam a primitive process. All Occam programs are built up as a combination of primitive processes. We will meet other primitive processes later.

Expressions

Expressions are made up of variables and literals (eg. 99, 42.3, 386.0E-2) in combination with operators. An expression is evaluated and produces a result. The result has a value and a data type. The simplest expressions are literals and variables. More complex expressions can be constructed using operators and parentheses. There is no operator precedence as in Fortran. All operators have the same priority. Hence parentheses MUST be used to define the hierarchy of a complex expression.

Note that the data types of operands in the expressions must be the same. Bit shifting is an exception.

We introduce some of the most useful operands and leave out others for the moment. Arithmetic operators:

```
x+y          --  add y to x
x-y          --  subtract y from x
x*y          --  multiply x by y
x/y          --  quotient when x is divided by y
x REM y      --  remainder when x is divided by y
```

This is all very like Fortran.

Here are some more examples of arithmetic expressions:

```
(2+3)*(4+5)       --answer 45
(2+(3*4))+5       --answer 19
2+(3*(4+5))       --answer 29
```

The following expression is illegal and would be rejected at compilation time:

```
2+3*4+5            --illegal !
```

The boolean operators AND, OR and NOT combine boolean operands and produce a boolean result. Examples of expressions using these operators:

```
NOT FALSE
x AND y
FALSE OR x
(a AND b) OR (c AND d)
```

Relational operators perform a comparison of their operands and produce a boolean result. Relational operators are:

```
=    equal
<>   not equal
<    less than
>    greater than
<=   less than or equal
>=   greater than or equal
```

Examples of expressions using relational operators.

```
x=y
(x=y) AND (a<>b)
(char>='a') OR (char<='z')
```

The SEQ construction

The SEQ construction is the simplest we shall meet. It says "do the following things in sequence". Example using Occam,

```
INT i, j, k :
VAL two IS 2(INT):
VAL three IS 3(INT):
SEQ
  i:= two + 2
  j:= three + i
  k:= (i+j) + 11
```

The Fortran equivalent is

```
PARAMETER ( ITWO=2, ITHREE=3)
I = ITWO+2
J = ITHREE+I
K = I+J+11
```

Unlike Fortran we have to specify explicitly that the three assignments must be done one after the other.

Why? Because as we shall see later they could be done in parallel!

The three assignments are indented by two spaces to indicate the extent of the construction. Other languages use {...} or begin ... end for this purpose, but Occam uses indentation. The end of any level of indentation marks the end of the construction. The construction is defined only within the range of its indentation, as is the scope of its local variables. The SEQ has been used to combine three primitive processes, three assignments, into a larger process.

Take care, a SEQ is compulsory in Occam whenever several Occam statements are to be executed in sequence.

The use of comments - a small digression.

```
INT i,j,k:
VAL two IS 2(INT):
VAL three IS 3(INT):
SEQ  -- a comment
     -- comments cannot be indented
     -- less than the following
     -- statement
  i:=two + 2
  j:=three + i
  k:=(i+j) + 11 -- note parentheses
```

The Replicated SEQ Construction

A process can be replicated to behave like a conventional counted loop.

For example;

```
INT a:
SEQ
  a:=0
  SEQ i= 0 FOR 100
    a:=a+i
```

The replicator has the form

base FOR count

Compare this with a Fortran DO loop.

```
      J=0
      DO 99 I=1, 100, 2
      J=J+I
99 CONTINUE
```

Note, the Fortran DO loop can specify a step, a feature not included in the replicated SEQ.

Note, you can jump out of a Fortran DO loop. You cannot jump out of a replicated SEQ.

If we were to introduce another assignment in our fragment of Occam above we would need to put in another SEQ.

i.e.

```
INT a, b
SEQ
  a:=0
  b:=10000
  SEQ i=0 FOR 100
    SEQ
      a:=a+i
      b:=b-i
```

The extra SEQ is required whenever the block of statements (component processes) to be replicated is longer than one.

An example of how to zero an array:

```
[100] INT array :
SEQ i=0 FOR 100
  array[i]:=0
```

* The equivalent Fortran would be.

```
      DIMENSION IRRAY(100)
      DO 1 J=1, 100
      IRRAY(J)= 0
1     CONTINUE
```

SKIP and STOP

SKIP and STOP are primitive processes. Assignment is another primitive process that we met earlier. We introduce SKIP and STOP now as we will need to use them when discussing the IF and CASE constructions. SKIP does nothing, its rather like a "no op" or Fortran CONTINUE. SKIP starts , does nothing and terminates.

STOP stops the flow of execution, it is a process that never terminates. A STOP in an Occam program will result in an error flag being set on the B008 board used in the lab exercises. When this occurs TDS will be terminated. Normally this should be avoided!

The IF Construction

All programming languages need to provide a way for programs to do different things according to a condition i.e. a test. One form of conditional choice is the IF construction.

An IF construction specifies a number of options (different processes) each of which has a test (a boolean expression) in front of it. Each test is evaluated in sequence and, for the first found to be TRUE, the associated option is executed. If none of the tests is true then a STOP is executed.

Example;

```
IF
  i=1 -- if i=1 is TRUE then
    j:= 1              -- set j equal to 1
  i=2 -- if i=2 is TRUE then
    j:= 10             -- set j equal to 10
  TRUE                 -- if neither the above
    SKIP               -- then execute a SKIP
```

Note that each test is indented two spaces wrt the IF, and each option is indented two spaces wrt its associated test.

The final test uses the boolean constant TRUE (which is always true) and acts as a catch all in case none of the above is true. SKIP is just a do nothing action. If we did not have this catch all then a STOP would occur if i is not equal to 1 or 2. We don't want this!

The "equivalent Fortran" would be:

```
IF(I.EQ.1) J=1
IF(I.EQ.2) J=10
```

IFs can be nested to make more complex choices , for example:

```
IF
  x=1
    IF
      y=1
        z:=100
      y=2
        z:=200
      y=3
        z:=999
      TRUE
        SKIP
  TRUE
    SKIP
```

The Replicated IF Construction

A conditional, like a SEQ, can be replicated. Example:

```
IF i=1 FOR 3
  x=i
    y:= i+10
```

This is just equivalent to:

```
IF
  x=1
    y:=1+10
  x=2
    y:=2+10
  x=3
    y:=3+10
```

We need a catch all in case none of the tests in the replicated IF is satisfied. This can be done by nesting the replicated IF within an outer IF:

```
IF
   IF i=1 FOR 3
     x=i
       y:= i+10
   TRUE
     SKIP
```

The CASE Construction

The CASE selection combines a number of options, one of which is selected by matching the value of a selector with the value of a constant expression (a case expression) associated with the option.

Example;

```
CASE direction
  up
    x:=x+1
  down
    x:=x-1
  ELSE
    SKIP
```

Note if no match is found a STOP would be performed. Occam provides an ELSE as a catch all.

It is possible to put more than one CASE expression on a single line, thus:

```
CASE letter
  'a','e','i','o','u'
    vowel := TRUE
  ELSE
    vowel := FALSE
```

The WHILE Construction

We have seen how a replicated SEQ can be used to execute a loop a specified number of times. The WHILE construct allows looping as long as a given condition holds.

