TEMPUS
Parallel Computing

Lecture notes

and

practical works on using of

t r a n s p u t e r s

London
1996

Contents

Introduction 5

THE CONCEPTION OF COMMUNICATING SEQUENTIAL PROCESSES (CSP) 6

THE PROGRAMMING LANGUAGE OCCAM, BASED ON CSP 7

Introduction 7

Basics 7
Processes and channels 7
Names 9
Types 9
Operators 14
Exercises 16

Primitive processes 17
Assignment process 17
Input process 18
Output process 18
SKIP 18
STOP 19
Exercises 19

Non-primitive processes 19
SEQ construction 19
PAR construction 20
Conditional processes 20
Selection processes 22
Repetitive processes 23
Alternative processes 24
Communicating processes 25
Replicators 28
Exercises 31

Subroutines 31
Abbreviations 31
Retyping 37
Procedures 38
Parameters 40
Functions 44
Function definitions 47
Exercises 47

Configuration 48
Processor 48
Allocations 48
Hard channels 49
Hardware protocols 49
Ports 50

PRACTICAL WORKS 51

Unit 1: Basics and Inmos Toolset 51

Unit 2 : Simple programs 52
Logic gates 52
Parallel transfer of an array 53

Unit 3: Pipeline (a sequential program) **54**
Sequential program for Newton-Raphson method 54
Codes of program 54
Benchmark timings 56

Unit 4: Pipeline (a parallel program) **57**
Parallel program for Newton-Raphson method 57
Codes of program 57
Benchmark timings 59

Unit 5: A simple network program **60**
Message passing between two processes 60
Codes of program 60

Unit 6: Transforming the simple network program to run on a network (of one transputer) **64**
Message passing between two transputers 64
Codes of program 64

Unit 7: Pipeline (a parallel multitransputer program) **69**
Parallel multitransputer program for Newton-Raphson method 69
Codes of program 69
Speed, speed up and efficiency 72

APPENDIX **74**

THE INMOS OCCAM 2 TOOLSET **75**

About Inmos **75**

Getting started **75**
Writing a simple program 75
Compiling a simple program 77
Linking a simple program 77
Creating a bootable file 77
Running a simple program 78
Using the simulator 78
Using imakef 78

The occam libraries **79**
Operations with libraries 79
Toolset constants 80
Compiler libraries 81
User Libraries 81
Exercises 83

ANSWERS TO EXERCISES **84**

REFERENCES **86**

CSP **86**

occam **86**

Architecture **86**

Occam 2 Toolset **87**

FUTHER READING **88**

CSP **88**

Occam **88**

Architecture **89**

Transputer applications **89**

PROGRAMS AND ALGORITHMS (A SORTED LITERATURE) 90

Grouping of parallel methods **90**

Dataflow-based data-partitioning **90**
Pipeline-s 90
Tree-structured graph 91
Process farming 91
“Divide et impera!” 91
Geometric parallelism 91

Function-parallel programs **91**
Real-time systems 92

Other methods **92**

THE FOLDER EDITOR ORIGAMI 94

Origami User Guide **94**
1. Introduction 94
2. Using Origami 96
Appendix A 98

Origami Tutorial **103**
--{{{ Moving around a document 103
--{{{ Inserting and deleting lines 113
--{{{ Moving and copying 115
--{{{ Summary of editing operations 117

Introduction

Transputers are high performance microprocessors that support parallel processing through on-chip hardware. They

- can be connected together by their serial links in application-specific ways and
- can be used as the building blocks for complex parallel processing systems.

The transputer is a complete microcomputer on a single chip. It has a very fast (single cycle) on-chip memory, on-chip inter-processor links, and a programmable memory interface that allows external memory to be added with the minimum of supporting logic.

The main parts of the transputer:

- processor;
- on-chip RAM;
- system services;
- link interface;
- application specific interface.

Multi-transputer system can be built very simply. The four high speed links allow transputers to be connected to each other in arrays, trees and many other configurations. The circuitry to drive these links is all on the transputer chip, and only two wires are needed to connect two transputers together.

In addition to providing a communication link between programs running on processors, transputer links

- allow memory to be examined without loading a program, and
- permit programs to be loaded and executed.

This allows whole networks of transputers to be loaded down a single transputer link.

Each single transputer supports parallel processing through a system of internal channels implemented as words in memory. Each transputer has

- a highly efficient built-in run-time scheduler ;
- processes waiting for input or output or waiting on a timer consume on CPU resources.

The programming language occam 2 has been designed to reflect the architecture of the transputer, and for maximum coding efficiency the whole system can be programmed in occam 2. The inherent security and code efficienty of occam and the ability to use the special features of the transputer make occam 2 a powerful tool for programming concurrent systems.

Transputers can also be programmed in C, Fortran Pascal and Ada, and their optimised design ensures efficient code. Where the programs need to exploit concurrency but still need to use languages other than occam 2, special occam code can be used to link modules together.

The conception of Communicating Sequential Processes (CSP)

From the study of process synchronisation problems grew out the theoretical model of Communicating Sequential Processes (CSP). Occam is in fact an able realisation of CSP conception, we can image it like one of its subset.

CSP is a mathematically-based notation for specifying the behaviour concurrent processes. Within the framework of CSP, a program is a collection of sequential processes, each of which may be executing concurrently with the others. The process may only interact or communicate via inter-process input/output operations - these input/output operations are the only interaction allowed between processes. The communicating processes are fully synchronised, in that when a process reaches an input (output) operation, it waits for the corresponding process to reach the matching output (input) operation. At this point the input/output operation is performed - the process are in synchronisation - and then both processes resume their execution at their own speeds. There is no queuing or buffering of messages.

A complex system may be completely (mathematically rigorously) specified in CSP. As indicated above, CSP embodies a notation for expressing the execution and interaction of a system comprising a collection of concurrent processes. The table shown overleaf contains a few examples of statements in CSP notation.

P ; Q	process P is followed sequentially by process Q
P ‖ Q	process P executes concurrently with process Q
P≤b ≥Q	process P is executed if Boolean b is true, else process Q is executed
b * P	process P is executed while Boolean b is true
x := e	expression e is assigned to variable x
c ! e	the value of expression e is output
C ? x	a value is input and assigned to variable x

The programming language occam, based on CSP

Introduction

Occam is a programming language which from the outset was designed to support concurrent applications i. e. those systems whose parts may operate independently and sometimes have a need to interact. The name occam is derived from the name of 14th century Oxford philosopher, William of Occam. He was particularly famous for a quotation known as Occam's Razor: "Entia non sunt multiplicanda prater necessitatem". Translated from the Latin, this says: "Entities should not be multiplied beyond necessity" or paraphrasing, "Keep things simple". This is the whole philosophy behind the occam programming language - programming is kept simple. Excess language constructs have not been specified in the occam definition. This simplicity helps understanding and us. Occam programs should be more reliable, more efficient and easier to reason about.

Occam is an ideal language for scientific and engineering applications, for industrial process control applications, and for embedded systems, be they for domestic, industrial or military consumation. Its real-time features endow occam with the facilities necessary for the development of real-time applications.

Basics

Processes and channels

In occam programming we refer to the parts of a program as **processes**. A process
- starts;
- performs a number of actions and then
- finishes.

In conventional programming languages such as BASIC, much of the activity of a program consists of changing the values, such as numbers or strings of characters, stored in variables. Take for example the next BASIC program:

```
10    LET A=2
20    LET B=A
30    PRINT B
40    END
```

The result of running this program is that the value in both variables A and B becomes 2, and the line 30 causes this value to be printed out of the screen. There is communication of a limited sort going on in this program:

- the PRINT command provides one-way communication between the program and the external device, the screen;
- there is also a sense in which the value 2 has been communicated from A to B, Thought we wouldn't normally dignify this act with the name 'communication', because there is only one BASIC program running and it's being executed one line after another. Instead we tend to regard the value 2 as being stored in both A and B.

Occam permits this sort of communication as a normal feature of programming. More importantly, occam doesn't mind weather the two programs which so communicate are running on different computers or are just two processes running concurrently on the same computer.

A channel is a one-way, point-to-point link from one process to one other process. A transfer over a channel is actually an act of copying; if the value is output from a variable, then the variable retains it value and a copy of it is sent over the channel. Occam uses symbol ! to mean output and ? mean to input so we could express the above examples by

where A is a channel and B is a variable. This reads as 'output 2 to A' and 'input from A to B'.

Since processes 1 and 2 are independent, they might well executed at different times. The act of transferring a value from one end of the channel to the other can only happen when both processes are ready. In other words, if the output in process 1 is executed before the input in process 2 executes, process 1 will automatically wait for process 2 before sending a value. Vice versa, if the input in process 2 were executed before process one had output, process 2 would wait for a value to appear. There is no way for a value to be output into 'thin air' and lost.

The two novel features which distinguish channels from the variable are:

- a channel can pass values either between two processes running on the same computer, or between two processes running on different computers. In the first case the channel would in fact be just a location in memory, rather like a variable. In the second case the channel could represent a real hardware link such as a Transputer link or other communication line. Both cases are represented identically in an occam program.

  The key idea is, that an occam channel describes communication in the abstract, and does not depend upon its physical implementation. You can trust write and test a program using channels without having to worry about exactly where the different processes will be executed. The program can be developed on a single processor workstation; when it's finished and proved you may decide to distribute various processes in the program onto different computers, and do so by making a few simple declarations at the beginning of the program.

- channels are patient and polite. If an input process finds that no value is ready it will wait until one is supplied, without explicit instruction from the programmer. Equally an output will not send until the responsibility for 'timekeeping' of the programmers shoulders.

Communication over self-synchronising channels is a novel and powerful part of occam, and it can render the writing of concurrent programs a far less formidable task than it is with conventional languages.

Names

In occam the names of objects can be as long as you like, and they must start with a letter of the alphabet. The rest of name, if there is one, can be made up of letters, digit and the dot character. Upper and lower case are distinguished by occam, so that tom and TOM are different names. These are valid names:

x Y fred chan3 Chan3 new.fred old.fred

Occam keywords such as SEQ, PAR and CHAN are always in upper case and they are reserved. In other words they cannot be used as names that you create.

These are not valid names:

3chan	--doesn't start with the letter
old-fred	--contains illegal character '-'
fred$	--contains illegal character '$'
CHAN	--reserved word CHAN

Types

Occam, like Pascal and many other languages, but unlike BASIC, requires that every object that is used by a program should have a type which tells occam what sort of object it is dealing with. Furthermore the type of an object must be specified before it can be used in a process. Noticed the colon, which is used to end all the different kinds of specification. This colon joins a specification to the process which follows it.

The type can be primitive type and array type. The primitive types are the following: channel type, timer type and data type. The data types are: integer type, byte, real and Boolean.

CHAN OF protocol

A communication channel is of type CHAN OF protocol. Each communication channel enables values to be communicated between two concurrent processes according to the specifies protocol. It is necessary to specify the data type and structure of the values that they are to carry. This is called the channel protocol. Examples:

```
CHAN OF INT keyboard:
CHAN OF INT screen:
CHAN OF BOOL signal:
```

Timer

A timer is of type timer. Each timer provides a clock which can be used by any number of concurrent processes. Example:

```
TIMER clock:
```

INT

It is an integer or whole number. INT16, INT32, INT64 are numeric types represented using 16, 32, 64 bytes respectively. Examples:

```
INT x, X:
INT a, b, c:
```

BYTE

It is a nonnegative integer between 0 and 255, very often used to represent characters. Example:

```
BYTE x:
```

REAL

It is a signed real value, represented according to IEEE standard 754-1985. REAL32, REAL64 are numeric types represented using 16, 32, 64 bytes respectively. Examples:

BOOL

The BOOL type is stored in 8-bits - TRUE having the numeric value 1, False having the numeric value 0. Example:

```
BOOL more:
```

Array

Array types are constructed from component types. An array type is a channel type, timer type or data type, depending on the types of its components. Two arrays have the same type if they have the same number of components and the types of their components are the same. In the array type [e] T, the value of e defines the number of the components in an array of the array type, and T defines the type of the components. In occam array variables are declared in the same way as single variables of any type, but with the number of components in brackets prefixed to the type specified. But the size of the array must be fixed when the program is compiled; it cannot be decided or changed while the program is running. Examples:

```
[20] INT fred:                      -- an array of 20
                                    -- integer called fred
[100] CHAN OF INT switchboard:      -- an array of 100
                                    -- channels caled
                                    -- switchboard
[12] BOOL jury:                     -- an array of 12
                                    -- truth values called
                                    -- jury
[4][5] INT matrix:                  -- an array of
                                    -- 4 elements,
                                    -- every element of
                                    --  each is [5] INT
fred[0]                             -- the first integer
                                    -- in fred
switchboard[20]                     -- the 21th channel in
                                    -- switchboard
jury[11]                            -- the last component
                                    -- of jury
```

Constants

A name can be given to a constant value by specifying it with:

|VAL type name IS value:

So we could write:

```
VAL INT year IS 365:
VAL INT leap.year IS 366:
```

The type can omitted as occam can deduce it from the value:

```
VAL year IS 365:    -- year is an integer
VAL spac IS ' ':    -- space is a BYTE
```

Possible ambiguities over BYTE and INT are resolved by explicitly specifying the type of the value.

Hexadecimal notation

Numeric constants can be entered in hexadecimal notation by preceding them with the # sign:
#FE (equivalent to 254 decimal)

Scope

The named objects of an occam process, such as constants, variables and channels must be declared in advance of any process in which they are used. The scope or range of definition of named objects is the range of definition of the process which immediately follows they declaration. The range of definition of a process is determined by its indentation, and so the scope of named objects is defined by the indentation of the following process. The named objects are said to be local to that process and any component processes.

```
INT item:       -- declaration of item
SEQ             -- scope of item
  item := 10
  .
  .
  .
  SEQ
    item := item+1
```

In this example, the variable item is available to the inner SEQ constructions since this variable is still in scope.

```
PAR
  INT fred:
  SEQ
    chan3 ? fred
    more processes
  INT jim:
  SEQ
    chan4 ? fred
    more processes
```

An error will be reported, because fred exists only inside the first SEQ and jim exists only inside the second SEQ. The second fred will therefore look to occam like an unspecified variable.

Consequence of scoping is that the value of a variable is undefined outside the scope of the declaring process. This means that program objects with different scopes may have the same name but lead independent existences i. e. they are totally separate objects, and may even have different types.

```
CHAN OF BYTE chan:
PAR
  BYTE char:          -- first declaration of char
  SEQ                 -- scope
    char := 'a'
    chan ! char
  BYTE char:
  SEQ
    chan ? char
```

A variable named char is declared for both sequential processes. However these variables are quite separate, each being local to its own sequential process. Channel chan is declared outside of PAR construction and so the channel is common to both component processes.

If an object is declared within the scope of an existing object with the same name, and then all references to that name refer to the more recent declaration i. e. the more recent declaration takes precedence in a clash of scope. The older declaration is suspended for the duration of the scope of the more recent declaration but resumes its role afterwards.

```
BYTE item:            -- declaration as BYTE
SEQ                   -- scope as BYTE
  item := 50 (BYTE)
  INT item:           -- declaration as INT
  SEQ
    item := 500
    chan2 ! item      -- gives 500
  chan1 ! item        -- gives 50 (BYTE declaration
                      -- restored)
```

where chan1 is specified as type BYTE, and chan2 is specified as type INT. In this example, the second declaration of variable item masks the first declaration for the duration of the second's scope. Any reference to item during this period will refer to the second declaration. Thus the output via channel chan2 produces the value 500 as the second declaration in force here. However the output via channel chan1 produces the value 50 since the first declaration is back in force - the second one is no longer in the scope.

Specifying a variable in occam does not initialise its value to zero. The value of a variable is undefined garbage until it has been assigned to or has input a value. The value of variable only has meaning during the execution of the process for which is it declared. It is important to realise that occam does not guarantee that the value of a local variable be kept from one execution of procedure to the next. If a value of a variable needs to be retained between executions, then the variable must be declared in a higher level process. Such a variable is called free variable.

```
INT x:
  WHILE x >= 0        -- unwise: x is garbage here
    SEQ
    input ? x
    output ! x
```

```
INT x:
SEQ
  x := 0
  WHILE x >= 0       -- correct
    SEQ
    input ? x
    output ! x
```

Type conversions

Sometimes it's convenient to convert one type to another in a program; it may for instance save having to declare several extra variables for a value that is only required once. Type conversion should be prevent values being used in inappropriate situations.

If **number** has been declared as INT and **digit** as byte, we could still add them together like this:

```
number := (number * 10) + (INT digit)
```

The reverse conversion, of INT to BYTE is only legal if the value is within the byte range of 0 to 255. For example, to output a number between 0 and 9 as a character we could write:

```
output ! BYTE (number + (INT '0') )
```

Values of type BOOL can be converted to type INT or BYTE and vice versa, using the following definitions:

INT TRUE or BYTE TRUE	is 1
INT FALSE or BYTE FALSE	is 0
BOOL 1	is TRUE
BOOL 0	is FALSE

so if value of **active** is FALSE, INT **active** is 0.

Operators

Arithmetic in occam

The basic arithmetic operation are these:

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

These operations can be performed on numbers of type INT or REAL.

All operators have the same priority in occam so parentheses must be used in complex expressions to enclose component operations and allow them to be treated as single operands. This also established the order of evaluation. For example:

```
fred = (2 + jim)
(fred + jane) < jim
```

Relational operators

They allow operands to be compared, the result of the comparison is one of the logical truth values TRUE or FALSE. The occam provides the following tests:

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

These tests may only be applied to two values of the same type, and they always yield a value of type BOOL. For example the test 2 <> 3 yields the value TRUE since 2 does not equal 3. The truth values TRUE and FALSE are occam constants which can be used in any situation where a test could be used; you may like to think of them as tests whose outcome is decided in advance.

Operands, which may be constants, variables or expressions, used with the equality (=) and inequality (<>) operators may be any primitive data type, but the operands used with the remaining relational operators must be of byte, integer or real types only. Characters and strings may be compared with relational operators, according to their ASCII ordering. For example, if comp is declared to be type BOOL, then

```
comp := (3 = 25)                                -- evaluates to FALSE
comp := (2 < 25)                                -- evaluates to TRUE
comp := (7.15(REAL(32) >= 3.25(REAL32)-- evaluates to TRUE
comp := ('A' < 'B')                             -- evaluates to TRUE
comp := (fred < mary)                           -- evaluates to TRUE
```

Boolean operators

Boolean operators comprise the usual logical connectives, AND, OR and NOT. They allow the logical combination of Boolean operands - either simple Boolean variables or Boolean expressions. The result is a Boolean value, TRUE or FALSE. For example:

```
(fred < mary) AND (alfa > beta)
```

is the logical AND combination of the Boolean expressions fred < mary and alfa > beta. The value of the result depends on the current value of the numeric variables being compared.

Bitwise operators

To allow low level operations on the individual bits in a value, occam provides bitwise operators.

Setting, masking

These operators allow various operations to be performed on the individual bits or pattern of bits comprising the value of a constant, variable or expression of integer type. The first operand represents the bit pattern which operates on this value, and itself may be a constant, variable or expression of integer type.

~	or	BITNOT	bitwise not
/\	or	BITAND	bitwise and
\/	or	BITOR	bitwise or
><		bitwise exclusive or	

For example:

```
int \/ #3F
```

shows the bitwise oring of a variable int with the constant #3F. The bit pattern of #3F is 0011 1111, and so this operation will set to one the 6 low-order bits of int.

Shift operators

The bit comprising the value of an integer constant, variable or expression may be shifted left or right a specified number of bit positions.

<<	left shift
>>	right shift

Such shifting is equivalent to multiplying or dividing by multiples of two. The number of bit positions to be shifted is given by the second operand. This operand, which itself may be a constant, variable or an expression, must be of type INT. The number of bits to be shifted must be not be greater than the number of bits allowed for the integer type of the first operand. The bit positions of the first operand, vacated by the shift operation, are filled with 0 bits. For example:

```
int << 3
```

shifts the value of the variable int left by three bit positions; this is equivalent to multiplying by 8.

```
int >>7
```

shifts the value of the variable into right by seven bit positions; this is equivalent to dividing by 128.

Exercises

1.1. What is the aim of the transputers?

1.2. What is a process?

1.3. Which of the following are valid occam names?

int
INT
column.data
column data
column_data

1.4. Compute the following occam expressions:

(6*3)+2
6*(3+2)
6*3+2
5 + 3 > 16
5 + (3 < 4)
(16 - 4) < 3
250(BYTE) + 10(BYTE)
250(INT16) + 100(INT16)
15 PLUS 3
15 LESS THAN 26

1.5. Specify the types of following objects:

a real variable with 32 bits
a byte variable with 32 bits
an integer variable with 32 bits
variables seconds, minutes, index as integer

1.6. Give the declarations for following:

an integer constant with 16 bits initialised to -16
a byte constant initialised to -16
a Boolean constant initialised to -16
a real constant with 32 bits initialised to -16
an integer constant with 16 bits initialised to -163265
a Boolean constant initialised to TRUE

1.7. Check which of the following declarations and expressions are valid in occam:

INT x
INT a; b; c:
BYTE x:
BOOL TRUE:
[32] INT jane:
[10] BOOL jim:
[4;5] INT matrix
VAL INT year IS 365:
VAL INT leap.year IS 366:
VAL min IS 12
VAL MaxVal IS '125:

Primitive processes

Assignment process

An assignment process changes the value of variable, just as it would in most conventional languages. The symbol for assignment in occam is :=. So the assignment process:

```
fred := 2
```

makes the value in variable **fred** two. The value assigned to a variable could be an expression such as:

```
fred := 2 + 5
```

and this expression could contain other variables:

```
fred := 5 - jim
```

Multiple assignment, assignment to more than one variable at the same time, is also possible in occam:

```
fred, john := 2, 3
```

This multiple assignment process makes the value in the variable **fred** two, and the value in the variable **john** three. This is really useful for swapping the value of variables:

```
fred, john := john, fred
```

Input process

An input process inputs the value from the channel into a variable. The symbol for input in occam is ?. The input process:

```
chan3 ? fred
```

takes a value from a channel called **chan3** and puts it into variable **fred**.

Input processes can only input values to variables. It is quite meaningless to input to a constant or to an other expression.

An input process cannot proceed, until a corresponding output process on the same channel is ready.

Output process

An output process outputs a value to a channel. The symbol for output in occam is !. The process:

```
chan3 ! 2
```

outputs the value 2 to a channel **chan3**.

The value output to a channel can be anything that you could assign to a variable, so it may be a variable or an expression, and the expression may contain variables.

An output process cannot proceed until a corresponding input process on the same channel is ready.

So communication over a channel can only occur when both input and output processes are ready. If during the execution of a program, an input process is reached before its corresponding output process is reached, the input will wait until the output becomes ready. Should the output be reached first, it will wait for its input.

A value communicated over a channel is copied to the input variable and the value of the output variable remains unchanged.

SKIP

The special process SKIP may be thought of as representing a process which does nothing. It might be used in a partly complemented program in place of process which will be written later, but which for the moment can be allowed to do nothing.

For example a process which is to drive an electric motor could be replaced by SKIP when testing the program without motor. There are also occasions when you want nothing to happen, but the syntax of the occam requires a process to be present.

STOP

The special process STOP may be thought of as representing a process which doesn't work, or is 'broken'. It might be used, like SKIP, to stand in for a process which has yet to be written.

For example a process to handle errors could be replaced by STOP in the early stages of testing a program.

The effect of 'broken' process tends to spread, because any process which communicates with a broken process will itself never finish, and hence it becomes broken too.

So far we have loosely used the term 'finish' when referring to process. Concurrent programming in occam requires us to be rather more precise than this. A process which completes all its actions is said to be terminate. Normally a process starts, proceeds and then terminates. A process which cannot proceed is said to be stopped by waiting for an event which will never happen, due to a programming error, in which case it is said to be deadlocked.

Correct termination of concurrent programs is not a trivial matter, since they may have many parallel processes which communicate with one another.

Exercises

1. What is a primitive process in occam?

2. Describe the primitive processes relating to communication!

3. Describe the SKIP and STOP primitive processes!

Non-primitive processes

Several primitive processes can be combined into a larger process by specifying that they should be performed one after the other, or all at the same time. This larger process is called a construction and it begins with an occam keyword which stated how the component processes are to be combined.

SEQ construction

The simplest construction to understand is the SEQ, short for sequence, which merely says 'do the following processes one after another'. Here is an example:

```
SEQ
  chan3 ? fred
  jim := fred + 1
  chan4 ! jim
```

This says, do in sequence, input from **chan3** to **fred**, assign **fred+1** to **jim** and output **jim** to **chan4.** In sequence means, to be more precise, that the next process does not start until the previous has terminated. A SEQ process therefore works just like a program in any conventional programming language; it finishes when its last component process finishes.

PAR construction

The PAR construction says: do the following processes all at the same time', I. e. parallel. All the component processes of a PAR start to execute simultaneously. For example:

```
PAR
  SEQ
    chan3 ? fred
    fred := fred + 1
  SEQ
    chan4 ? jim
    jim := jim + 1
```

says: at the same time, input from **chan3** to **fred** and then add one to the result, while receiving input from **chan4** to **jim** and then adding one of the result.

The important thing is to keep in mind is that in a PAR, written order of the component processes is irrelevant as they are all performed in the same time. PAR is not quite so easy to understand as SEQ, because the idea of things happening simultaneously in computer programs is new to many programmers.
All component processes in a PAR start at the same time, and the PAR construction itself terminates when all its component processes have terminated and that is all we need to know.

Conditional processes

In addition to repetition, all programming languages need to provide a way for programs to choose to do different things according to condition i. e. the result of a test. In occam one form of conditional choice is provided by the construction called **IF**.

IF can take any number of processes, each of which has a test placed before it, and make them into a single process. Only one of the component processes will actually be executed, and that will be the first one (in the order in which they are written) whose test is true:

```
IF
  x = 1
    chan1 ! y
  x = 2
    chan2 ! y
```

In this fragment of program (we assume **x, y, chan1** are **chan2** and declared elsewhere), the value of **y** will either be output on **chan1** or **chan2** depending upon whether the value of **x** is 1 or 2.

The tests **x = 1** and **x = 2** are Boolean expressions which are used to choose which component of the **IF** is to be executed. The component parts of the **IF**, each composed of a Boolean expression and a process, are called choices.

What if the value x were 3? Then the **IF** process would cause the program to stop just as if **STOP** had been executed. The program can only proceed if one of the choices is executed.

(An **IF** with no choices in it just acts like a **STOP** A **PAR** or **SEQ** with no component processes on the other hand acts like **SKIP** i. e. the program continues as if it were not there at all.)

In many cases it will not be acceptable to have the program stop if x not either 1 or 2. In that case we must add another choice which will be executed no matter what the value of x. This is accomplished by using **TRUE**.

```
IF
  x = 1
    chan1 ! y
  x = 2
    chan2 ! y
TRUE
  chan3 ! y
```

Now y will be output on **chan3** if x has any other value but 1 or 2, because of test on the last choice is always be executed by default if no previous choice has been executed. If we wanted nothing to happen at all when x was not 1 or 2 we could say:

```
IF
  x = 1
    chan1 ! y
  x = 2
    chan2 ! y
TRUE
  SKIP
```

This provides one example of the utility of **SKIP**; occam requires some sort of process after the guard and won't allow just blank space.

A better way of writing the above is to explicitly state each case as in the following example:

```
IF
  x = 1
    chan1 ! y
  x = 2
```

```
    chan2 ! y
(x <> 1) AND (x <> 2)
  SKIP
```

This is a much better way to write conditional processes as it is totally unambiguous. For convenience and simplicity in this tutorial we will, in places continue to use **TRUE** as a condition. In real programs you should avoid doing so.

To make more complex choices, **IF**s can be nested by using an **IF** process in a choice of another **IF** construct.

```
IF
  x = 1
    chan1 ! y
  x = 2
    IF
      chan2 ! y
    TRUE
    chan3 ! y
  TRUE
    SKIP
```

In this process **chan3** is used for the output if x is 2 and y has any other value than 1.

Selection processes

Like many other programming languages, occam provides a further means of making a choice depending upon the value of variable. Such a construction is called a selection in occam and provides an efficient means of selecting one of a number of options in a **CASE**.

CASE can take any number of processes, each of which has a list of one or more expressions, placed before it and combine them into an single process. Only one of the component processes will actually be executed, and that will be the first one (again, in the order in which they are written) with an expression, which has the same values the selecting variable:

```
CASE x
  1
    chan1 ! y
  2
    chan2 ! y
```

In this fragment of program (which is similar to the example, used to describe IF), the value of y will either be output on **chan1** or **chan2** depending upon whether the value of x is 1 or 2. Typically, the constants used in **CASE** will be named, and also there can be more than one **CASE** expressions:

```
CASE x
  i, j
    chan1 ! y
  k
    chan2 ! y
```

In this program fragment each constant expression has been given a name: i, j and k. The value of y will be output on **chan1** if x has the same value as i, and will be output on **chan2** if x has the same value as k.

What if the value of x was none of these values? Then the process would cause the program to stop just as if **STOP** had been executed. The program can only proceed if one of the option is executed, just as an **IF** may only proceed if a choice is executed.

Once again (as we saw with **IF**), in many cases it will be not acceptable to have the program stop if x does not have the same as one of the case expressions. We must then add a further option which will execute no matter what the value of x. This is accomplished by using **ELSE**:

```
CASE x
  i, j
    chan1 ! y
  k
    chan2 ! y
  ELSE
    chan2 ! y
```

Now y will be output on **chan3** if x has any value other than i, j or k.
The component parts of the **CASE**, each composed of an expression and a process, are called options.

Repetitive processes

All programming languages provide some means of looping, i. e. performing an action repeatedly. In general it's convention to distinguish two kinds of repetition: repeat for a specified number of times, or repeat while a given condition holds. occam has both types of repetition. The second conditional loop is performed by a construction called **WHILE**, which includes a test such as x < 0 or fred = 100. The resulting process is executed while the test result is true, or looked at another way, until it becomes false.

For example:

```
INT x:
SEQ
  x := 0
  WHILE x >= 0
    SEQ
      input ? x
      output ! x
```

will continue to read values from channel **input** and send them to **output** so long as the value is not less than zero. Every time the inner **SEQ** process terminates, the **WHILE** process will be performed again and the test repeated. This continues so long, as the test result is **TRUE** i. e. so long as **x** is greater or equal to zero. When a negative value is received the **WHILE** process terminates.

The net effect of this process is to buffer (i. e. store) a single value on its way from input to output. occam programs are often designed by making the major processes communicate on a channel to buffer, filter, or transform the transmitted values, almost as if they were electrical components rather than programs.

The logical values **TRUE** and **FALSE** can be used as constant in an occam program, anywhere that a test could be used so:

```
WHILE TRUE
  INT x:
```

```
  SEQ
    input ? x
    output ! x
```

will continue to read values for ever (or until you pull the plug), whereas:

```
WHILE FALSE
  INT x:
  SEQ
    input ? x
    output ! x
```

is a pointless sort of process which terminates immediately and will read no values at all.

Alternative processes

A process may use several channels for input, or several for output, or both, in order to communicate with one other process. So we can make choices according to the state of channels. This is made possible by the ALT construction, whose name is short for alternation.

ALT joins together any number of components into a single construction, but the component part of an ALT called alternatives are some more complicated.

The simplest kind of ALT has as each alternative an input process followed by a process to be executed. The ALT watches all the input processes and executes the process associated with the first input to become ready. Thus ALT is basically a first-past-the-post race between a group of channels, with only the winner's process being executed:

```
CHAN OF INT chan1, chan2, chan3 :
INTx:
ALT
  chan1 ? x
     .... first process
  chan2 ? x
     .... second process
  chan3 ? x
     .... third process
```

If chan2 were the first to produce an input, then only the second process would be executed.

Here choice is being decided in the time dimension, the inputs causing the program to wait until one of them is ready.

Communicating processes

Communication between two processes is the essence of occam programming. At its simplest it requires two processes executing in parallel and a channel joining them:

```
INT x:
CHAN OF INT comm:
PAR
  comm ! 2
  comm ! x
```

This trivial program merely outputs the value 2 from one process and inputs it into the variable x in the second. Its overall effect is exactly as if we had a single process which assigned 2 to x.

Communication between the component processes of a **PAR** must only be done using channels, occam doesn't allows us to pass values between parallel processes using shared variable.

In fact if a component of a **PAR** contains an assignment or input to a variable, then the variable must not be used at all in any other component:

```
INT x, y :
PAR
  SEQ
    x := 2
    ... more processes
  SEQ
    y := x      -- illegal!
    ... more processes
```

Keeping variables local to component processes and using channels to communicate values is the right way to do it.

This may seem like a severe restriction to programmers who have experience with conventional languages. It will certainly be the biggest source of errors when first programming in occam.

Like a prohibitions it will be more easily borne if the reason for it understood. The reason is both simple and necessary.

Parallel processes run at the same time, and in general they run asynchronously i. e. at they own pace, only coming into synchronisation with each other briefly when forced to by communication over a channel.

If occam allowed one parallel process to read from a variable which has its value altered in another parallel process, what value will be read? It depends upon whether or not the other process has altered it yet, and this can't be known since the processes are asynchronous. And what if the altering process chooses to alter the variable's value at the precise moment that the second process is reading it? What would the value be then?

Such a scheme is obviously unworkable, hence the prohibition. But couldn't we organise it so that a variable warns the other process that it has had its value charged? We could indeed; the resulting object already exists in occam and is called a channel Q. E. D.

In occam variables are used for storing values, while channels are used for communicating values.

Lets now return to the main track with a more complicated example of **PAR** which performs some arithmetic on a value before passing it on:

```
CHAN OF INT comm:
PAR
  INT x:
  SEQ
    input ? x
    comm ! 2 * x
  INT y:
```

```
SEQ
comm ? y
  output ! y + 1
```

Here we have two channels called **input** and **output** which lead to other processes or perhaps o the outside world. We assumed that they have been declared elsewhere in a larger program. This piece of program uses two processes working in parallel one of which multiplies an input value by two , the other adds one to the result and sends it on its way to the output. The times-two process and the add-one process communicate on channel **comm**.

In this case it worries you this is not a particularly useful thing to do; it is purely for illustration. It would be much simpler to do times-two and add-one in a single SEQ process, or indeed in a single expression. But later on when we have more of occam at our disposal, we shall see how this sort of thing can be very useful indeed. At this early stage, all examples of communicating PARs will tend unfortunately to appear trivial.

Its been said several times already that an occam channel is a one-way link between a pair of processes, but it is useful to now examine exactly what this implies. In a communicating PAR construct it means that:

1. Only two component processes of the PAR may use any particular channel, one as the sender and the other as receiver.

```
CHAN OF INT comm:
PAR
  SEQ
    comm ! 2
  INT y:
  SEQ
    comm ? y
  INT x:  -- ILLEGAL! two processes
  SEQ     -- inputting from the same channel
    comm ? y
```

2. The sender process must only contain outputs to the channel and the receiver must only contain inputs from the channel

```
CHAN OF INT comm:
PAR
  SEQ
    comm ! 2
  INT y:
  SEQ
    comm ? y
    comm ! y+1
     -- ILLEGAL! input and output
     -- from the same channel in the same process
```

For two-way communication between two processes we would need two channels:

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

```
comm2 ! 3
```

The effect is that each process sends a value to the other; x ends up with the value 3 and y with the value 2. The order of the inputs and outputs in each SEQ matters very much here and it's important to understand why.

If we were to write:

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

then the program would never terminate; we have the dreaded deadlock.

Why deadlock? Because both SEQs wait patiently for an input to become ready. But since each is waiting for the other to output, neither can proceed to make the necessary output! It's rather like those comical scenes when two people passing in a narrow doorway repeatedly step to the same side to make way, so repeatedly blocking each other. Swapping the input and output in either process resolves the deadlock.

Take care: sequence your programs to ensure that two parallel processes are never each waiting for a sequentially later output from the other. This is the only circumstance in which occam requires you to worry about such matters, but watch out for it. Like certain stalemates in the game chess, it may be disguised in complex processes.