Example;

```
INT x,y,z,sum :
SEQ
  x:=0
  y:=0
  z:=0
  WHILE x <= 100
    SEQ
      y:=y+x
      z:=z+(x*100)
      x:=x+1
  sum:=x+(y+z)
```

Input/Output

The Fortran language allows users to perform I/O by means of READ and WRITE statements. These statements allow programs to access peripherals, such as disks, magnetic tapes, keyboards, screens, etc.

We will introduce the idea of Occam channels and explain how I/O is performed from an Occam program to peripherals, attached to the APOLLO, via channels. Later we will see how Occam channels are used for all inter-process communications.

i.e. channels are used for

a. Communication between different Occam processes in the same Transputer (between a user program and TDS or between two parts of a user program).

b. Communication between different Occam processes in different Transputers.

First Ideas about Channels

Values can be communicated over channels using the primitive processes input and output. The input process

```
chan1 ? fred
```

asks for a value from a channel named chan1. When the value arrives it is put in variable fred. Think of the ? as saying "where's my value".

The output process

```
chan3 ! jane
```

takes a value jane and sends it down a channel called chan3. Think of the ! as saying "here you are here's your value".

The following code inputs a value from one channel and outputs it on another:

```
SEQ
    chan1 ? fred
    chan3 ! fred
```

Where do the channels come from and go to?

One example is a user program running under TDS. Two channels called "`screen`" and "`keyboard`" are made available to the user program by TDS. Values sent to the channel "`screen`" end up on the APOLLO screen. Values read from the channel "`keyboard`" contain input typed on the APOLLO keyboard. Communication between the user program and TDS uses a special set of protocols. Normally users make use of procedures from the TDS libraries to perform I/O. These procedures hide the underlying protocol from the user, see Fig (6).

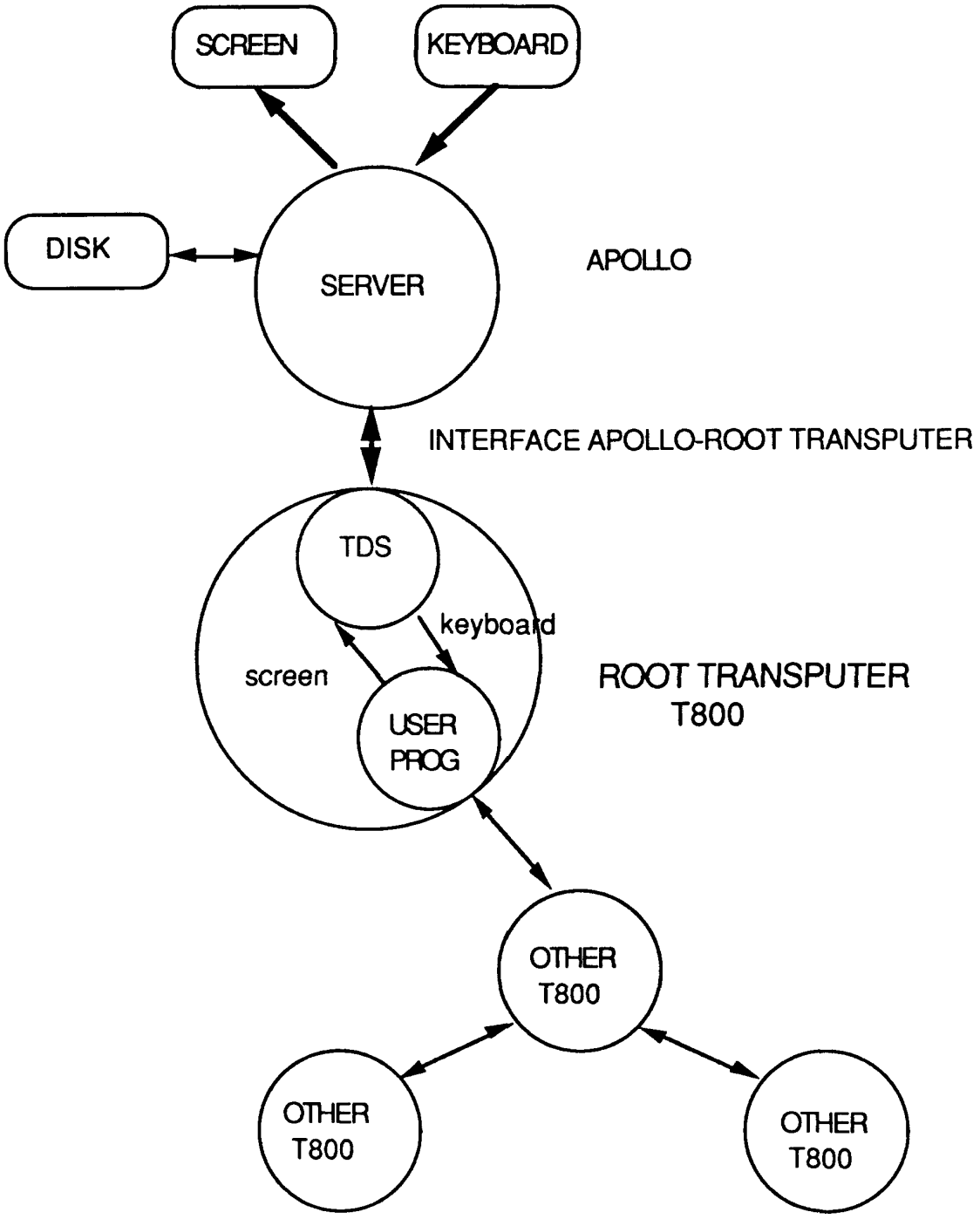

SCREEN
KEYBOARD
DISK
SERVER
APOLLO
INTERFACE APOLLO-ROOT TRANSPUTER
TDS
keyboard
screen
USER
PROG
ROOT TRANSPUTER
T800
OTHER
T800
OTHER
T800
OTHER
T800

FIG (6)

USERIO Library

For example

Operations via channel "screen"

write.int	write integer
write.char	write character
goto.xy	position cursor
write.full.string	write string
write.real32	floating point write
newline	give new line
clear.eos	clear screen
up	move cursor up one line

Operations via channel "keyboard"

read.echo.char	read/echo a character
read.text.line	read a line of text
read.hex.int	read hex integer
read.echo.int	read and echo integer
read.echo.real32	floating point read/echo

Compare this with some Fortran I/O.

```
    WRITE(5,100) ISTAT
100 FORMAT(/,'STATUS=",I3)
    READ(6,120) INT, FP
120 FORMAT(I10,F8.4)
```

Example of I/O Using Occam and TDS to write hello on the screen

```
#USE userio              -- library containing I/O
INT any:
SEQ
  write.full.string(screen,"hello!*c*n")
  write.full.string(screen,"press any key")
  read.char(keyboard,any)
```

Notes

#USE userio specifies use of the library containing the I/O procedures.
*c moves the cursor to the first character of the current line.
*n moves the cursor to the next line.
A character is read from the keyboard to allow "hello " to be read before returning to TDS.