Replicators

One of the most powerful features of occam is that it allows the construction of arrays of processes in addition to data and channel arrays.

The device replicator is used together with one of the occam constructs SEQ, PAR, ALT and IF to create an array of similar processes of the corresponding kind. Individual processes in a repliated construct can be referred to using the replicator index, in just the same way that components of an array are selected using a subscript. The general form of a replicator is:

```
REP index = base FOR count
  process
```

where Rep is one SEQ, PAR, ALT or IF. Take care: if the count of replicator is zero, the process behaves like a single construct with no components i. e. SEQ will act like SKIP (do nothing) but IF and ALT act like STOP (stop the process).

Replicated SEQ

The most straightforward replicated construct is the replicated SEQ. If input is specified as a channel then

```
INT x:
```

```
SEQ i = 0 FOR 5
  input ? x
```

says “create five replicas of the input process and execute them in sequence”. The efect is as we had written:

```
INT x:
SEQ
  input ? x
  input ? x
  input ? x
  input ? x
  input ? x
```

which in its turn is as if we had specified a loop with five iterations:

```
INT x, i:
SEQ
  i := 0
  while i < 5
    SEQ
      input ? x
      i := i + 1
```

In other words a replicated **SEQ** is equivalent to a counted loop. Using a replicated **SEQ** is more concise than using **WHILE** because there`s no need to specify and increment and index variable and test its value each time round.

The replicator index variable, which can be given any name, increases by one from the base value for count times, and can be used to reference particular pass through the loop. Note that the index does not need to be separately specified, and always of type INT. An array of any size could be filled with input values like this:

```
SEQ i = 0 FOR SIZE big.array
  input ? big.array[i]
```

which has the same effect as:

```
SEQ
  input ? big.array[0]
  input ? big.array[1]

  input ? big.array[(size big.array - 1)]
```

occam allows either or both base and count values in the replicated SEQ to be variables rather than constant.

Take care: It is forbidden to input or assign to the replicator index. Ther is no way to cause partial execution of a replicated SEQ (or any other replicated construct); remember that they are not loops but arrays or processes which only terminate when all their processes are terminated.

Replicated PAR

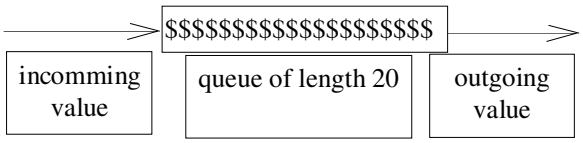
A replicated **PAR** builds an array of structurally similar parallel processes. Any process can be referred to by means of the replicator index.

The replicated **PAR** is of paramount importance in occam programming. Used in conjuction with an array of channels, it permits an economical and elegant expression of some of the

stock data structures used by programmers, such as buffers and queues, but futhermore it allows the exploitation of multiple concurrent processors using pipelining and other techniques.

Technical note: Current implementation of occam do not allow the count in a replicated **PAR** to be a variable value. This is in order that the compiler can know all the reseources needed by a process at compile time.

As an example let`s look first a simple queue. A queue is precisely what you would expect from the everyday meaning of the word, a structure through which a number of values pass, with the first value to arrive being also the first value to leave:



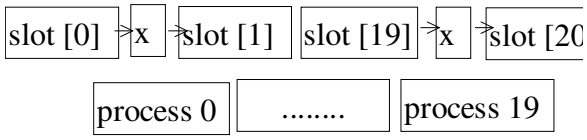
Queues are used extensively in programming, often as buffers for processes which consume data more slowly then they are being supplied. In order that no data should be lost, they are buffered or queued to wait their tum. Such buffers are often called FIFO buffers, short for First In/First Out.

A typical application would be in a computer terminal`s keyboard. Some programs may accept characters slower than a fast typist can enter them. If a queue is used to accept the characters, then users can type as fast as they wish, the queue growing as they type ahead of the program.

In occam a queue can be simulated by an array of parallel processes passing data from one to the other like a bucket chain:

```
[21]CHAN OF INT slot :
  PAR i = 0 FOR 20
    WHILE TRUE
      INT x :
      SEQ
        slot [i] ? x
        slot [i+1] ! x
```

The replicated **PAR** sets up 20 parallel processes each of which continually transfers values between two slots in the queue, which is represented by an array of 21 channels. The net effect is that each value has to pass through the whole queue before leaving from **slot [20]**.



As it stands this is not a complete program, because the first process in the queue needs somewhere to input the values from, and the last process needs somewhere to outputs to. Otherwise each will be waiting and the whole queue will be deadlocked. We assume that the queue is part of the larger program which feeds data into **`slot [0]`** and bleeds data from **`slot [20]`**, like this:

```
PAR
...feed
...queue
...bleed
```

If the supply of data were temporally interupted, data already in the queue would continue to be passed along, the earlier processes waiting automatically by the very nature of the occam input process.

When the supply starts again, the early processes can proceed, but later processes anywhere in the line to be waiting even as the flow of data goes on - only if the queue is allowed to become empty of full will all the processes be waiting.

Replicated **`PAR`** exemplies the occam approach to programming and its difference from conventional languages.

Exercises

1. Write an occam fragment to display twenty asterisks on the screen!

2. Write a program fragment to perform the matrix multiplication of two-dimensional square matrices.

3. Write an occam fragment which will act as a multiplexor for a series of outputs

Subroutines

Abbreviations

An abbreviation is an occam language feature for producing a succinct alias, or new name, for an occam expression or element. Within an occam process the abbreviation is then used

instead of the original expression or element. This feature allows simplification of complicated occam statements. Essentially, the abbreviation behaves as a macro for the expression or element. In execution the effect of an abbreviation is equivalent to the substitution of the abbreviation name by the original expression or element. The usual scoping rules apply to an abbreviation. When a new name is in force due to an abbreviation, the old name of the expression or element may not be used within the scope of the abbreviation. We can remark that the constant declaration is a simple form of abbreviation. For example:

```
VAL maximum IS 1000:
```

Abbreviations are categorised as being either expression abbreviations or element abbreviations.

Expression abbreviations

This form is used to abbreviate the values of expressions. The value of an abbreviated expression must remain constant whilst the abbreviation is in scope. The simplest format is:

VAL *type name* IS e*xpression*:

where

- *type*, the type of the abbreviation, is one of the primitive data types. The inclusion of type is an optional and may be omitted if the type my be determined from the data type of the expression.
- *name* is an occam identifier of the abbreviation, and
- *expression* is a valid occam expression, such as a constant, a variable, an array component or some combination of these. The type of the expression must be the same as type, if type is specified. Any variables used in expression must not be changed by assignment or input within the scope of the abbreviation. Any array component used in the expression must have a valid subscript.

The use of the reserved word VAL underlines the constant value of this type of abbreviation.

Example 1

```
VAL Return is '*c':

VAL AbsolutZero IS –273;

VAL TwoPi IS 2.0 (REAL32) * 3.14159 (REAL32):

VAL BufferFull IS COUNT > Limit:

VAL INT16 Maximum IS 32767 (INT16):

VAL REAL32 Volume IS (Length*Breadth)*Height:

VAL REAL64 Error IS Theory – Experiment:
```

So for example **TwoPi** may be used to represent the value of the expression **2.0 (REAL32) * 3.14159 (REAL32)** and **Volume** may be used to represent the value of the expression **(Length*Breadth)*Height**, calculated with the current values of the variables **Length, Breadth** and **Height** at the time **Volume** is referenced.

Hence the assignments

```
Circumference := 2.0 (REAL32) * 3.14159 (REAL32) * Radius
```

and

```
ThisVolume := (Length*Breadth)*Height
```

may be written more succinctly as

```
Circumference := TwoPi * Radius
```

and

```
ThisVolume := Volume
```

As noted above, because of the restriction to constant values, it is important that the value of any variable used in expression is not be changed by either assignment or input within the scope of abbreviation i. e. while the abbreviation is active. So, for example, **Length, Breadth** and **Height** must remain constant within the scope of **Volume**.

Example 2

```
VAL TerminalMask IS #FF << 8:
```

or

```
VAL Shift IS 8:
VAL Mask IS #FF:
VAL TerminalMask IS Mask << Shift:
```

shows the use of abbreviations to define a further abbreviation.

Other versions of the expression abbreviation allow for an abbreviation of a table of expressions an array of values. Such a mechanism is convenient for data initialisation. The format is

VAL *[size] type name* IS *expression*:

or

VAL *[ ] type name* IS *expression*:

where the second alternative demonstrates that the actual specification of the array size is optional. Again the specification of type may be omitted if it may be determined from the data type of the expression.

Example 3

```
VAL [10] INT Primes IS [2,3,5,7,11,13,17,19,23,29]:
SEQ Index = 0 FOR 10
  Chan ! Primes [Index}
```

depicts the use of a table to initialise an array. The type may be omitted as it may be determined from the table components, so the above abbreviation may be written as

```
VAL Primes IS [2,3,5,7,11,13,17,19,23,29]:
```

The first version is useful when it is required to emphasise the size of the array.

Example 4

```
VAL [5] [2] BYTE Vowels IS [['a','A'], ['e','E'], ['i','I'], ['o','O'], ['u','U']:
```

and

```
VAL [5] [2] Vowels IS [['a','A'], ['e','E'], ['i','I'], ['o','O'], ['u','U']:
```

are equivalent initialisations of the two-dimensional array **Vowels**.

Example 5

```
VAL WhiteQueen IS ChessBoard [3] [0]:

VAL Punctuation IS ['.', ',', ':', ';', '?', '!']:

VAL SummerMonths IS {"Jun", "Jul", "Aug"]:

VAL [] INT ValidRange IS [Spectrum FROM Start FOR Length}:
```

are further examples of abbreviations using arrays - the last demonstrating the abbreviation of a segment. So it is possible to have statements of the form

```
NextPieceToMove := WhiteQueen
```

instead of

```
NextPieceToMove := ChessBoard [3] [0]
```

and

```
Acceptable := ValidRange
```

instead of

```
Acceptable := [Spectrum FROM Start FOR Length]
```

(where **Acceptable** is an array of the same type as **ValidRange**.

Example 6

```
VAL Vowels IS ['a', 'e', 'i', 'o', 'u', ]:
BYTE Char:
INT VowelCount:
SEQ
  VowelCount:= 0
  WHILE TRUE
```

```
SEQ
  InChan ? Char
  IF
    IF Index = 0 FOR 5
      Char = Vowels[Index]
        VowelCount + 1
    TRUE
      SKIP
```

represents an occam fragment to count the number of vowels appearing in a piece of text, where the channel, **InChan** is declared to be type BYTE.

Element abbreviations

This form of abbreviations used to give a new name to an element. (occam gives the generic title *element* to variables of the primitive data types, channel and timer types, and also to arrays-components, segments or whole arrays.) The format is:

type element IS *element*:

where *type* is the type of the abbreviation, and which may be omitted if the type may be determined from the type of *element*. If *type* is present, the type of the element must match the *type*. This type of abbreviation is not limited to constant values. Any change in the value of the abbreviation is reflected by a change in the value of the element being abbreviated.

For example,

```
INT Rook IS Castle:

REAL32 Average IS Results [99]:

StatusLine IS Screen {20}:

INT Element IS Array [Subscript]:
```

So for example, **Results [99]** would now be referenced by **Average**.

When the abbreviation is an alias for any array component, then certain rules must be observed within the scope of the abbreviation.

- any variable selecting i. e. indexing the array component must remain constant.

For example:

```
INT Middle:
[50] REAL32 List:

   -- initialise array

Middle := 25
Pivot IS List[Middle]: --specify Pivot as an abbreviation


  -- scope of Pivot
```

The value of **Middle** may not be changed by input or assignment within the scope of definition of **Pivot**.

- no reference must be made to *any* component of the array, except via abbreviations.

For example:

```
INT Middle:
[50] REAL32 List:

   -- initialise array

Middle := 25
Pivot IS List[Middle]:
Initial IS List[0]:

  -- process array with abbreviations Pivot and Initial
```

The array component **List[Middle]** must not be referenced as such within the scope of **Pivot** - any reference to **List[Middle]** is only valid via a reference to **Pivot**. Moreover, references to other components of **List**, for example, **List[0]**, are illegal within the scope of **Pivot**. Other components of **List** may only be referenced via other abbreviations for these components; for example, declaring the abbreviation **Initial** for **List[0]** and referencing **List[0]** via this abbreviation.

As with the previous category of abbreviation, an array version exists for this category.

The format is:

[size] type name IS *element*:

or

[ ] type name IS *element*:

Example 7

```
[30] INT WholeRange:
MiddleRange IS [WholeRange FROM 10 FOR 10]:
```

declares that **MiddleRange** is an alias for a segment of the array, **WholeRange**, and may be referenced as an array. So, for example, **MiddleRange[4]** refers to **WholeRange[4]**.

Abbreviations for efficiency

One important use of abbreviations is the production of efficient code - both in terms of memory space and execution time - for handling large arrays.

For example, rather than write

```
VAL Start IS 1000:
[5000] INT Vector:
SEQ
  Vector[Start+Offset1} := Value
```

```
  Vector[Start+Offset2} := Value
```

(where **Offset2**, **Offset2** etc. have been declared as type INT constants), it is better to write

```
VAL Lenght IS 3000:
VAL Start IS 1000:
[5000] INT Vector:
Array IS [Vector FROM Start FOR Length]
SEQ
  Vector[Start+Offset1} := Value
  Vector[Start+Offset2} := Value
```

where the large array **Vector** has been abbreviated by the segment **Array**, .

Since the large array has been abbreviated by a segment, and the segment is indexed by a constant, the occam compiler has no need to generate run-time range-checking code. All the checking may be performed at compile-time. This leads to efficiency in saving memory space and execution time.

Retyping

Retyping allows a given constant or variable of one data type to be expressed as a different data type, essentially mapping the given bit pattern to a named constant or variable of the different type. Retyping differs from the previously discussed data conversion in that retyping (as the name suggests) only changes the *type* of the given constant or variable and does not alter the bit pattern to produce an equivalent value of a different type. For example, the data conversion from an integer value to a real value involves the change from two's complement form to the IEEE floating point form. Retyping, moreover, is a specification and not an operation, as the data conversion is.

The format of retyping declaration is :

VAL *type name* RETYPE *expression*:

where *name* is the occam identifier of the constant (*expression*) or variable (*element*) being retyped.

The retyped constant or variable is governed by the usual scoping rules. Within the scope of the retyping, the name of the constant or variable being retyped may not be used.

Example 8

```
INT32 PackedNumber:
[4] BYTE SmallNumber RETYPES PackedNumber:
```

retypes the integer **PackedNumber** as a byte array **SmallNumber**. Individual bytes of the integer may then be referenced via the array. The size of the array must be such so as to correspond to the word-size of the integer being retyped.

Procedures

Procedure is a means of giving a name to an occam process, and, as such, leads to more compact, transparent and structured programs. Instead of the statements of the same process

being repeated many times within the program, the process may be defined once and then referenced many times by referring to its name.

One possible drawback of an occam procedure is that recursion is not supported. Memory allocation in occam is static - fixed at compile-time, and static allocation does not allow recursion.

An occam procedure is another example of a specification statement. Constant and variable declarations were earlier examples of specification statements.

The procedure has the format:

PROC *name* ()
  procedure body
:

i. e. an occam procedure definition consists of

- a *procedure heading* comprising the keyword PROC, *name*, the procedure identifier and a pair of matching brackets,

- followed by the *procedure body*. The procedure body must be indented by two spaces and may consist of a primitive or more complex process. Like any other process, this process may contain local declarations of constants, variables, etc. required by the procedure.

The procedure, like other specifications is terminated by a colon. However, the terminating colon of a procedure must appear on a line by itself, directly aligned with the P of PROC. The procedure is bound to the following process in the same way as a constant or variable declaration. It is governed by similar scoping rules i. e. it is before any process which references them. Procedures may be nested within other procedures.

Constants and variables, specified before the specification of a procedure but which are still in scope when the procedure is in scope, are accessible from the procedure.

Example 9

```
PROC WriteErrorMessage
  SKIP
```

specifies a trivial procedure named **WriteErrorMessage**. The body of this procedure solely comprises the primitive SKIP process. Thus , essentially this procedure behaves as a procedure stub - representing an as yet unwritten procedures are represented by procedure stubs, to be filled in later as the development processes.

Example 10

```
PROC Delay ()
  VAL Limit IS 5000:
  INT Count:
  SEQ
    Count := 0
    WHILE Count < Limit
      Count:=Count+1
```

depicts a procedure which produces a delay by (rather wastefully) looping until an incremented variable reaches a certain value. (A more efficient method of producing a delay is to use and occam timer.)

Procedures are only executed when invoked from another process. A procedure is invoked from another process by referring to the procedure's name - this is called an instance of the procedure.

The format is

name ()

where *name* is the procedure identifier of the procedure being invoked. (The procedure brackets must be present.)

Example 11

```
PROC Delay ()                    --  |
  VAL Limit IS 5000:             --  |
  INT Count:                     --  |
  SEQ                            --  |procedure specification
    Count := 0                   --  |
    WHILE Count < Limit          --  |
      Count:=Count+1             --  |


SEQ
  Delay ()           -- instance procedure
```

shows an instance of the procedure *Delay* is a following process.

Unlike procedures in some other programming languages with a branch and return mechanism, an instance of an occam procedure produces an in-line expansion of the procedure body. It behaves as a macro expansion.

For example, using the *Delay* procedure once more.

```
.
.     -- procedure specification

SEQ                              --  |
  VAL Limit IS 5000:             --  |
  INT Count:                     --  | instance of Delay
  SEQ                            --  | procedure expanded
    Count := 0                   --  | in-line
    WHILE Count < Limit          --  |
      Count:=Count+1             --  |
```

Parameters

Procedures may have parameters which allow the effect of the procedure to be applied to different values of variables, if required, each time an instance of a procedure occurs.

PROC *name (parameter 1, ... parameter n)*
procedure body
:

where *parameter 1, ... parameter n* are the formal parameters of the procedure, each separated by a comma if there is more than one parameter. The parameters may be of any occam type. This type must be completely specified within the procedure heading. Parameters of the same type may be grouped together with a single specifier of the type.

Parameters may be one or two kinds
- constant, or
- variable

If the value of a parameter remains constant i. e. unchanged within the procedure, then the type specification of the parameter in the procedure heading should be preceded by the reserved word VAL. If the value of a parameter may be changed i. e. it is a variable within the procedure, then VAL should be omitted from the parameter specification.

For example

```
PROC SkipSpaces (VAL INT Number)
```

indicates the value of the parameter. Number, remains constant within the procedure (an instance of this procedure would use an evaluated expression as a parameter) whilst

```
PROC Exchange (INT Item1, Item2)
```

indicates the values of the parameters, **Item1** and **Item2**, may be changed within the procedure (an instance of this procedure would use named variables and parameters), and

```
PROC MaxMin (VAL INT Item1, Item2, INT Max, Min)
```

indicates the values of the first two parameters, **Item1** and **Item2** remains constant, whilst the values of the second two, **Max** and **Min** are variable and may be modified within the procedure.

Example 12

```
PROC SkipSpaces (VAL INT Number)
  VAL Space IS ' ':
  BYTE Char:
  INT Skip:
  SEQ
    Skip := Number
    WHILE Skip := Number
    WHILE Skip > 0
      SEQ
        Chan ? Char
        IF
          Char = Space
            Skip := Skip - 1
          TRUE
            SKIP
```

is a procedure to skip given number of spaces, specified as the parameter value **Number**. The channel **Chan** is assumed to e declared with type BYTE. It would be preferable to pass this channel name to the procedure as a channel parameter (see example 18)

Example 13

```
PROC Exchange (INT Item1, Item2)
  SEQ
```

```
Item1, Item2 := Item2, Item1
```

represents the procedure to swap the values of its two parameters.

Example 14

```
PROC Circle (VAL REAL32 Radius REAL32 Area, Circumference)
  VAL Pi IS 3.14159 (REAL32):
  SEQ
    Area := Pi * (Radius * Radius)
    Circumference := 2.0 (REAL32) * (Pi * Radius)
```

is a procedure to calculate the area and circumference of a circle, given to radius.

Example 15

```
PROC RunningAverage (VAL REAL32 Data, REAL32 Average, INT Count)
SEQ
   Average := ((Average * (ROUND Count)) + Data) / ROUND (Count + 1))
  Count := Count + 1
```

is a procedure to calculate the running average of a list of numbers, given one number at a time.

In an instance of procedure within a process, actual parameters replace the formal parameters. Actual parameters must agree in type and number with the formal parameters. The format of a procedure instance with parameters is just an extension of a simple format

name (actual1 ... actualn)

For example

Exchange (Int1, Int2)

In execution the formal parameter behaves as an abbreviation for the value of the actual parameter; a VAL type parameter behaving as an expression abbreviation and a non-VAL type behaving as an element abbreviation.

For example,

SkipSpaces (N)

is equivalent to

```
VAL Number IS N:                --|
VAL BYTE Space IS ' ':          --|
  BYTE Char:                    --|
  INT Skip:                     --| in-line expansion
  SEQ                           --| of SkipSpaces (N)
  .                             --|
  .                             --|
  .                             --|
```

Any change in the value of a non-VAL formal parameter in the procedure body produces a corresponding change in the value of the actual parameter when used in an instance of the procedure.

For example

```
    PROC  Circle  (VAL  REAL32  Radius  REAL32  Area,
Circumference)
      VAL Pi IS 3.14159 (REAL32):
      SEQ
        Area := Pi * (Radius * Radius)
        Circumference := 2.0 (REAL32) * (Pi * Radius)
        .
        .
    SEQ
      Circle (R, A, C)        --| instance of procedure
```

is equivalent to

```
    .
      -- procedure specification
    .
    SEQ                                    --|
      VAL Radius IS R:                     --| in-line
      Area IS A:                           --| expansion
      Circumference IS C:                  --| of procedure
      VAL Pi IS 3.14159 (REAL32):          --| instance
      SEQ                                  --|
```

The actual parameters R, A and C are abbreviated to formal parameters **Radius**, **Area** and **Circumference** within the procedure will cause corresponding changes to the values of A and C.

In occam, when a variable passed as an actual parameter to a procedure, it is as if the variable replaces the formal parameter throughout the procedure. Anything that is done to the formal parameter, is done to the variable, which may therefore have its value changed.

Example:

```
    PROC decrement (INT number)
      number := number -1
    :
```

If we use **decrement (x)** as an instance of the procedure, the value of x will be reduced by one when **decrement (x)** terminates.

Take care, the behaviour differs from that found in certain other widely used languages. The commonly used call-by-value convention (available in C and Pascal) has the effect of evaluating the variable (actual) parameter, and using the result as the initial value of the formal parameter, which behaves as a local variable of the procedure body. Consequently, as assignment to the formal parameter has no effect on the actual parameter. The occam convention is more nearly equivalent to Pascal's call-by-reference (or VAR) parameters. This point is emphasised because it may trip up programmers who are experienced in these other languages.

Sometimes it is preferable that a procedure should not alter the value of a variable passed to it as a parameter.

Example:

```
PROC delay (VAL INT interval)
  INT n:
  SEQ
    n:= interval
    WHILE n> 0
      n:= n-1
```

decant the value of the parameter into a local variable (n in this example) and do any manipulations on this local value. Use this method if you need to translate Pascal procedures or similar with value parameters, into occam.

If only the original value of a variable is needed in a procedure, i. e. if the formal parameter is never altered by assignment of input, then a more efficient program may result if we explicitly say that only the value is to be passed , using VAL.

Parameters may be any occam-type. In particular, this includes channels and arrays. In specification of the procedure heading, occam does not require the size of any formal array parameter to be declared. Instead, an empty array dimension may be supplied. Used in conjunction with the SIZE operator, this shows quite general array handling procedures to be written, without having to be specific about the size of arrays catered for.

Example 16

```
PROC Initialise ([] INT Buffer)
  SEQ Index = 0 FOR SIZE Buffer
    Buffer [Index] := 0
```

is a procedure which may be called with any size of array as parameter; the use of the SIZE operator within the procedure body catering for any size of array.

Example 17

```
PROC Transpose (VAL [] [] REAL32 Matrix, {} {} REAL32
TranMatrix)
  SEQ Row = 0 FOR SIZE Matrix
    SEQ Column = 0 FOR SIZE Matrix
      TranMatrix [Row, Column] := Matrix [Column, Row]
```

represents a procedure to calculate the transpose of a two-dimensional *square* matrix.

Example 18

```
PROC Buffer (CHAN OF BYTE InChan, OutChan)
  WHILE TRUE
    BYTE Char
    SEQ
      InChan ? Char
      OutChan ! Char
```

is a simple buffer process written as procedure. The channels used by the buffer are passed as parameters. The ability to pass channel names as parameters is an important feature of occam and is frequently used in programs.

Functions

In addition to procedures the occam language definition also includes functions which are another form of process. In common with functions in other programming languages, the occam function returns a value as a result of some computation within the function. The simplest format is

```
type FUNCTION name (parameters)
  declarations
  VALOF
    function body
    RESULT expression
:
```

where

- *type* is one of the primitive types. The function returns a value of this type.
- *name* is the function identifier,
- *parameters* are the optional parameters, separated by commas. The kind of any parameter used in function must be VAL, and
- *function* body is an occam process which effects the computation of the function, and may be a primitive process or a more complex one. The result of the computation is returned via the value of *expression* (which is composed of any combination of parameters, constants, variables, etc. specified in *declarations* and literals). The expression must result in a value which has the same data type as *type*. The function body may contain further local declarations of constants, variables etc. required by the function.

The occam reserved word, VALOF, must be indented by two spaces with respect to the first letter of the specification, and the function body and the reserved word, RESULT, indented a further two spaces. Like the procedure, a function is terminated by a colon. The colon must appear on a line by itself directly underneath the first letter of the type specification.

Example 19

```
INT FUNCTION Maximum (VAL INT X, Y)
  INT Max:
    VALOF
      SEQ
        IF
          X>Y
            Max := X
          TRUE
            Max := Y
      RESULT
:
```

is a function which delivers the maximum of two integers.

Example 20

```
INT FUNCTION Factorial (VAL INT Number)
  INT Fact:
    VALOF
      SEQ
        Fact := 1
        SEQ Index = 1 FOR Number
          Fact := Index * Fact
      RESULT
:
```

depicts a function for calculating the factorial (n! = 1 * 2 * 3 * ...n) of given integer.

Example 21

```
REAL32 FUNCTION Average (VAL [] REAL32 List)
  REAL32 Sum:
    VALOF
      SEQ
        Sum = 0.0 (REAL32)
        SEQ Index = 0 FOR SIZE List
          Sum := Sum + List [Index]
      RESULT Sum / REAL32 ROUND (SIZE List)
```

represents a function which calculates the average for a list of reals.

Instance of a function behave as an in-line expansion of the function body and any formal parameters behave as expression abbreviations of the actual parameters. Also, like a procedure, a function is referenced by referring to its name. However, unlike a procedure, because the function behaves like an expression, it is referenced via an assignment statement. Thus,

variable := name (actual parameters)

For example,

```
[100] REAL32 Data
SEQ
.
.     -- input data into array
.
Mean := Average (Data)   reference function Average
```

A special and interesting feature of the occam function is that it is guaranteed not to produce any side-effects. To enable this feature, certain conditions must be observed in the use of the function

- the function body most not contain any parallel or alternation constructions
- the function body must not contain any inputs or outputs
- only variables declared within the function may be assigned to.

More sophisticated functions may be specified which return more than one value. The simple format of the function is expanded to cater for this situation.

type1, ... type n FUNCTION *name (parameters)*
 declarations
 VALOF
  function body
  RESULT *expression1, ... expression n*
:

where the list of types, separated by commas, must much in type and number the list or expression, likewise separated by commas.

For example,

```
REAL32, REAL32 FUNCTION Statistics (VAL REAL32 List)
```

would be the function heading for a function which returned the mean and standard deviation of a list of numbers.

Such a function must be referenced via a multiple assignment statement.

For example,

```
Mean, StdDev := Statistics (Data)
```

represents an instance of this multi-valued function.

Function definitions

A function definition provides a convenient notation for the specification of simple functions which are expressible as a single expression. Essentially, function definition have a null function body.

```
REAL32 FUNCTION InchesToCms (VAL REAL32 Inches)
```
IS 2.54 (REAL32) * Inches

```
REAL32 FUNCTION Area (VAL REAL32 Length, Breadth) IS
Length * Breadth

BOOL FUNCTION BufferFull (VAL INT Count) IS Count > 1024:

REAL32 FUNCTION Disc (VAL REAL32 A, B, C) IS (B*B) - (4.0
(REAL32) * (A*C):

BOOL FUNCTION IsaDigit (VAL BYTE Char) IS (Char >= '0')
AND (Char <= '9'):
```

are examples of function definitions.

Like a simple function, a function definition may be generalised to deliver multiple values.

Exercises

1. Write abbreviations for
   a) the number of degrees per radian (180 degrees = pi radians).
   b) the number of minutes in a day.
   c) the conversion of miles per hour to kilometres per hour (50 mph = 80 kph)

2. Comment the validity of the following abbreviations

a) VAL [5] BYTE Greetings IS “Hello”:

b) VAL [] BYTE Greetings IS “Hello”:

c) VAL Creetings IS “Hello”:

3. Write a procedure to reverse the digits of the three-digit number.

4. Write a set of procedure necessary for manipulating a stack, for example Push, Pop. The stack is to be implemented using an array.

5. Write a procedure to assemble characters read in to an array and record the size.

6. Write a function to find the minimum of three integers.

7. Write a function to test if a word, stored as a list of characters in an array, is palindrome or not. Assume the first space read signifies the end of the word. (A palondrome is a word which reads the same forwards as well beckwards.)

8. Write a function definition to convert from miles per houur to kilometres per hour (50 mph = 80 kmph)

9. Write a function definition to convert a Farenheit temperature to a Celsius temperature (Celsius = (Fahrenheit -32) * 5/9).

10. Write a function to test wheter a given character is vowel or not.

11. Write a multi-value function which returns the mean and standard deviation of a list of values.

Configuration

The key idea is, that the configuration does not affect the logical behavior of a program, it does enable the program to be aranged so that the performance requirements are met.

Processor

The actual processor in the network on which a set of processes are to be execute and the type of the orcessor are identified with the PROCESSOR statement.

PROCESSOR *nuber type*

where

- *number* is an integer value identifying the particular processor. This is just a logical numbering and is used to facilitate debugging, and

- *type* is the type of the processor - T2, T4 or T8 - and used to check that the program has been compiled for the correct processor:

T2	-	16-bit transputer T212
T4	-	32-bit transputer T414
T8	-	32-bit transputer T800

Allocations

Occam programs may be designed, written, tested and debugged on a single processor workstation, and then transferred to a network of parallel computers.

The final stage of such a development cycle is to allocate parallel processes in the program to different processors. This allocation is performed by replacing PAR with PLACED PAR in the appropriate parts of the program. PLACED PAR is followed by a placement, which consist of the number of a processor and a process to be run on it.

Hard channels

The four communication links of a transputer have fixed adresses in the transputer's memory space. These addresses are the first eight words of the internal memory. Conventionally, the addresses are mapped into occam identifiers using VAL abbreviation, as follows:

```
VAL link0out IS 0, link0in IS 4 :
VAL link1out IS 1, link1in IS 5 :
VAL link2out IS 2, link2in IS 6 :
VAL link3out IS 3, link3in IS 7 :
```

Each physical link will support two occam channels - one input channel and one output channel. The *hard* channels, which provide iner-processor communication or communication with external devices, muts be mapped onto he transputer links. *Soft* channels, on the other hand, are implemented via memory locations since the processes using the such channels for interprocess communication reside on the same transputer. Any link may be choosen to act as a channel to a communicating processor provided it is of correct type, namely input or output. A link on one transputer need not be connected to a correspondingly named link on another transputer. Thus, for example, **`link1out`** on one transputer may be connected to **`link1in`** on another transputer, but may equally well be connected to **`link0in, link2in, link3in`**. The corresponding physical links must be connected together between transputers as per the configuration.

Each collection of occam processes, which will execute on a single transputer in the network, must be grouped together to form an outher procedure with the additon of a procedure heading and terminator. The parameters for this outher procedure are necessary hard channels for communicating with the outer procedures on other transputers. It is an Inmos characteristic, that each outher procedure must be separately compiled.

Each hard channel between two processes on different transputers must be specified in two separate PLACE statements; once as an input channel of one transputer and once as an output channel for the other transputer. The order of declarations must follow the general format:

```
global declarations
PLACED PAR
  PROCESSOR
    local declarations
    procedure instance
```

Hardware protocols

Some hardware devices will require various protocols to be adhered to, that is certain special commands and non-data characters may need to be sent (and received) to control the devices, in addition to the actual data.

These are very hardware dependent and cannot be covered in any detail here.

Ports

In addition to hard channels, occam can address I/O ports as used in conventional computer systems. A port declaration has a data type, e. g.

```
PORT OF BYTE seriall:
```

and the allowed processes are input and output only:

```
serial1 ! 'a'
serial2 ? x
```

This allows ports to be used like channels, rather than like variables, which is more in keeping with the occcam style of using channels for all communication and variables for storage.

Ports behave like occam channels in that only one process may input from a port, and only one process may output to a port. Thus ports provide a secure method of accessing external memory mapped status registers etc.

Note that there is noo synchronisation mechanism assotiated with port input and output. Any timing constraints which result from the use of asynchronous external hardware will have to be programmed explicitly. For example, a value read by a port inut may depend uppon the time at which the input was executed, and inputting at an invalid time would produuce unusable data.

During applications development it is recomended that the peripherial is modelled by an occam process connected via channel.

Practical works

Unit 1: Basics and Inmos Toolset

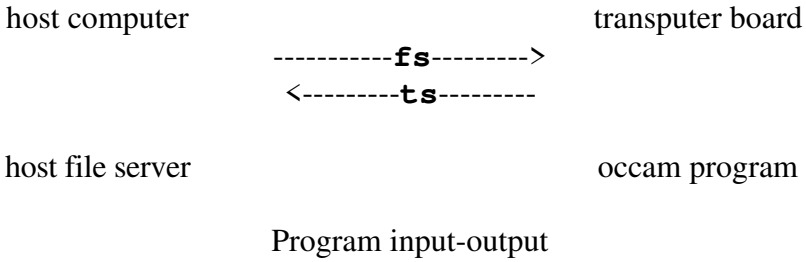
Theory	Practical tasks
Basics of occam and Inmos Toolset	Programs Simple, Hello world

1. Type in the program simple using any text editor.

2. Compile, link, create a bootable file and finally run it.

3. Make the preparing works using imakef.

4. Analyse the program on base of next additional information:



so.write.string

```
PROC so.write.string (CHAN OF SP fs, ts,
                      VAL []BYTE string)
```

Writes the string **string** to the screen.

so.read.echo.line

```
PROC so.read.echo.line (CHAN OF SP fs, ts,
                        INT len, []BYTE line
                        BYTE result)
```

Reads a line of text from the keyboard with echo on the screen. The characters read are in the first **len** bytes of **line**. The line is read until 'RETURN' is pressed at the keyboard. The line is truncated if **line** is not large enough. A newline or carriage return is not included in line.

The result returned can take any of the following values:

spr.ok **>=spr.operation.failed**	The read was successful. If **result** takes a value **>=spr.operation.failed** then this denotes a server returned failure.

so.write.nl

```
PROC so.write.nl (CHAN OF SP fs, ts)
```

Writes a newline sequence on the screen.

so.write.string.nl

```
PROC so.write.string.nl (CHAN OF SP fs, ts
                         VAL []BYTE string)
```

As **so.write.string** but appends a newline sequence to the end of the string.

so.exit

```
PROC so.exit (CHAN OF SP fs, ts
              VAL INT32 status)
```

Terminates the serve, which returns the value of **status** to its caller. If **status** has the special value **sps.success** then the server will terminate with a host specific 'success' result. If **status** has the special value **sps.failure** then the server will terminate with a host specific 'failure' result.