Some Concluding Comments about I/O

TDS offers a lot of possible options for screen and keyboard I/O, it also provides file access which we have not described. It is easy to get confused, so we recommend you learn initially just enough to get by. TDS I/O is specific to TDS! This means you will not be able to use the I/O calls outside the TDS environment. This is a nuisance and a good reason for not spending too much time learning a whole lot of detail. TDS provides no graphics calls.

Note: In general you should choose to perform all I/O from the same process in the root Transputer. TDS offers I/O only on the root Transputer. You yourself have to route any I/O from a network to the root. An RPC (Remote Procedure Call) is implemented between the APOLLO and the root Transputer.

Some Basic Ideas About Parallel Processing

Most computers, basically, operate in the following way:

A processor is connected to a memory system which can store numbers. Some of the numbers are the data to be processed and other numbers are instructions to the processor to tell it what to do. The instructions are passed to the processor one after the other.

Thus the execution of most computer programs is sequential, one instruction after the other. Most computer languages assume this form of computer hardware and reflect its sequential nature. A program has a single thread of control running through it. Computers are very often employed to model, service and control real world phenomena. The real world is highly parallel and it is rich in communications. Lets look at examples of computing problems with intrinsic parallelism in them:

When simulating an electronic circuit, made up of an array of integrated circuits or gates or transistors, the inputs and outputs of the devices are all changing "simultaneously". The output of a gate will depend on values communicated to its inputs.

Every parameter in an industrial plant may need to be monitored "at the same time, all the time".

An airline reservation system needs to service many enquiries "concurrently" and change its database if you book a seat.

Very often computers of the traditional type described above mimic concurrent events sequentially (using timesharing). However, it is often worth looking into the use of a number of closely cooperating processors working together, "in parallel", to solve a problem. This has been done in fact, though often for only a small number of processors, in an adhoc way, or at great cost and effort. Or perhaps all three!

Until recently, neither the processor hardware nor the programming languages have existed to easily synthesize systems comprised of large arrays of processors. The Transputer and Occam are a first step in the direction of easing the difficulty. However, it would be naive to pretend that applying parallel processing techniques to real situations is always easy.

A Knitting Analogy

The sequential Occam representation of someone knitting a sweater could be as follows;

```
SEQ
  ... knit body
  ... knit left sleeve
  ... knit right sleeve
  ... knit neck
  ... sew sweater.
```

We have used the SEQ construct to emphasise the sequential nature of the knitting and have used folds to describe the different operations required. This sequential approach is ok for some purposes, but supposing time starts to matter. Supposing a new sweater is needed quickly! The overall task of producing a sweater could be given to a team of knitters working in parallel. Suppose there is one knitter for each of four pieces plus someone to sew the final garment. The Occam for this can be written as follows:

```
SEQ
  PAR
    ... knit body
    ... knit left sleeve
    ... knit right sleeve
    ... knit neck

  ... sew sweater
```

Note we have now made use of a new construction, the PAR, to combine the four knitting processes which are going on in parallel.

There is now a need for synchronization and communication. The sewing cannot start until all knitting is finished and the four separate pieces given to the sewer. Unless all knitting activities are "well balanced" then production is inefficient with everyone waiting for the slowest person to finish. Synchronization and communication are important in both parallel knitting and parallel processing.

Lets put these ideas into the Occam knitting program more explicitly:

```
PAR
  SEQ
    ... knit body
    bodychannel ! body
  SEQ
    ... knit right sleeve
    rightchannel ! rightsleeve
  SEQ
    ... knit left sleeve
    leftchannel ! leftsleeve
  SEQ
    ... knit neck
    neckchannel ! neck
  SEQ
    PAR
      bodychannel  ? body
      rightchannel ? rightsleeve
      leftchannel  ? leftsleeve
      neckchannel  ? neck
    ... sew sweater
```

Occam Processes and Channels

Writing a program in Occam is carried out by combining a number of simple processes into a larger process. A process starts, performs a series of actions, and terminates. More than one process can be running at a time and messages can be passed between these concurrent processes. Channels are used to pass messages. Channels can link processes running on the same processor or on different processors. The use of channels not only takes care of communication but also takes care of synchronization.

If a process asks for input, and no value is ready, then it will wait until one is supplied. Similarly, an output will not take place between sender and receiver until both are ready. Fig (7) shows how synchronisation on channels takes place.

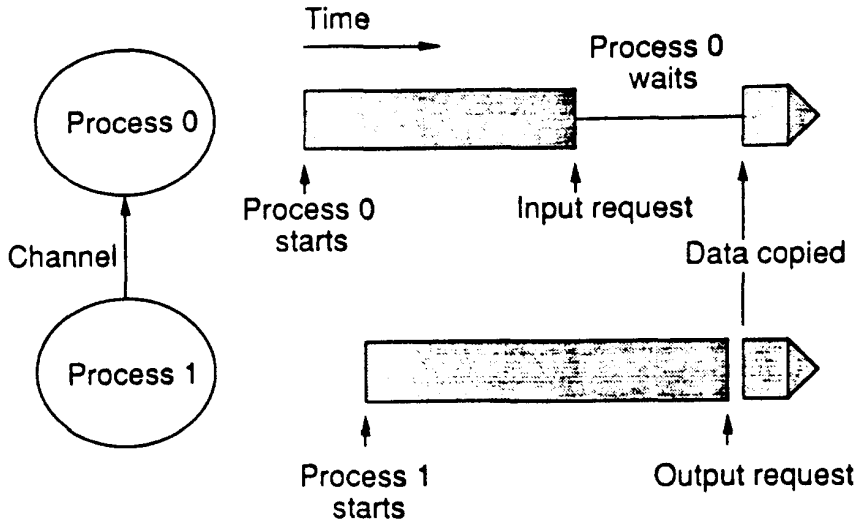


FIG (7)

More Formal Aspects of PAR and Channels

Consider the following simple piece of Occam:

```
INT a, b:
SEQ
  a:=5
  b:=a+100
```

We have constructed a compound process made up of two primitive processes, two assignments, which are run in sequence one after the other. However we could also combine the two assignments with a parallel PAR construct which would specify "do the following processes both at the same time".

A completely trivial example could look like this:

```
INT a, b, c :
CHAN OF INT comm:
PAR
  SEQ
    a:=5
    comm ! a    -- here's my value of a
  SEQ
    comm ? c    -- give me your value
    b:=c+100    -- form sum as before
```

`CHAN OF INT comm:` declares a channel called comm and specifies that an integer can be sent down it. Declaring a channel is like declaring a variable, a declaration must precede subsequent use. When you define what is sent over a channel you are specifying the channel protocol. In general you use the PROTOCOL keyword to do this.

Examples of channel declarations and protocols;

```
CHAN OF [20] INT fred:
PROTOCOL values IS REAL32; INT; BOOL:
CHAN OF values bert:
```

`CHAN OF ANY annie:` allows anything to be sent down the channel called `annie`.