1.5. Modify the program simple giving values to the variables and appearing this values on the screen.

1.6. Solving the exercises of chapter Basics of part occam.

Unit 2 : Simple programs

Primitive processes, arithmetic, channel communication	Writing simple programs to receive practises in using the occam and the Inmos Toolset

Logic gates

Write a set of occam programs which will simulate various logic gates, for example AND, OR and NOR.

```
-- logic.top
-- an AND gate is
CHAN OF BOOL InPin1, InPin2, OutPin
BOOL Level1, Level2
SEQ
  PAR
    InPin1 ? Level1
    InPin2 ? Level2
  OutPin ! Level1 AND Level2
```

Parallel transfer of an array

Write an occam fragment for transferring a [25] REAL32 array between two component prcesses of aPAR process.

```
-- partrans.top
CHAN OF [25] REAL32 Chan:
PAR
  [25] REAL32 X:
  SEQ
    Chan ! X
  [25] REAL32 Y:
  SEQ
    Chan ! Y
```

Unit 3: Pipeline (a sequential program)

Pipeline	introduction replicated SEQ, VAL, REAL

Sequential program for Newton-Raphson method

We use the Newton-Raphson estimate technique for evaluating square roots. This method starts off with the value of number whose square-root is required an initial estimate of square-root. The Newton-Raphson formula is then applied in an iterative manner, each time producing a better estimate of the square-root from the previous value. If the iteration is performed a large number times, the final estimate will be close approximation to the real value.

The Newton-Raphson formula for calculating the square-root of a number x is

$$y_{i+1} = 1/2(y_i + x/y_i)$$

where y_i and y_{i+1} are successive estimations of the root y.

This formula may be expressed in occam as

```
est := ((est + (x/est)) * 0.5 (REAL32))
```

where x is the number whose square-root is required and the initial value for est is given. Usually, the initial value for est is taken to be

```
est := x * 0.5 (REAL32)
```

Codes of program

pipeseq.tsr

```
  # USE userio

  --{{{  comments on program
  -- A Newton-Raphson program for extracting the square roots
of
  -- the first R integers. No paralellism involved
  -- N is the number of iterations to be used
  --
  -- A pseudocode equivalent of program is:
  --
  --R=5, N=10
  --FOR r = 1 TO R

      x:= r, est:= 0.5x     -- feed in integers

      FOR n = 1 to N        -- main iterations
        est:= (est + x/est)/2

      Write x and est       -- print square roots
```

```
--}}}

--{{{  VALs etc
VAL Half IS 0.5 (REAL32):
VAL R IS 5:
VAL N IS 10:
--}}}

SEQ

  --{{{  running
  write.full.string (screen, "Started running --")
  newline (screen)
  --}}}

  --{{{  main program
  SEQ R = 1 FOR R
    REAL32 x, est
    SEQ
      --{{{  feed in integers
      x := (REAL32 ROUND r)
      est := x * Half
      --}}}
      --{{{  main iterations
      SEQ
        est := ((est + (x/est)) * Half)
      --}}}
      --{{{  print square roots
      --{{{  VALs
      VAL Ip IS 5:
      VAL Dp IS 3:
      --}}}
      SEQ
        write.full.string(screen, "x = ")
        write.real32 (screen, x, Ip, Dp)
        write.full.string (screen, "sqrt(x) = "
        write.real32 (screen, est, Ip, Dp)
        newline (screen)
      --}}}

  --}}}

  --{{{  finished and pause
  INT Any:
  SEQ
    write.full.string (screen, "Finished running --")
    keyboard ? Any
  --}}}
```

Benchmark timings

Run the program for the first 20000 integers, for example. You can use stopwatch to time it now to keep things simple.

Unit 4: Pipeline (a parallel program)

Pipeline	Introducing of PAR, replicated PAR, PROTOCOL

Parallel program for Newton-Raphson method

The Newton-Raphson sequential program can be rewritten using several parallel processes, which can be run on the host. There can be no speed-up, and there is a slow down because of the overhead involved in implementing the parallelism on one transputer. It is, in fact, easier to develop a program destined for a network of transputers on the host, since it avoids much time-consuming effort with the network at the debugging stage. Conceptionally, having a written program for a single machine, it is an “easy task” to convert the program into one suitable for running on the network. However, this “easy task” is not always as straight forward as often is claimed.

Unless at development stage, one keeps quite clear what the eventual configuration will be, and writes to program accordingly, then there may have to be a fairly substantial rewrite of the program to transfer it to the network, so that it will run efficiently.

Codes of program

```
# USE userio

{{{  comments on program
-- A Newton-Raphson program for extracting the square roots
-- of the first integers.
-- Parallel program for host transputer

-- N is the number of interations to be used
}}}

{{{  VALs etc
VAL Half IS 0.5 (REAL32):
VAL R IS 5:
VAL N IS 10:
}}}

{{{  CHANs and PROTOCOL
PROTOCOL x.est IS REAL32; REAL32:
[N+1] CHAN OF x.est Pipe:
}}}

SEQ

{{{  running
write.full.string (screen, "Started running--")
newline (screen)
}}}
```

```
{{{  main program
PAR

  {{{  feed in integers to pipeline
  SEQ r = 1 FOR R
    REAL32 x, est:
    SEQ
      x:=(REAL32 ROUND r)
      est := x * Half
      Pipe[0] ! x; est
  }}}

  {{{  main pipeline
  PAR n = 1 FOR N
    Input IS Pipe[n-1]:
    Output IS Pipe[n]:

    {{{  nth iteration
    SEQ r = 1 FOR R
      REAL32 x, est

      SEQ
        Input ? x; est
        est := ((est + (x/est)) * Half)
        Output ! x; est
    }}}

  }}}

  {{{  bleed square roots
  SEQ x = 1 FOR R
    REAL32 x, est
    SEQ
      Pipe[N] ? x; est

      {{{  print square roots
      {{{  VALs
      VAL Ip IS 5:
      VAL Dp IS 3:
      }}}

      SEQ
        write.full.string (screen, "x=")
        write.real32 (scrreen, x, Ip, Dp)
        write.full.string (screen, est, Ip, Dp)
        write.real32 (screen, est, Ip, Dp)
        newline (screen)

      }}}

  }}}

}}}

{{{  finished and pause
```

```
INT Any:
SEQ
  write.full.string (screen, "Finished running --"
  keyboard ? Any
}}}
```

Benchmark timings

Once the program is working, change R to 20000 and again time the program.

Unit 5: A simple network program

Processes for minimal program to demonstrate hoe to use a network	Use of procedures (PROC), IF and ALT constructs

Message passing between two processes

Writing a minimal program, which demonstrates, how to use a network. The keyboard and screen handler run on the host and communicate with application which runs on an other transputer. All key strokes are sent to application by the keyboard handler. The application watches out for a special character (the digit '2'), forwards it to the screen handler, which then displays the massage "Hello! From transputer T4!" All other characters are ignored, with the exception of a special stop character ("@"). Every key pressed is also sent directly to the screen handler for echoing on the screen.

Codes of program

```
--{{{  comments on program
-- Accepts characters from the keyboard, and shoes to screen.
-- Also sends it on to application, which watches out for
-- the ch. '2', and then send this on to the screen handler.
-- This version is for the host alone.
--}}}

#USE userio

--{{{  CHANs
CHAN OF INT To, App, From, echo:
--}}}

--{{{  PROC Keyboard.handler
PROC Keyoard.handler (CHAN OF INT keyboard, CHAN OF INT To,
App, echo)

-- Accepts character input from keyboard and sends it on
-- (i) to echo to screen and (ii) to the application.

--{{{  Declarations
INT ch:
VAL stopch IS INT '@':
BOOL running:
--}}}

SEQ
  running := TRUE

  WHILE running
    SEQ
      keyboard ? ch
```

```
      PAR
        echo ! ch
        To.App ! ch

      --{{{  finished?
      IF
        ch = stopch
          running := FALSE
        ch <> stopch
          SKIP
      --}}}
:
--}}}
--{{{  PROC Screen.handler
PROC Screen.handler (CHAN OF INT From.App, echo, CHAN OF ANY
screen)
  -- Accepts input from either application or echo
  -- and sends to screen

  --{{{  Declarations
  INT ch:
  BOOL running:

  --{{{  VALs
  VAL stopch   IS   INT  '0':
  VAL app.row  IS   10:       -- position for application
  VAL app.col  IS   5:        -- printing
  VAL echo.row IS   5:        -- position for echo printing
  VAL echo.col IS   5:
  --}}}

  --}}}

  SEQ
    running := TRUE

    WHILE running
      ALT

        --{{{  input from echo
        echo ? ch
          SEQ

            --{{{  clear previous character
            goto.xy (screen, echo.col, echo.row)
            write.full.string (screen, "       ")
            -- az utolso sor nem write.char?
            --}}}
            --{{{  print new character
                goto.xy (screen, echo.col, echo.row)
                write.char (screen, BYTE ch)
            --}}}

            --{{{  finished?
```

```
            IF
              ch = stopch
                running := FALSE
              ch <> stopch
                SKIP
            --}}}
        --}}}

        --{{{  input from application
        From.App ? ch
          SEQ
            --{{{  clear previous message
                goto.xy (screen, app.col, app.row)
                write.full.string (screen, "
")
            --}}}
            --{{{  print new message
                    goto.xy (screen, app.col, app.row)
                    write.char (screen, BYTE ch)
                    write.full.string (screen, "Hello! From
Application!")

            --}}}
        --}}}
:
--}}}
--{{{  PROC Application
PROC Application (CHAN OF INT To.App, From.Ap)

  -- Watches for the ch. '2' from keyboard handler and sends
  -- it on to screen handler. All other chars,
  -- (except stopch) are ignored.

  --{{{  Declarations
  BOOL running:
  INT ch:
  VAL To.App.ch     IS   INT  '2':
  VAL stopch   IS   INT  '@':
  --}}}

  SEQ
    running := TRUE
      SEQ

        To.App ? ch

        --{{{  if a '2' send on to screen handler
        IF
          ch = To.App.ch
            From.App ! ch
          ch <> To.App.ch
            SKIP
        --}}}
        --{{{  if a 'stopch' finish application
        IF
```

```
          ch = stopch
            running := FALSE
          ch <> stopch
            SKIP
        --}}}
:
--}}}

PAR
  Keyboard.handler        (keyboard, To.App, echo)
  Screen.handler          (From.App, echo. screen)
  Application             (To.App, From.App)
```

Unit 6: Transforming the simple network program to run on a network (of one transputer)

Configuration, multi-transputer systems	How to transform it? Hard channel allocations

Message passing between two transputers

We have to make several changes on the program of the UNIT before to run on the host and one other transputer.

Codes of program

```
--{{{  EXE key and screen

--{{{  comments on program SIMPNET2
-- Accepts characters from the keyboard, and shoes to screen.
-- Also sends it on to application, which watches out for
-- the ch. '2', and then send this on to the screen handler.
-- This version is for the host and one other transputer on
-- the network
--}}}


#USE userio

--{{{  hard channel placement values
VAL link0out IS 0:
VAL link1out IS 1:
VAL link2out IS 2:
VAL link3out IS 3:
VAL link0in  IS 4:
VAL link1in  IS 5:
VAL link2in  IS 6:
VAL link3in  IS 7:
--}}}
--{{{  CHANs
CHAN OF INT To, App, From, echo:
--}}}
--{{{  PLACEMENTs
PLACE To.App   AT link2out:
PLACE From.App AT link2in:
--}}}

--{{{  PROC Keyboard.handler
PROC Keyoard.handler (CHAN OF INT keyboard, CHAN OF INT To,
App, echo)

-- Accepts character input from keyboard and sends it on
```

```
-- (i) to echo to screen and (ii) to the application.

--{{{  Declarations
INT ch:
VAL stopch IS INT '@':
BOOL running:
--}}}

SEQ
  running := TRUE

  WHILE running
    SEQ
      keyboard ? ch

      PAR
        echo ! ch
        To.App ! ch

      --{{{  finished?
      IF
        ch = stopch
          running := FALSE
        ch <> stopch
          SKIP
      --}}}
:
--}}}
--{{{  PROC Screen.handler
PROC Screen.handler (CHAN OF INT From.App, echo, CHAN OF ANY
screen)
  -- Accepts input from either application or echo
  -- and sends to screen

  --{{{  Declarations
  INT ch:
  BOOL running:

  --{{{  VALs
  VAL stopch   IS   INT  '0':
  VAL app.row  IS   10:       -- position for application
  VAL app.col  IS   5:        -- printing
  VAL echo.row IS   5:        -- position for echo printing
  VAL echo.col IS   5:
  --}}}

  --}}}

  SEQ
    running := TRUE

    WHILE running
      ALT

        --{{{  input from echo
```

```
        echo ? ch
          SEQ

            --{{{  clear previous character
            goto.xy (screen, echo.col, echo.row)
            write.full.string (screen, "       ")
            -- az utolso sor nem write.char?
            --}}}
            --{{{  print new character
                goto.xy (screen, echo.col, echo.row)
                write.char (screen, BYTE ch)
            --}}}

            --{{{  finished?
            IF
              ch = stopch
                running := FALSE
              ch <> stopch
                SKIP
            --}}}
        --}}}

        --{{{  input from application
        From.App ? ch
          SEQ
            --{{{  clear previous message
                goto.xy (screen, app.col, app.row)
                write.full.string (screen, "
")
            --}}}
            --{{{  print new message
                     goto.xy (screen, app.col, app.row)
                     write.char (screen, BYTE ch)
                     write.full.string (screen, "Hello! From
Application!")

            --}}}
        --}}}
:
--}}}

PAR
  Keyboard.handler       (keyboard, To.App, echo)
  Screen.handler         (From.App, echo. screen)
--}}}
--{{{  PROGRAM Application

--{{{  SC PROC Application
PROC Application ()
  --{{{  Comments on Proc

  -- Watches for the ch. '2' from keyboard handler and sends
  -- it on to screen handler. All other chars,
  -- (except stopch) are ignored.
  --}}}
```

```
  --{{{  Declarations
  --{{{  VARs
  BOOL running:
  INT ch:
  --}}}
  --{{{  VALs
  VAL To.App.ch     IS   INT  '2':
  VAL stopch   IS   INT  '@':
  --}}}
  --{{{  hard channel placement values
  VAL link0out IS 0:
  VAL link1out IS 1:
  VAL link2out IS 2:
  VAL link3out IS 3:
  VAL link0in  IS 4:
  VAL link1in  IS 5:
  VAL link2in  IS 6:
  VAL link3in  IS 7:
  --}}}
  --{{{  PLACEMENTs
  PLACE To.App AT link0in:
  PLACE From.Appp   AT link0out:
  --}}}

  --}}}

  SEQ
    running := TRUE
    WHILE running
      SEQ

        To.App ? ch

        --{{{  if a '2' send on to screen handler
        IF
          ch = To.App.ch
            From.App ! ch
          ch <> To.App.ch
            SKIP
        --}}}
        --{{{  if a 'stopch' finish application
        IF
          ch = stopch
            running := FALSE
          ch <> stopch
            SKIP
        --}}}
:
--}}}

PLACED PAR
  PROCESSOR T4
    Application ()

--}}}
```

Unit 7: Pipeline (a parallel multitransputer program)

Pipeline	Introducing of PAR, replicated PAR, PROTOCOL

Parallel multitransputer program for Newton-Raphson method

In that case we turn to writing the Newton-Raphson program on a network of several computers (I. e. more than one in the network).

We arrange for the total number of iterations N to be shared as equally as possible amongst the M transputers in the network (excluding the host).

Codes of program

```
--{{{  EXE feedmn and bleed
# USE userio
--{{{  comments on program
-- A Newton-Raphson program for extracting the square roots
-- of the first integers.

-- Program starts up and then feeds the pipeline running
-- and also bleeds the pipeline for displaying results if required


-- Parallel program for host transputer + other T4 transputer

-- N is the number of interations to be used
-- R is the number of square root to be extracted
--}}}
PROC feed.bleed ()
  --{{{  hard channel placement values
  VAL link0out          IS 0:
  VAL link1out          IS 1:
  VAL link2out          IS 2:
  VAL link3out          IS 3:
  VAL link0in  IS 4:
  VAL link1in  IS 5:
  VAL link2in  IS 6:
  VAL link3in  IS 7:
  --}}}
  --{{{  CHANs and PROTOCOL
  PROTOCOL x.est IS REAL32; REAL32:
  CHAN OF x.est Pipe0, PipeN:
  --}}}
  --{{{  PLACEMENTs
  PLACE Pipe0 AT link2out:
  PLACE PipeN AT link3out:
  --}}}
```

```
  --{{{  VALs etc
  VAL Half IS 0.5 (REAL32):
  VAL R IS 5:
  --}}}
  --{{{  main feed and bleed program
  PAR

    --{{{  feed in integers to pipeline
    SEQ r = 1 FOR R
      REAL32 x, est:
      SEQ
        x:=(REAL32 ROUND r)
        est := x * Half
        Pipe[0] ! x; est
    --}}}

    --{{{  bleed square roots
    SEQ x = 1 FOR R
      REAL32 x, est
      SEQ
        Pipe[N] ? x; est

        --{{{  print square roots
        --{{{  VALs
        VAL Ip IS 5:
        VAL Dp IS 3:
        --}}}

        SEQ
          write.full.string (screen, "x=")
          write.real32 (scrreen, x, Ip, Dp)
          write.full.string (screen, est, Ip, Dp)
          write.real32 (screen, est, Ip, Dp)
          newline (screen)

        --}}}

    --}}}

  --}}}
:

--{{{  running
INT Any:
SEQ
  write.full.string (screen, "Started running--")
  newline (screen)
  keyboard ? Any
--}}}
--{{{  feed and bleed
  feed.bleed ()
--}}}
--{{{  finished and pause
  write.full.string (screen, "Finished running --"
```

```
  keyboard ? Any
--}}}

--}}}
--{{{  PROGRAM parmn
--{{{  comments on program
-- A Newton-Raphson program for extracting the square roots
-- of the first integers

-- Parallel program for host transputer + other T4 transputer

-- N is the number of interations to be used
-- R is the number of square root to be extracted
--}}}

PROTOCOL x.est IS REAL32; REAL32:

--{{{  SC Iterations
--{{{  Iterations
PROTOCOL x.est IS REAL32; REAL32;

PROC Iterations (CHAN OF x.est Input, Output, VAL INT m, M)

  --{{{  main pipeline

  --{{{  VALs etc
  VAL Half IS 0.5 (REAL32):
  VAL R IS 5:                -- No of Integers
  VAL N IS 10:                -- No of iterations
  --}}}
  --{{{  number of iterations on each transputer

  -- The total number of iterations is shared as
  -- equally as possible amongst the M transputers
  -- (ie Nm = N DIV M on each).
  -- However, there will be (N REM M) iterations left
  -- over; one iteration is given to each of the first
  -- (N REM M) transputers
  Int Nm, N.Its
  SEQ
    Nm := N/M
    IF
      --{{{  first N REM M transputers
      m <= (N REM M)
        N.Its := Nm + 1
      --}}}
      --{{{  the others
      TRUE
        N.Its := Nm
      --}}}
  --}}}

    SEQ r = 1 FOR R
      REAL32 x, est:
```

```
    SEQ
      Input ? x; est

      SEQ itno = 1 FOR N.Its
      est := ((est + (x/est)) * Half)
      Output ! x; est
 --}}}

--}}}

--}}}

--{{{  hard channel placement values
VAL link0out  IS 0:
VAL link1out  IS 1:
VAL link2out  IS 2:
VAL link3out  IS 3:
VAL link0in   IS 4:
VAL link1in   IS 5:
VAL link2in   IS 6:
VAL link3in   IS 7:
--}}}
--{{{  VALs etc
VAL M IS 3    -- Number of transputers in pipeline
--}}}
--{{{  CHANs and PROTOCOL
[M+1] CHAN OF x.est Pipe:
--}}}


PLACED PAR m = 1 FOR M
  PROCESSOR m T4
  PLACE Pipe[m-1] AT link0in:
  PLACE Pipe[m] AT link1out:
  Iterations (Pipe[m-1], Pipe[m], m, M)

--}}}
```

Speed, speed up and efficiency

We can now demonstrate the speed of a network of transputers compared an IBM PC and also investigate how speed-up and efficiency depend on the number of computers on the network.

Appendix

The Inmos Occam 2 Toolset

About Inmos

Inmos is a leader in the development and design of high-performance integrated circuits and is a pioneer in the field of parallel processing. As developers of the transputer and the occam parallel processing language, Inmos has established standards for the parallel processing.

The corporate headquarters, product design team and world-wide sales are based at Bristol, UK. Inmos products are manufactured at the Inmos Newport, Duffryn facility which began operations in 1983. This is an 8000 square metre building with 3000 square metre cleanroom operating to Class 10 environment in work areas. Subcontractors in Korea, Taiwan, Hong Kong and the UK are used to assemble devices.

Getting started

That lesson shows you how to compile, link and run a simple example program on a single processor. The tutorial assumes that you have a boot from link board containing T425 processor.

Writing a simple program

The example program simple.occ reads a name from the keyboard and displays a greeting on the screen. The source of the program can be written using any text editor. The program uses the library hostio.lib and incorporates the include file hostio.inc.

The program is illustrated bellow:

```
#INCLUDE "hostio.inc" --  -- contains SP protocol
PROC simple (CHAN OF SP fs, ts, []INT memory)
  #USE "hostio.lib"
  []BYTE buffer RETYPES memory:
  BYTE result:
  INT length:
  SEQ
    so.write.string      (fs, ts,
                 "Please type your name :")
    so.read.echo.line    (fs, ts, length, buffer,
                 result)
    so.write.nl          (fs, ts)
    so.write.string      (fs, ts, "Hello ")
    so.write.string.nl   (fs, ts,
                 [buffer FROM 0 FOR length])
    so.exit              (fs, ts, sps.success)
:
```

The first line in the program loads the file hostio.inc. This file contains the definition of protocol SP, used to communicate with the host file server, and a number of constants that are used in conjunction with the host i/o library.

The procedure simple is then declared. All the working code is contained within this procedure. Single processor program must always use a similar parameter list, to enable the communication with the host. Program input and program output is supported by the host file server, which is resident on the host computer. Access to the host file server is via the i/o libraries. Whenever routines from these libraries are used the channel **fs** and **ts** (from server and to server) must be passed to the routine so that it can communicate with the host file server. (Fig. 4.1.)

The vector **memory** contains the free memory remaining on the transputer evaluation board after the program code has been loaded and the workspace allocated. It is calculated by subtracting the area occupied by the program code and workspace from the value specified in the IBORDSIZE host environment variable. The **memory** vector is passed to the program as a vector of type int, where it can be used. By allowing programs to be run on boards with different memory sizes, this vector aids program portability between different boards.

The server library hostio.lib is referenced by the #USE directive. This library contains all the procedures used by the program.

Before the body of the procedure a number of variables are declared. First, the memory array is retyped as byte array. This enables the program to use the free memory on the board as a character buffer.

The variables **length** and **result** are then declared for use by the program. The variable **length** refers to the number of characters in the name read from the keyboard, and **result** is used by the library routine to indicate whether or not the read was successful. The result is ignored by this example for the sake of simplicity; it is assumed that the screen writes and keyboard reads always succeed.

The working code is contained within a SEQ, indicating that the statements which follow are to be executed sequentially. All of the statements are calls to library routines in hostio.lib. The code prompts for a name, reads the name from the keyboard, and types a greeting on the screen.

The last statement calls a library procedure which terminates the server, returning control to the host operating system. Without this statement the program would finish and appear to hang, and the server would have to be terminated explicitly by interrupting the program.

Compiling a simple program

In order to compile the program in its simplest form i.e. with all defaults enabled the following command line should be used:

```
oc simple
```

Because the file has the default extension of .occ you can omit it when invoking the compiler. The compiler will create a file called **simple.tco**, containing the code compiled for T4xx HALT mode. The compiler will perform the necessary syntax , alias and change usage checks and will insert code to perform run-time error checking. By default the compiler enables interactive debugging with idebug.

Linking a simple program

To use the result of your compilation it must be linked with libraries that it uses. To link the program type:

```
ilink simple.tco hostio.lib -f occama.lnk          (UNIX)
ilink simple.tco hostio.lib /f occama.lnk          (MS-DOS)
```

The linked program will be written to the file **simple.lku**. As no output file is specified, the file is named after the input file and the default link extension **.lku** is added. The library **hostio.lib** is the server library used by this program.

The 'f' option specifies a linker indirect file containing commands and directives to **ilink**. The file **occama.lnk** is the correct file to use for T4 series transputers. It identify various libraries including compiler libraries which are required to be linked with the program. These files are provided as a short-hand method of specifying such libraries to the linker.

Creating a bootable file

Before the program can be run it must be made bootable. This involves adding bootstrap information to make the program loadable and is achieved using the collector tool **icollect**. One of the following commands should be used depending on the host in use.

```
icollect simple.lku -t                             (UNIX)
icollect simple.lku /t                             (MS-DOS)
```

By the default **icollect** expects the input file to have been produced by the configurer. Because the example program is going to run on a single processor there is no need to configure it. The 't' option instructs the collector to build a bootable file from the linked unit. The bootable program will be written to the file **simple.btl**.

icollect will also create a configuration binary file as a by-product of creating the bootstrap. Configuration binary files describe the network configuration, in this case the single transputer. This file will have the extension **.cfb** and is created by **icollect** for use by the

debugger. For multitransputer programs the configurer is used to create configuration binary files.

Running a simple program

To run the program it must be loaded onto a transputer board using the host file server tool **iserver**. To load and run the program use one of the following commands :

```
iserver -se -sb simple.btl          (UNIX)
iserver /se /sb simple.btl          (MS-DOS)
```

The 'sb' option specifies the file to be booted and loads the program onto the transputer board.. It has the effect of resetting the board, opening communication with the host, and loading the program onto network. The 'se' option directs the server to terminate if the program sets the error flag.

Using the simulator

If you are using the simulator to run the example program use one of the following commands:

```
isim -bq simple.btl                 (UNIX)
isim /bq simple.btl                 (UNIX)
```

The 'bq' option specifies batch quiet mode which causes the simulator to run the program and then terminate.

Using **imakef**

As an alternative method of program development the toolset Makefile generator **imakef** can be used. This tool can produce a Makefile for any type of file than can be built the toolset tools. **imakef** serves two purposes:

- It enables the user to generate a target file automatically (e. g. a bootable file) without having manually perform the intermediate stages of program development i. e. compiling, linking, configuring etc
- For more complex programs, comprising several modules, it simplifies the incorporation of changes to the program by identifying dependencies and incorporating them into the Makefile.

In order for **imakef** to be able to identify file types, a different system of file extensions must be used to that used in the examples above.

To create a Makefile for the example program, use the following command:

```
imakef simple.b4h
```

The **.b4h** extension informs **imakef** that we wish to build a bootable program for T4xx in the default HALT error mode. **imakef** will create a Makefile called **simple.mak** containing full instructions on how to build the program.

To build the program run the **make** program on **simple.mak**. The entire program will be automatically compiled, linked and made bootable, ready for loading onto the transputer.

```
make -f simple.mak                    (UNIX)
make /f simple.mak                    (MS-DOS)
```

To run the program:

```
iserver -se -sb simple.b4h            (UNIX)
iserver /se /sb simple.b4h            (MS-DOS)
```

If you are using the simulator to run the example program use one of the following commands:

```
isim -bq simple.b4h                   (UNIX)
isim -bq simple.b4h                   (UNIX)
```

The occam libraries

Operations with libraries

The occam libraries include:
- the compiler libraries which are automatically referenced by the compiler; and
- a number of user libraries to assist with common programming tasks.

The user libraries provide standard mathematical functions, host i/o and file management procedures, file stream i/o support and many other functions. A full list of all the toolset libraries with brief descriptions of their contents can be found in the next table:

Compiler Libraries	Multiple length integer arithmetic Floating point functions 32 bit IEEE arithmetic functions 64 bit IEEE arithmetic functions 2D block move library Bit manipulation and CRC library Arithmetic instruction library

snglmath.lib	Single length mathematical functions
dblmath.lib	Double length mathematical functions
tbmaths.lib	T4 series optimised math library
hostio.lib	Host file server library
streamio.lib	Stream i/o library
string.lib	String library
convert.lib	Type conversion library
crc.lib	Block CRC library
xlink.lib	Extraordinary link library
debug.link	Debugging support library
callc.link	Mixed languages support library
msdos.link	DOS specific hostio library

Using the occam libraries

If a library routine is used in a program then the libraries must be declared with the #USE directive and the declaration must be in scope, where the routine is used. The scope of a library, as with all occam declarations, is determined by its level of indentation in the occam text. An example showing how to declare a library given below:

```
#USE "hostio.lib"
```

Linking libraries

All libraries used by the program or program module must also be linked with the main program. This includes the compiler libraries even though they are automatically referenced when the program is compiled.

Listing library contents

You can used the ilist tool to examine the contents of a library and determine which routines are available. The tool displays procedural interfaces for routines in each library module and the code size and work space requirements for individual modules. It can also be used to determine the transputer types and error modes for which the code was compiled. For example:

```
ilist hostio.lib -a                    (UNIX)
ilist hostio.lib /a                    (MS-DOS)
```

Toolset constants

Constants and protocols used by the toolset libraries are defined in six include files which are supplied with the toolset. Two of six provide constants and definitions for the hostio and streamio libraries, a third provides mathematical and trigonometric constants, the fourth contains the absolute transputer link addresses, the fifth contains the rates at which the two transputer clocks increment on the transputer and the sixth provides constants to support the DOS specific library routines. All files containing constant definitions must be declared in the program before the library references them. Files of constants:

File	Description

hostio.inc	Constants for the host file server interface
streamio.inc	Constants for the stream i/o interface
mathvals.inc	Maths constants
linkaddr.inc	Addresses of transputer links
ticks.inc	Rates of the two transputer clocks
msdos.inc	DOS specific constants

Compiler libraries

The compiler libraries are used internally by code generated by the compiler. They are found automatically by the compiler on the path specified by the ISEARCH host environment variable and do not need to be referenced by the #USE directive. Separate libraries are supplied for the different processor-categories, for example for the T4 family the occama.lib.

File names of the compiler libraries must not be changed.

User Libraries

Math libraries

The single and double length libraries contain the same set of functions in single and double length forms. By convention the double length forms begin with the letter "D". Function names are upper case.

The TB optimised library is a combined single and double length library containing all the single and double length functions optimised for the T400, T414, T425 processors. The standard maths libraries can also be used by T400, T414, T425 processors, but optimum performance on these processors can be achieved by using the optimised functions. The accuracy of the T400/T414/T425 optimised functions is similar to that of the standard single length functions, but results returned may not be identical because different algorithms are used. Functions in the optimised library have the same names as the equivalent functions in the single and double length libraries. This means that the optimised library cannot be used together with either the signal or double length library on the same processor.

Host file server library

The host file server library contains routines that are used to communicate with the host file server. The routines are independent of the host on which the server is running. Using routines from this library you can guarantee that programs will be portable across all implementations of the toolset.

The result value from many of the routines in this library can take the value >= **spr.operation.failed** which is a server dependent failure result.

In the procedure descriptions fs is the channel from the host file server, and ts is the channel to the host file server. The SP (server protocol) protocol used by the host file server channels is defined in the include file **hostio.inc**. The hostio routines are divided into six groups: five groups reflect function and use, and the sixth miscellaneous group. The five specific groups are:

- file access and management;
- general host access;
- keyboard input;
- screen output;
- file output.

Stream i/o library

The streamio library contains routines for reading and writing to files and to the terminal at a higher level of abstraction than the hostio library. The file **`streamio.inc`** defines the KS and SS protocols and constants used by the streamio library routines.

The result value from many of the routines in this library can take the value >= **`spr.operation.failed`** which is a server dependent failure result.

The streamio routines can be classified into three main groups:
- stream processes;
  stream input procedures;
  stream output procedures.

String handling library

This library contains functions and procedures for handling strings and scanning lines of texts. They assist with the manipulation of character strings such as names, commands and keyboard responses.

The library provides routines for
- identifying characters;
- comparing strings;
- searching strings;
- editing strings;
- scanning lines of text.

Type conversion library

This library contains procedures for converting numeric variables to strings and vice versa.

String to numeric conversions return two results, the converted value and a Boolean error indication. Numeric to string conversions return the converted string and an integer which represents the number of the significant characters written into the string.

Block CRC library

This library provides two functions for generating CRC codes from byte strings.

Debugging support library

The debugging support library provides four procedures. Two procedures are provided to stop a process, one on a specified condition. The third procedure is used to insert debugging messages and the fourth procedure is a timer process for analysing deadlocks.

DOS specific hostio library

The MS-DOS host file server library allows programs to use some facilities specific to the IBM PC.

Exercises

1. Get the 'Hello ' program to run!

2. Describe the different steps of program-processing!

Answers to exercises

Notice: Not all the exercises are answered here!

1.3.

Valid.
Invalid - occam reserved word.
Valid.
Invalid. Space not permitted in the names.
Invalid. Underscore not permitted in the names.

1.4.

20
30
Illegal.
Illegal.
Illegal.
FALSE
Illegal.
350

1.5.

REAL32 real1:
Not possible. The length of BYTE variables is 8 bits.
INT32 int1:

1.6.

VAL INT(16) int1 IS -16:
Not possible. BYTE is a nonnegative integer.
Not possible. BOOL can receive value only TRUE or FALSE.
VAL REAL(32) real1 IS -16(REAL32):
Not possible. The minimum value for a negative INT16 is -32768.
VAL BOOL bool1 IS TRUE:
INT seconds, minutes, index:

1.7.

Invalid. Missing colon from the end.
Valid.
Valid.
Invalid. TRUE is a reserved word of occam.
Valid.
Valid.
Invalid.
Valid.
Valid.
Invalid. Missing colon from the end.

Invalid. ‘ is not permitted.

References

CSP

John Galletly: Occam 2, Pitman Publishing, 1990

occam

G. R. R. Justo: Introduction To Parallel Programming Languages and Occam, School of Computer Science and Information Systems Engineering

John Galletly: Occam 2, Pitman Publishing, 1990

R. D. Dowsing: Introduction to concurrency using occam, VNR International, 1988

G. Jones and M. Goldsmith: Programming in occam 2, Prentice-Hall, 1988 (*A detailed book on programming in occam 2. It contains many lengthy examples of occam 2 program including the Game of Life, process farming and Huffman encoding.*)

John Wexler: Concurrent programming in occam 2, Ellis Hornwood Limited, 1989

David Ellison: Understanding occam and the transputer, Sigma Press, 1991

A Tutorial Introduction to Occam Programming, Computer Systems Architects, 1990

Stephen J. Turner (Ed.): Tools and Techniques for transputer applications, OUG-12, Proceedings of the 12th occam User Group Technical Meeting, IOS Press, 1990

The transputer instruction set: a compiler writer's guide, Inmos, 1988

The transputer instruction set: a compiler writer's guide (T805) (logical architecture), Inmos, 1990, pp. 155 (C) - ABC titles in one book

Kacsuk-Ferenczi: Parhuzamos és konkurrens programozas soktranszputeres rendszereken, BME Mérnoktovabbképzo Intézet, Budapest, 1993

Architecture

Transputer Architecture and Overview (physical architecture, all copied)), Inmos, 1990, pp. 41 (A) - ABC titles in one book

Transputer Technical Specifications: T805 (physical architecture, first pages copied), Inmos, 1990, pp. 81 (B) - ABC titles in one book

The T9000 transputer, Hardware reference manual (physical architecture, first pages copied), Inmos 1993, order code: DBT9000RMST/1, Inmos document number: 72 TRN 238-01, pp 332.