A channel is a one way communication path between two processes running in parallel. There is one and only one sender and one and only one receiver. If a two way communication is required a second channel must be added. You can think of input on a channel as a remote assignment. Communication between the component processes of a PAR should only be done using channels. The following is illegal:

```
INT a, b :
PAR
  SEQ
    a:=5
  SEQ
    b:=a+100 -- this is illegal the compiler will object
```

If Occam allowed this, what value of"a" would be added to 100? We don't know anything about the time ordering of the two components of the PAR.

Shared variables between parallel processes are not encouraged by Occam! Although read only access is permitted. Keeping variables local to component processes and using channels to communicate is the right approach. Thus :

```
CHAN OF INT comm:
PAR
  INT a :
  SEQ
     a:=5
     comm ! a    -- here's my value of a
  INT b, c :
  SEQ
    comm ? c     -- give me your value
    b:=c+100     -- form sum
```

Note that the process formed by the PAR construct terminates when all component processes have terminated.

Deadlock

Be careful to avoid deadlock between two processes running in parallel. e.g.

```
CHAN OF INT comm1, comm2 :
PAR
  INT x :
  SEQ
    comm2 ? x
    comm1 ! 2
  INT y :
  SEQ
    comm1 ? y
    comm2 ! 3
```

Both processes are waiting for input that never comes. The PAR will never finish.

The ALT Construction

We have seen how to make choices according to the values of variables using the IF construct.

```
INT x:
IF
  x=1
    ... process 1
  x=2
    ... process 2
  TRUE
    ... process 3
```

An ALT construction makes choices on the basis of the state of input channels. For example;

```
CHAN OF INT chan1, chan2, chan3 :
INT x :
ALT
  chan1 ? x
    ... process 1
  chan2 ? x
    ... process 2
  chan3 ? x
    ... process 3
```

The ALT watches all the input processes and executes the process associated with the first input to become ready. It is a first past the post race between a group of input channels with only the winner's process being executed. Note, channel ready means the sender is ready to communicate, it does not mean the input has completed. The ALT terminates when one of the alternative processes has been run.

An ALT may include a test in addition to an input, just like IF tests. The associated process is then only chosen if its input is ready and the test is true.

```
CHAN OF INT chan1, chan2, chan3 :
INT x :
ALT
  (y<0) & chan1 ? x
    ... process 1
  (y=2) & chan2 ? x
    ... process 2
  (y>0) & chan3 ? x
    ... process 3
```

The conditions that have to be satisfied before any of the alternative processes are run are called guards, i.e.

```
(y>0) & chan3 ? x
```

includes a guard, if y>0 and chan3 is ready then the value from chan3 is transferred into x and process 3 is run.

How to build a Multiplexor

```
WHILE TRUE
  ALT
    left ? packet
      stream ! packet
    right ? packet
      stream ! packet
```

The code merges the inputs coming from the left and right channels and outputs on a single channel called stream. If left is ready and right is not then input is taken from left and passed to the output channel stream. If right is ready and left is not then input is taken from right and passed out to stream. If both are ready at the same time one of the two alternatives is chosen. Note, an Occam ALT does not specify which alternative is chosen (see PRI ALT later). If neither channel is ready the process waits until an input becomes available. See Fig (8).

Multiplexors (mux) are important when using TDS and Occam. For example, if you wanted to have two processes running in parallel both outputting to the screen you would have to put in a multiplexor (Fig 9).

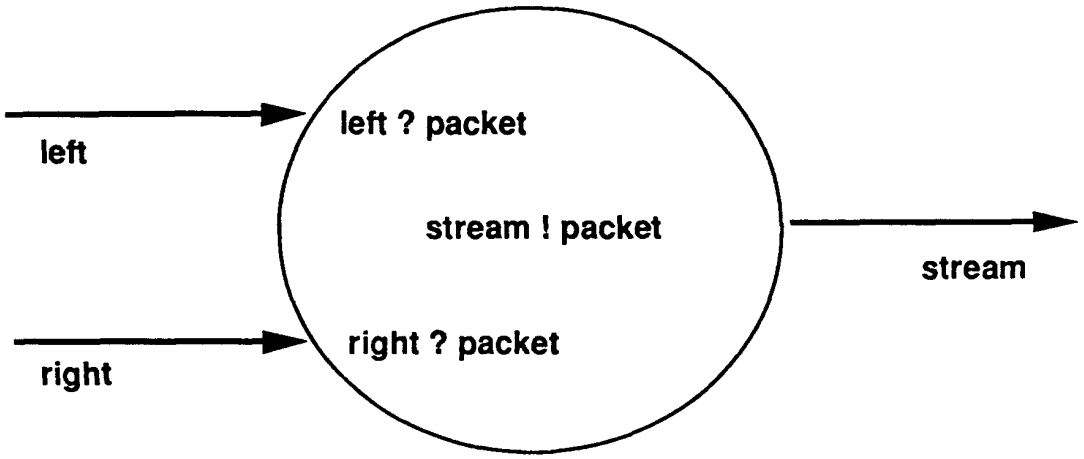

left
right
left ? packet
stream ! packet
right ? packet
stream

FIG (8)

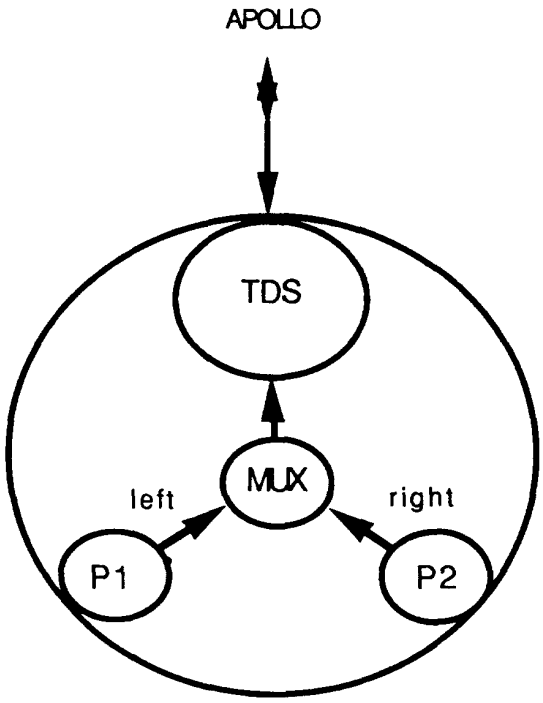

APOLLO
TDS
MUX
left
right
P1
P2

FIG (9)

Abbreviations

The term abbreviation can be confusing as it means more than one thing in Occam! Earlier we saw the notation for the declaration of constants:

```
VAL k IS 1024 (INT32):
```

this is one form of abbreviation.

The Occam distinguishes two types of abbreviation, abbreviations of expressions and abbreviations of elements.