The T9000 transputer, Instruction set manual (logical architecture, first pages copied), Inmos 1993, order code: DBT9000UMST/1, Inmos document number: 72 TRN 240-01, pp 474

The transputer development and iq systems databook, SGS-Thomson, 1991

Transputer Applications Notebook: Architecture and Software, SGS-Thomson

Transputer technical notes, INMOS, 1989

Occam 2 Toolset

Occam 2 Toolset, User manual, Computer Systems Architects, 1990

Occam 2 Toolset User manual - part 1 - User guide and tools, INMOS, 1991

Occam 2 Toolset User manual - part 2 - occam libraries, INMOS, 1991

Transputer Technical Specifications, Inmos, 1990

Futher reading

CSP

C. A. R. Hoare: Communicating Sequential Processes, Prentice-Hall, 1985 (*A very thorough theoretical treatment of CSP, the basis of occam. It is must for any serious reader.*)

Occam

Occam 2 Reference Manual, Inmos, Prentice-Hall, 1988

D. Pountain and D. May: A tutorial introducing to occam programming, Blackwell Scientific Publications, 1987 (*A very readable and easy-going introduction to occam 2*)

A. Burns: Programming in occam 2, Addison-Wesley, 1988 (*A good book on occam 2. It includes sections on the transputer, the transformation on occam programs and a comparison between occam and Ada.*)

K. Bowler, R. Kenway, G. Pawley and D. Roweth: (*An introduction to occam 2 programming, Chartwell-Bratt (A good introductory book to occam 2. It contains interesting chapters dealing with configuration, parallel programs and a survey of parallel architectures.*)

J Wexler (Ed.): Developing Transputer Applications (OUG-11)

A. Bakkers (Ed.): Applying Transputer Based Parallel Machines (OUG-10)

C. Askew (Ed.): Occam and the Transputer - Research and Applications (OUG-9)

J. Kerridge (Ed.): Developments Using Occam (OUG-8)

T. Muntéan (Ed.): Parallel Programming of Transputer Based Machines |(OUG-7)

James L. McClelland, David E. Rumelhart: Explorations in parallel distributed processing, A handbook of models, programs and exercises, MIT, 1988 (in library of WU)

Cok, R. S.: Parallel programs for transputer, Prentice-Hall, Englewood Cliffs, N. J. 1991 (*literature for occam programming style*)

Architecture

Transputer Databook, SGS - Thomson

Transputer Applications Notebook: Systems and Performance, SGS-Thomson

Transputer Reference Manual, Prentice Hall, Inmos, 1988 (in library of WU)

Transputer applications

Digital signal processing, Inmos

Military and Space Transputer Databook, Inmos

Graphics Databook, Inmos

G. S. Stiles (Ed.): Transputer Research and Applications 1 (NATUG 1)

J. A. Board, (Ed.): Transputer Research and Applications 2 (NATUG 2)

L. Freeman and C. Phiollips (Eds.): Application of Transputers

Harp: Transputer applications (from Internet)

Programs and algorithms (a sorted literature)

whole occam program
not whole occam program
other language programs
only theory

Grouping of parallel methods

- *Kacsuk: 109-154*
- *Ellison: 159-160*

Dataflow-based data-partitioning

Pipeline-s

General introduction
- *Kacsuk 110-111*
- *Inmos: 8-59*
- *Galletly 192-193*

Generation of prim-number
(Kacsuk: 57-60)

Pipeline sorter
- Kacsuk: 111-119 (with C code)

Sequential and parallel program for Newton-Raphson method
- Ellison: 19-25 (sequential)
- Ellison: 25-32 (parallel)
- Ellison: 55-79 (on the network)

Formatted input and output
- (Jones: 107-133+appendix)

Parallel matrix multiplication (two-dimansional pipeline)
- Jones: 133-139+appendix
- Inmos: 58-64
- Kacsuk: 120-122 (with C code)

Tree-structured graph

Sorter
- Kacsuk: 123-126 (with C code)

Data structures as processes
- *Dowsing: 102-119 (binary trees, balancing the tree)*

Process farming
- Ellison: 85-123

The Mandelbrot program
- Ellison: 86-87

The graphics
- Ellison: 89-90

Integral-calculation
Galletly: 210-224
Kacsuk: 126-130 (with C code)

"Divide et impera!"

Integral-calculation
- Kacsuk: 130-134 (with C code)

Quick-sort
- Kacsuk: 134-135 (with C code)

Geometric parallelism

General intoduction
Galletly: 197-210 (not whole program)

Picture-processing
Kacsuk: 134-137 (with C code)

The graphics
- Ellison: 89-90

Function-parallel programs

Buffer with limitations
- Kacsuk: 139-141 (with C code)

"Philosophers on the lunch"

- Kacsuk: 141-146 (solution on the base of CSP and semaphores with C code)

Readers and writers
- Kacsuk: 146-158 (solution on the base of CSP and semaphores with C code)
- *Dowsing 51-61 (solution on the base of multi-slot buffers with separate index processes, multi-slot buffer as an array of seaprate slots, pipelined buffer))*
- *Dowsing 62-73 (solution on the base of crossbar swich and courier)*

Real-time systems
- *Dowsing 135-151*

Regular processing
Timing out of communications
Message reception
Servicing high-priority devices in parallel
An automated teller example
Simple controller/sensor example

Higher level communication primitives (decouupling, channel sharing, rendezvous, rendezvous with continuations

Other methods

Packing and unpacking routines
- Jones: appendix

Distributed implementation of buffers
- Jones: appendix

Process farming
- Jones: 92-95+appendix

Handling interrupts
- Jones: 96-105+appendix

Monitoring communication
- Jones: 140-159+appendix

Convey's game of life
(Jones: 159-174+appendix)

Huffman minimum redundancy coding
(Jones: 175-203+appendix: Simple Huffman coder + adaptive Huffman coder)

Comparing arrays
Wexler: 173-175

Time-of-day server
Wexler: 175-179

Background working
Wexler: 179-181

Multiplexing messages from concurent processes
Wexler: 181-186

Filters for protocol conversion
Wexler: 186-190

Multiplexing to use fewer channels
Wexler: 190-193

Message exchange
Wexler: 193-197

Basic communications mechanisms
- *Dowsing: 36-50 (shared variable, encapsulated shared variable, sinchronised communication, single slot buffer, multi-slot buffer, signal, binary semaphore, general semaphore, multi-slot buffer with separate index processes. multi-slot buffer as an array of single slots, pipelinde buffer)*

Communication amongst physically distributed systems
- *Dowsing: 74-79 (packet switching, circuit switching, multiply anonymous processes, server model, stream model)*

Symulating digital logic: case study in occamm
- *Dowsing: 82-101 (logic simulation, timing simulation)*

Simulation of an operating system
- *Dowsing: 121-134*

The folder editor Origami

Origami User Guide

1. Introduction

1.1. Origami

Origami is a 'folding editor', similar to, and inspired by, the editor included in the Inmos TDS (Transputer Development System). Those of us who have become familiar with the TDS editor have long bemoaned the lack of a similar editor for use with other languages, for example Pascal and C.

Origami is not a word processor (although many of its features would be useful in a word processor), but it comes into its own as a program editor for structured languages. The structuring of the code is obvious from the screen display (if Origami is sensibly used!), although no actual code may be visible.

Existing files, not produced using Origami, may be imported into Origami, and then folded up for future use, and ease of development. These files can be further edited either using Origami or some other editor (if you can bear to use one after Origami!), and compiled in the normal way.

1.2. The Folding Editor

A 'folding editor' extends the principle of tree structured directories into a text file. This allows the simultaneous display of large amounts of text, by 'folding' sections of textway behind a descriptive heading. This results in a tree structure very similar to the sub-directory structure of, for example, MS-DOS.

With suitable structuring of the text it should be possible in most circumstances to ensure that no display exceeds a single screen at any one time. To access text which is folded away you can either ENTER the containing fold, in which case the contents of the fold and its header are the only displayed text, or you can OPEN the fold, in which case the contents are displayed in the context of the surrounding text.

The advantage of this system is that it eliminates the need for seemingly endless paging through long files to find the section of interest, allowing you to move down the tree structure, following the (hopefully) descriptive headers, to locate the text you require.

1.3. Examples of Origami Folding

As an example, a Turbo Pascal file might look like this at the top level (eg immediately after starting Origami):

```
program test;
...  CONSTs
...  TYPEs
...  VARs
...  procedures
...  main body of code
```

OPENing the VARs fold might lead to the following display:

```
program test;
...  CONSTs
{{{  VARs
VAR a, b, c : integer;
   foo     : boolean;
}}}
{{{  VARs
VAR  i : integer;
    x : real;
}}}
...  procedures
...  main body of code
```

ENTERing the procedures fold might give:

```
{{{  procedures
...  statistical procedures
...  string handling procedures
...  procedure user_input
}}}
```

Each of these folds can be ENTERed further until actual code is reached - for example if we ENTER the 'procedure user_input' fold we might find:

```
{{{  procedure user_input
procedure user_input(prompt : str_type; var reply : str_type);
...  local declarations
...  local procedures
  begin
    repeat
      write(prompt);
      readln(reply);
    until verify_reply(reply);
  end;
}}}
```

2. Using Origami

2.1. Getting Started

This is the binary distribution of the origami folding editor for MSDOS machines.

It was written by Martin Green, Charlie Lawrence and this version is maintained by Johan Sunter (sun@rt.el.utwente.nl).

To get this running, you need to set the environment variables ORIGAMIHOME and ORIGAMITERM. ORIGAMIHOME needs to be set to the path of the directory where the pc.hlp and pc.key files will reside. The ORIGAMITERM environment variable must be set to 'pc. e.g if the origami files live in C:\DOS, you would set

```
set ORIGAMIHOME=C:\DOS
set ORIGAMITERM=pc
```

If you have given a wrong path, then you will receive the next error-massage:

```
Unable to open   path\vt100.key
Unable to open   path\pc.key
```

This distribution contains the following files

README	this
ORIGAMI.EXE	The editor executable
PC.HLP	The help file for the editor
PC.KEY	The keymap file for the editor
ORIGAMI.TXT	The origami 'documentation'
TUTOR.TXT	The editor tutorial text
AUTHORS	Details of the authors and maintainer

Hence a good start would be 'origami origami.txt' or 'origami tutor.txt'.

If you want the source distribution, it can be obtained from the following anonymous FTP site:

```
Site:      unix.hensa.ac.uk
Directory: /parallel/software/folding-editors/origami.tar.Z
```

Or send email containing the single line 'send help' to:

```
netlib@unix.hensa.ac.uk
```

for details of how to transfer it by email.

Dave Beckett
(djb1@ukc.ac.uk)

Origami is contained in ORIGAMI.EXE file, and hence invoked by typing 'origami' after the MS-DOS system prompt (provided of course that a path exists to the directory containing ORIGAMI.EXE). A file name may be typed after 'origami' - if this is not done a prompt for a filename will be issued.

If the file being read contains tab characters, Origami will ask for the tab size (in spaces) in order to preserve your formatting.

2.2. Editing Your File

2.2.1. The Screen Format

On starting an Origami session, a command bar, containing the word 'Origami', appears near the top of the screen. This line will remain throughout your edit, and any prompts or warnings issued by the editor will appear on this line.

If the current line (the line in which the cursor is placed, and which is marked by a '<' at the right of the screen) is too long to fit across the screen, the overflowing section is displayed on the top line of the screen, above the command bar.

Lines (other than the current line) which overflow the screen are indicated by a diamond symbol at the end of the displayed line.

2.2.2. Editing Keys

In Origami the keypad keys, the function keys, and certain ALT and CONTROL keys invoke editing functions which are similar in the main to those of the TDS editor. A list of these functions is given in Appendix A. Several keys have different functions in TDS and Origami, and these are indicated in the appendix.

2.2.3. The Editor Buffers

There are three buffers in Origami. The 'move buffer' which can contain one line of text, the 'pick buffer' into which several lines can be accumulated before inserting it back into the file, and the 'delete buffer', which holds the last deleted line (in each case a line can be a closed fold, with all its contents). The operation of these buffers in Origami is slightly different from in TDS - the move and pick buffers are lost on exit from Origami, but the delete buffer may be recalled at any time up to exiting Origami, and the contents are inserted before the current position, and not at their original location as in TDS.

2.3. Compiling Origami Files

In order for a saved file to pass a particular compiler, the fold structure must appear to the compiler as comments. As comments differ from one language to another, this means choosing the language before saving a new file for the first time, although on reading a file back into Origami the language type is retained (provided the file contained at least one fold). As a consequence of this you should take care not to include strings which the compiler would recognise as comments within the fold header, as this will lead to nested comments - not allowed in some languages.

2.4. Limitations of Origami

Origami stores your complete file in memory while editing, so the file size is limited. This limits the number of lines to about 3000 on a 640k PC. The length of a line is limited to 150 characters, and the depth of folding is limited to 256.

Appendix A

In the description of key functions below, ^ indicates a control key and 'ALT_' indicates an alt key. 'ALT_' followed by a number relates to a numeric key on the MAIN keyboard (not the keypad).

Most of the keys have identical or similar action to the TDS editor.

The Function Keys for the PC version.

SHIFT_F1 Toggles between edit and browse modes.

ALT_7 Prompts for search and replace strings, and finds the first occurrence of the search string. If you terminate search string input with an Up Arrow (keypad 8) Origami will search backward. Note that terminating any question with up or down arrow will change the search direction.

ALT_8 Repeats the last search without prompting for strings.

F1 Displays the HELP screen.

SHIFT_F2 File or Unfile a fold.

F2 Returns information on the fold depth and indentation of the current line, plus information about any fold the cursor is on.

SHIFT_F3 Places the 'pick buffer' before the current line.

ALT_F3 Removes a line from the file and adds it to the end of the 'pick buffer'.

F3 If the move buffer is empty it removes a line from the file and adds it to the 'move buffer'. Otherwise the 'move buffer' is inserted before the current line. Note that the 'move buffer' is lost on exiting Origami, unlike TDS.

ALT_F4 Copies a line into the end of the 'pick buffer' without deleting it from the file.

F4 Duplicates the current line in the file.

SHIFT_F5 Allows you to transfer to editting another file, saving the current one if required.

F6 Allows you to select the language for which the code is to be saved - ie the format in which folds are to be saved.

SHIFT_F6 Save File. File name and language type are

prompted for.

ALT_F7 Transfers the current line to the 'delete buffer', and places the old contents of the delete buffer in 'dispose chain'.

SHIFT_F7 Move left one word.

F7 Moves the cursor to the start of the current line - ie the first non-space (or the left margin if the line is empty).

ALT_F8 Transfers the line currently stored in the delete buffer, back into the file before the current line. Repeating ALT_F8 MAY allow earlier deletions to be restored.

SHIFT_F8 Move right one word.

F8 Moves the cursor to the end of the current line - ie the last non-space (or the left margin if the line is empty).

^F9 Define macro. Remembers key presses until ^F9.

SHIFT_F9 Cursor moves to the top line of the currently entered fold.

ALT_F9 Page up.

F9 Line up.

^F10 Call macro.

SHIFT_F10 Cursor moves to the bottom line of the currently entered fold.

ALT_F10 Page down.

F10 Line down.

Keypad Keys

^L Refresh screen.

Home Enter the fold at the current line.

^Home Delete to end of line.

Up Arrow Move cursor up a line (if not at top of entered fold).

PgUp Exit fold.

^PgUp Remove fold at the current line - deletes the fold structure and inserts the contents of the fold in its place.

Left Arrow Move the cursor one character left.

^Left Arrow Move the cursor one word left.

Right Arrow Move the cursor one character right.

^Right Arrow Move the cursor one word right.

End Open fold, and show the contents in the context of the current level.

^End Finish. If at top level, save file and exit from Origami.

Down Arrow Move cursor down a line (if not at bottom of entered fold).

PgDn If the cursor is within an opened fold, close this fold. If at top level, changes to the current file are saved.

Ins Create fold. Use twice - first to indicate the top of a new fold, then to indicate the bottom of it. Lines to be included within the fold must not be indented less than the fold itself. Several other keys are disabled while Create Fold is in operation.

Del Delete the character at the cursor position.

ALT Keys

ALT_1 Attach a file to a fold.

ALT_2 When the cursor is on a closed fold this key will copy the first line enclosed by that fold onto the fold line.

ALT_5 Insert File. Prompts for a filename, and imports this file into the file being edited at the current position.

ALT_8 Search. Begins a search for the current search string in the current search direction.

ALT_9 Replace. If the cursor is at the beginning of a string matching the search string then the replace string will be substituted for it.

ALT_0 Comment fold. The contents of the (closed) fold the cursor is on will be completely enclosed by the appropriate comment brackets.

Control Keys

Most of these duplicate the action of other keys.

^A Auto-fold: takes all lines to the first blank line, folds them up, inserts the first line as the fold header, and positions the cursor on the next non-blank line.

^B End of line

^E Delete to end of line.

^F Move left one word.

^G Goto line. The line number is prompted for.

^O Edit/browse.

^Q Quit from Origami.

^R Remove fold.

^V Special character.

^T Top of fold.

^U Put pick.

^Y Delete line.

Bk Space Deletes the character to the left of the cursor.

^Bk Space Undeletes the last deleted character.

Origami Tutorial

This is the folding editor tutorial for the origami folding editor.

It will give you a series of instructions which you are to follow. As you work through the tutoral, you will change this file, so make sure that you have a backup copy of the tutorial if you want to repeat it. The [HELP] key (which is F1 on the PC keyboard) or the appropriate .hlp file will explain all the logical key names that are mentioned throught.

The tutorial file is structured as a set of folds, just like all documents produced with the occam programming system. So as you work your way through the tutorial you will be learning how to use the fold operations of the editor! On starting the tutorial, try to ensure that you press only the keys instructed. Later on, you will be expected to decide which keys to press.

To start the tutorial, find the cursor moving keys on the terminal you are with "... Moving around a document". When the cursor has been placed there, press [ENTER FOLD]. This will start the tutorial section 1.

--{{{ Moving around a document

This section of the tutorial describes the operations available for opening and closing folds, and for moving around within them. Folds are the principal means for storing text and other forms of data produced by the occam programming system.

A fold is just a sequence of lines which have been "folded up" so that the lines are no longer visible. Instead a single line is displayed, called the "fold line", which usually contains a comment indicating what the lines were.

An operation called [ENTER FOLD] (which you have already used to get this far!) makes the lines in the fold available for editing.

The line marked with "... more in here" below is a fold which contains the rest of this section of the tutorial. To continue the tutorial, move the cursor down until the cursor is on this line. Then press [ENTER FOLD] to continue the tutorial.