Abbreviations of expressions is used to specify a name for a constant:

```
VAL number.of.beers.drunk IS 99 (INT32):
```

This declares a constant of value 99.

```
INT32 y:
SEQ
  y:=32
  VAL x IS y:
  PAR
    ...process
```

This declares an integer constant x which has the current value of y.

```
REAL32 x,z:
SEQ
  x:=10.0 (REAL32)
  z:=99.0 (REAL32)
  VAL REAL32 y IS (x*x) + z:
```

This declares a real32 constant y which contains the current value of the expression (x*x) + z.

No assignment can be made to this type of abbreviation:

```
INT32 y,z:
VAL INT32 x IS y:
SEQ
  z:=x+10 (INT32)                    -- this is allowed

INT32 y:
VAL INT32 x IS y:
SEQ
  x:=100 (INT32)                     -- this is not allowed
```

i.e x is a read only quantity.

After the abbreviation of y we cannot assign to y :

```
INT32 y,z:
VAL INT32 x IS y:
SEQ
  z:=x+10  (INT32)  -- this is allowed
  y:=x+15   (INT32)  -- this is not allowed
```

Abbreviations of elements are used to abbreviate elements of an array or to rename a variable:

```
INT32 y:
SEQ
  x IS y:
```

x is a variable which is used instead of y

```
[25] INT32 one.dimensional.array:
SEQ
  INT32 x IS one.dimensional.array[8]:
```

This declares an integer x which represents the 8th element of the array.

```
[100] INT32 z:
SEQ
  [] INT32 x IS [z FROM 10 FOR 13]:
```

This declares an array [13]x which is equivalent to the array z[10] to z[22]. This is a read/write abbreviation, in that you can assign values to the abbreviation.

We cannot use name of the variable which has been abbreviated after the abbreviation:

```
INT32 x:
SEQ
  y IS x:
  x:=0            -- This is not allowed
```

A part or all of a multidimensional array can also be abbreviated, fig(10). Suppose we have:

```
[9][6] INT two.d.array:
SEQ
  [] INT one.d.array IS two.d.array[3]:
```

This results in the 3rd i'th row being abbreviated.

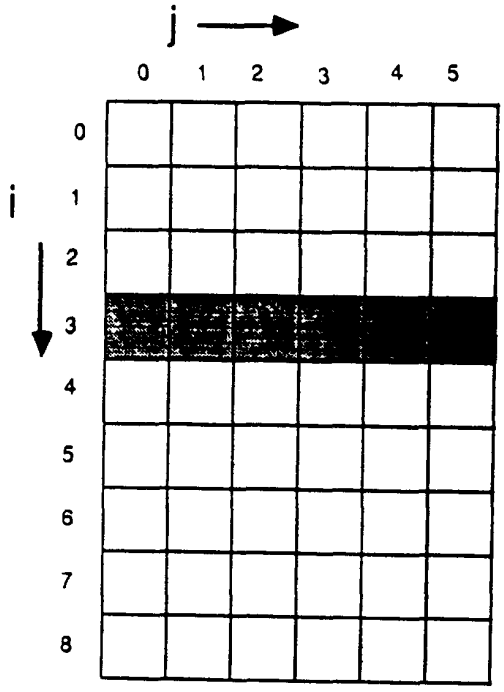


FIG (10)

Also if we have

```
[10][20][30] INT y:
```

Then

```
[][] INT x IS y[5]:
```

will give a two dimensional array [20][30]x.

If the original array can be thought of as

```
[i] [j] [k] three.d.array
```

then we will have

```
[j] [k] two.d.array
```

as the abbreviation; i.e. the i'th row has been abbreviated.

What use are abbreviations?

We can make a program more readable; instead of

```
m:=x[10][25][234][453]
n:=x[10][25][234][453]
p:=x[10][25][234][453]
```

one could have

```
INT a IS x[10][25][234][453]:
SEQ
  m:=a
  n:=a
  p:=a
```

Program execution time can be decreased by abbreviating array elements. In the above unabbreviated example the Transputer would have to calculate the address of x[10][25][234][453] three times. In the abbreviated example the Transputer would only have to calculate the address once for the abbreviation a.

Note that you can also have:

```
[20][20] INT two.d.array:
SEQ
  VAL [] INT x IS two.d.array[10]:
```

This gives a new one dimensional array, however you cannot assign to the elements of the array, e.g.

```
x[3]:=0          -- is not allowed.
```

The compiler will say 'Cannot write to x'.
If you abbreviate an array element, you cannot use or change the other array elements:

```
[10] INT x:
INT y IS x[3]:
SEQ
  y:= 0                  -- allowed
  x[4]:=0                -- not allowed
```

Procedures

A procedure in Occam is similar to a FORTRAN subroutine:

```
SUBROUTINE sub1(i,j,x)
  i=j
  x=x+2.5
RETURN
END
```

The occam equivalent would be:

```
PROC sub1(INT i,j, REAL32 x)
  SEQ
    i:=j
    x:=x+2.5(REAL32)
:
```

Note that the statements are indented wrt the 'PROC' and that the ':' signifies the end of the procedure:

```
PROC my.proc (INT z)
  SEQ
    ... body of procedure
:

SEQ
  my.proc(x) -- 1st call
  ... some more statements
  my.proc(y) -- 2nd call
```

Formal Parameters are the variables passed in/out of the procedure. In FORTRAN we can always modify these variables in the subroutine. In Occam there are two methods of passing parameters:

```
PROC sub1(INT x)
```

allows x to be modified in the procedure (this is called 'passing the parameter by reference') and

```
PROC sub1(VAL INT x) -- note the 'VAL'
```

means that x is read only for this procedure and cannot be modified within it (this is called 'passing the parameter by value').

Note that if the PROC does not have any parameter you still have to include the brackets in the declaration:

```
PROC my.proc()    -- note the empty ()
  SEQ
    ... procedure body
:
SEQ
  my.proc()       -- note the empty ()
```

Passing parameters by value 'adds' a level of 'safety' to programs as it means that variables cannot be 'accidently' modified in a procedure.

SIZE

This is an Occam keyword which returns the size of an array:

```
[123] INT my.array:
INT array.size:
SEQ
  array.size:=SIZE my.array
```

would assign 123 to array.size.

SIZE is useful in procedures etc. which are passed arrays as parameters and the procedure does not know before hand the size of the array.

e.g.

```
PROC my.procedure([] INT ip.array)
  SEQ
    SEQ i=0 FOR SIZE ip.array
      ip.array[i]:=i
:
```

Functions

An Occam function is similar to a FORTRAN function and allows a procedure to return a result to the caller e.g. SIN(x), COS(x), TAN(x) are all functions:

```
#USE snglmath
REAL32 z,angle:
SEQ
  z:=SIN(angle)
  z:=SIN(angle) + COS(angle)
```

The TDS system includes many libraries of functions e.g. mathematical functions (SIN, TAN, SQRT), i/o libraries etc. A 'simple' function which takes an integer parameter and returns an integer result looks like:

```
INT FUNCTION my.func(VAL INT parameter)
  INT my.result:
  VALOF -- note new keyword 'VALOF'
    SEQ
      ... body of function using parameter
    RESULT my.result                    -- new keyword
:
```

The 'VALOF' keyword instructs the Occam compiler that a result will be generated by the following instructions. The 'RESULT' keyword assigns the value to this result. The formal parameters must be of type VAL i.e. passed by value.

Multi-Transputer Programs

In this section we explain how multi Transputer programs are written and how Transputer networks are configured.

EXE's and PROGRAMS

At the moment you have been writing programs for a single processor (known as EXE's). An EXE is a 'special' type of program which runs on the first Transputer in the APOLLO which communicates with the APOLLO screen, keyboard etc. Fig (11)

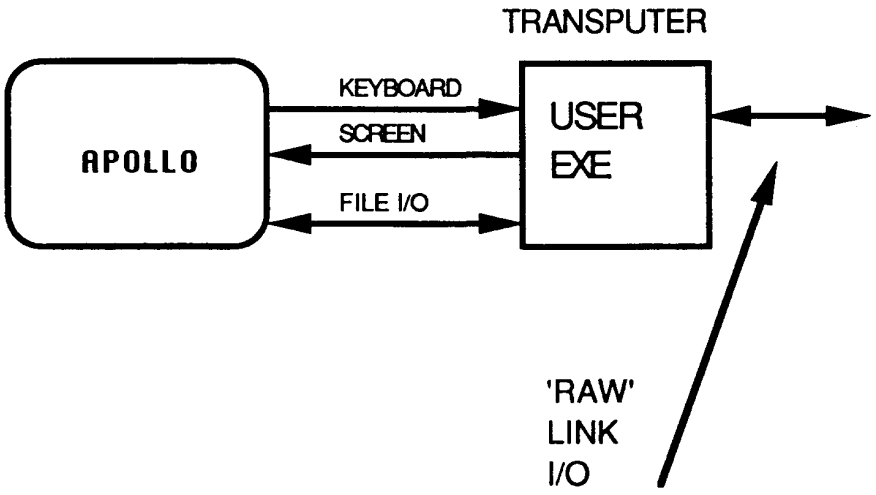
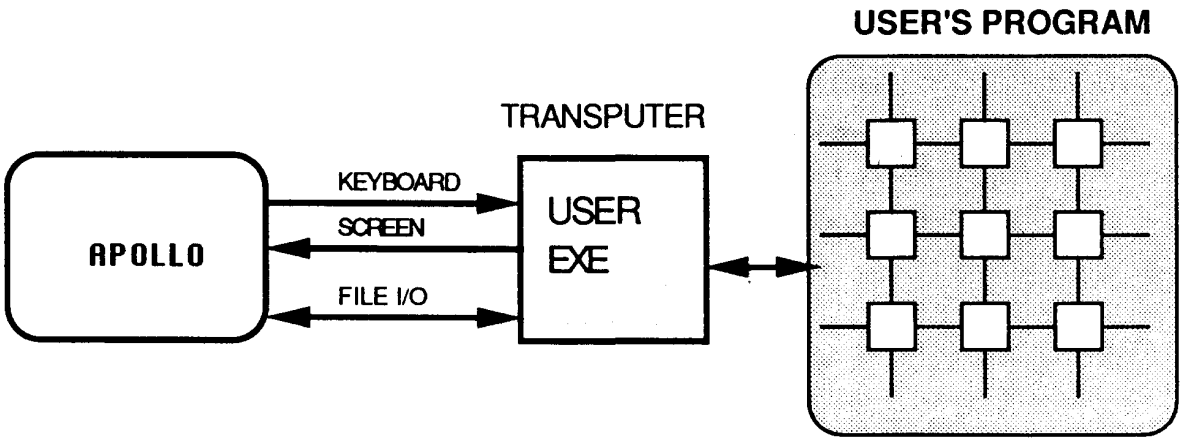


FIG (11)

An EXE runs on a single Transputer, the code which runs on more than one processor is known as an PROGRAM.

A PROGRAM can run on more than one Transputer. It contains a special Configuration Language section which specifies which Transputer a process will run on and how the links should be connected between Transputers. It has no access to the keyboard, screen etc. and can only perform 'raw' link i/o. Fig (12



FIG(12)

Occam as a Configuration Language

The Occam allows:

Processes to be PLACED on the various processors in the network.

The assignment of Occam channels in the processes to physical Transputer links.

The definition of the types of processors in the network.

The configuration language uses the statements:

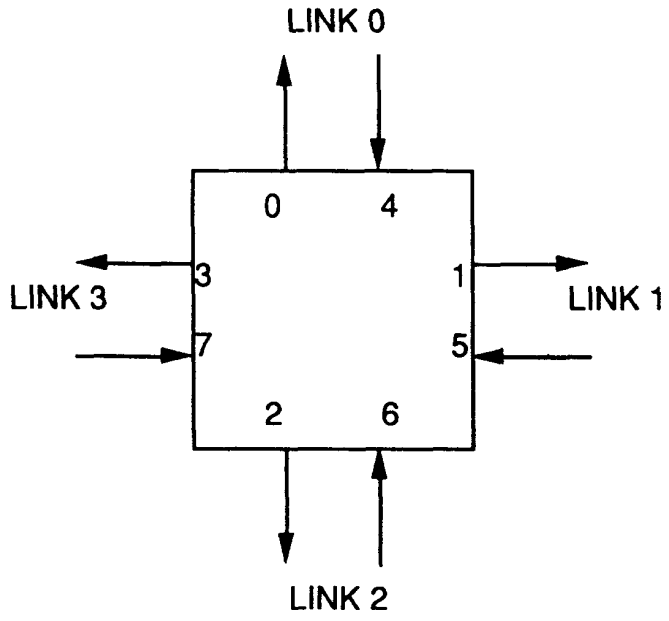
```
PLACED PAR

PROCESSOR number transputer.type

PLACE channel AT link.number:
```

where:

```
number                  identifies the processor
tranputer.type          is either T2,T4 or T8
channel                 is an occam channel name
link.number             is the Transputer link number
```

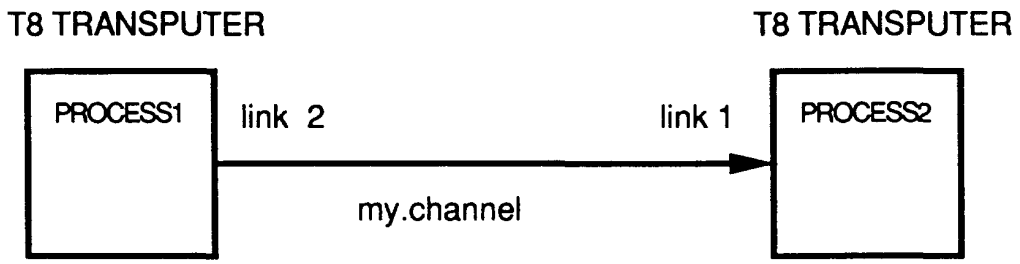
The `link.numbers` specify the link inputs or link outputs, Fig(13):

```
0,1,2,3          Output Links 0 to 3
4,5,6,7          Input Links 0 to 3
```



FIG(13)