--{{{ more in here

You will have noticed that whenever you press [ENTER FOLD] the screen is cleared, and a new set of text lines is displayed on the screen. These new lines are the contents of the fold. These are bracketted between two lines marked with "{{{" and "}}}". These two lines are called "crease lines", and the top one of them ("the top crease") also contains the text which was on the fold line where you pressed [ENTER FOLD].

As well as [ENTER FOLD], there is a function called [EXIT FOLD], which reverses the effect of the last [ENTER FOLD]. However, don't press it just yet! First move the cursor down to the line below marked "use of [EXIT FOLD]" and then press [ENTER FOLD].

--{{{ use of [EXIT FOLD]

Having pressed [ENTER FOLD] again, you have now displayed the contents of the fold "use of [EXIT FOLD".

[EXIT FOLD] is used to reverse the effect of the last [ENTER FOLD] and put you back into the position you were in when [ENTER FOLD] was last pressed.

So when you press [EXIT FOLD] now you should see a fold line marked with "use of [EXIT FOLD]", surrounded by the lines which were there when you pressed [ENTER FOLD] on that line.

You should then follow the instructions underneath that line.

Press [EXIT FOLD] now.
--}}}

If all has gone well you should have returned to here. Now move onto the fold line below, and press [ENTER FOLD]. The tutorial will continue there.

--{{{ more in here
As well as [ENTER FOLD] and [EXIT FOLD], there are two more fold operations available. These are called [OPEN FOLD] and [CLOSE FOLD].

Whereas [ENTER FOLD] clears the screen and makes the contents of the fold the only lines visible, [OPEN FOLD] makes the contents of a fold visible in the surrounding context. Think of a fold as a doorway into another room. If you ENTER the doorway you leave behind the room you were in, and you can only see the contents of the new room. If you simply OPEN the door you can see the contents of the other room, but you stay in the first room, and you can still see its contents.

Now try opening a fold, by moving to the fold below and pressing [OPEN FOLD].

--{{{ use of [OPEN FOLD]
Notice that when the fold is opened the lines below are pushed down to make room for the contents of this fold.

Now use [CLOSE FOLD] to reverse the effect of the [OPEN FOLD] operation.
--}}}

Now move to the fold below and enter it.

--{{{ more in here
So far you have only moved down the screen. You may use the cursor keys on your terminal to move around the screen.
These are [CURSOR UP], [CURSOR DOWN], [CURSOR LEFT] and [CURSOR RIGHT].

Move to the fold below and press [OPEN FOLD].

--{{{ moving around the screen
Now use [CURSOR DOWN] and [CURSOR RIGHT] to place the cursor on the 'i' of the line below. Use [CURSOR LEFT] if you go too far to the right! A line of text

A function [START OF LINE] is available to move the cursor to the first significant character of a line (that is, the first character which is not a space). Try [START OF LINE] now.

The cursor should now be on the 'A'. A function [END OF LINE] is available to move the cursor to the end of a line (that is, just after the last significant character). Try [END OF LINE] now.

The cursor should now follow the final 't'. Now press [CLOSE FOLD].
--}}}

Now enter the fold below and continue the tutorial.

--{{{ scrolling the screen

All the folds you have seen so far you have been able to view by moving around the screen and making use of the fold operations. Folding provides a way of structuring some text so that an overall view of the text can be seen on the screen. The fold operations become the principal means of moving around a document.

However, sometimes a fold has more lines in it than there are lines on the screen. This fold is an example. You can tell this because there is no bottom crease matching the crease at the top of the screen. There are more lines below the portion of the fold you can see, ending with a bottom crease.

The lines from the top crease to the bottom crease of the last entered fold are called the current view. The screen can be seen as a window into that view. There are scrolling functions to move the window up and down the view.

[LINE UP] and [LINE DOWN] move the window up and down the view a line at a time.

Try pressing [LINE DOWN] a number of times until the screen stops scrolling.

Note that the scrolling stops when the bottom crease of the view becomes the last line of the screen.

Now move the cursor down to the line below ("more on scrolling") and press [OPEN FOLD].

--{{{ more on scrolling
When a fold is opened, the lines are inserted into the view between top and bottom creases. Now press [LINE DOWN] a few more times until scrolling stops.

You may open a number of folds and add their contents to the view. Try moving the cursor down to the fold below marked "yet more on scrolling" and press [OPEN FOLD] there.
--}}}

--{{{ yet more on scrolling
Single line scrolling can become tiresome when there are many lines in the view. The functions [PAGE UP] and [PAGE DOWN] are provided to move the screen up and down the view a page (that is, a screenful) at a time.

Try pressing [PAGE UP] followed by [PAGE DOWN] now.

If you press [PAGE UP] and then repeat [LINE UP] until the screen stops scrolling, the top of the view will be reached. This is indicated by the top line containing the top crease of the view, marked with the text "scrolling the screen".

Try that now, and then press [PAGE DOWN] twice to get back here.

Now move to the fold below, and open that, using scrolling to view it.

--{{{ finishing up

Folds which you have opened to insert their contents into the view may also be closed using [CLOSE FOLD]. This function closes the nearest fold surrounding the line on which the cursor is placed.

Put the cursor here ---> <--- then press [CLOSE FOLD] and [OPEN FOLD].

Notice where the cursor ends up.

You have now finished the section on scrolling. When you now press [EXIT FOLD] it will reverse the effect of the last [ENTER FOLD]. The folds you have opened since then will automatically be closed. Press [EXIT FOLD] now.
--}}}

<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb> Page 2
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb> Page 3
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>

<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb> Page 4
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb> Page 5
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>

<blurb>
<blurb>
--}}}
--}}}

When you have finished that, read the contents of the following fold, which provides a summary of the functions you have seen so far.

--{{{ Summary of moving operations
The functions available for moving around the fold structure are :-
[ENTER FOLD]
  Makes the contents of a fold available for editing.

[EXIT FOLD]
  Reverses the effect of the last [ENTER FOLD].

[OPEN FOLD]
  Inserts the contents of a fold into the current view.

[CLOSE FOLD]
  Removes the contents of an open fold from the view.

[CURSOR UP], [CURSOR DOWN],
[CURSOR LEFT], [CURSOR RIGHT]
  Move the cursor around the screen.

[START OF LINE], [END OF LINE]
  Move the cursor to the start or the end of a line.

[LINE UP], [LINE DOWN]
  Move the screen up and down the view a line at a time.

[PAGE UP], [PAGE DOWN]
  Move the screen up and down the view a page at a time.
--}}}

Once you have finished with this, press [EXIT FOLD].
--}}}

Once you have finished with this, press [EXIT FOLD].
--}}}

Once you have finished with this, press [EXIT FOLD].
--}}}

Once you have finished this , you should press [EXIT FOLD] to return to the top level of the tutorial. Continue with the other sections in the same way.
--}}}

--{{{ Creating folds and inserting text

Now that you have learned how to browse around the fold structure, you should not need to be given explicit instructions on what to do with each fold in this document. It is generally best to enter each fold to access its contents, and then exit it when it is finished.

The folds below describe how to insert and edit text, and how to create and remove folds. Work through them in the order they appear, and then return to the level above to move on to the next section of the tutorial.

--{{{ Inserting and deleting characters

It is fairly straightforward to insert text. In general any characters typed are inserted at the cursor position. So for example you can insert some text on the line below by moving the cursor to the end of the line and typing " discontent"

"Now is the winter of our
made glorious summer by this son of YORK.
" Richard III- Shakespeare

There are two character deletion functions :-

[DELETE], which deletes the character to the left of the cursor, and [DELETE RIGHT], which deletes the character under the cursor. Try deleting the last few characters of the line and retyping them.

If you type [RETURN] when the cursor is at the end of the line a new line is created. The cursor is positioned under the first character of the line. This allows you to create indented text without having to adjust the cursor every time you make a new line. Try typing [RETURN] now, followed by some text.
--}}}

--{{{ Creating and removing folds

A fold can be created using the [CREATE FOLD] function. Two presses of [CREATE FOLD] are needed, one to mark the top of the fold and one to mark the end of a fold.

--{{{ A simple example of fold creation

Try making a fold from the following lines. First put the cursor on the line marked "line 1", in the first column of the screen, and press [CREATE FOLD].

Then move the cursor down until it is on line 4, and press [CREATE FOLD] again.

line 1
line 2
line 3
line 4

Note that the fold has been made from the line on which [CREATE] fold was first pressed, up to (but not including), the line on which the second press of [CREATE FOLD] was done.

The cursor has been placed after the "..." which indicates the fold. Type some text at the cursor position ; for example "lines 1-3". Whenever a fold is made it is a good idea to type some text here to indicate what the fold contains. This text is known as the fold header.

Open and close the fold you have made to confirm that it contains lines 1 to 3.
--}}}

--{{{ Fold indentation

Every fold has an indentation associated with it. The indentation may be changed by inserting or deleting spaces on the fold line before the fold marker (that is, the "...").

Try this on the following fold. First open it to see its contents, then close it. Then, with the cursor in the first column of the screen, type four spaces.

```
--{{{  a fold to indent
There are two lines
in this fold.
--}}}
```

These spaces should appear before the fold marker. Then press [OPEN FOLD] again. By indenting the fold line, you have indented the contents of the fold as well!

--{{{ Some comments on insertion and deletion of text in folds It is not possible to insert or delete text to the left of the leftmost column of an open fold (that is, the column in which the crease marks "{{{"
and "}}}" begin).

Only spaces may be inserted or deleted before the fold marker of a closed fold.

The markers which indicate fold or crease lines consist of the three characters ("...", "{{{", or "}}}") and two spaces following. Text may not be inserted or deleted within these markers.

```
 --{{{  a fold to indent
 There are two lines
 in this fold.
 --}}}
```

You may wish to try insertions or deletions on the fold above to confirm these restictions on where text may be inserted or deleted. Try deleting characters before an indented fold line marker to reduce the indentation of the fold.
--}}}

Try pressing [ENTER FOLD] on the following (indented) fold line :-

```
 --{{{  This fold indented by two spaces
 Note that the view of the contents of this fold starts at the left-hand
 edge of the screen even though the fold was indented.

 Having entered a fold, we are only interested in its contents, not in its
 relation to the surrounding lines. So the indentation is ignored for the
 moment.

 This allows us to view quite deeply indented text (such as occam program
 text) without having a large amount of blank space on the left hand side
 of the screen.

 [EXIT FOLD] returns us to where we were before.
```

```
  --}}}
--}}}
```

--{{{ Creating indented folds

When a fold is created, the indentation of the newly created fold is taken from the position of the cursor when [CREATE FOLD] is first pressed.

Try creating a fold within the following program text. To avoid problems, make sure all the program text is on the screen before starting. First move the cursor to the 'S' of "SEQ" and press [CREATE FOLD]. Then move beyond the last line of the program and press [CREATE FOLD] again.

```
IF
  record[i] = key
    --{{{  found key
    SEQ
      found := TRUE
      position := i
    --}}}
  record[i] <> key
    i := i + 1
```

The message "Cannot create : lines in fold have incorrect indentation" will appear. This is because the lines you have asked to be created in the fold contain one line "record[i] <> key", whose indentation is less than that of the fold being created. Instead move the cursor to that line and press [CREATE FOLD] there so that there are three lines in the fold.

Type a comment such as "found key" on the fold you have created.

This example shows how indented folds fit well into occam program text, in which indentation is significant.

If the indentation of a fold you have made is not right, you can always adjust it by inserting or deleting spaces before the fold marker, as previously described.

```
--}}}
```

Between the two presses of [CREATE FOLD], the only operations allowed are cursor moves and screen scrolling operations. The message "Creating fold..." appears to remind you what you are doing.

A fold can be removed using the [REMOVE FOLD] function.

--{{{ Example of fold removal

A fold may be removed by placing the cursor on the fold line and pressing [REMOVE FOLD]. The contents of the fold are inserted into the current view instead of the fold. The indentation of the inserted lines are the same as when the fold is opened.

For example, try opening the folds in the following text, closing them again and then removing them using [REMOVE FOLD].

```
SEQ
  --{{{
```

```
  going := TRUE
  WHILE going
    SEQ
      input ? command
      IF
        command = finish
          going := FALSE
        command <> finish
          SKIP
  --}}}
```

Now try creating the folds again! The result should look like the contents of this fold :-

```
--{{{  copy of original program text above
    SEQ
SEQ
  WHILE going
      input ? command
      IF
          SKIP
          SKIP
          SKIP
          going := FALSE
        command <> finish
  going := TRUE
  going := TRUE
  going := TRUE
        command = finish
          SKIP
--}}}
--}}}
```

--{{{ Altering program indentations

The [CREATE FOLD] and [REMOVE FOLD] functions, along with the ability to alter the indentation of a fold, are used together to change the indentation of a block of program.

For example, the following program fragment has a section which is incorrectly indented.

```
IF
  traffic.light = red
  SEQ                    --  These lines
    foot.on.brakes ()    --  should be indented
    stopping := TRUE     --  two more spaces
  traffic.light = green
    foot.on.accelerator ()
```

The lines indicated need to be indented more. Start by creating a fold around these lines. Put the cursor on the 'S' of "SEQ", press [CREATE FOLD]. Then move down to the 't' of "traffic.light = green" and press it again.

Then indent the created fold by doing [START OF LINE] and inserting two spaces. Now press [REMOVE FOLD] to insert the correctly indented text into the program.

Although this may seem a bit long-winded at first, it will become much easier as the fold operations become more familiar!
--}}}
--}}}

--{{{ Inserting and deleting lines

The folds below describe a number of techniques for inserting and deleting lines in a document.

--{{{ Using [RETURN] and [DELETE]

We have seen above how lines may be created by using [RETURN] at the end of a line. [RETURN] may also be used to split lines in two. Try placing the cursor between the arrows on the line below and typing [RETURN].

Try splitting this line here --> <--
Line underneath

The upper line is split into two, with the right-hand section inserted as the line below. Now try pressing [DELETE]. The lines are joined together again.

However, a fold line may not be split, or joined with another line. Try going to the line below, and splitting it between the arrows. Also try using [DELETE] on the 'L' of the line underneath

--{{{ try splitting this fold line here --> <--
[RETURN] at the end of the line above
inserts a line at the start of this fold.
--}}}
Line underneath

Similar restrictions apply to crease lines. However, [RETURN] may be used at the end of these lines. Try opening the fold marked "try splitting", moving to the end of the crease line and pressing [RETURN].
--}}}

--{{{ Using [LINE DELETE] and [LINE UNDELETE]

As well as the functions described above for joining and splitting lines, there is a function which deletes a line from the document altogether. To use this, move to the line you want to remove and press [DELETE LINE].

Try moving to the line marked "A line to delete" below and deleting it.

A line
A line to delete
--{{{ a fold line to delete
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>

<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
--}}}
Another line

Try pressing [DELETE LINE] again. A fold line may also be deleted! Since there may be a lot of text contained in the fold, this makes [DELETE LINE] quite a powerful operation. So it should be used with care!

If you delete a line accidentally you can get it back using [UNDELETE LINE]. Try pressing that now. Undeletion is an "undo" type of operation which should be used immediately after the deletion, if you had cause to regret it. Any operations other than cursor moves or screen scrolling operations cause the line to disappear for good.

A crease line may not be deleted. Try opening the fold below and deleting its contents. The deletions stop when you reach the end of the fold.

--{{{ a fold line
delete me
and me
--}}}
--}}}
--}}}

--{{{ Moving and copying

The editor provides two functions which may be used together to copy sections

--{{{ Moving lines

[MOVE LINE] can be used twice to move a line from one part of the document to another. The first press of [MOVE LINE] is used to pick up the line - it picks up the line which the cursor is on. The second press is used to put the line down - it inserts the picked line above the line which the cursor is on.

Try [MOVE LINE] on the following text - use it to place the lines in order. First move the cursor to the line marked "March" and press [MOVE LINE]. This line will disappear - it is now in the move buffer. Now move the cursor to the line marked "April". Press [MOVE LINE] again. It will insert the "March" line in the correct place.

March
January
February
April

Any set of editor functions may be used between the presses of [MOVE LINE]. However, when you finish a session with the occam programming system and return to operating system level, the contents of the move buffer are lost.

The fact that the move buffer only contains one line means that it is easy to reverse if you press it accidentally - it doesn't insert a long sequence of lines into your document. If you want to move a sequence of lines around, fold them up first.

[MOVE LINE] can be used on a fold line in exactly the same way as on a text line. So major reorganisations of a folded document can be done with a few key presses. For example, try reorganising the following document:

```
--{{{  The Perils of Pauline
--{{{  Chapter 3
--{{{  the rest of the chapter
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
--}}}
And they both lived happily ever after.

THE END.
--}}}
--{{{  Chapter 2
The tall dark stranger staying at the inn that night introduced himself to Pauline.
"My name is Ramon" he said, in a husky voice.
--{{{  rest of chapter
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
--}}}
--}}}
--{{{  Chapter 1
It was a cold and rainy night as Pauline set out
on the most momentous journey of her life.
--{{{  the rest of the chapter
<blurb>
<blurb>
<blurb>
```

<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
<blurb>
--}}}
--}}}
--}}}
--}}}

--{{{ Copying lines

The [COPY LINE] key copies the line which the cursor is on, inserts the new

Copy me (a text line)

--{{{ Copy me (a fold line)

All my contents
are copied too!
--}}}
--}}}
--}}}

--{{{ Summary of editing operations

The keys you have used in this section of the tutorial are the following:

Character keys : insert characters at the cursor position

[RETURN] : splits a line in two

[DELETE] : deletes the character to the left of the cursor
It may be used at the left edge of a line to join
lines.

[DELETE RIGHT] : deletes the character under the cursor

[DELETE LINE] : deletes the current line (including fold lines)

[UNDELETE LINE] : used after a [DELETE LINE], restores the line

[MOVE LINE] : used twice to move a line; once to pick it up, and
once to put it down

[COPY LINE] : duplicates the current line

[CREATE FOLD] : used twice to create a fold, once to mark the top of
the fold, and once to mark the bottom

[REMOVE FOLD] : on a fold line, removes the fold and inserts its
contents at that position in the document
--}}}
--}}}

You can now press [EXIT FOLD] to return to startup fold. As you have been making changes to this file as part of the tutorial, a new copy of the file will be written out as you exit.