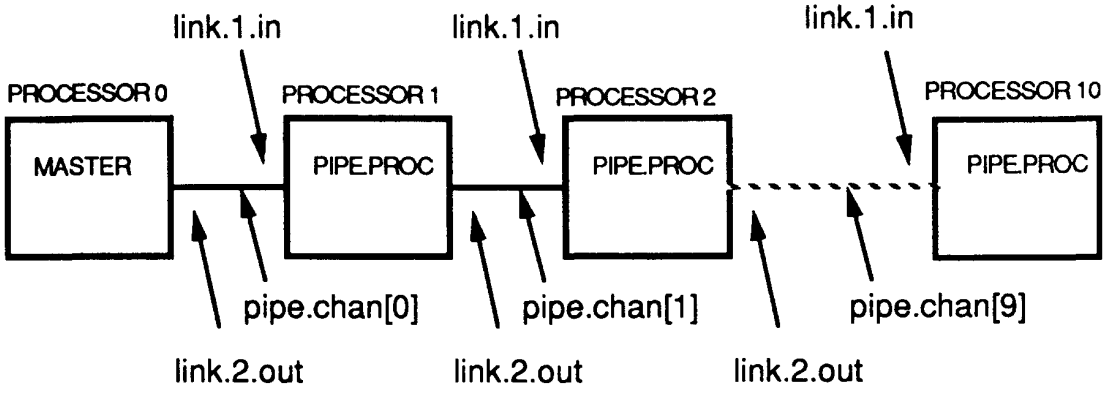
The details of the PLACED PAR can best be described by an example, fig(14):



FIG(14)

```
... SC process1(CHAN OF ANY out.chan)
... SC process2(CHAN OF ANY in.chan)

CHAN OF ANY my.channel:
VAL link.2.out IS 2:
VAL link.1.in  IS 5:
PLACED PAR

  PROCESSOR 1 T8
    PLACE my.channel AT link.2.out:
    process1(my.channel)

  PROCESSOR 2 T8
    PLACE my.channel AT link.1.in:
    process2(my.channel)
```

A PLACED PAR can be replicated, suppose we have some processors in a pipeline, Fig(15):



FIG(15)

Then the configuration language will be:

```
... SC pipe.proc
... SC master.proc

[10] CHAN OF ANY pipe.chan:
VAL link.2.out IS 2:
VAL link.1.in  IS 5:

PLACED PAR

  PROCESSOR 0 T8
    PLACE pipe.chan [0] AT link.2.out:
    master.proc(pipe.chan [0])

  PLACED PAR i=1 FOR 9
     PROCESSOR i T8 -- note 'i'

       VAL to.next.proc IS i:
       VAL from.last.proc IS i-1:

       PLACE pipe.chan[to.next.proc] AT link.2.out:
       PLACE pipe.chan[from.last.proc] AT link.1.in:

       pipe.proc(i,pipe.chan[to.next.proc],
                 pipe.chan[from.last.proc])
```

Note: i can be passed as a parameter

Timers

The Transputer has two internal clocks. A low process priority clock which counts in ticks of 64μs and a high priority clock which counts in ticks of 1μs. A timer behaves like a channel connecting the processor clock to a process. It is different from a normal channel in the following ways:

It can only be used for input; It is always ready to communicate; It can be shared among concurrent processes on the same processor.

A timer counts in 'ticks', thus reading the clock returns the current value of the timer in 'ticks'. How to use Timer:

declaration: `TIMER my.timer:`
use: `my.timer ? the.time.now`

To time a section of code:

```
INT start.time,finish.time:
REAL32 execution.time:
TIMER my.timer:

SEQ
  my.timer ? start.time
  ... section of code to time
  my.timer ? finish.time

  execution.time:=( REAL32 ROUND (finish.time MINUS start.time))*
                  0.000064 (REAL32)
```

Delays

A process can be blocked until a specified time by using the Occam AFTER keyword:

```
my.timer ? AFTER wait.time
```

where wait.time is an integer expression. This will wait until the time equals 'wait.time'

A delay can be generated by first getting the current time

```
my.timer ? time.now
```

and then specifying the time to wait for:

```
my.timer ? AFTER (time.now PLUS delay)
```

Note the use of the PLUS and MINUS operators, this is a modulo arithmetic operation (see later). Delay will be specified in 'ticks'. We can also have arrays of timers;

```
INT now:
[10] TIMER clock:

clock[0] ? now
clock[0] ? AFTER (now PLUS 1)
```

Modulo Arithmetic

The modulo keywords are: `PLUS, MINUS, TIMES`

What use are modulo operators?

They perform the same operation as the corresponding arithmetic operators except that there is no overflow checking. The Transputer is a 32-bit processor, with 31-bits representing the number and the 32nd bit representing the sign (+ or -). The maximum positive integer is thus:

```
#7FFFFFFF
```

The operation:

```
#7FFFFFFF+1
```

will set the Transputer overflow bit and cause an error, as the number generated is greater than the maximum integer representation.

The use of the PLUS operator will not create an overflow, the result of

```
#7FFFFFFF PLUS 1
```

would be

```
#80000000
```

What use is this?

Modulo arithmetic is essential when performing arithmetic with timers. An analogy is: If the time is 23.55 (11.55 p.m.) what will the time be in 1 hour? The answer is not 24.55 but 00.55 i.e. the time has 'wrapped around'.

The same ideas apply with Occam timers. Suppose the timer returned the value #7FFFFFFF:

```
clock ? time.now
```

and we want a delay of just one tick, which will be at the time #80000000:

```
clock ? AFTER (time.now + 1)
```

However the addition #7FFFFFFF+1 would cause an overflow. We want to wait for the absolute time #80000000, the use of the PLUS operator would calculate this quantity correctly:

```
clock ? AFTER (time.now PLUS 1)
```

Note the MINUS operator used in the code timing example.

Scope

Associated with each variable name is a region of a program in which it is valid. Variables are only valid in regions of the program starting directly after their declaration, indented to the right and ending when an occam statement returns to the same level of indentation as the declarations:

```
INT my.variable:
SEQ
  ... my.variable scope valid

... my.variable scope invalid
-- as we are back at the same level of
-- indentation as original declaration
```

e.g.

```
INT x:
SEQ
  x:=x+1 -- scope of x valid
  y:=x+2 -- scope of x valid
```

whereas:

```
INT x:
SEQ
  x:=x+1 -- scope of x valid
y:=x+2   -- scope of x invalid
```

After the scope of a variable is finished the variable ceases to exist in the program. This allows a name to be re-used for a 'local' variable. e.g.

```
INT local.variable:
SEQ
  local.variable:=100
  ... code where local.variable is 100
... more of program

-- At this point the scope of local.variable has ended and thus ceases to exist in the
-- program


INT local.variable:
SEQ
  local.variable:=200
  ... code where local.variable is 200
```

Priority Processes

The Transputer has two priority levels for processes: high and low. If a high priority process and a low priority process are ready to execute at the same time then the high priority process will execute and will block the low priority process. High priority processes are different from low priority processes in that they are not time-sliced, but run to completion or a communication block.

PRI PAR

High priority processes are declared using the occam PRI PAR statement. The rules for its use are:

The PAR contains only two Occam processes. The first textually declared process will run at high priority, the second textually declared process will run at low priority.

The Occam looks like:

```
PRI PAR
  SEQ
    ... process 1
  SEQ
    ... process 2
```

process 1 will be at high priority, process 2 will be at low priority.

As high priority processes will execute to the exclusion of low priority processes then they should be short and execute quickly. Interrupt service routines or device-driver routines are examples of things which would need to run at high priority.

How do you get more than one process at high priority?

```
PRI PAR
   PAR -- this 'par' process is at high priority
     ... proc 1 at high priority
     ... proc 2 at high priority
     ... proc 3 at high priority
   SEQ -- this 'SEQ' process is at low priority
     ... proc 4 at low priority
```

PRI ALT

Can also assign a priority to processes in an ALT, this is useful so that we can define which process will execute if the inputs to the ALT are ready at the same time. e.g.

```
ALT
  chan.1 ? x
    ... process 1
  chan.2 ? y
    ... process 2
```

If chan.1 and chan.2 are ready at the same time the Occam definition does not define which process will execute (it is a compiler implementation detail)

However, with

```
PRI ALT
  high.priority.input ? x
    ... process 1
  lower.priority.input.1 ? y
    ... process 2
  lowest.priority.input.2 ? z
    ... process 3
```

The inputs have a 'sliding scale' of priority, starting with the first textually declared input as the highest priority and the next textually declared input at a lower priority and so on.

Replication of PAR and ALT

The replication of PAR is identical to the replication of SEQ except that the replication count must be a constant and not a variable. The reason for this is that all Transputer workspace is allocated at compile time, you cannot have 'dynamic' generation of processes:

```
PAR i=0 FOR 10
  ... process -- is allowed
```

But

```
INT x:
SEQ
  ... process using x
  x:=10*x
  PAR i=0 FOR x -- is NOT allowed
    ... process
```

Replication can be useful for executing the same process or procedure in parallel e.g.

```
#USE snglmath
[100] REAL32 sin.values:
PROC my.procedure(VAL INT proc.num, REAL32 result)
  VAL pi IS 3.142(REAL32):
  SEQ
    result:=SIN((REAL32 ROUND proc.num)*pi)
:

PAR i=0 FOR 100
  my.procedure(i,sin.values[i])
```

would give 100 versions of my.procedure running in parallel each calculating the SIN and assigning it to sin.values[i].

ALT can be replicated (a variable number of times):

```
[10] CHAN OF ANY input.channel:
[10] INT x:
ALT i=0 FOR 10
  input.channel[i] ? x[i]
    my.proc(i)
```

In this way an array of channels can be scanned for input.

Recursion not allowed

Recursion is a term to describe a procedure or process calling itself. It is not allowed in FORTRAN but 'C' and Pascal allow it. It requires dynamic allocation of stack space etc and is not allowed in Occam.

Bitwise operations

To allow low level operations on the individual bits in a value, 0ccam provides bitwise operators;

Bitwise AND	=	/\
Bitwise OR	=	\/
Bitwise EOR	=	><
Bitwise NOT	=	~

e.g.

```
INT x,y:
SEQ
  x:=#FFFF1234
  y:=x /\ #0000FFFF
```

'y' would have the value #00001234

Conversion of Data Types

There are some simple rules for the conversion between data types. For conversion of bit-precisions of the same type:

```
INT32 i:
INT16 j:
REAL32 x:
REAL64 y:
SEQ
  i:= INT32 j
  j:= INT16 i
  y:= REAL64 x
```

However, when converting from 64 to 32 or 32 to 16-bit variables, the number must 'fit' into the specified number of bits:

```
INT16 x:
INT32 y:
SEQ
  y:=#55    (INT32)
  x:= INT16 y                -- OK
  y:=#10000   (INT32)
  x:= INT16 y                -- Overflow
```

For conversion between data types it can be specified whether the conversion ROUNDS or TRUNCATES bits during the conversion: e.g.

```
REAL32 x:
REAL64 y:
INT32 i:
SEQ
  x:= REAL32 ROUND i   -- rounded conversion
  x:= REAL32 TRUNC i   -- truncated conversion
  y:= REAL64 ROUND i   -- rounded conversion
  y:= REAL64 TRUNC i   -- truncated conversion
  i:= INT32 ROUND x    -- rounded conversion
  i:= INT32 TRUNC x    -- truncated conversion
```

Conclusions

We have seen that Occam and Transputers provide an important starting point for learning about parallel processing. However, in many applications there is a need to be able to program multiprocessors in familiar high level languages such as Fortran and C. Here there is a requirement to somehow incorporate parallel processing concepts into languages which were originally conceived for sequential programming. This has be done for the Transputer by INMOS and others , but that is another story, maybe at another CERN School of Computing.

Further Reading

M. Homewood et al., The IMS T800 Transputer, IEEE Micro vol 7., no. 5, October 1987.

Occam 2 Reference Manual, Prentice Hall, London , 1988.

J.Galletly, Occam 2, Pitman Publishing, London, 1990.

D. Pountain and D. May, A tutorial introduction to Occam programming, BSP books, London, 1987.

INMOS, Transputer development system, Prentice Hall, London, 1988.

Dick Pountain, Virtual channels: the next generation of Transputers, BYTE Magazine, E&W 3, April 1990.

A.J.G. Hey, Transputers and Occam, 1988 CERN School of Computing, CERN 89-06

D.May, The influence of VLSI technology on computer architecture, Phil. Trans. R. Soc. Lond. A 326, 377-393 (1988).

Acknowledgements

The 1989 and 1990 CERN Schools of Computing offered students the opportunity to learn about parallel processing using the Occam language to program arrays of Transputers. A series of formal lectures gave a basic grounding in Occam and a complementary set of practical exercises were organized in two teaching laboratories. In 1989 we used 12 IBM Personal Computers and in 1990 12 Apollo workstations. Each PC or workstation was equipped with at least 4 Transputers.

The write up we have provided for the 1990 School proceedings contains an expanded version of the lectures given by Ian Willers and myself. This material was originally presented by David Jeffery and I at the 1989 School and revised with the help of Rekha Gupta.

I wish to thank the many people who contributed to the success of the practical courses at the 1989 and 1990 schools.

IBM and Hewlett Packard provided personal computers and workstations.

INMOS and Cresco Data provided Transputer software.

INMOS and Transtech provided Transputer boards.

Southampton University and INMOS loaned us their experts.

Mike Jane of the Rutherford-Appleton Laboratory organized Transputer loans with his normal patience and tolerance.

Bob O'Brien did a wonderful job of organizing the Apollo loans, arranging for equipment to be flown in specially from the USA for the 1990 School.

Brian Martin made a simple but vital adaptor board for plugging Transputer boards into the Apollos.

The following people acted as lab demonstrators; Bob Dobinson, Rekha Gupta, Andy Hamilton, Andy Jackson, David Jeffery, Henrik Kristensen, William Lu and Ian Willers.

Bob Dobinson, ECP Div,
CERN, November 1990